



Programmation en langage C

Avant-propos



Règles du jeu notation / LLMs ✓✗

- 👍 Les TPs ne sont **pas** notés
- 🧠 L'utilisation de Copilot / ChatGPT ou d'autres LLMs est **interdite**
- 📋 Notation : 1 exam de mi-parcours, 1 exam final, 1 note de "participation"

FAQ 🚑

- 😊 "Oups j'ai utilisé un LLM" → "Oups je t'ai enlevé des points de participation"
- 🤦 "Oui mais je n'étais pas au courant / j'ai oublié / c'était pour autre chose" → Même tarif pour tout le monde
- 😰 "Je suis anxieux concernant la difficulté de l'exam" → Pas de surprises à l'exam, pas de sujets exploratoires

Avant-propos



Objectifs



- Gagner en **discipline**, le C est permissif, le compilateur donne peu de garanties (contrairement à Rust par exemple)
- Comprendre le fonctionnement d'un **processus**
- Avoir un "**langage d'ancrage**", suffisamment haut niveau pour raisonner en langage naturel, suffisamment bas niveau pour comprendre la mise en œuvre du processus. Il s'agit donc de **lever les couches d'abstraction** (comme en Python) ne permettant pas de comprendre les limites de performance
- Avoir des **bases informatiques solides**, et vivre la transition C → JS / Python comme une récompense plutôt qu'une transition JS / Python → C comme une punition.
- **Influencer positivement l'écriture de code haut niveau**, en cherchant comment peuvent être implémentées les fonctions des lib standard, par exemple append(val) ou insert(idx, val) des listes en Python
- **Faciliter l'accès à certains domaines**, comme l'embarqué : robotique, spatial, IoT, etc.

Avant-propos



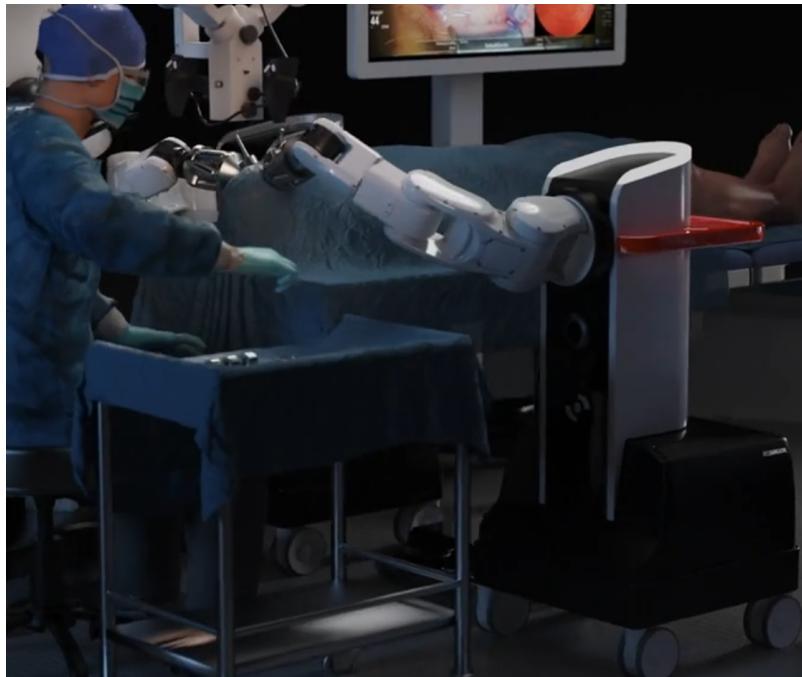
Historique

- Le langage C est développé par Dennis Ritchie et Brian Kernighan en 1972
- Première version de glibc en 1987
- ANSI C en 1989, ISO en 1990 (C90)
- Standards C99, C23 (en 2024)
- Les OS modernes sont écrits (parfois à 100%) en C
- Une base de nombreux langages (le Python est écrit en C)
- De nombreux langages sont basés sur la syntaxe du C comme C++, Objective-C, C#, Java, ou à minima fortement influencés par sa syntaxe, comme Javascript, Go, Rust, Swift, etc.

Avant-propos



Exemple d'application :



Robot pour la chirurgie de la rétine

Exemple de technologies :



Interfaces utilisateurs
bureau, mobile
et embarqué



Microcontrôleurs



UNREAL
ENGINE

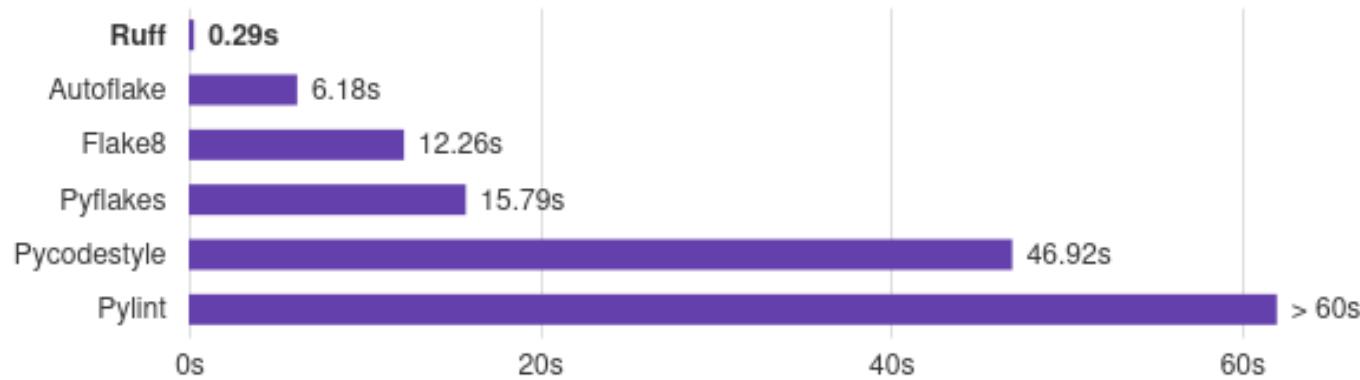
Avant-propos



Un enjeu DDRS

L'informatique a un impact environnemental certain, la sobriété en tant qu'utilisateur et une chose, mais il reste important de réfléchir à l'efficience énergétiques des systèmes en tant qu'ingénieur

Exemple DevOps : utilisation d'outils de qualité logicielle. Ces outils sont exécutés à chaque pipeline d'addition de code sur le gestionnaire de version. Ils sont souvent exécutées dans votre IDE, potentiellement à chaque modification.



Ruff peut remplacer tous les autres, et c'est le seul implémenté en langage de bas niveau

Avant-propos



Plan de bataille

- Environnement de développement
- Variables, Types, Mémoire, Instructions
- Pile d'exécution et Processus
- Blocs, Fonctions, Opérateurs, Conditions
- Boucles, Tableaux, Chaînes de caractères
- Pointeurs & Structures
- Allocation dynamique & Casting
- Interpréteur Python
- Préprocesseur & VLAs
- Enums, Labels, Switchs & Unions
- Pointeurs génériques & pointeurs de fonction
- Dynamique Dispatch
- Qualité logicielle & Ownership
- Prototypes, Headers, Compilation
- Makefile
- Templates & Generics simulés

Avant-propos



⚠️ Attention le cours est complet..

ok tout ça est présent, sur le reste du plan?

Oui — globalement ton plan est **complet et mature** (on sent le C “qui fait mal”, pas le C jouet 😊).





Environnement de développement

Environnement



Ubuntu (quelques raccourcis)

- **CTRL + ALT + ← / →** pour changer d'espace de travail
- **CTRL + ALT + SHIFT + ← / →** pour changer d'espace de travail en emportant la fenêtre active
- **CTRL + ALT + T** pour ouvrir un nouveau terminal
- **ALT + TAB** pour changer de fenêtre active
- **F11** ou **Super (windows) + ↑ / ↓** pour le mode plein écran sur une fenêtre (maximiser/minimiser)
- **Super (windows) + ← / →** pour mettre une fenêtre sur la partie gauche / droite de l'écran (**Echap** pour annuler la section pour la seconde moitié)
- Bien choisir son terminal, par exemple, zsh + terminator

Se créer un dossier de travail

- Créer un dossier avec **mkdir (make directory)**
- Lancer **VSCode** avec ce dossier comme environnement

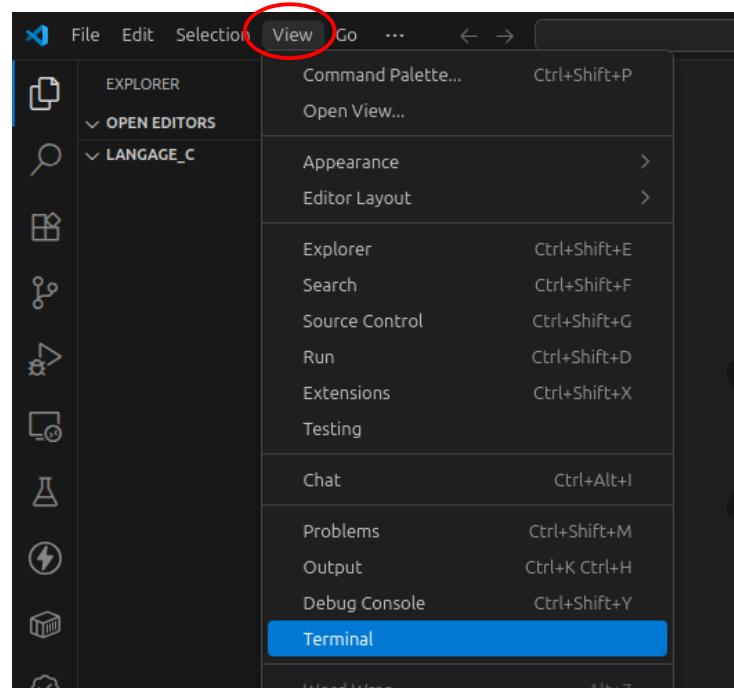
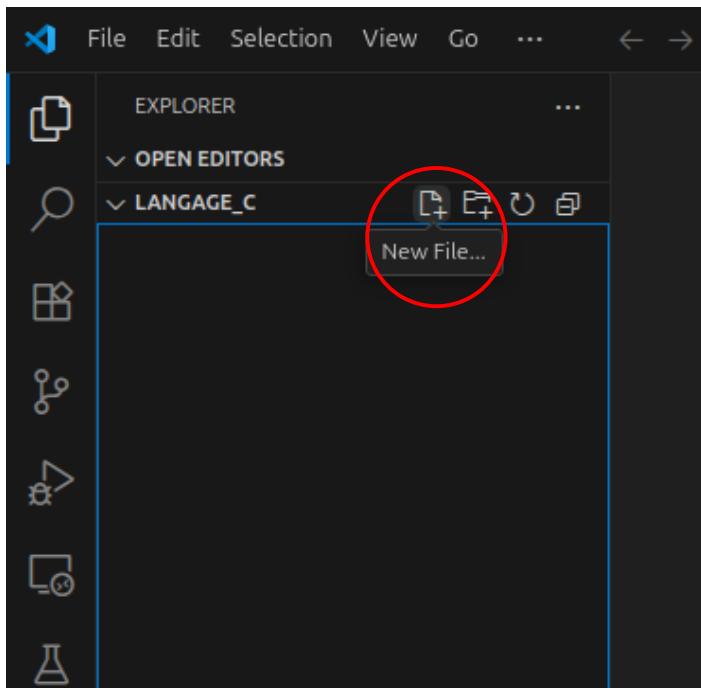
A screenshot of a terminal window titled "olivier@olivier-lirmm:~". It shows two tabs open, both with the command "(base) ~" and a small terminal icon. The window has standard window controls (minimize, maximize, close).A screenshot of a terminal window titled "olivier@olivier-lirmm:~". The user is running the command "mkdir langage_c". The terminal shows the command being typed and the resulting output: "(base) ~ mkdir langage_c". A red arrow points from the "Se créer un dossier de travail" section towards this terminal window.

Environnement



VSCode

- Créer un nouveau fichier
- Ouvrir un terminal intégré



Environnement



VSCode (raccourcis utiles)

- **ALT + SHIFT + ↑ / ↓** pour passer en mode multicurseur
- **CTRL + D** pour sélectionner le mot, rappuyer sur **D** pour passer en mode multicurseur sur la prochaine occurrence
- **F2** pour renommer un symbole (il ne s'agit pas d'un simple rechercher/remplacer)
- **CTRL + /** pour commenter une ou plusieurs lignes (en fonction de la sélection)
- **CTRL + K** suivi de **CTRL + S** pour apprendre de nouveaux raccourcis ou pour les modifier

Il n'existe pas de raccourcis sur clavier français pour passer du terminal à l'éditeur

Vous pouvez créer de nouveaux raccourcis. Ci-dessous la configuration pour mettre en place un toggle de focus sur le terminal / éditeur avec **CTRL + !** (clic droit → "change when expression" pour ajouter la condition)

Terminal: Focus on Terminal View	Ctrl + !	!terminalFocus	User
View: Focus Active Editor Group	Ctrl + !	terminalFocus	User

Environnement



Fichiers C et exécution d'un programme

- Les fichiers sources auront l'extension **.c** les fichiers d'entête l'extension **.h**
- On compilera avec **gcc** (*GNU C Compiler* puis *GNU Compiler Collection*), dans un premier temps sans options particulières
- On appréciera cependant l'option **-o** (pour **output**). Les fichiers binaires n'auront pas d'extension. Sans l'option **-o**, gcc crée un fichier nommé **a.out**.
- On exécutera le binaire avec **./<nom_executable>** (ou **./a.out**)

```
● (base) → examples gcc minimal_prog.c -o minimal_prog
● (base) → examples ./minimal_prog
    Welcome to the IG3 C Programming Course
○ (base) → examples
```



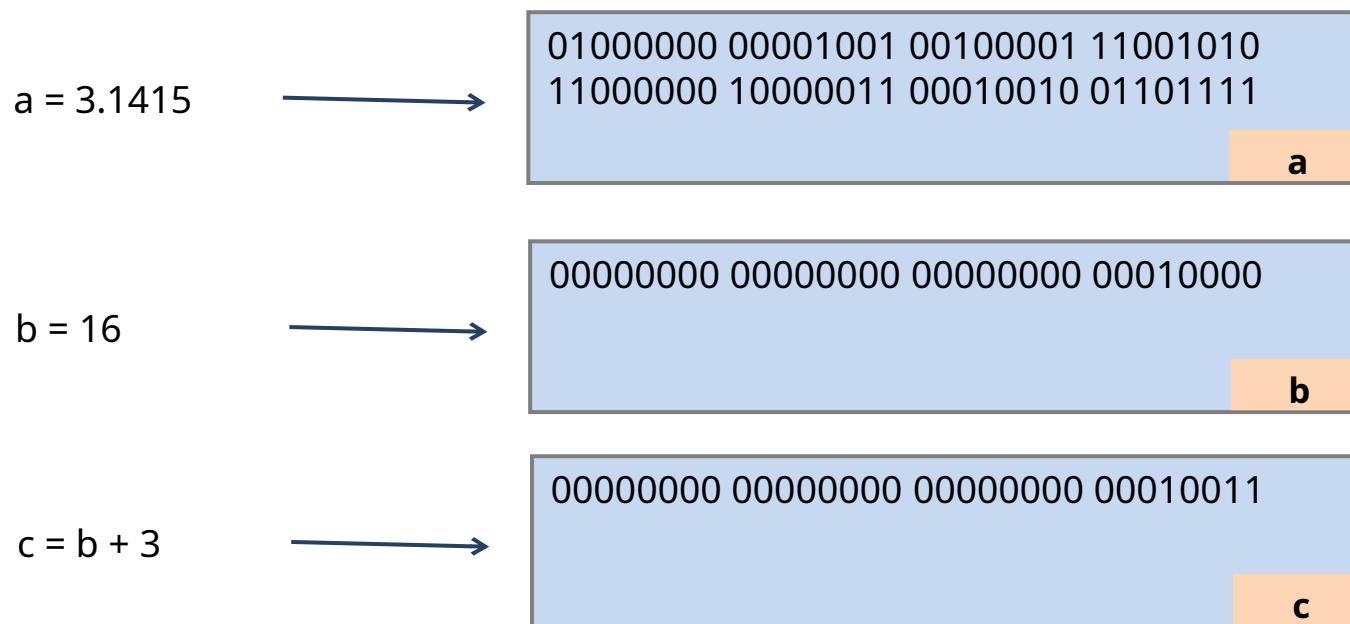
Variables, Types, Mémoire, Instructions

Variables, Types, Mémoire, Instructions



Les variables

- Une variable est un **label**, une étiquette vers une zone mémoire contenant des données
- C'est à travers ce label que le programmeur manipule la mémoire (lecture et écriture)

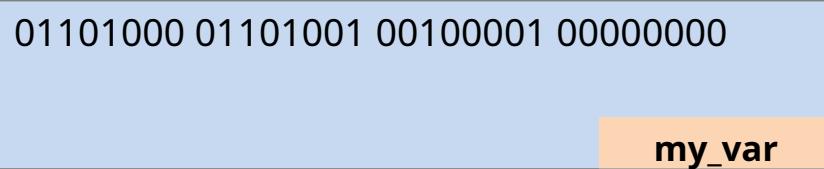


Variables, Types, Mémoire, Instructions



Les variables

- On souhaite manipuler cette zone mémoire (4 octets)



- Qu'est-ce que "my_var" est censée représenter ?
 - Les nombres **104, 105, 33 et 0** ?
 - Les nombres **26729 et 8448** ?
 - Le nombre **1751720192** ?
 - La chaîne de caractères « **hi!** » ?
 - Le nombre **4.4036805x10²⁴** ?

→ **il nous manque une notion**

Variables, Types, Mémoire, Instructions



Les types (primitifs)

- Définir la **représentation / interprétation** des données
 - Exemple : l'octet 00001011 en mémoire représente le nombre **42**, ou le caractère '*****' en fonction du type qu'on "lui" associe.
- Définir la **taille** de la zone mémoire associée au label
 - Exemple : un "**char**" est sur un octet, un "**int**" généralement sur 4 octets.
- Définir les **types d'opérations**
 - Exemple : la multiplication des nombres flottants n'est pas câblée comme la multiplication des entiers
 - **mais il nous manque encore une notion**

Variables, Types, Mémoire, Instructions



La mémoire

Encodage d'un int : une zone mémoire de 4 octets stockant l'entier **1165517165**, en binaire **01000101 01111000 01100001 01101101** * :

0x00000083 (131)	0 1 1 0 1 1 0 1
0x00000082 (130)	0 1 1 0 0 0 0 1
0x00000081 (129)	0 1 1 1 1 0 0 0
0x00000080 (128)	0 1 0 0 0 1 0 1

L'adresse de cette case est celle de son premier octet, ici **128 (0x80 en hexadécimal)**

0x00000083 (131)	1165517165
0x00000082 (130)	
0x00000081 (129)	
0x00000080 (128)	

On parlera d'**adresses hautes** et d'**adresses basses**

→ mais il nous manque une dernière notion

adresses hautes

0x00000083 (131)
0x00000082 (130)
0x00000081 (129)
0x00000080 (128)

...

1165517165

stack

free

On ignore ces sections pour l'instant

adresses basses

Variables, Types, Mémoire, Instructions



Instructions

- Un programme est une suite d'instructions exécutées par le processeur
 - En langage C la granularité sera moins fine, car une instruction C peut être traduite par plusieurs instructions processeur, et on peut distinguer des types d'instructions :
 - Expressions (et affectations)
 - Conditions,
 - Appels de fonctions
 - Etc.
 - Une instruction en C se termine par un **point virgule**
- Ok, inventaire de la caisse à outils : nous avons donc des **variables**, des **types**, de la **mémoire**, des **instructions**

C'est parti

Variables, Types, Mémoire, Instructions



Déclaration d'une variable typée

1) Type : mot-clef

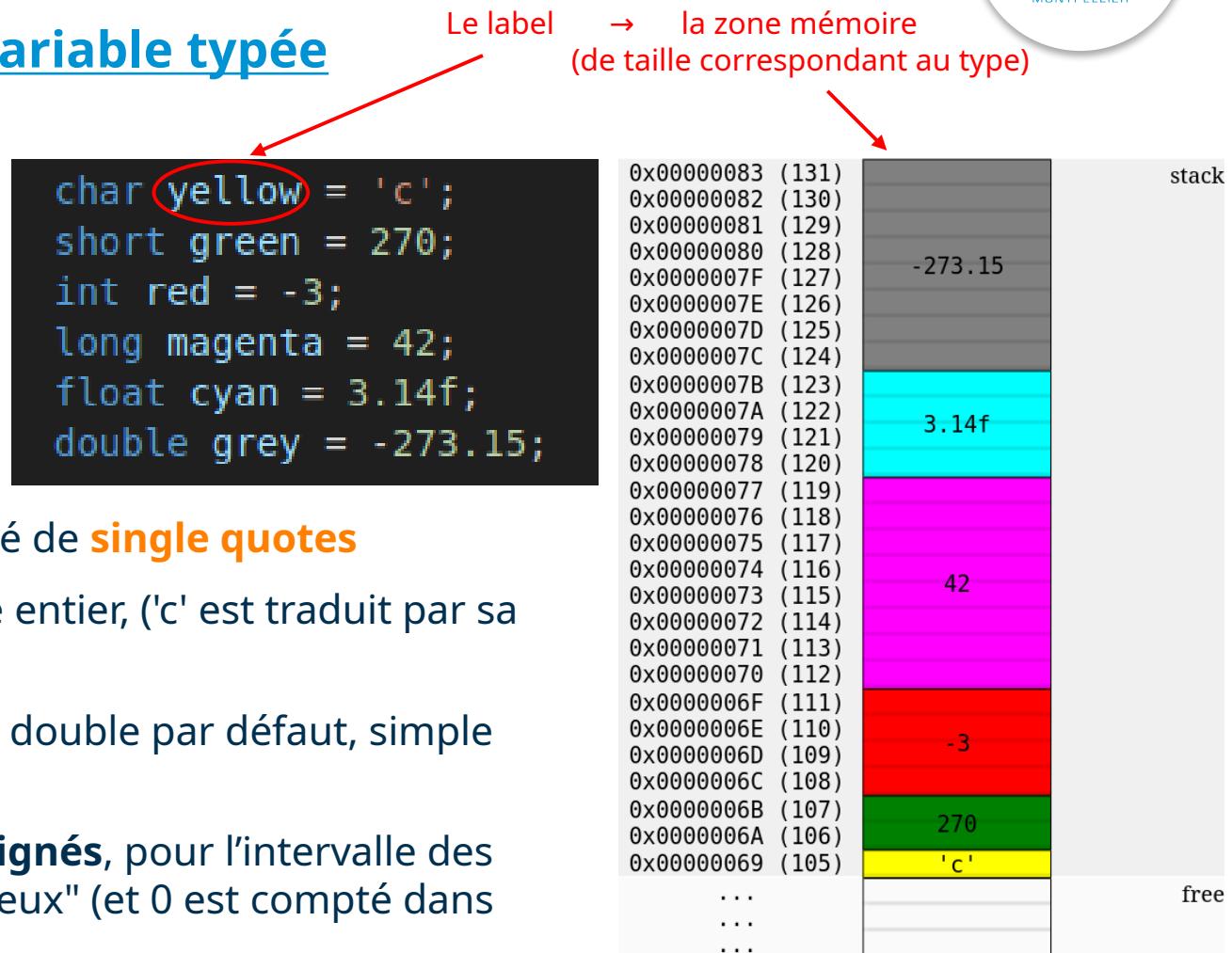
2) Nom : le label ou étiquette

3) "=" : **initialisation**

4) Valeur

Remarques :

- Un caractère est entouré de **single quotes**
- "char" est aussi un type entier, ('c' est traduit par sa valeur en ascii)
- Une valeur flottante est double par défaut, simple avec le suffixe "**f**".
- Les types entiers sont **signés**, pour l'intervalle des valeurs, on "coupe en deux" (et 0 est compté dans la partie positive)



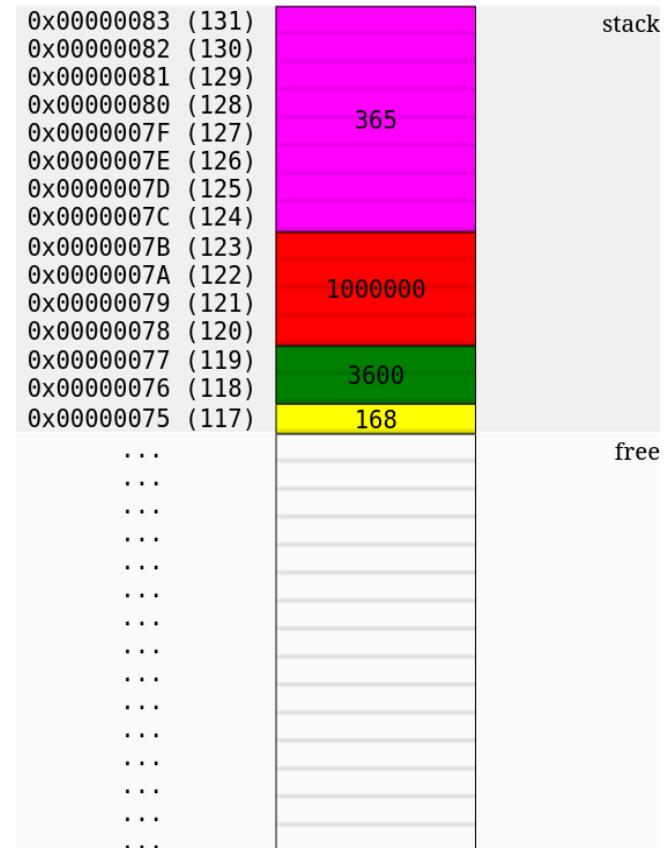
Variables, Types, Mémoire, Instructions



Les types non signés

- Déclaration avec le mot-clef "**unsigned**" devant le type entier
- La range (intervalle) est adaptée, e.g. de [0, 255] pour un "**unsigned char**" contre [-128, 127] pour le "**char**"

```
unsigned char yellow = 168;
unsigned short green = 3600;
unsigned int red = 1000000;
unsigned long magenta = 365;
```



Représentation et langage courant

- Vulgairement, les mots "variable" et "valeur" peuvent être confondus
- Pour des raisons de visibilité : les représentations du cours ne présenteront pas forcément les données en binaire

Variables, Types, Mémoire, Instructions



Affichage sur la sortie standard

- Utilisez la fonction "**printf**" définie dans "**stdio.h**" (standard Input/Output header)
- La fonction attend au moins un argument, une chaîne de caractères pouvant contenir des "format specifiers" (par exemple, "%d"). Dans les cas simples, le "**format specifier**" est composé d'un "**length modifier**" optionnel et d'un "**conversion modifier**".
- La fonction attend ensuite les arguments à formater, ce nombre est donc variable et peut être nul

```
float zero = -273.15;
int var1 = 42, var2 = 123;
printf("La valeur du zero absolu est %f\n", zero);
printf("var1 vaut %d et var2 vaut %d\n", var1, var2);
printf("La taille d'un int est %lu\n", sizeof(int));
```

Intéressant..

```
La valeur du zero absolu est -273.149994
var1 vaut 42 et var2 vaut 123
Un int est encodé sur 4 octets
```

- Dans l'exemple ci-dessus, **float** et **int** ont la taille par défaut, le **length modifier** n'est pas requis. On utilise les **conversion modifier** "d" (pour "decimal integer") et "f" (floating-point number)
- L'opérateur **sizeof** rend par contre un entier long non signé (unsigned decimal integer)

Variables, Types, Mémoire, Instructions



Premier programme complet

- La fonction **main** est le point d'entrée
- La fonction main doit retourner **0** si tout se passe bien
- Compilez avec **gcc** (GNU C Compiler). Omettre l'option `-o <name>` donnera un exécutable nommé "a.out"
- **Importez les headers** (fichiers d'entête) pour l'affichage et la lisibilité du code

The screenshot shows a terminal window with the following content:

```
C minimal_prog.c ×  
home > olivier > Documents > Lectures > C > examples > C minimal_prog.c > main(void)  
1 #include <stdio.h> // includes printf  
2 #include <stdlib.h> // includes EXIT_SUCCESS  
3  
4 int main(void) {  
5     printf("Welcome to the IG3 C Programming Course\n");  
6     return EXIT_SUCCESS; // EXIT_SUCCESS is equal to 0  
7 }  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL  
● (base) → examples gcc minimal_prog.c -o minimal_prog  
● (base) → examples ./minimal_prog  
Welcome to the IG3 C Programming Course  
○ (base) → examples
```

The terminal shows the compilation of the file `minimal_prog.c` into an executable `minimal_prog`, followed by its execution which prints the welcome message to the console.

Le texte derrière `//` sur une ligne unique est un commentaire

Le texte entre `/*` et `*/` sur une ou plusieurs lignes est un commentaire

Variables, Types, Mémoire, Instructions



Ok pour "%d", "%f", "%lu"... et le reste ? please RTFM

- Les LLMs peuvent se tromper, consomment beaucoup d'énergie, et vous ne perdrez pas forcément beaucoup plus de temps pour des informations simples
- Utilisez la section 3 du system's manual pager → `(base) → ~ man 3 printf`
 - Testez **man** pour printf
 - Tapez "**/modifier**" et entrée
 - Tapez "**n**" pour "next occurrence"
 - Tapez "**q**" pour "quit"

```
printf(3)           Library Functions Manual      printf(3)
NAME
    printf, fprintf, dprintf, sprintf, snprintf, vprintf,
    vfprintf, vdprintf, vsprintf, vsnprintf - formatted output
    conversion

LIBRARY
    Standard C library (libc, -lc)

SYNOPSIS
    #include <stdio.h>
    /modifier
```

Pour rechercher le mot "modifier"

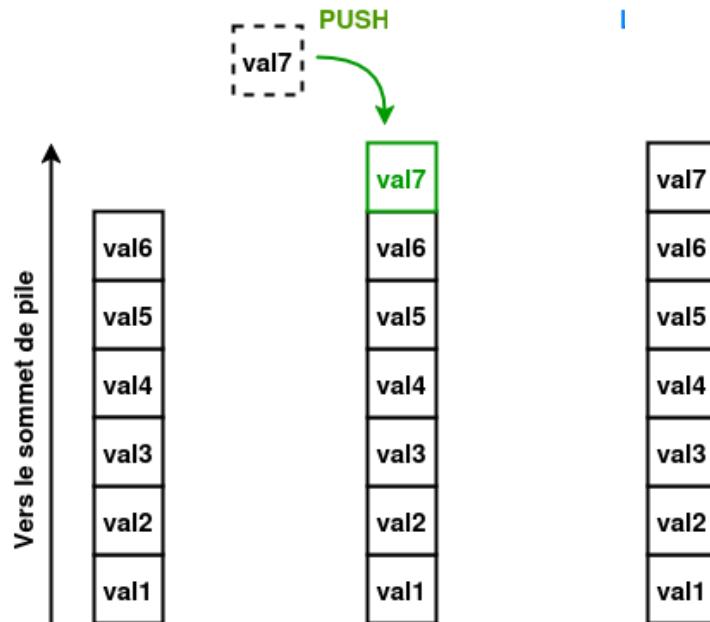


Pile d'exécution et Processus

Pile d'exécution et Processus

Une pile est une structure linéaire LIFO (Last In First Out)

- On ne peut ajouter un élément qu'au sommet de la pile (**PUSH**)

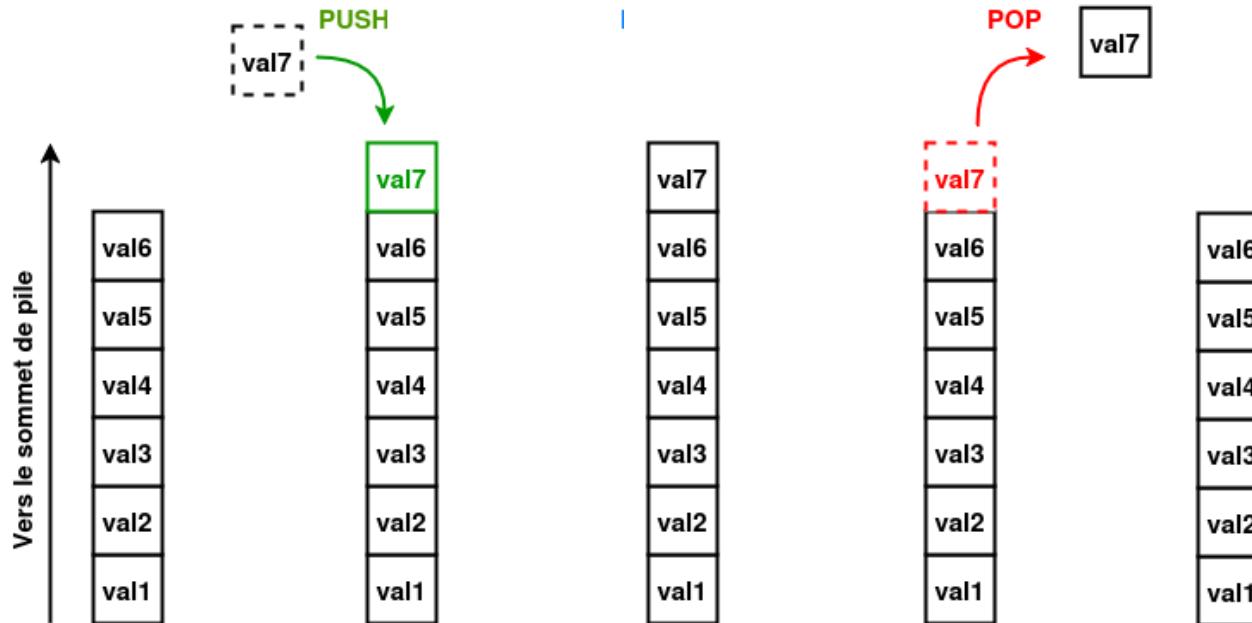


Pile d'exécution et Processus



Une pile est une structure linéaire LIFO (Last In First Out)

- On ne peut ajouter un élément qu'au sommet de la pile (**PUSH**)
- On ne peut enlever que l'élément au sommet de la pile (**POP**)

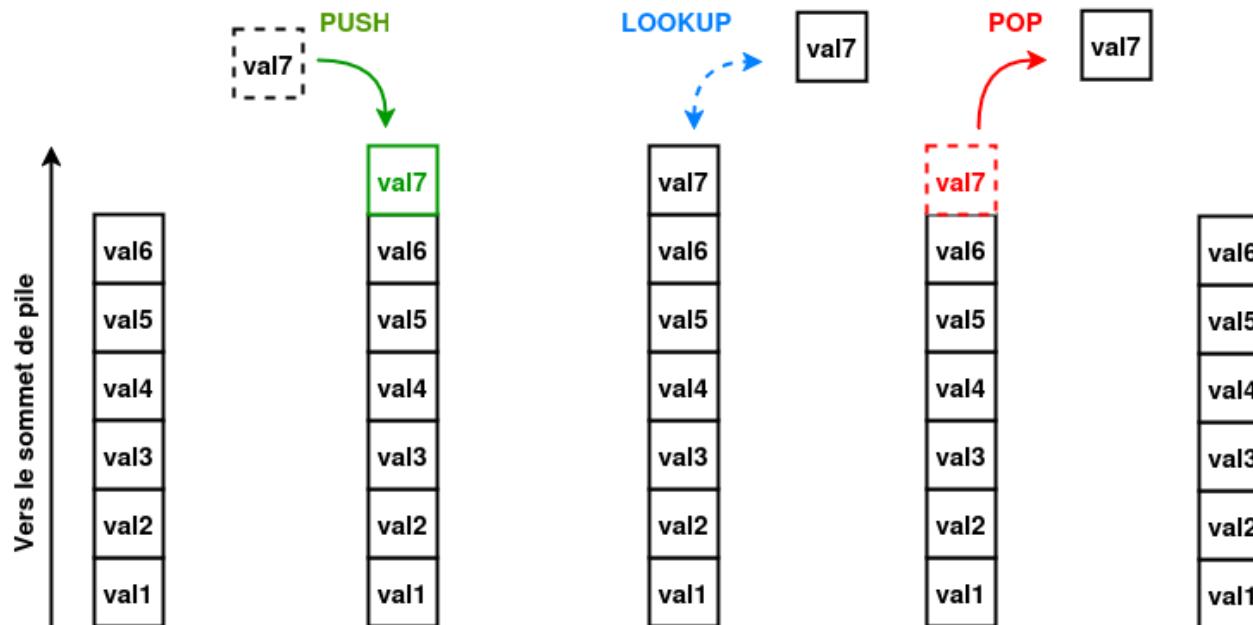


Pile d'exécution et Processus



Une pile est une structure linéaire LIFO (Last In First Out)

- On ne peut ajouter un élément qu'au sommet de la pile (**PUSH**)
- On ne peut enlever que l'élément au sommet de la pile (**POP**)
- On peut également ajouter des opérations pour inspecter le sommet de pile (**LOOKUP**) ou pour connaître sa taille, savoir si elle est vide

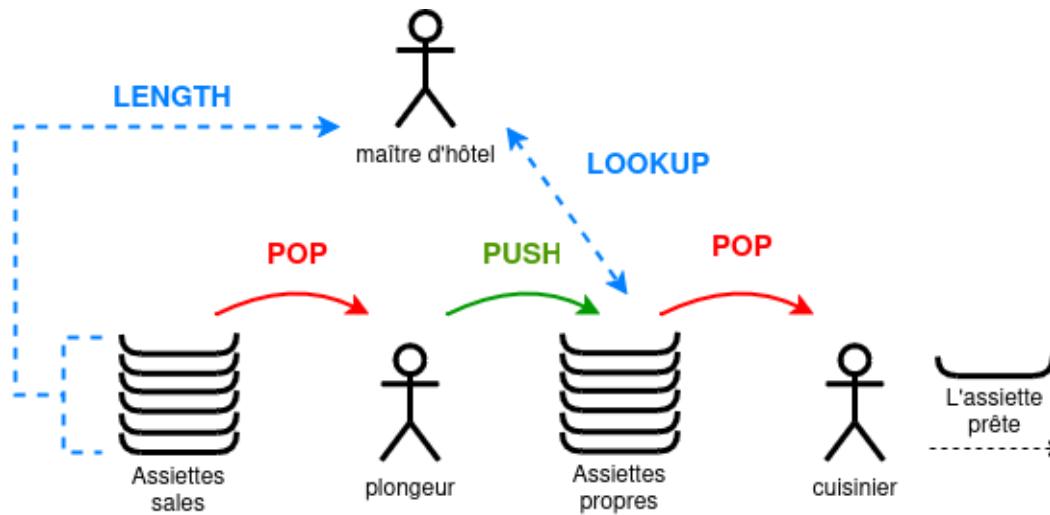


Pile d'exécution et Processus

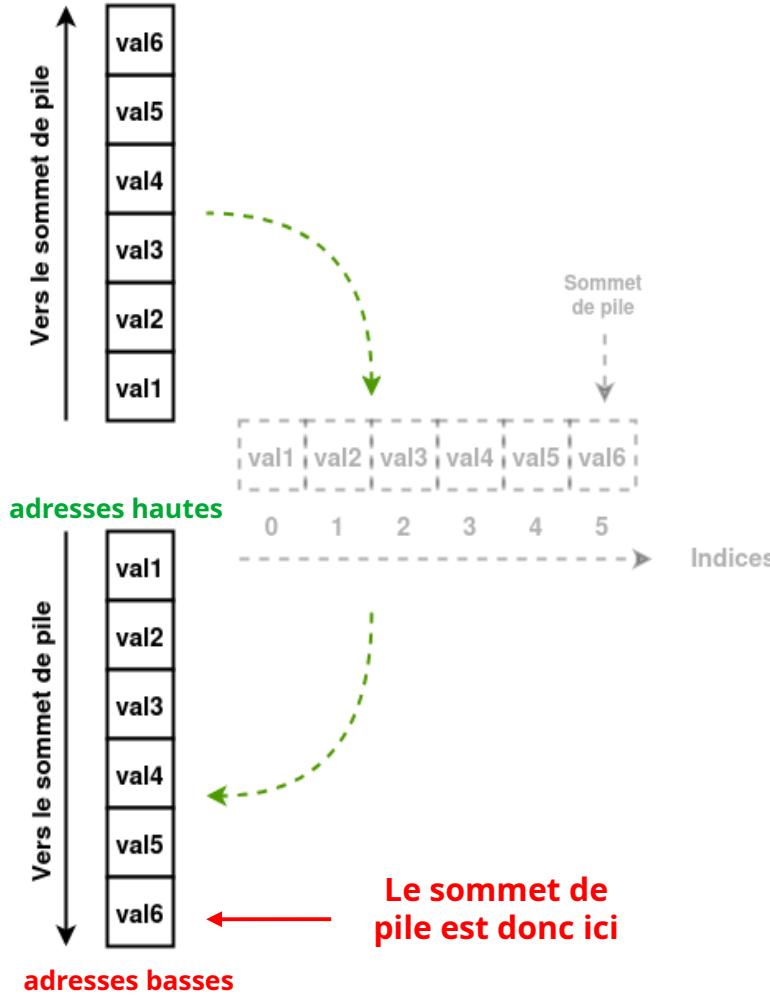


Exemple : Organisation du restaurant

- Le **plongeur** possède une pile d'assiettes sales, il les **POP** une par une, les lave, et les **PUSH** dans la pile d'assiettes propres
- Le **cuisinier** ne s'occupe que de la pile d'assiette propre, il **POP** l'assiette en haut de pile pour la préparer
- Le **maître d'hôtel** vérifie la propreté de chaque assiette sur le point d'être **POP** par le **cuisinier** par **LOOKUP**. Il peut aussi assigner un commis aux cuisines à la plonge en cas d'urgence (si le nombre d'assiettes sales devient trop élevé)



Pile d'exécution et Processus



Pour la suite, on doit considérer une partie de la mémoire comme **une pile**

- Mais la pile sera à l'envers !
- Le fond de pile sera dans les **adresses hautes**
- Le sommet de pile vers les **adresses basses**

Pile d'exécution et Processus



Un processus travaille sur une zone mémoire constituée de sections remarquables

La pile d'exécution (stack) :

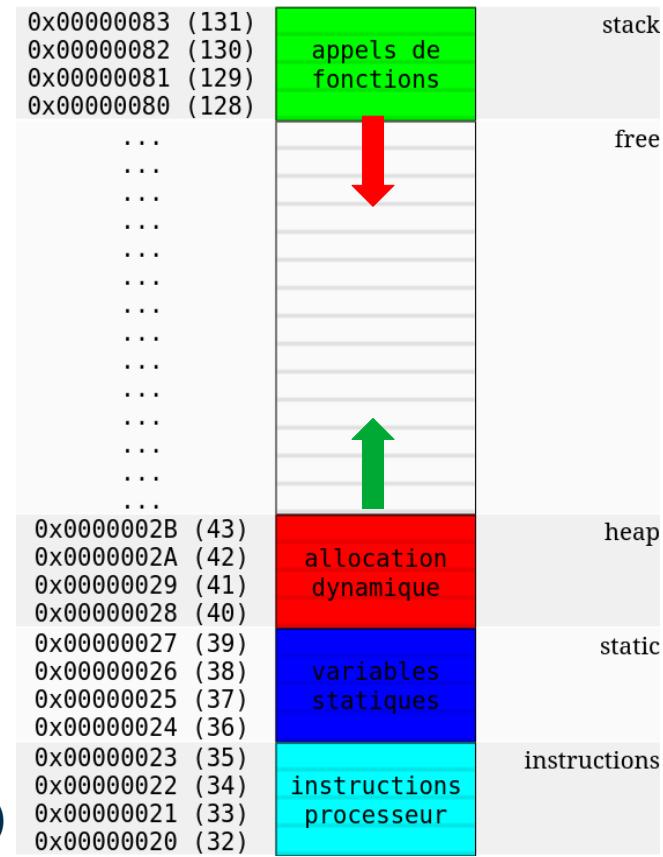
"Montée à l'envers", elle stocke les données liées aux **appels de fonctions** :

- Les paramètres et variables internes des fonctions
- Les valeurs de retour des fonctions
- **Grandit vers les adresses basses**

Le tas (heap) :

Il stocke les données créées **dynamiquement** :

- Les données partagées, volumineuses, ou à longue durée de vie
- Les données des structures dynamiques, c.a.d. créées dynamiquement ou à tailles dynamiques (e.g. les listes)
- **Grandit vers les adresses hautes**



Pile d'exécution et Processus

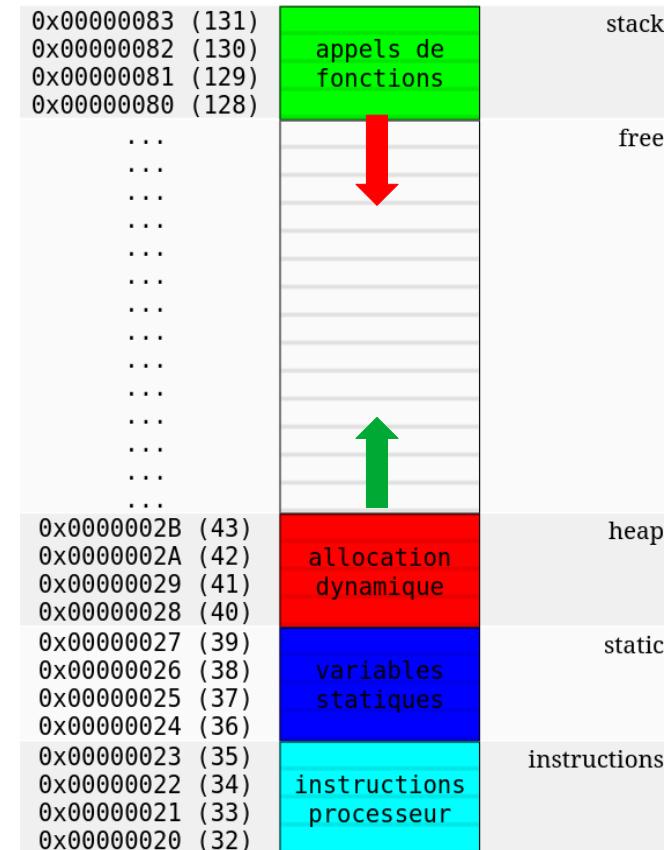


IMPORTANT : il ne s'agit pas d'un cours de système

- Les layouts présentés sur la partie droite des slides (celle-ci incluse) peuvent omettre des données ou des sections entières en fonction de l'**OS** (Operating System), de l'**ABI** (Application Binary Interface), du compilateur, etc.

→ **On se concentre sur la mémoire qui nous est accessible par adresse**

- Les adresses sont en hexadécimal pour vous habituer mais les valeurs ne sont pas représentatives de la réalité
- Le fonctionnement "système" n'est pas couvert : registres CPU impliqués, les instructions assembleurs, etc. Seul le fonctionnement "logique" nous intéresse.



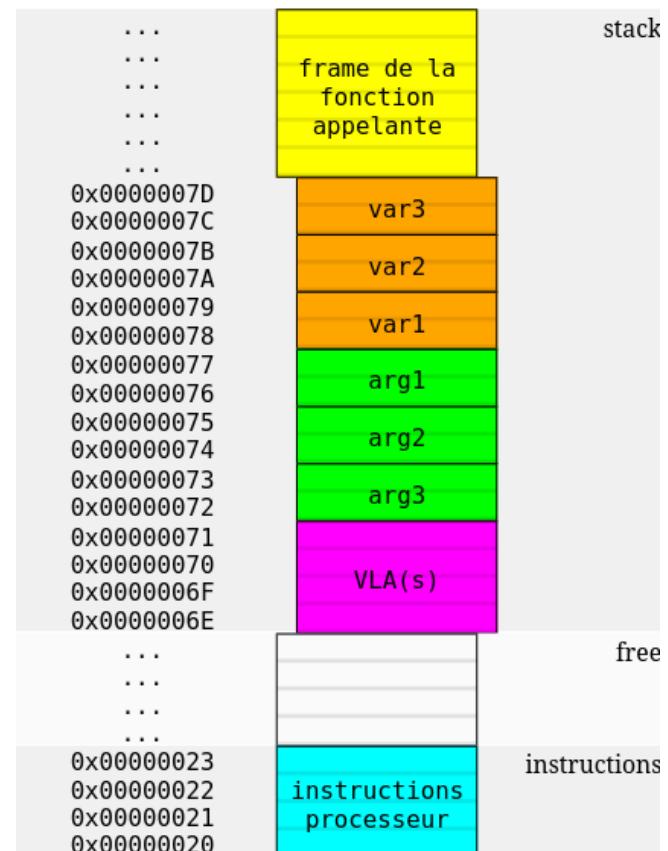
Pile d'exécution et Processus



IMPORTANT !

- Nous allons utiliser un certain ordre pour les valeurs poussées sur la stack, **cela ne reflète pas la réalité**
- Nous utiliserons un agencement proche de gcc mais pas identique
- Dans l'exemple de droite, on peut par exemple observer un réagencement si var2 est un tableau, il pourrait se retrouver au dessus de var 3
- Il ne s'agit de toute façon pas d'apprendre par cœur l'agencement d'un compilateur donné
- On ne représente pas de zone mémoire pour le retour des fonctions

TLDR : Les images d'agencement de la mémoire d'un processus ne seront **jamais** contractuelles





Blocs, Fonctions, Opérateurs, Conditions

Blocs, Fonctions, Opérateurs, Conditions



Une fonction est un **bloc** de code autonome

- Elle **peut** posséder des entrées. Exemple : la fonction "**printf**" possède au moins un argument, notre "**main**" n'en présentait pas.
- Elle **peut** posséder une valeur de sortie. Exemple : l'entier **0** en cas de succès pour la fonction "**main**".
- La valeur de retour peut être ignorée. Exemple : nous n'avons rien fait du retour de "**printf**" (votre réflexe pour savoir ce que rend printf ? → **man 3 printf**)
- Elle présente un **bloc** de code, contenant une ou plusieurs déclarations de retour

On peut aborder la notion de fonction sans connaître la notion de bloc

mais ce n'est pas une bonne idée

Le suspense est insoutenable... **Qu'est-ce qu'un bloc ?**

Blocs, Fonctions, Opérateurs, Conditions



Les blocs

- Permet de grouper des instructions
- Permet de limiter **le scope** (**la portée**) des variables
- ⚠ Seuls**, ils ne créent **pas** de nouvelles **frames** sur la pile
- On peut parler de durée de vie d'une variable (**lifetime** en Rust)

Aucune variable "b"
n'est dans le scope
ligne 10

The screenshot shows a terminal window with a C file named 'bloc.c'. The code demonstrates variable scope within a function:

```
C bloc.c 1 x
home > olivier > Documents > Lectures > C > examples > C bloc.c > ...
1 #include <stdlib.h>
2
3 int main(void) {
4     int a = 2;
5     {
6         int b = 3;
7         int c = a + b;
8     } // b and c leave the scope
9
10    int d = a + b; // Illegal
11    return EXIT_SUCCESS;
12} // a leaves the scope
```

The terminal output shows a compilation error for line 10:

```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL zsh - examples + v ⌂ ⌂ ... ^ ⌂
① (base) ➔ examples gcc bloc.c
bloc.c: In function 'main':
bloc.c:10:17: error: 'b' undeclared (first use in this function)
  10 |         int d = a + b; // Illegal
               ^
bloc.c:10:17: note: each undeclared identifier is reported only once for
each function it appears in
○ (base) ➔ examples ⌂
```

A red arrow points from the text "Aucune variable 'b'" to the error message in the terminal output.

Blocs, Fonctions, Opérateurs, Conditions



Déclaration d'une fonction :

1) Type de retour

2) Nom

3) Arguments : une liste entre parenthèses de variables typées séparées par des virgules

4) Un bloc, les arguments de la fonction sont dans le même scope que les variables déclarées dans ce bloc

5) Retour : si besoin, une ou plusieurs déclarations de retour avec le mot-clef "return"

Le type de retour sera "**void**" pour une fonction ne renvoyant rien ("**return;**" est toujours possible)

Une fonction ne prenant pas d'argument présentera seulement "**(void)**" ou "**()**" en tant que liste d'arguments

A screenshot of a code editor showing a C program named main.c. The code includes #include directives for stdio.h and stdlib.h, a main function that checks the argument count, prints usage information if there are fewer than two arguments, and returns EXIT_FAILURE or EXIT_SUCCESS based on the number of arguments. The code editor interface shows tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab displays the output of running the compiled program with one argument and then with two arguments, showing the usage message and the successful execution message.

```
C main.c  x
C main.c > ↗ main(int, char * [])
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // argc: arg count
5 // argv: arg values
6 int main(int argc, char* argv[]) {
7     if (argc != 2) {
8         printf("Usage: %s <argument>\n", argv[0]);
9         return EXIT_FAILURE;
10    }
11    printf("You typed: %s %s\n", argv[0], argv[1]);
12    return EXIT_SUCCESS;
13 }
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
• (base) ➔ examples gcc main.c -o main			
• (base) ➔ examples ./main			
			Usage: ./main <argument>
• (base) ➔ examples ./main hello!			
			You typed: ./main hello!
• (base) ➔ examples			

Main prend en réalité deux arguments, mais la déclaration "**int main (void)**" est toujours viable.

"**void main (void)**" est à éviter, l'OS attendra un retour (en plus des standards C)

Blocs, Fonctions, Opérateurs, Conditions



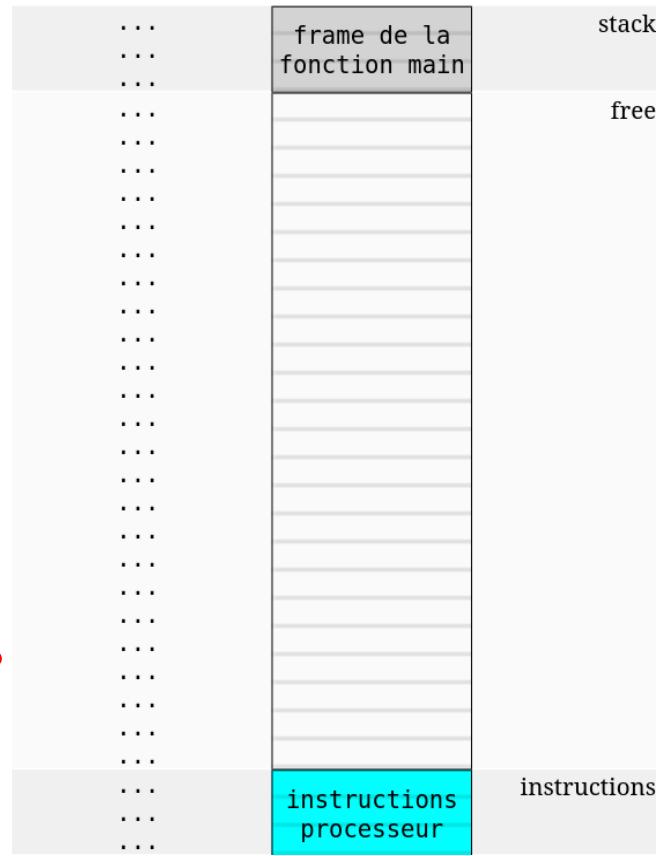
```
c frames.c > ...
1 #include <stdio.h>
2
3 void function_2(){
4     printf("function 2\n");
5 }
6
7 void function_1(){
8     printf("function 1\n");
9     function_2();
10}
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

On vient de compiler le programme de gauche (avec **gcc**) en un binaire, et on l'exécute (**a.out**) !

Les instructions sont binaires (dur de visualiser)

on suit donc l'exécution en regardant le fichier **.c**

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
```



Blocs, Fonctions, Opérateurs, Conditions

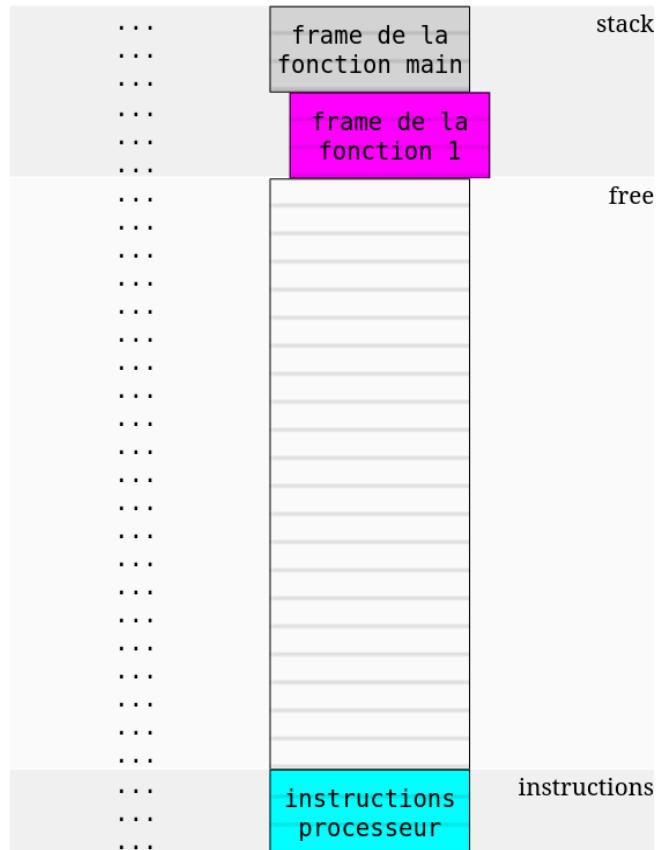


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Appel à **fonction 1** depuis la fonction main

Création de frame et on empile !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
```



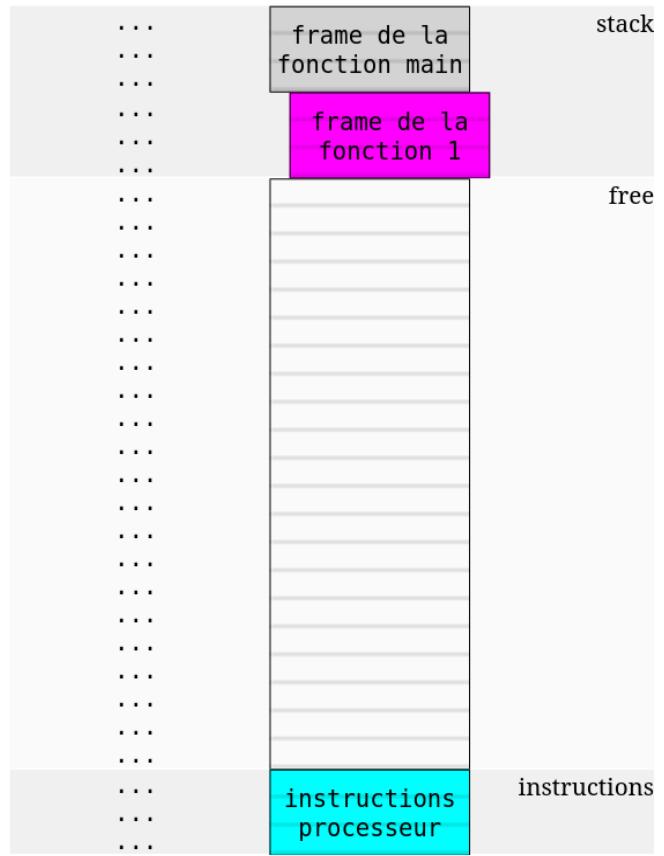
Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Affichage !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
```



Blocs, Fonctions, Opérateurs, Conditions

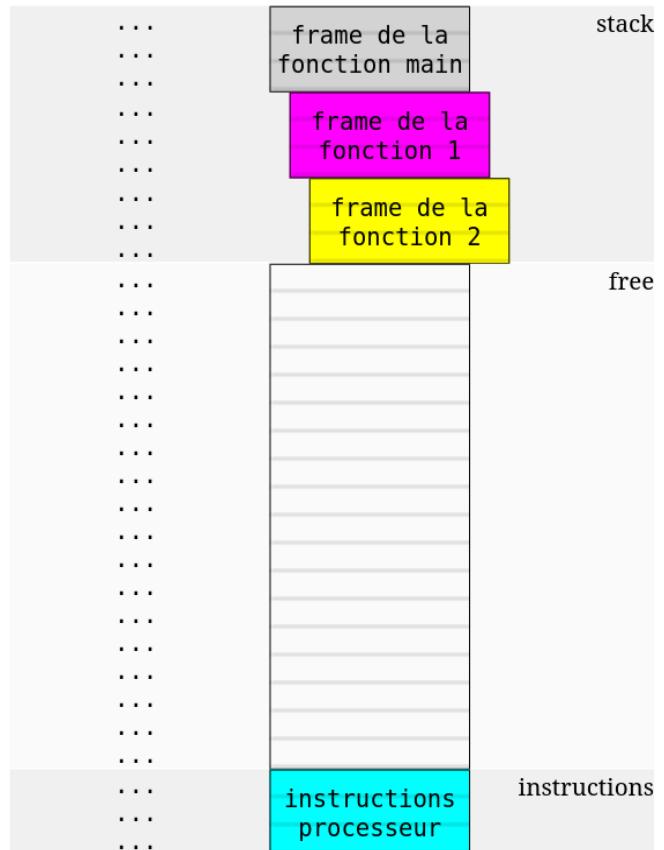


```
C frames.c > ...
1  #include <stdio.h>
2  void function_2(){
3      printf("function 2\n");
4  }
5
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Appel à **fonction 2** depuis la fonction **fonction 1**

Création de frame et on empile !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
```



Blocs, Fonctions, Opérateurs, Conditions

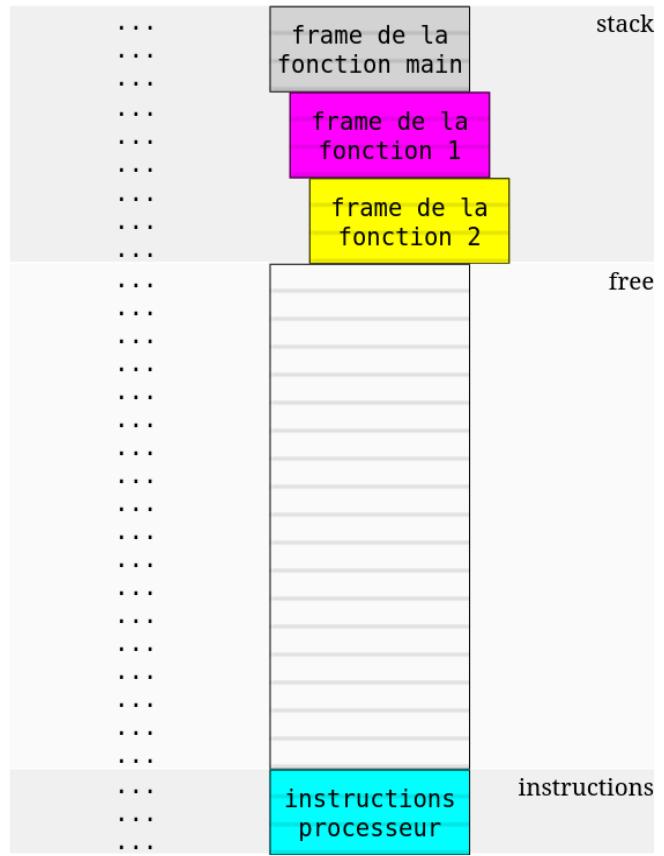


C frames.c > ...

```
1 #include <stdio.h>
2
3 void function_2(){
4     printf("function 2\n");
5 }
6
7 void function_1(){
8     printf("function 1\n");
9     function_2();
10}
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Affichage !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
```



Blocs, Fonctions, Opérateurs, Conditions

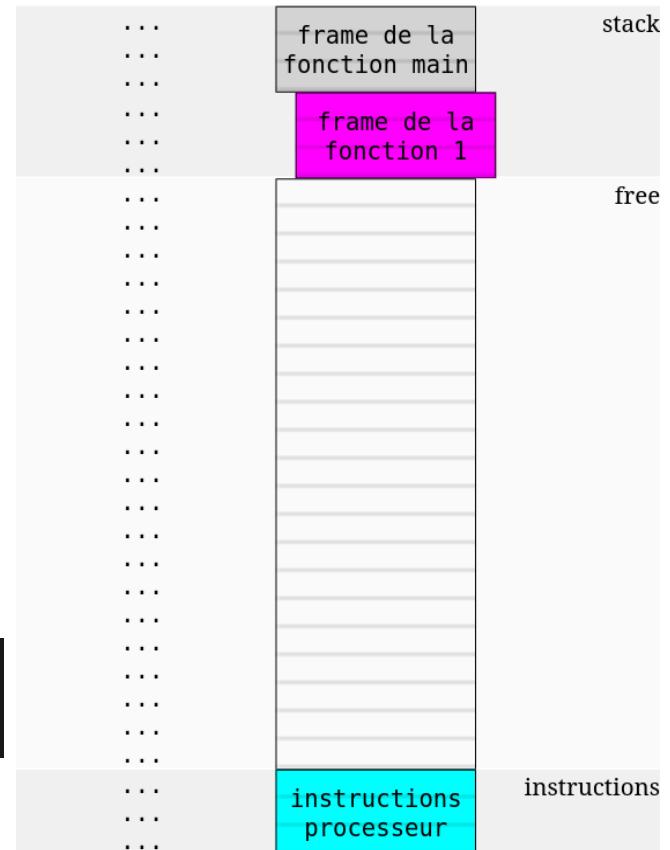


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

On sort de la **fonction 2**

On dépile la frame pour
revenir à la frame
précédente (**fonction 1**)

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
```



Blocs, Fonctions, Opérateurs, Conditions

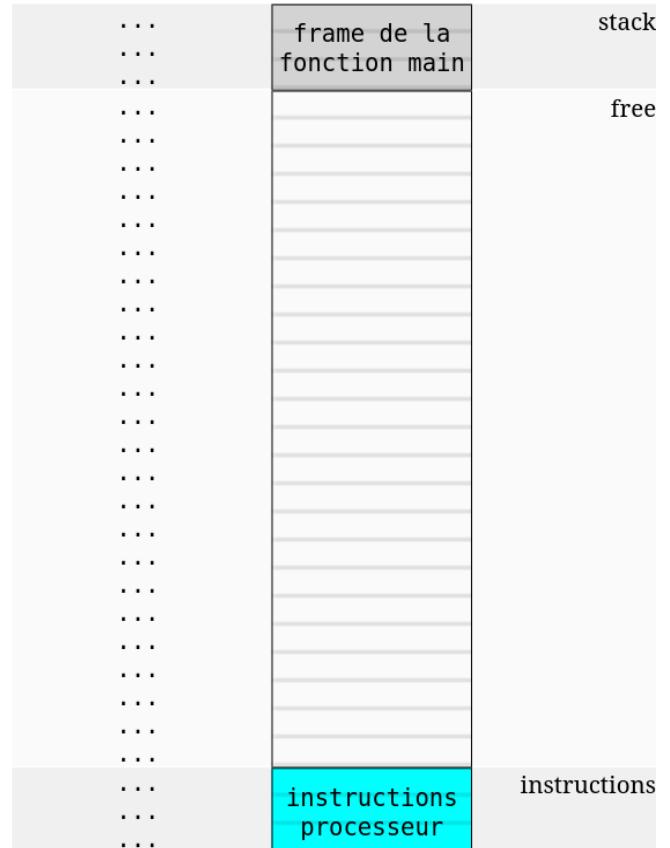


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

On sort de la **fonction 1**

On dépile la frame pour
revenir à la frame
précédente (**main**)

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
```



Blocs, Fonctions, Opérateurs, Conditions

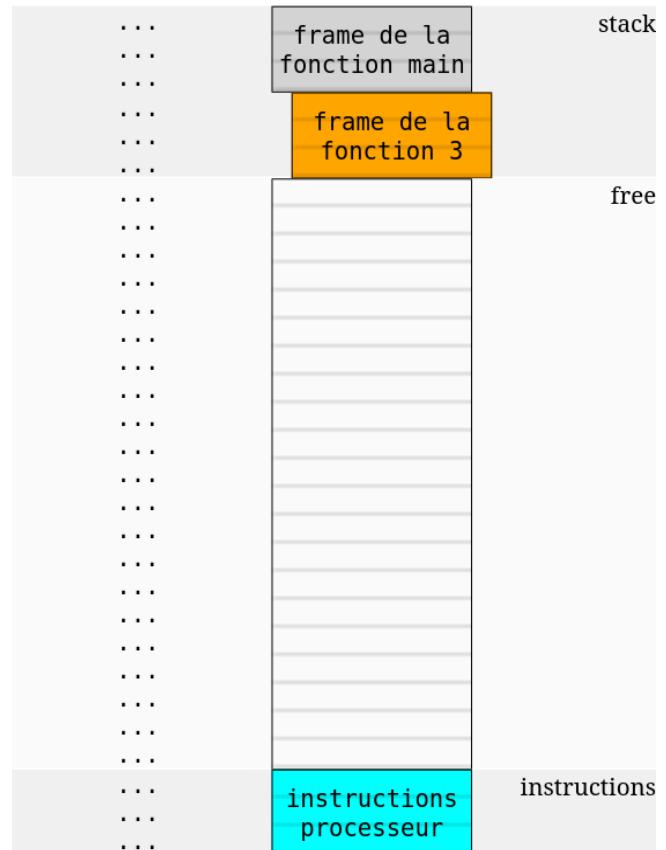


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Appel à **fonction 3** depuis la fonction **main**

Création de frame et on empile !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
```



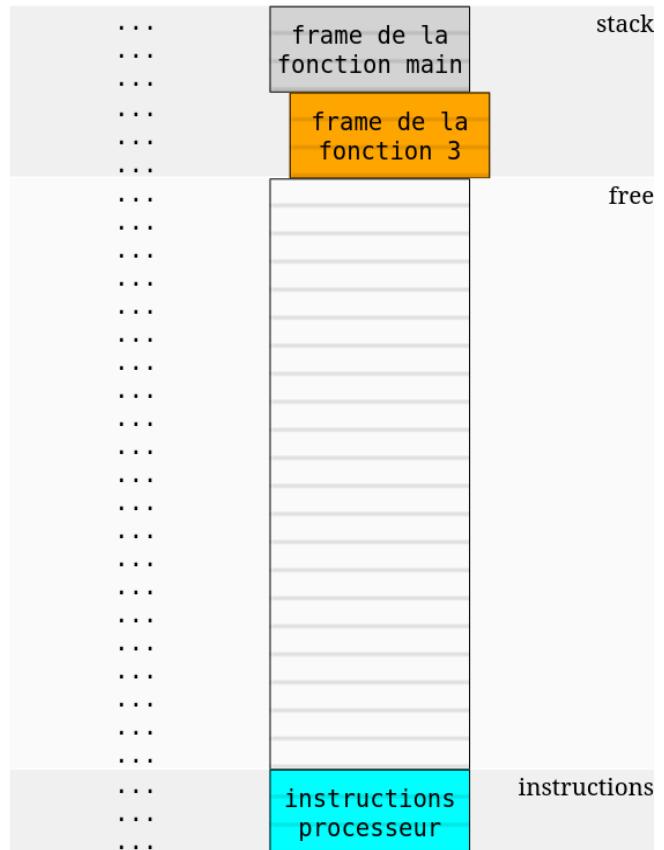
Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 } →
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Affichage !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
```



Blocs, Fonctions, Opérateurs, Conditions

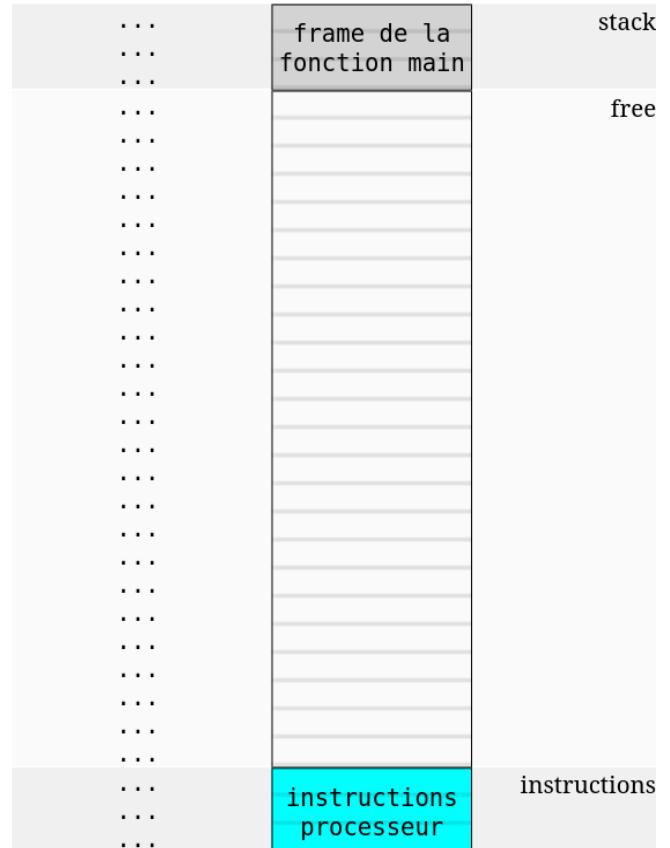


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

On sort de la **fonction 3**

On dépile la frame pour
revenir à la frame
précédente (**main**)

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
```



Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

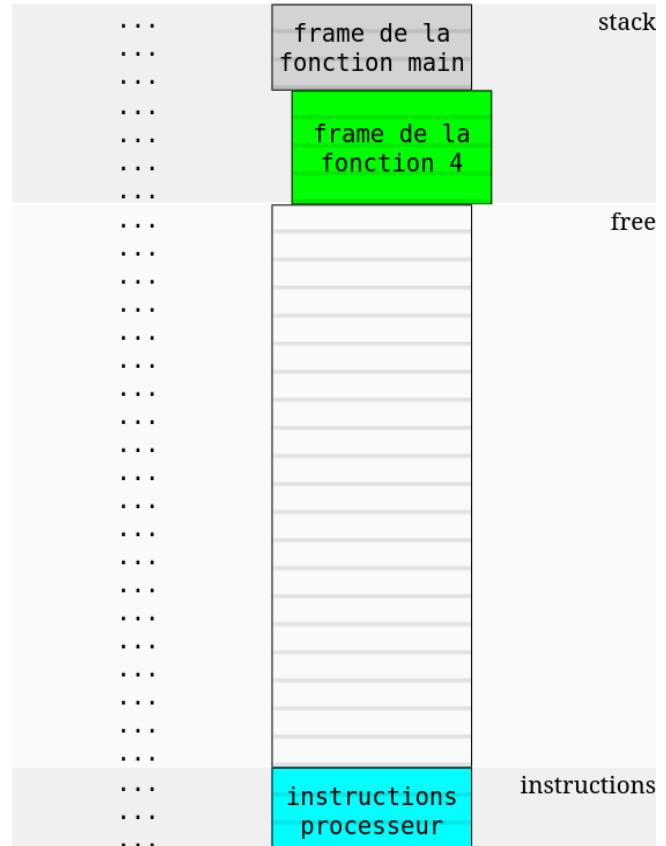
Appel à **fonction 4** depuis la fonction **main**

On passe 3 à la fonction

Création de frame et on empile !

Les frames pour la fonction 4 sont grandes car elles ont l'argument "val"

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
```



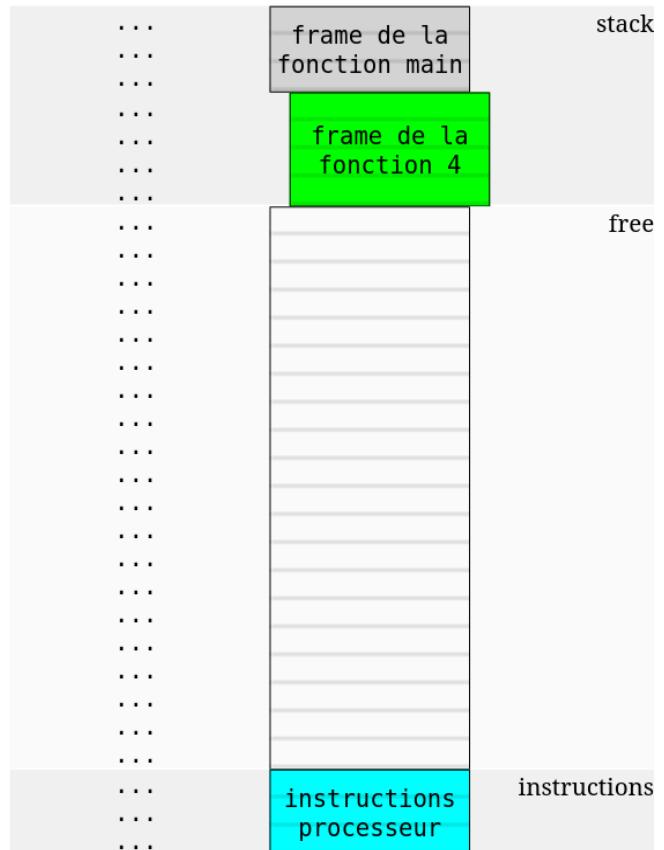
Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Affichage !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
```



Blocs, Fonctions, Opérateurs, Conditions

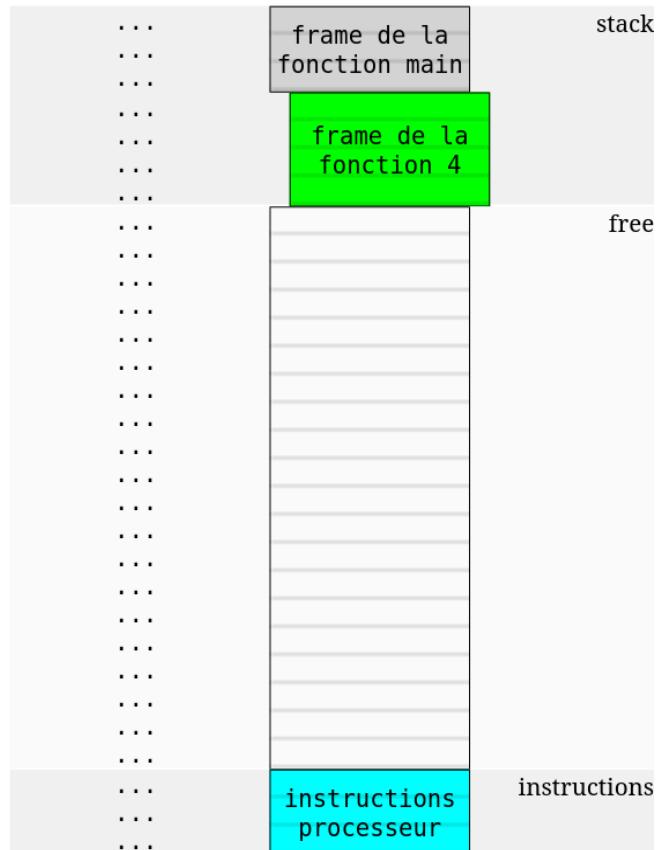


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

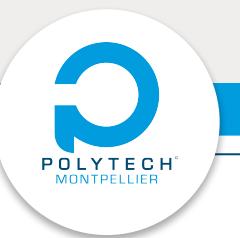
Affichage !

val vaut 3 dans cette
frame : le test réussi

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
```



Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

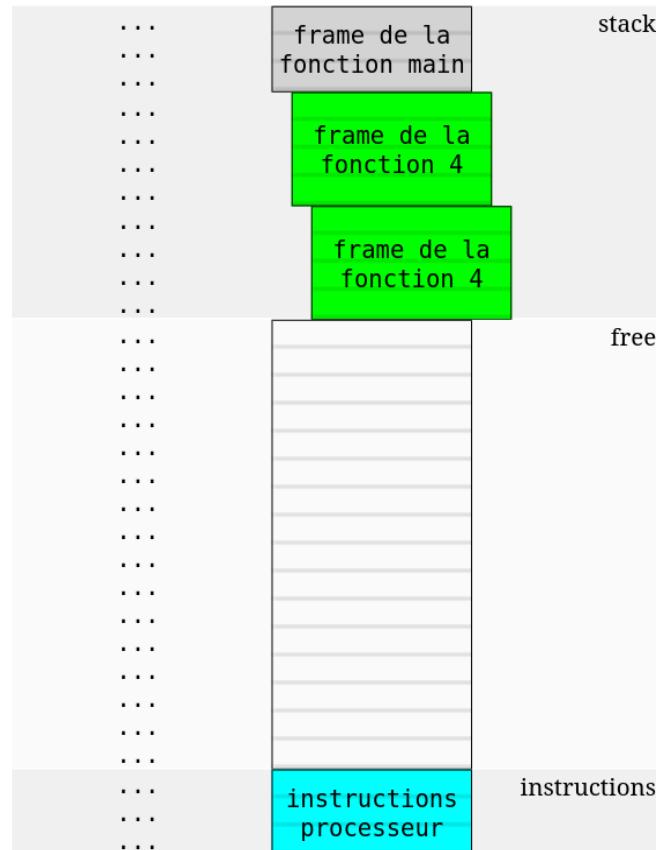
Appel à **fonction 4** depuis la fonction **fonction 4**

La fonction 4 est récursive !

On passe 2 à la fonction

Création de frame et on empile !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
```



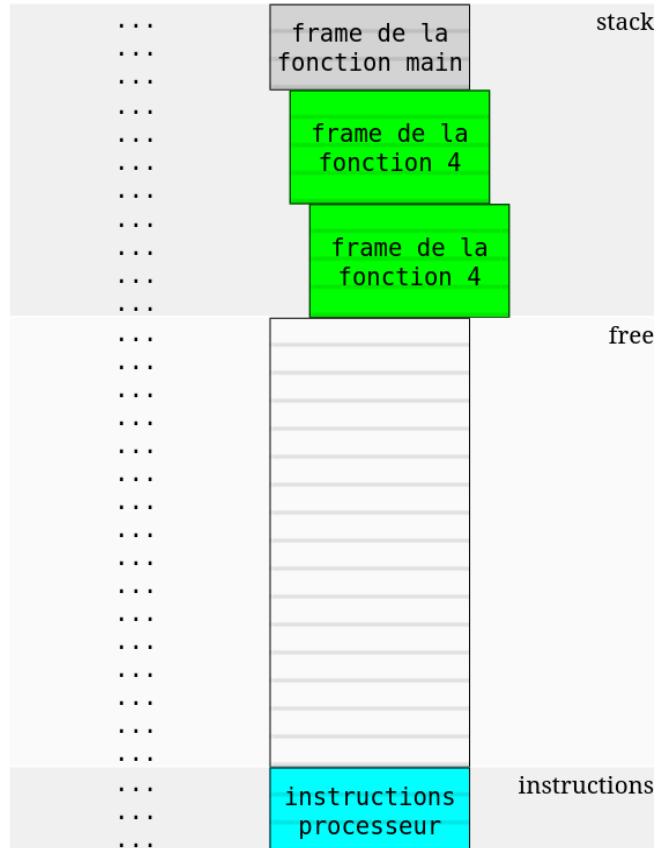
Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Affichage !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
```



Blocs, Fonctions, Opérateurs, Conditions

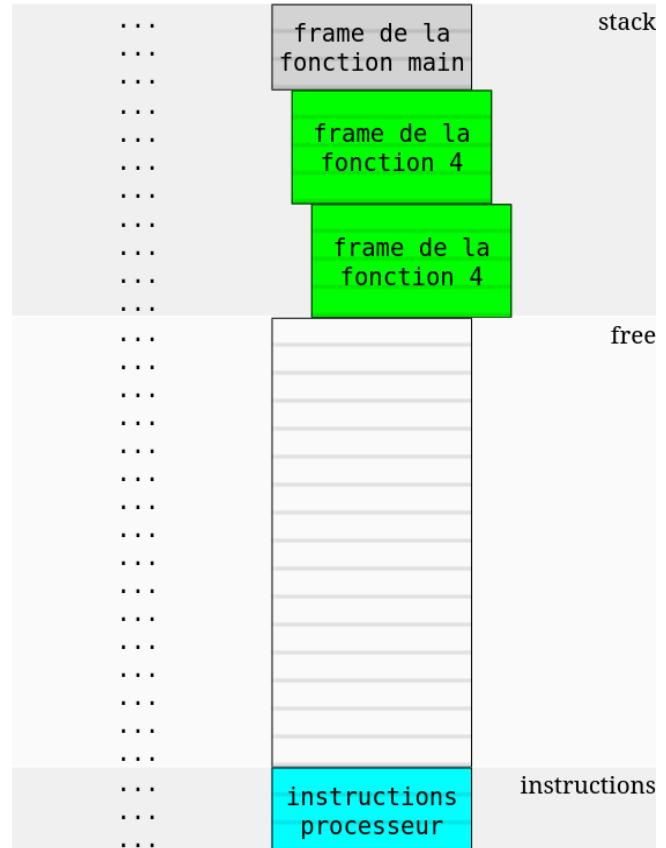


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Affichage !

val vaut 2 dans cette
frame : le test réussi

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
```



Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

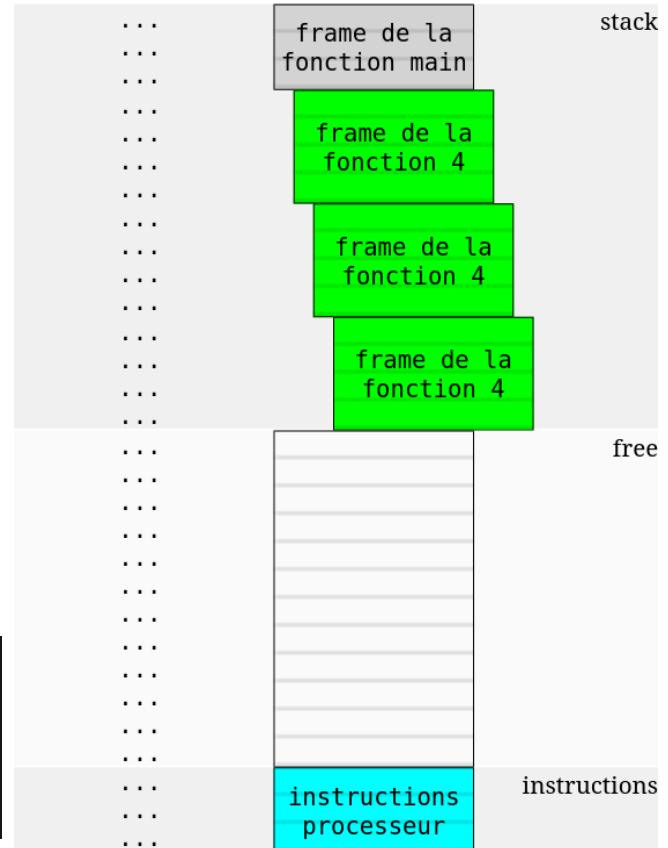
Appel à **fonction 4** depuis la fonction **fonction 4**

On continue la récursion !

On passe 1 à la fonction

Création de frame et on empile !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
```



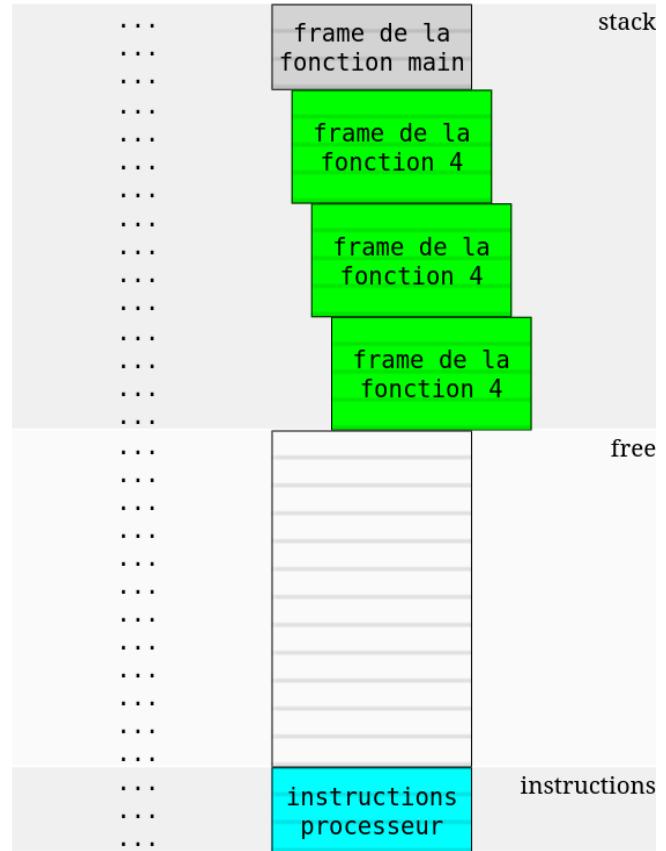
Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Affichage !

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
function 4
```



Blocs, Fonctions, Opérateurs, Conditions

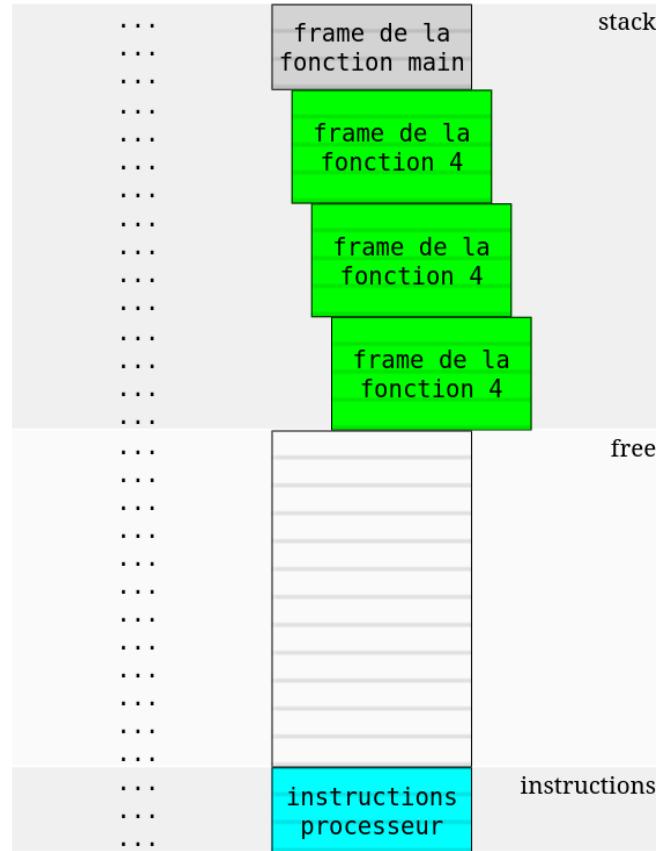


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Affichage !

val vaut 1 dans cette
frame : le test échoue

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
function 4
```



Blocs, Fonctions, Opérateurs, Conditions

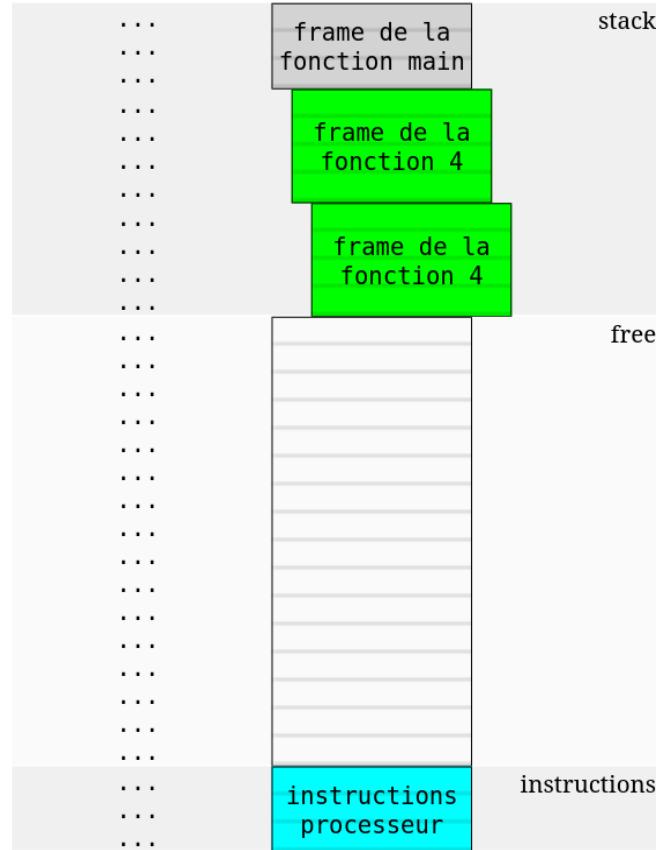


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

On sort de la **fonction 4**

On dépile la frame pour
revenir à la frame
précédente (**fonction 4**)

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
function 4
```



Blocs, Fonctions, Opérateurs, Conditions

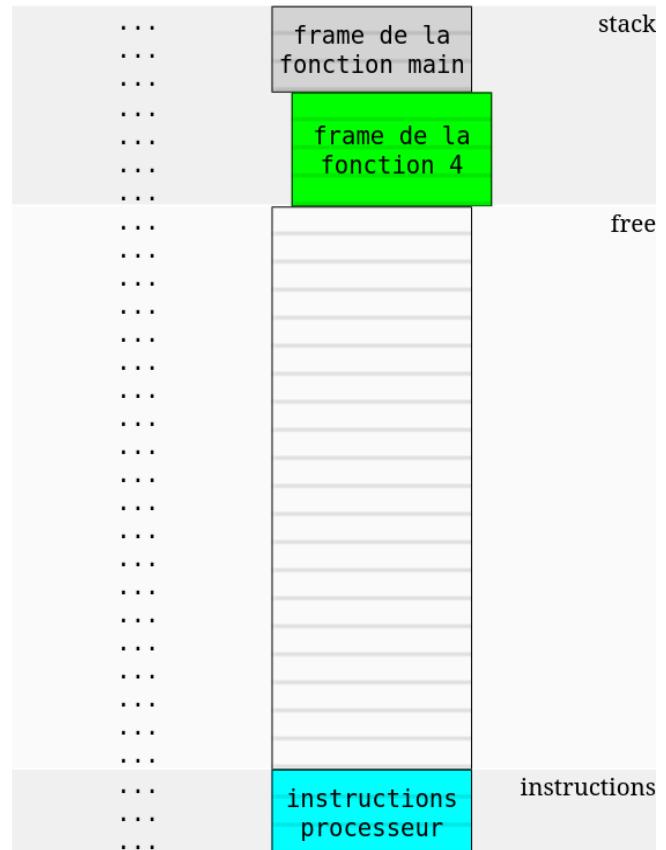


```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

On sort de la **fonction 4**

On dépile la frame pour
revenir à la frame
précédente (**main**)

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
function 4
```



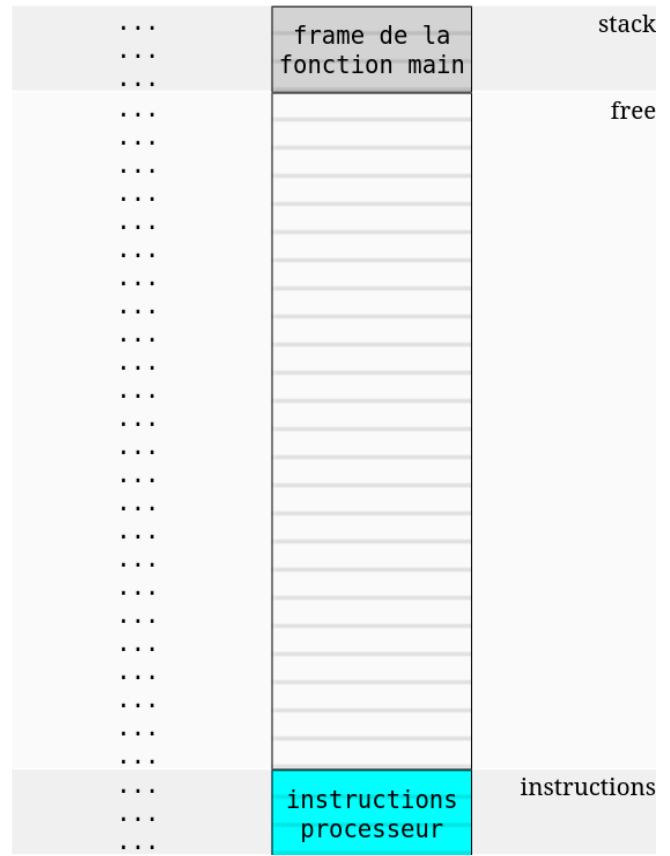
Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

On sort de main

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
function 4
```



Blocs, Fonctions, Opérateurs, Conditions



```
C frames.c > ...
1  #include <stdio.h>
2
3  void function_2(){
4      printf("function 2\n");
5  }
6
7  void function_1(){
8      printf("function 1\n");
9      function_2();
10 }
11
12 void function_3(){
13     printf("function 3\n");
14 }
15
16 void function_4(int val){
17     printf("function 4\n");
18     if(val > 1) {
19         function_4(val - 1);
20     }
21 }
22
23 int main(void){
24     function_1();
25     function_3();
26     function_4(3);
27     return 0;
28 }
```

Le processus se termine, la mémoire du processus est libérée, on récupère la main sur la ligne de commande

```
(base) ✘ frames git:(main) ✘ gcc frames.c
(base) ✘ frames git:(main) ✘ ./a.out
function 1
function 2
function 3
function 4
function 4
function 4
(base) ✘ frames git:(main) ✘
```

Blocs, Fonctions, Opérateurs, Conditions

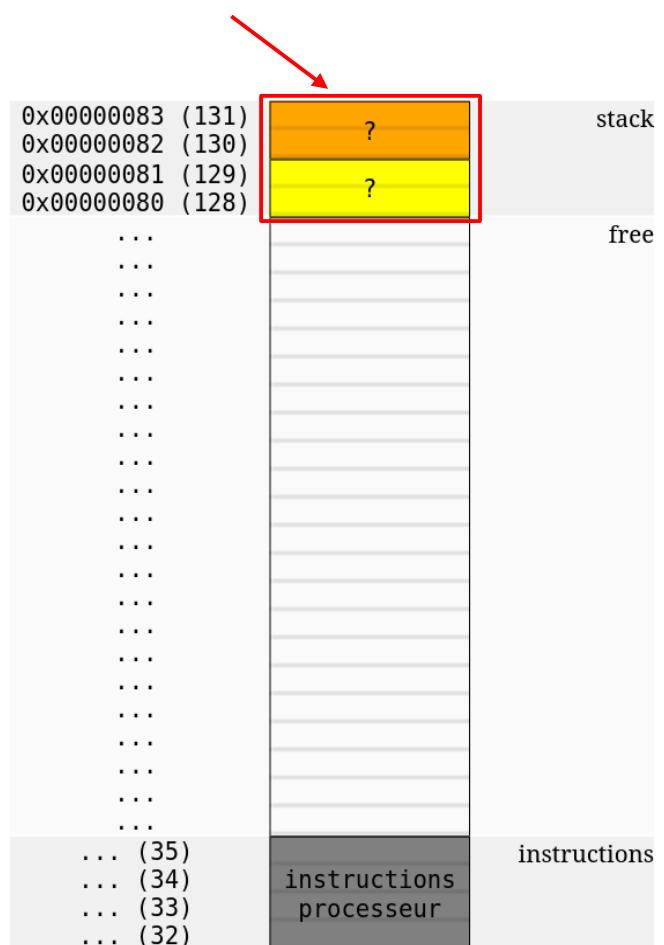


Exécution des fonctions

Un appel de fonction crée une **frame**, ici notre frame pour **main** présente deux variables

```
1 #include <stdlib.h>
2
3 int add_with_ten(short cyan, short green) {
4     short red = 10 + cyan + green;
5     return red;
6 }
7
8 int main(void) {
9     short yellow = 2;
10    short orange = 3;
11    orange = add_with_ten(yellow, orange);
12    return EXIT_SUCCESS;
13 }
```

La frame pour **main**



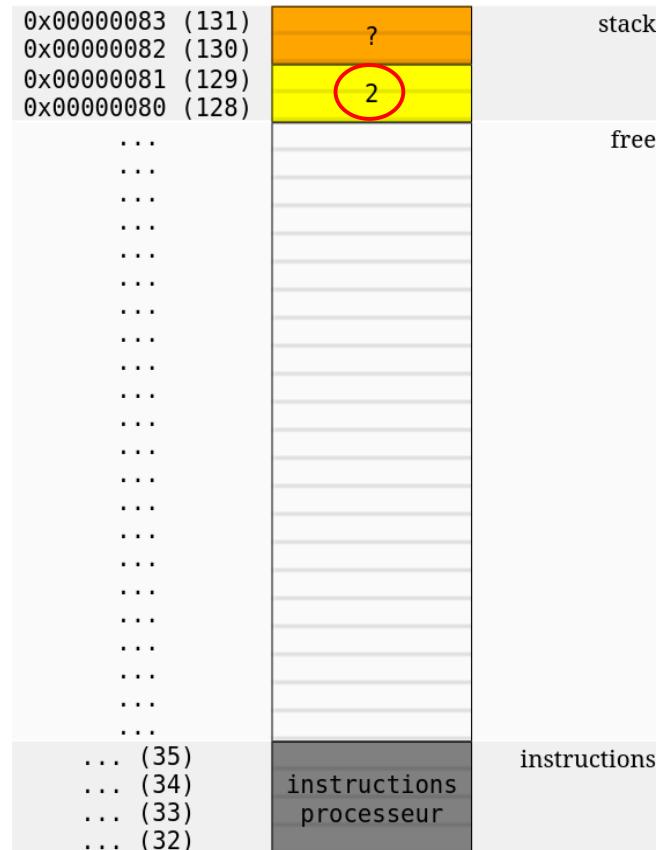
Blocs, Fonctions, Opérateurs, Conditions



Exécution des fonctions

Initialisation, on met à jour la zone mémoire associée au label

```
1 #include <stdlib.h>
2
3 int add_with_ten(short cyan, short green) {
4     short red = 10 + cyan + green;
5     return red;
6 }
7
8 int main(void) {
9     short yellow = 2;
10    short orange = 3;
11    orange = add_with_ten(yellow, orange);
12    return EXIT_SUCCESS;
13 }
```



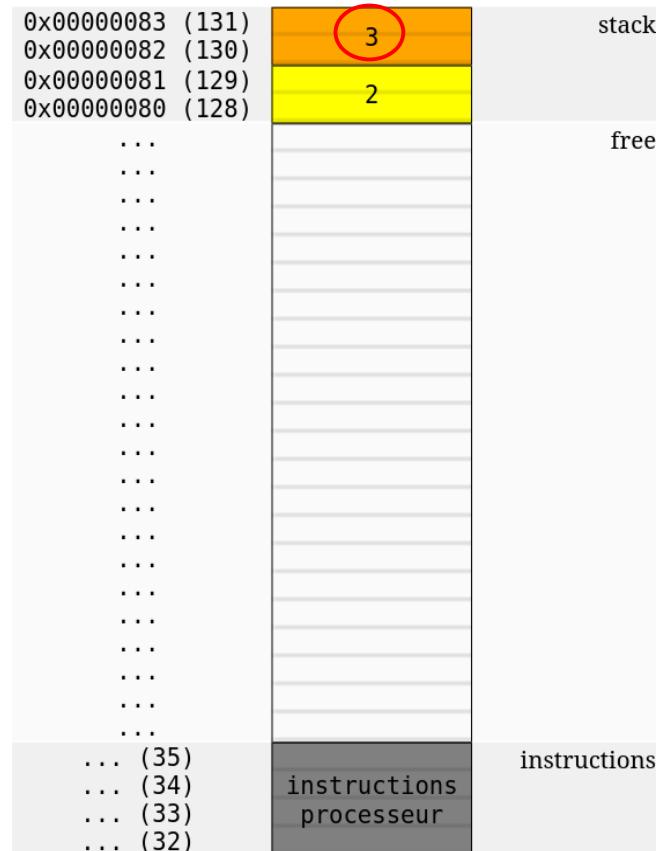
Blocs, Fonctions, Opérateurs, Conditions



Exécution des fonctions

Nouvelle **initialisation**, on met à jour la zone mémoire associée au label

```
1 #include <stdlib.h>
2
3 int add_with_ten(short cyan, short green) {
4     short red = 10 + cyan + green;
5     return red;
6 }
7
8 int main(void) {
9     short yellow = 2;
10    short orange = 3;
11    orange = add_with_ten(yellow, orange);
12    return EXIT_SUCCESS;
13 }
```



Blocs, Fonctions, Opérateurs, Conditions

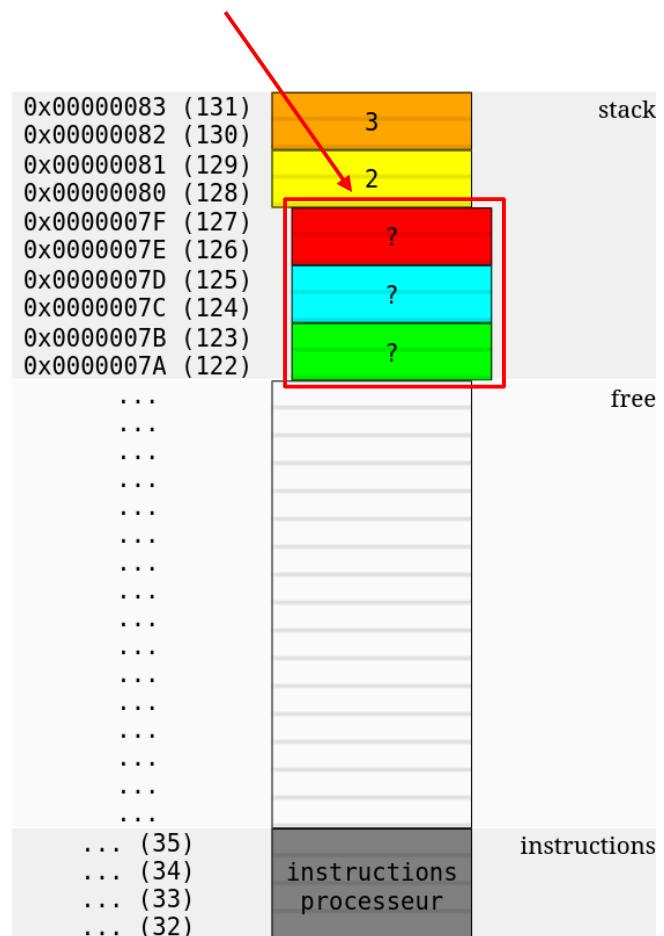


Exécution des fonctions

Appel de fonction, on crée une **nouvelle frame**

```
1 #include <stdlib.h>
2
3 int add_with_ten(short cyan, short green) {
4     short red = 10 + cyan + green;
5     return red;
6 }
7
8 int main(void) {
9     short yellow = 2;
10    short orange = 3;
11    orange = add_with_ten(yellow, orange);
12    return EXIT_SUCCESS;
13 }
```

La frame pour **add_with_ten**, l'indentation traduit la profondeur d'appel



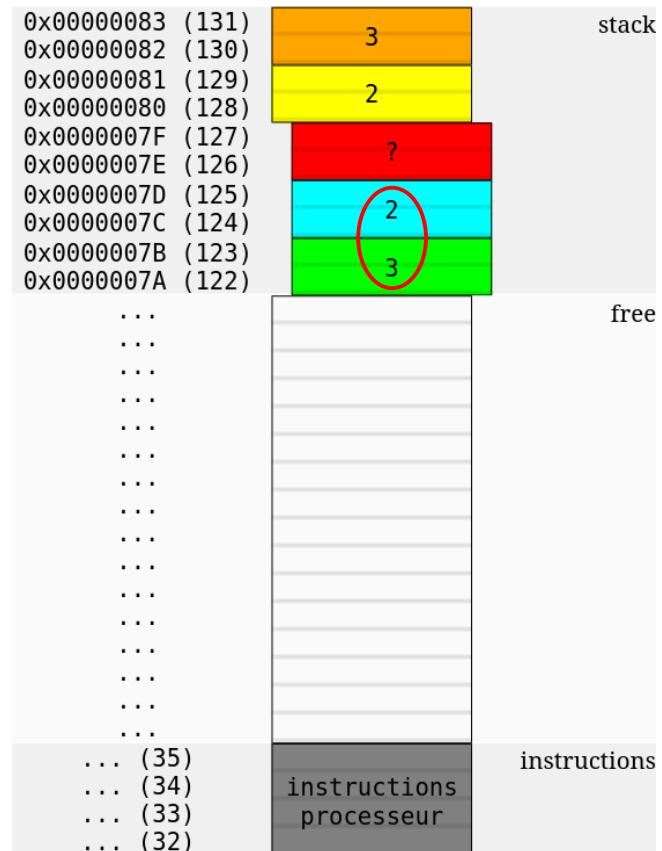
Blocs, Fonctions, Opérateurs, Conditions



Exécution des fonctions

Les arguments sont variables de bloc, on met à jour les zones mémoire associées

```
1 #include <stdlib.h>
2
3 int add_with_ten(short cyan, short green) {
4     short red = 10 + cyan + green;
5     return red;
6 }
7
8 int main(void) {
9     short yellow = 2;
10    short orange = 3;
11    orange = add_with_ten(yellow, orange);
12    return EXIT_SUCCESS;
13 }
```



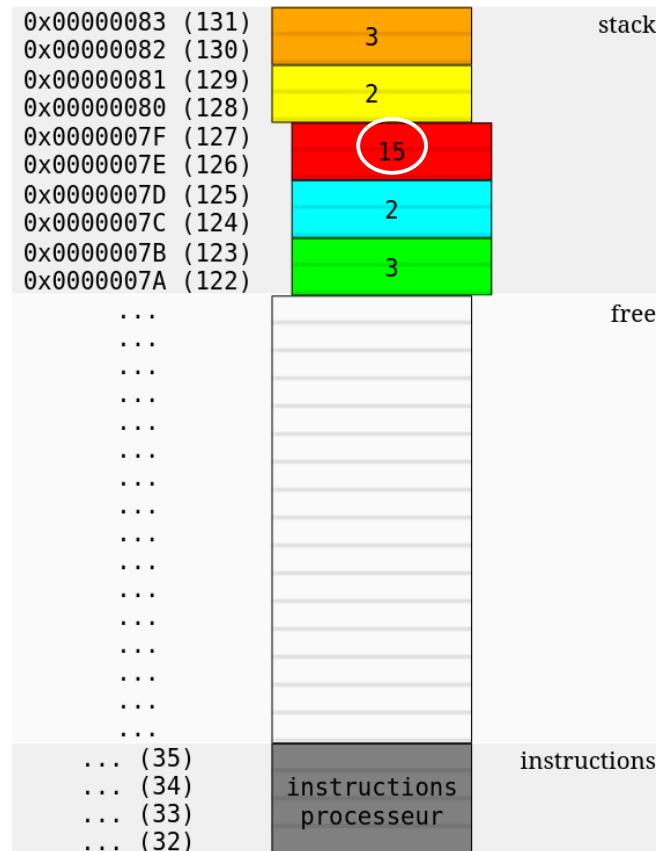
Blocs, Fonctions, Opérateurs, Conditions



Exécution des fonctions

Initialisation de la variable interne, arguments et variables internes sont considérés pareillement

```
1 #include <stdlib.h>
2
3 int add_with_ten(short cyan, short green) {
4     short red = 10 + cyan + green;
5     return red;
6 }
7
8 int main(void) {
9     short yellow = 2;
10    short orange = 3;
11    orange = add_with_ten(yellow, orange);
12    return EXIT_SUCCESS;
13 }
```



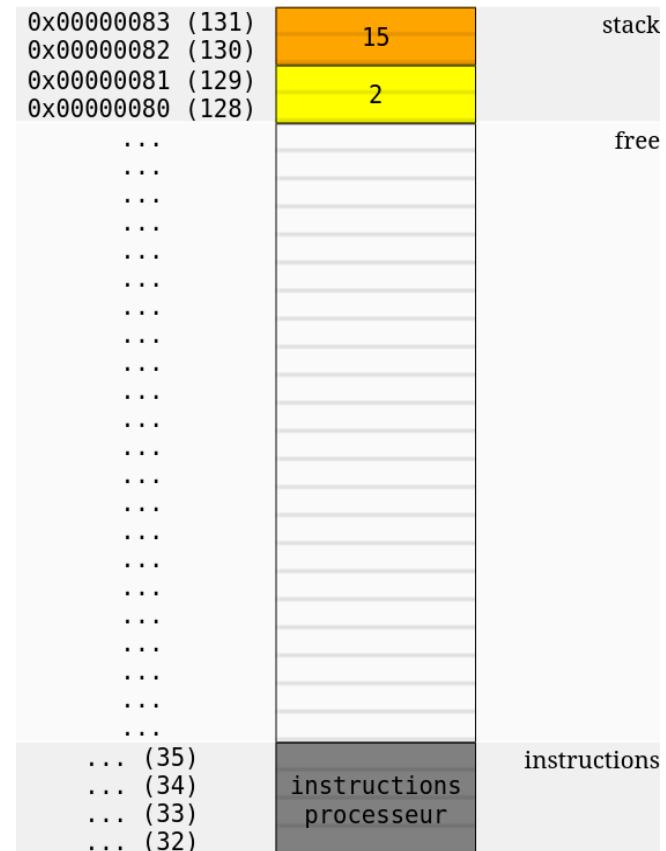
Blocs, Fonctions, Opérateurs, Conditions



Exécution des fonctions

La valeur de **retour** est stockée "quelque part*" au retour de fonction, puis copiée dans orange : **affectation**

```
1 #include <stdlib.h>
2
3 int add_with_ten(short cyan, short green) {
4     short red = 10 + cyan + green;
5     return red;
6 }
7
8 int main(void) {
9     short yellow = 2;
10    short orange = 3;
11    orange = add_with_ten(yellow, orange);
12    return EXIT_SUCCESS;
13 }
```



* L'**ABI** (Application Binary Interface) est au programme du cours de système, nous ne voyons ici qu'une version simplifiée d'un layout de processus

Blocs, Fonctions, Opérateurs, Conditions

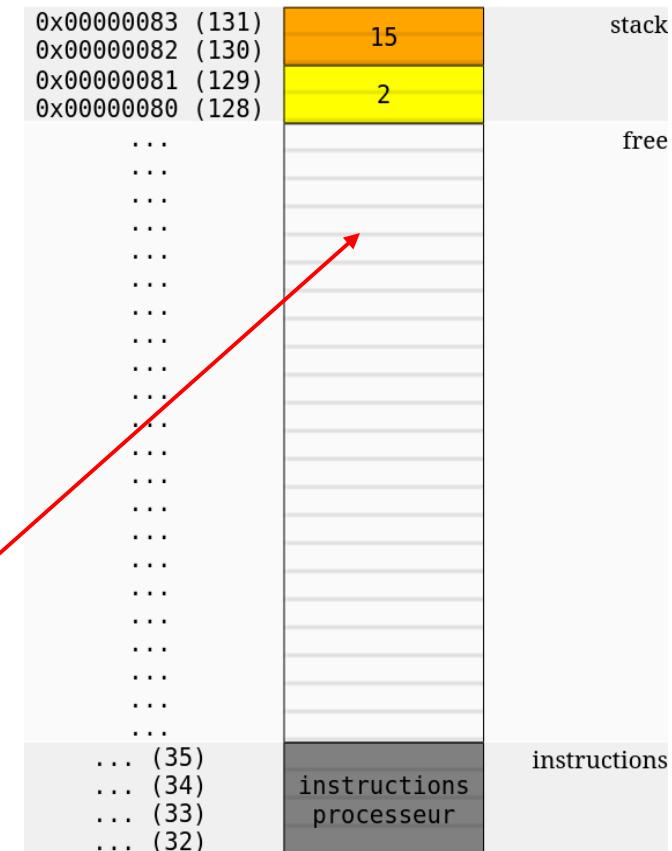


Concernant les frames

Les frames ne sont pas réellement "dépilées", c'est juste le "pointeur" de sommet de pile qui est modifié

→ Les valeurs d'une frame tout juste "dépilée" sont **toujours en mémoire**

Les valeurs **2**, **3** et **15** (sur 2 octets) sont toujours présentes aux adresses **126**, **124** et **122**, maintenant dans la zone "**free**" après l'appel à la fonction **add_with_ten**



Blocs, Fonctions, Opérateurs, Conditions



Opérateurs rencontrés

Nous avons déjà vu, sans les nommer, quelques opérateurs de base :

- L'opérateur d'affection / assignation =
- L'opérateur + pour l'addition
- L'opérateur unaire - pour les valeurs négatives (on peut évidemment soustraire avec)
- L'opérateur **sizeof** pour connaître la taille en mémoire d'un type (et d'autres choses)

→ Ils appartiennent à la famille des opérateurs d'assignation et à celle des opérateurs arithmétiques

Attention : l'opérateur égal a un sens différent pour
l'**initialisation** (int a = 2) et l'**affectation** (a = 3)

Une assignation est une simple copie de la mémoire de y pour écraser celle de x

Opérateurs d'affectation

Opérateur	Traduction	Exemple	Résultat
=	affectation simple	x = y	assigne la valeur de y à x
(op)=	affectation composée	x (op)= y	x (op)=y est équivalent à x = x (op) y

Blocs, Fonctions, Opérateurs, Conditions



Opérateurs arithmétiques

Opérateur	Traduction	Exemple	Résultat
+	Addition	$x + y$	l'addition de x et y
-	Soustraction	$x - y$	la soustraction de x et y
*	Produit	$x * y$	la multiplication de x et y
/	Division	x / y	le quotient de x et y
%	Reste	$x \% y$	Reste de la division euclidienne de x par y
+ (unaire)	Signe positif	$+x$	la valeur de x
- (unaire)	Signe négatif	$-x$	la négation arithmétique de x
++ (unaire)	Incrément	$x++$ ou $++x$	x est incrementé ($x = x + 1$). L'opérateur préfixe $++x$ (resp. suffixe $x++$) incrémente x avant (resp. après) de l'évaluer
-- (unaire)	Decrémentation	$x--$ ou $--x$	x est décremémenté ($x = x - 1$). L'opérateur préfixe $--x$ (resp. suffixe $x--$) décrémente x avant (resp. après) de l'évaluer

Remarques :

- % ne fonctionne que si les opérandes sont entiers (sinon erreur de compilation)
- ++ et -- fonctionnent sur les flottants
- / est une division euclidienne si et seulement si les deux opérandes sont entières

Blocs, Fonctions, Opérateurs, Conditions



Pour traiter les conditions il nous manque 2 familles supplémentaires

Opérateurs relationnels

Opérateur	Traduction	Exemple	Résultat (0 sinon)
<	inférieur	$x < y$	1 si x est inférieur à y
\leq	inférieur ou égal	$x \leq y$	1 si x est inférieur ou égal à y
>	supérieur	$x > y$	1 si x est supérieur à y
\geq	supérieur ou égal	$x \geq y$	1 si x est supérieur ou égal à y
\equiv	égalité	$x \equiv y$	1 si x est égal à y
\neq	in inégalité	$x \neq y$	1 si x est différent de y

Pour les opérateurs logiques :

Le type booléen n'existe pas, seule la valeur 0 sera testée pour "false", une autre valeur serait "true"

Opérateurs logiques

Opérateur	Traduction	Exemple	Résultat
$\&\&$	ET logique	$x \&\& y$	1 si x et y sont différents de 0
$\ $	OU logique	$x \ y$	1 si x et/ou y sont différents de 0
!	NON logique	$!x$	1 si x est égal à 0. Dans tous les autres cas, 0 est renvoyé.

Par exemple,
-1 $\&\&$ 2 rendra 1

Blocs, Fonctions, Opérateurs, Conditions



Control flow basique avec if / else :

Permet d'exécuter **conditionnellement une instruction ou un bloc** et de proposer une alternative permettant aussi d'exécuter **une instruction ou un bloc** :

- 1) If
- 2) Condition entre parenthèses
- 3) Bloc ou instruction unique (à éviter)

Et si besoin

- 4) Else pour définir l'alternative
- 5) Bloc ou instruction unique (à éviter)

→ **L'alternative n'est pas exécutée si la condition est remplie**

→ **Toujours dans un bloc pour visibilité et éviter les erreurs !**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     // Ne pas faire
6     if (1)
7         printf("If légal mais peu lisible \n");
8
9     // Ne pas faire
10    if (0) {
11        printf("Ne s'affichera pas\n");
12    } else
13        printf("Else légal mais peu lisible \n");
14    printf("---\n");
15
16    // Faire
17    if (2 && -1) {
18        printf("2 && -1 rend 1\n");
19    } else {
20        printf("2 && -1 rend 0\n");
21    }
22    return EXIT_SUCCESS;
23 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples ./conditions
If légal mais peu lisible
Else légal mais peu lisible
---
2 && -1 rend 1
(base) → examples
```

Blocs, Fonctions, Opérateurs, Conditions



```
c elif.c > main(void)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5
6     unsigned char note = 17;
7     // 4 conditionnelles à la suite
8     if (note > 15) {
9         printf("Super\n");
10    }
11    if (note <= 15 && note > 10) {
12        printf("Ok\n");
13    }
14    if (note <= 10) {
15        printf("Aïe\n");
16    }
17
18     // 1 ou 2 conditionnelles
19    if (note > 15) {
20        printf("Super\n");
21    } else if (note > 10) {
22        printf("Ok\n");
23    } else {
24        printf("Aïe\n");
25    }
26    return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) ➜ examples git:(main) ✘ gcc elif.c -o elif
(base) ➜ examples git:(main) ✘ ./elif
Super
Super
(base) ➜ examples git:(main) ✘
```

Sinon si / elif / else if :

Le "sinon si" permet d'éviter l'enchaînement de tests si les conditions s'excluent.

Par exemple : la note **17** est supérieure à **15**, ce qui est "**Super**" mais aussi supérieure à **10** ce qui est éligible à l'appréciation "**Ok**", or on veut afficher la meilleure appréciation possible.

Pour ce type de tests, deux solutions :

- Enchaîner les **if**, mais il faut tester intervalle par intervalle, **ce qui ajoute de la redondance, des tests supplémentaires..**
- Éliminer un intervalle déjà testé avec un **else**, et tester le suivant avec un **if** : "**else if**". Dans l'encadré vert l'intervalle **note > 15** est déjà exclu (ou traité).

Blocs, Fonctions, Opérateurs, Conditions



Particularité des conditions

Pourquoi se fatiguer si on connaît le résultat dès l'évaluation de la première opérande ?

```
home > olivier > Documents > Lectures > C > examples > C conditions_2.c >
3   int main() {
4       // vrai (ici 1) ou X est toujours vrai
5       if (1 || printf("Eval ?\n"));
6       // faux (ici 0) et X est toujours faux
7       if (0 && printf("Eval ?\n"));
8
9       // Pour les deux conditions suivantes,
10      // le premier opérande ne suffit pas
11      if (0 || printf("This should !\n"));
12      if (1 && printf("This should !\n"));
13
14  }
15
16
17
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
(base) → examples git:(main) ✘ gcc conditions_2.c
(base) → examples git:(main) ✘ ./a.out
This should !
This should !
(base) → examples git:(main) ✘
```



Boucles, Tableaux, Chaînes de caractères

Boucles, Tableaux, Chaînes de caractères



Control flow itératif : les boucles

Il existe 3 (+ 1 non recommandée) façon de faire des boucles en C :

- La boucle **while** pour exécuter un bloc tant qu'une condition est vraie (= ne vaut pas 0)

```
while(1) {  
    // boucle infini  
}
```

- La boucle **do while**, très similaire mais on exécute le bloc au moins une fois

```
do {  
    // exécuté une fois  
} while (0);
```

- La boucle **for** proposant un contrôle très fin sur l'itération avec une syntaxe permettant **une initialisation**, **une condition** (comme **while/do while**), **un incrément**

```
for (int i = 0; i < 10; i++) {  
    // i va de 0 à 9  
}
```

Remarque : dans des versions plus anciennes du C, la variable **i** aurait été initialisée **avant la boucle** :

```
int i;  
for (i = 0 ; i < 10 ; i ++)
```

Caractéristiques en mémoire des tableaux en C:

- Il s'agit d'une **séquence d'éléments de même type (et donc taille)**
- Les éléments sont **alignés en mémoire (stockage contigu)**
- Les tableaux sont **stockés dans la pile**
- Comme tous les éléments de la pile, **la taille d'un tableau ne peut pas être modifiée**
- L'index de la première valeur est **0**

Boucles, Tableaux, Chaînes de caractères



Initialisation des tableaux

- Le label est **suffixé par des crochets**, la taille peut y être renseignée
- Les valeurs peuvent être renseignées **entre accolades**

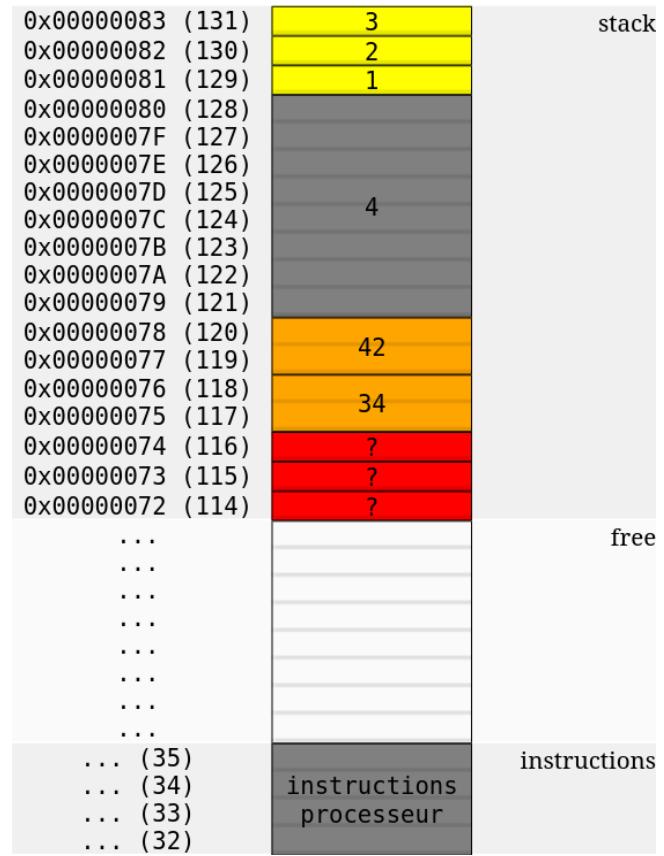
```
C arrays.c  x
C arrays.c > ⌂ main()

3 int main() {
4     // taille de 3, mais pas initialisé
5     char array_red[3];
6
7     // taille de 2, initialisé
8     short array_orange[2] = {34, 42};
9     unsigned long size_of = sizeof(array_orange);
10    printf("Taille de array_orange : %ld\n", size_of);
11
12    // Le compilateur en déduit une taille de 3
13    char array_yellow[] = {1, 2, 3};
14
15    // Illégal
16    // char array_4[];
17    return 0;
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Nombre d'octets d'une variable tableau

(base) → examples ./arrays
Taille de array_orange : 4
(base) → examples



Boucles, Tableaux, Chaînes de caractères

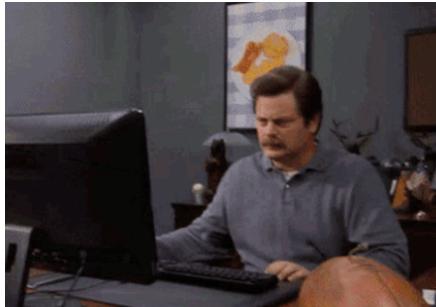


Utiliser {0} est la méthode classique pour initialiser toutes les cases d'un tableau à 0, attention à ne pas en déduire la logique de cette syntaxe...

Boucles, Tableaux, Chaînes de caractères



Utiliser {0} est la méthode classique pour initialiser toutes les cases d'un tableau à 0, attention à ne pas en déduire la logique de cette syntaxe...



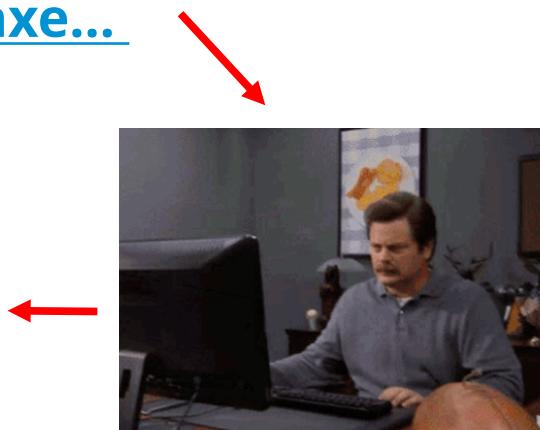
Boucles, Tableaux, Chaînes de caractères



Utiliser {0} est la méthode classique pour initialiser toutes les cases d'un tableau à 0, attention à ne pas en déduire la logique de cette syntaxe...

```
C array_init.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      char array_1[4] = {0};
6      for (int i = 0; i < 4 ; i++){
7          printf("%d ", array_1[i]);
8      }
9      printf("\n");
10
11     char array_2[4] = {[42]};
12     for (int i = 0; i < 4 ; i++){
13         printf("%d ", array_2[i]);
14     }
15     printf("\n");
16
17     return EXIT_SUCCESS;
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL



Boucles, Tableaux, Chaînes de caractères

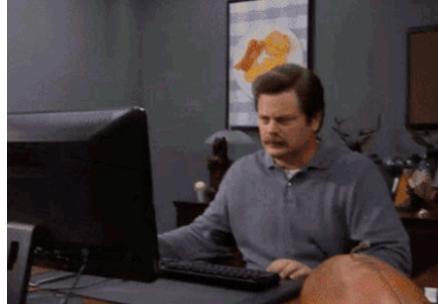


Utiliser {0} est la méthode classique pour initialiser toutes les cases d'un tableau à 0, attention à ne pas en déduire la logique de cette syntaxe...

```
C array_init.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      char array_1[4] = {0};
6      for (int i = 0; i < 4 ; i++){
7          printf("%d ", array_1[i]);
8      }
9      printf("\n");
10
11     char array_2[4] = {[42]};
12     for (int i = 0; i < 4 ; i++){
13         printf("%d ", array_2[i]);
14     }
15     printf("\n");
16
17     return EXIT_SUCCESS;
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples gcc array_init.c -o array_init
(base) → examples ./array_init
0 0 0 0
42 0 0 0
(base) → examples
```



- J'aime pas trop beaucoup ça.

Boucles, Tableaux, Chaînes de caractères

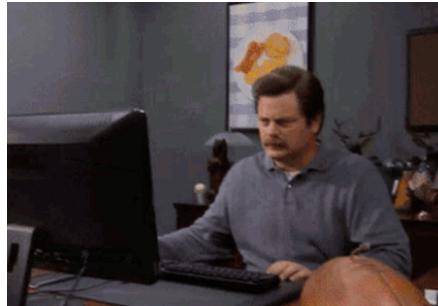


Utiliser {0} est la méthode classique pour initialiser toutes les cases d'un tableau à 0, attention à ne pas en déduire la logique de cette syntaxe...

```
C array_init.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      char array_1[4] = {0};
6      for (int i = 0; i < 4 ; i++){
7          printf("%d ", array_1[i]);
8      }
9      printf("\n");
10
11     char array_2[4] = {[42]};
12     for (int i = 0; i < 4 ; i++){
13         printf("%d ", array_2[i]);
14     }
15     printf("\n");
16
17     return EXIT_SUCCESS;
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples gcc array_init.c -o array_init
(base) → examples ./array_init
0 0 0 0
42 0 0 0
(base) → examples
```



https://www.gnu.org/

You don't have to explicitly initialize all of the array elements. For example, this code initializes the first three elements as specified, and then initializes the last two elements to a default value of zero:

```
int my_array[5] = { 0, 1, 2 };
```

- J'aime pas trop beaucoup ça.

Boucles, Tableaux, Chaînes de caractères

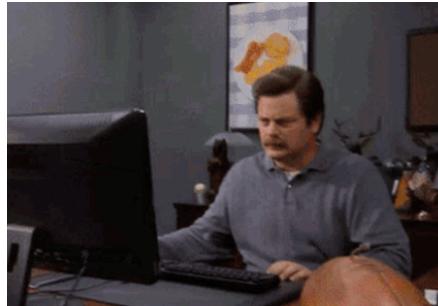


Utiliser {0} est la méthode classique pour initialiser toutes les cases d'un tableau à 0, attention à ne pas en déduire la logique de cette syntaxe...

```
c array_init.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      char array_1[4] = {0};
6      for (int i = 0; i < 4 ; i++){
7          printf("%d ", array_1[i]);
8      }
9      printf("\n");
10
11     char array_2[4] = {[42]};
12     for (int i = 0; i < 4 ; i++){
13         printf("%d ", array_2[i]);
14     }
15     printf("\n");
16
17     return EXIT_SUCCESS;
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples gcc array_init.c -o array_init
(base) → examples ./array_init
0 0 0 0
42 0 0 0
(base) → examples



https://www.gnu.org/

You don't have to explicitly initialize all of the array elements. For example, this code initializes the first three elements as specified, and then initializes the last two elements to a default value of zero:

```
int my_array[5] = { 0, 1, 2 };
```



Boucles, Tableaux, Chaînes de caractères

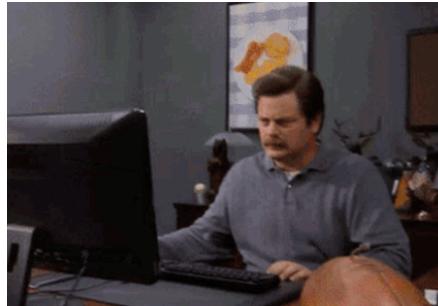


Utiliser {0} est la méthode classique pour initialiser toutes les cases d'un tableau à 0, attention à ne pas en déduire la logique de cette syntaxe...

```
c array_init.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      char array_1[4] = {0};
6      for (int i = 0; i < 4 ; i++){
7          printf("%d ", array_1[i]);
8      }
9      printf("\n");
10
11     char array_2[4] = {[42]};
12     for (int i = 0; i < 4 ; i++){
13         printf("%d ", array_2[i]);
14     }
15     printf("\n");
16
17     return EXIT_SUCCESS;
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples gcc array_init.c -o array_init
(base) → examples ./array_init
0 0 0 0
42 0 0 0
(base) → examples



G https://www.gnu.org/

You don't have to explicitly initialize all of the array elements. For example, this code initializes the first three elements as specified, and then initializes the last two elements to a default value of zero:

```
int my_array[5] = { 0, 1, 2 };
```



→ Pas d'initialisation partielle !

Boucles, Tableaux, Chaînes de caractères

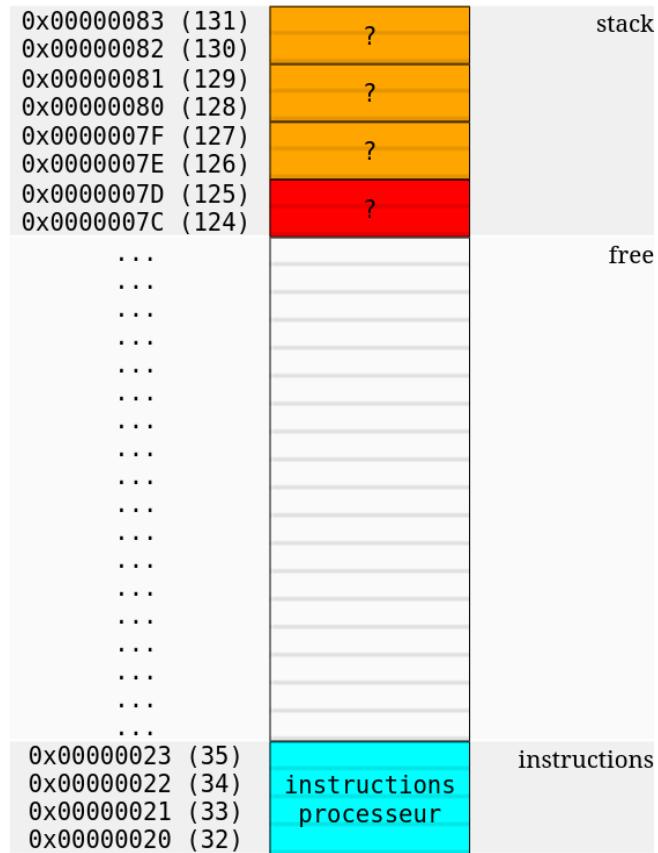


Accès aux valeurs

- Le tableau fait partie de la frame
- Comme pour les variables, s'il n'est pas initialisé, les valeurs en mémoire sont indéfinies (il peut y avoir n'importe quoi)

```
C array_access.c X

C array_access.c > ...
1 #include<stdlib.h>
2
3 int main() {
4
5     short val;
6     short array_orange[] = {1, 2, 3};
7     array_orange[1] = 42;
8     val = array_orange[2];
9
10    return EXIT_SUCCESS;
11 }
12 }
```



Boucles, Tableaux, Chaînes de caractères

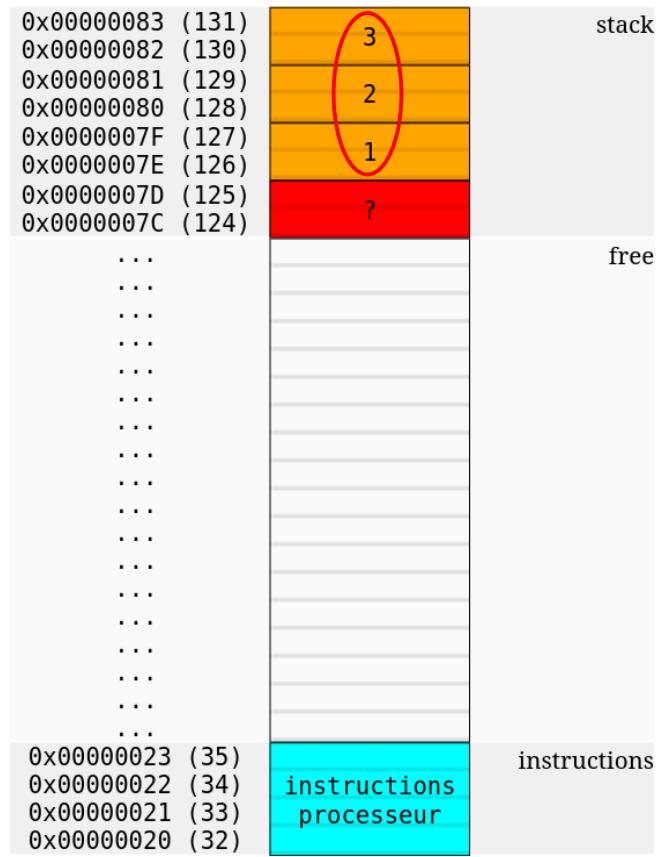


Accès aux valeurs

- L'initialisation des données est opérée lors de sa déclaration
- ! La syntaxe **array_orange = {1, 2, 3}** est toutefois interdite **après l'initialisation** : les tableaux ne sont pas assignables, seules leurs cases le sont !

```
C array_access.c X

C array_access.c > ...
1 #include<stdlib.h>
2
3 int main() {
4
5     short val;
6     short array_orange[] = {1, 2, 3}; →
7     array_orange[1] = 42;
8     val = array_orange[2];
9
10    return EXIT_SUCCESS;
11 }
12
```



Boucles, Tableaux, Chaînes de caractères

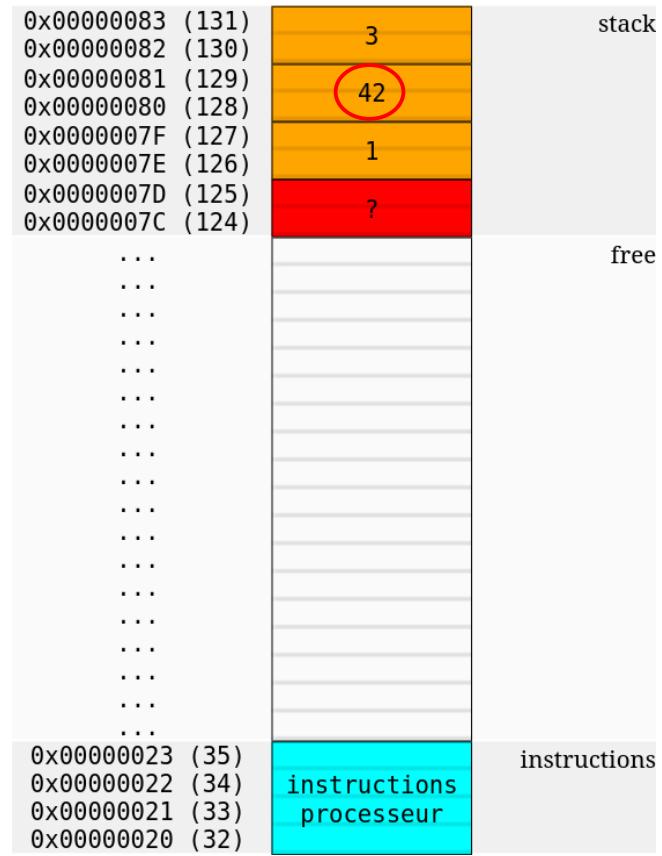


Accès aux valeurs

- Le changement d'une valeur se comporte comme une assignation simple (**copie** de la valeur)
- Ici, un littéral est renseigné (42), le comportement est le même avec une variable

```
C array_access.c X

C array_access.c > ...
1 #include<stdlib.h>
2
3 int main() {
4
5     short val;
6     short array_orange[] = {1, 2, 3};
7     array_orange[1] = 42;
8     val = array_orange[2];
9
10    return EXIT_SUCCESS;
11 }
12
```



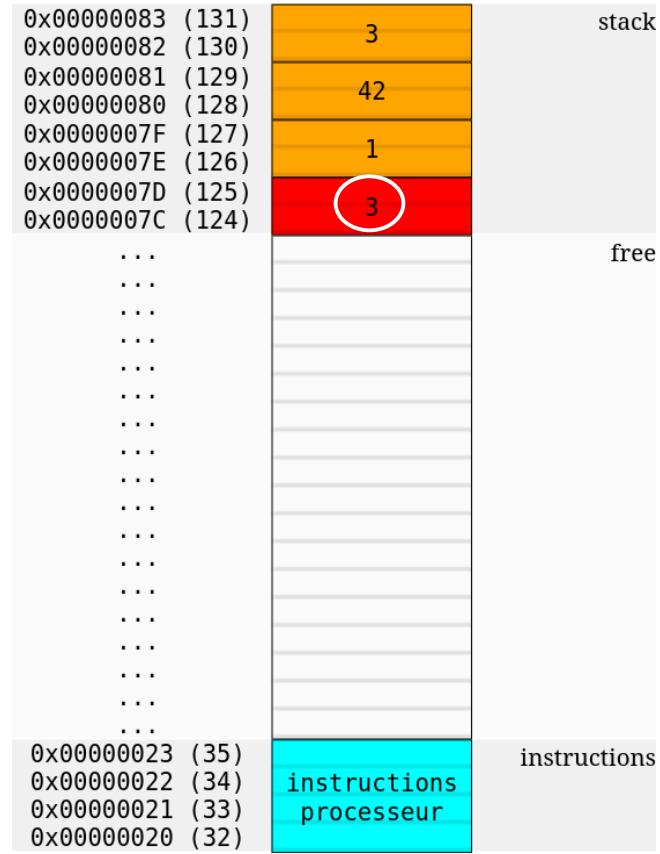
Boucles, Tableaux, Chaînes de caractères



Accès aux valeurs

- Récupérer une valeur du tableau se fait aussi par **copie**
- Le tableau reste donc inchangé

```
C array_access.c X
C array_access.c > ...
1 #include<stdlib.h>
2
3 int main() {
4
5     short val;
6     short array_orange[] = {1, 2, 3};
7     array_orange[1] = 42;
8     val = array_orange[2];
9
10    return EXIT_SUCCESS;
11 }
12
```



Boucles, Tableaux, Chaînes de caractères



⚠ Il n'y a pas de sécurité enfant

- Il n'y a **pas de contrôle** sur la longueur du tableau pour opérer un accès
- Ici on va lire une **zone mémoire hors du tableau**, le **comportement est indéfini**
- On va chercher la case mémoire "**comme si**" le tableau faisait une taille d'au moins 100 cases en partant de la première
- On peut voir que des données sont présentes en mémoire à cet endroit (**11735**)
- On lit ou modifie 2 octets dans une zone où des données importantes peuvent être présentes..
→ **C'est la responsabilité du programmeur d'éviter les dépassemens de tableau**

A screenshot of a terminal window showing a C program named "array_access.c". The code defines a small array "array_orange" with three elements (1, 2, 3), then sets the fourth element to 42. It then attempts to read from index 100 of the array, which is outside its bounds. The terminal output shows the program running and printing the value 11735, which is the memory address of the byte at index 100 of the stack. The line "val = array_orange[100];" and the terminal output "val = 11735" are circled in red.

```
c array_access.c x
C array_access.c > ⌂ main()
1 #include<stdlib.h>
2 #include<stdio.h>
3
4 int main() {
5
6     short val;
7     short array_orange[] = {1, 2, 3};
8     array_orange[1] = 42;
9     val = array_orange[100];
10    printf("val = %hd\n", val);
11
12    return EXIT_SUCCESS; // mouais..
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) ➔ examples ./a.out
val = 11735
- (base) ➔ examples

Il y a plus à dire sur les tableaux, nous y reviendrons !

Boucles, Tableaux, Chaînes de caractères



Exécution d'une boucle

```
c loop.c    x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5       → short arr[] = {29, 83, 34};
6
7       for (int i = 0; i < 3; i++) {
8           printf("arr[%d] : %hd\n", i, arr[i]);
9       }
10
11       return EXIT_SUCCESS;
12   }
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

0x00000083 (131)	?	stack
0x00000082 (130)	?	
0x00000081 (129)	?	
0x00000080 (128)	?	
0x0000007F (127)	?	
0x0000007E (126)	?	
0x0000007D (125)	?	
0x0000007C (124)	?	
0x0000007B (123)	?	
0x0000007A (122)	?	
...	...	free
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
0x00000023 (35)	instructions	
0x00000022 (34)	processeur	
0x00000021 (33)		
0x00000020 (32)		

Boucles, Tableaux, Chaînes de caractères



Exécution d'une boucle

```
c loop.c    x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

0x00000083 (131)	?	stack
0x00000082 (130)		
0x00000081 (129)		
0x00000080 (128)		
0x0000007F (127)		
0x0000007E (126)		
0x0000007D (125)		
0x0000007C (124)		
0x0000007B (123)		
0x0000007A (122)		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
0x00000023 (35)		instructions
0x00000022 (34)		
0x00000021 (33)		
0x00000020 (32)		
	instructions	
	processeur	

Boucles, Tableaux, Chaînes de caractères



Exécution d'une boucle

```
c loop.c x
c loop.c > main(void)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5
6     short arr[] = {29, 83, 34};
7
8     for (int i = 0; i < 3; i++) {
9         printf("arr[%d] : %hd\n", i, arr[i]);
10    }
11
12    return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

0x00000083 (131)	stack
0x00000082 (130)	
0x00000081 (129)	
0x00000080 (128)	
0x0000007F (127)	
0x0000007E (126)	
0x0000007D (125)	
0x0000007C (124)	
0x0000007B (123)	
0x0000007A (122)	
...	free
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
...	
0x00000023 (35)	instructions
0x00000022 (34)	
0x00000021 (33)	
0x00000020 (32)	processeur

Boucles, Tableaux, Chaînes de caractères



Exécution d'une boucle

```
c loop.c    x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

0x00000083 (131)	0	stack
0x00000082 (130)	34	
0x00000081 (129)	83	
0x00000080 (128)	29	
0x0000007F (127)	...	
0x0000007E (126)	...	
0x0000007D (125)	...	
0x0000007C (124)	...	
0x0000007B (123)	...	
0x0000007A (122)	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
0x00000023 (35)	instructions	
0x00000022 (34)	processeur	
0x00000021 (33)		
0x00000020 (32)		

Boucles, Tableaux, Chaînes de caractères



Exécution d'une boucle

```
c loop.c    x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

0x00000083 (131)	0	stack
0x00000082 (130)	34	
0x00000081 (129)	83	
0x00000080 (128)	29	
0x0000007F (127)	...	
0x0000007E (126)	...	
0x0000007D (125)	...	
0x0000007C (124)	...	
0x0000007B (123)	...	
0x0000007A (122)	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
...	...	
0x00000023 (35)	instructions	
0x00000022 (34)	processeur	
0x00000021 (33)		
0x00000020 (32)		

Boucles, Tableaux, Chaînes de caractères

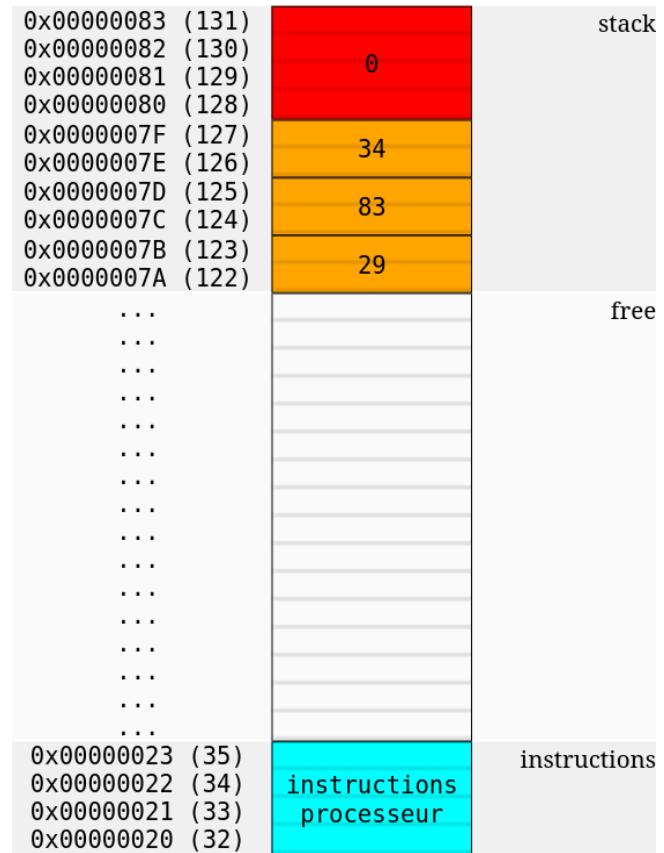


Exécution d'une boucle

```
c loop.c    x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) ➜ examples ./loop
arr[0] : 29



Boucles, Tableaux, Chaînes de caractères

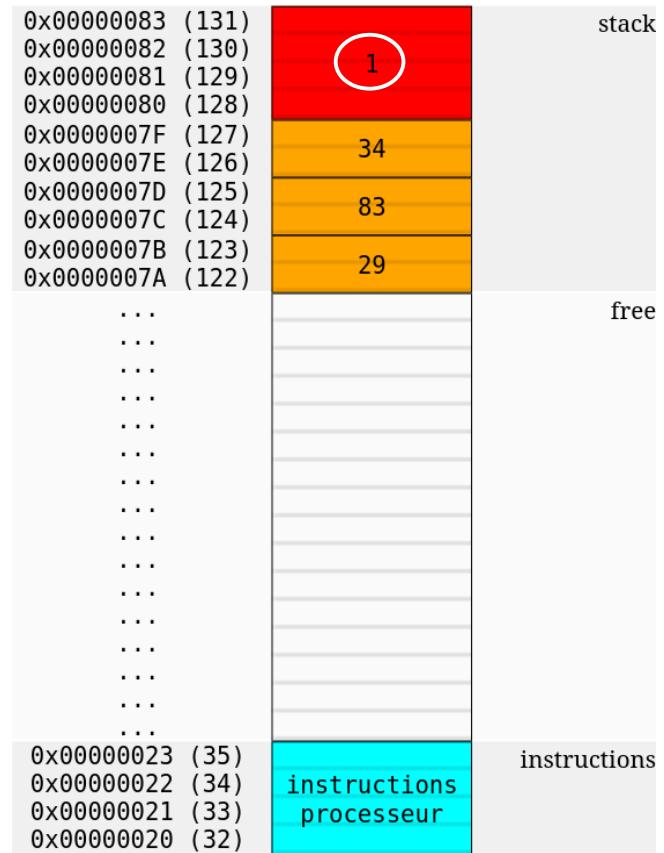


Exécution d'une boucle

```
c loop.c  x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop
arr[0] : 29



Boucles, Tableaux, Chaînes de caractères

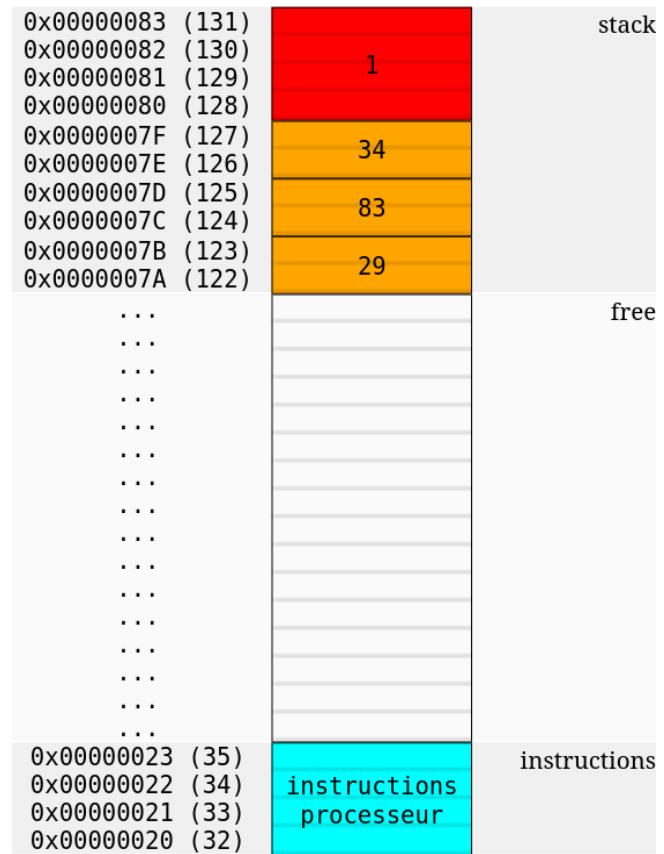


Exécution d'une boucle

```
c loop.c x
c loop.c > main(void)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5
6     short arr[] = {29, 83, 34};
7
8     for (int i = 0; i < 3; i++) {
9         printf("arr[%d] : %hd\n", i, arr[i]);
10    }
11
12    return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop
arr[0] : 29



Boucles, Tableaux, Chaînes de caractères

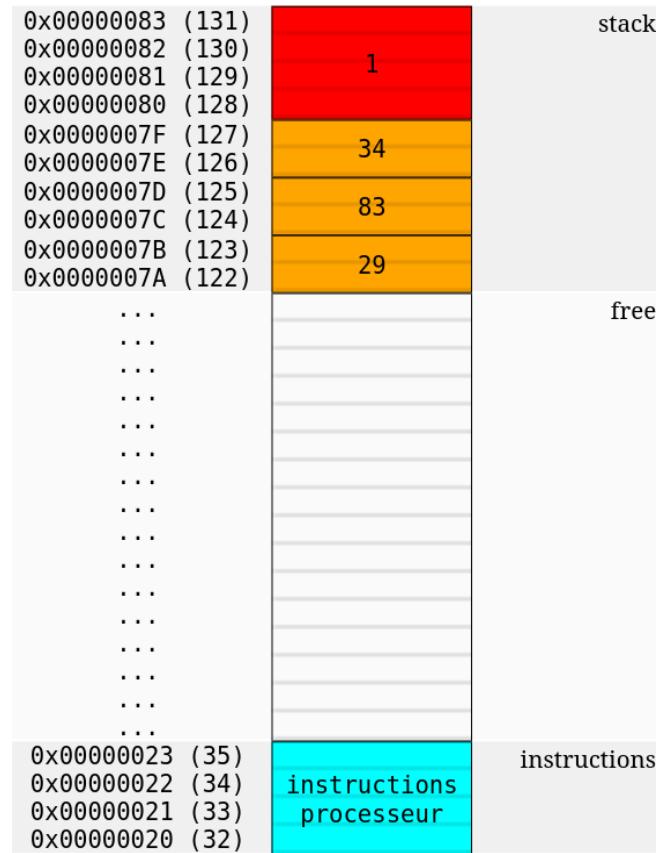


Exécution d'une boucle

```
c loop.c  x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop
arr[0] : 29



Boucles, Tableaux, Chaînes de caractères

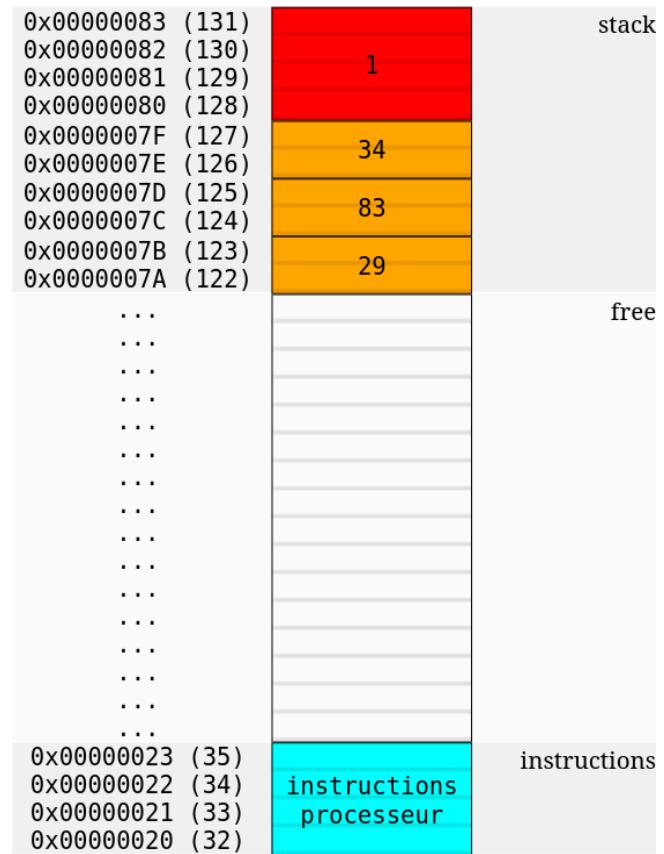


Exécution d'une boucle

```
c loop.c  x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) ➔ examples ./loop
arr[0] : 29
arr[1] : 83



Boucles, Tableaux, Chaînes de caractères



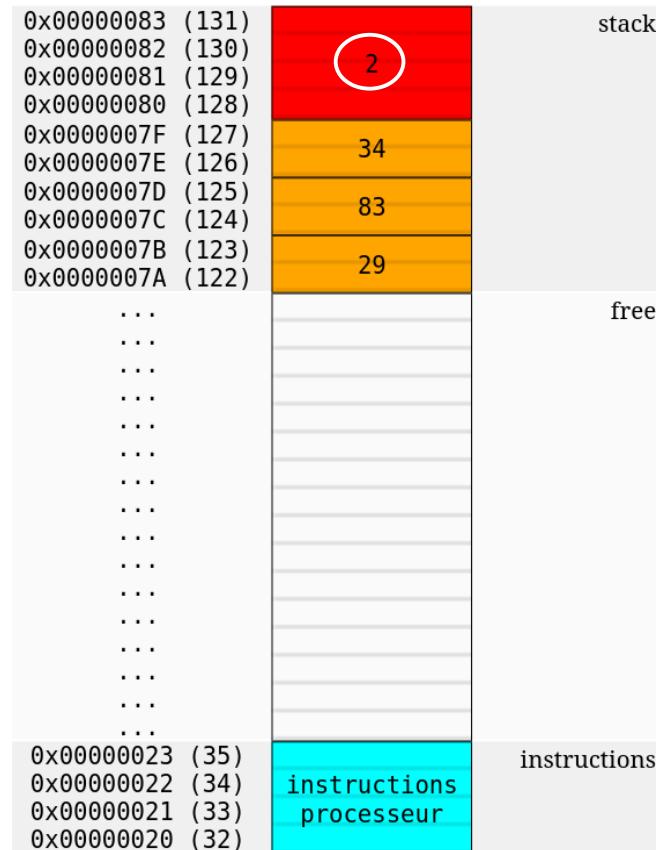
Exécution d'une boucle

```
c loop.c  x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

```
arr[0] : 29
arr[1] : 83
```



Boucles, Tableaux, Chaînes de caractères

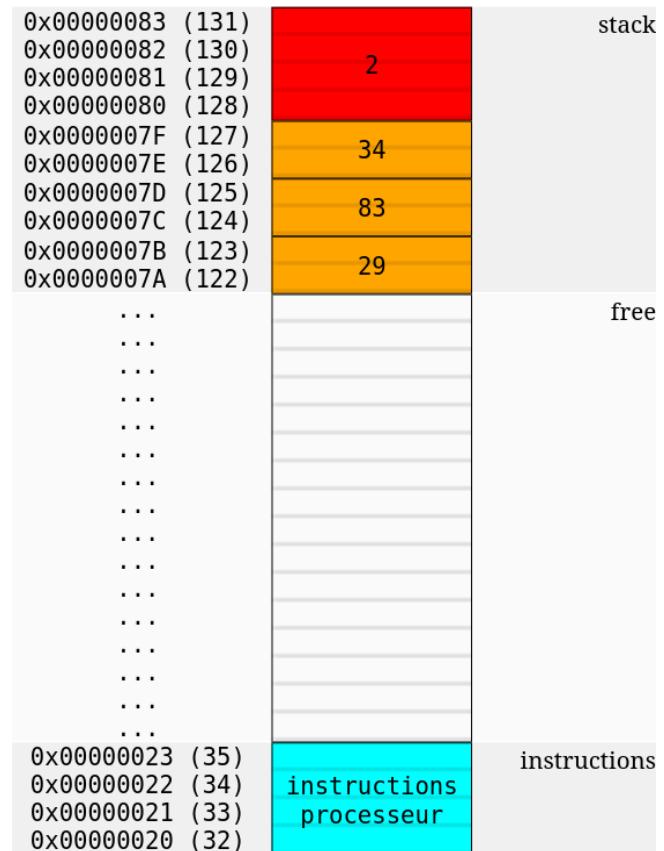


Exécution d'une boucle

```
c loop.c  x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop
arr[0] : 29
arr[1] : 83



Boucles, Tableaux, Chaînes de caractères

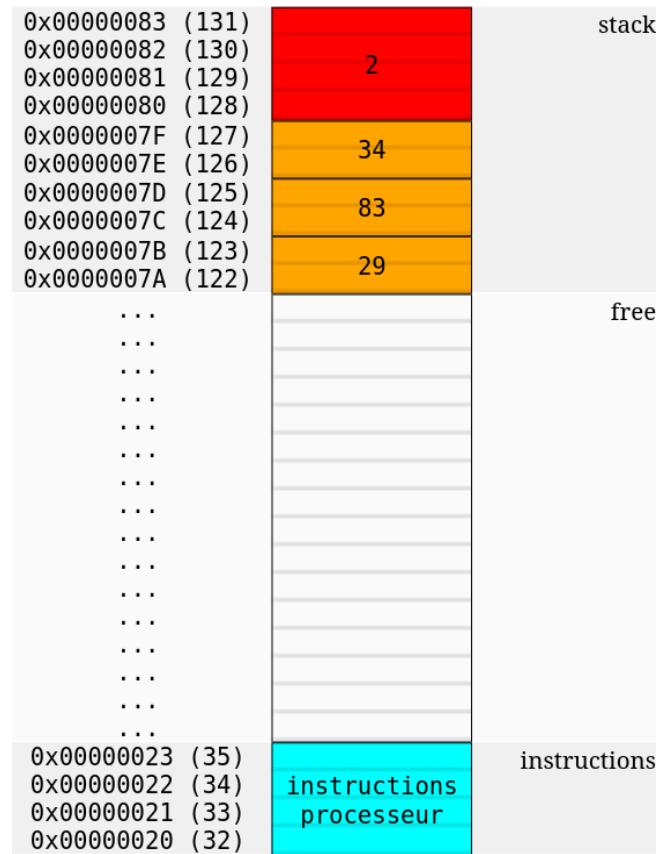


Exécution d'une boucle

```
c loop.c  x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop
arr[0] : 29
arr[1] : 83



Boucles, Tableaux, Chaînes de caractères



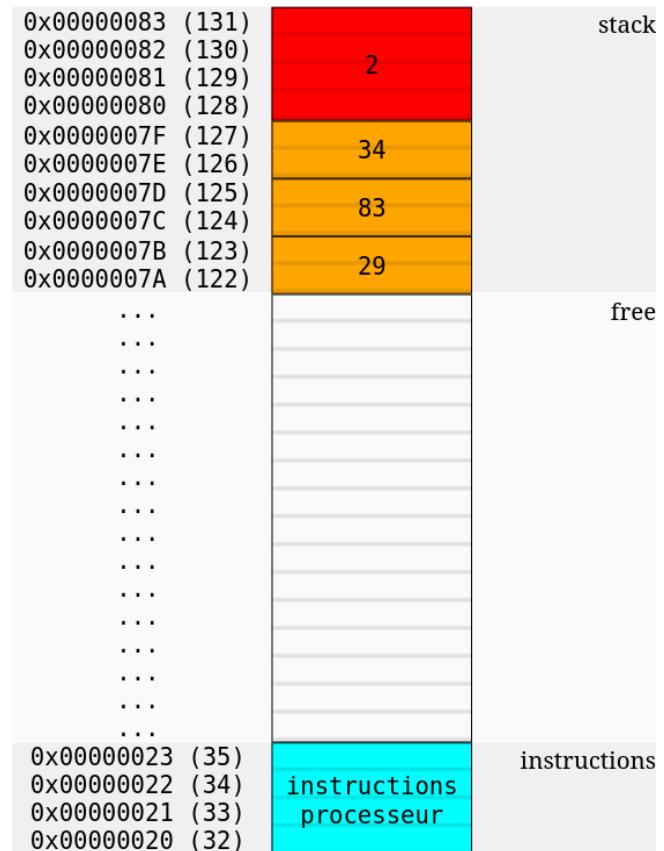
Exécution d'une boucle

```
c loop.c  x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) ➔ examples ./loop

```
arr[0] : 29
arr[1] : 83
arr[2] : 34
```



Boucles, Tableaux, Chaînes de caractères



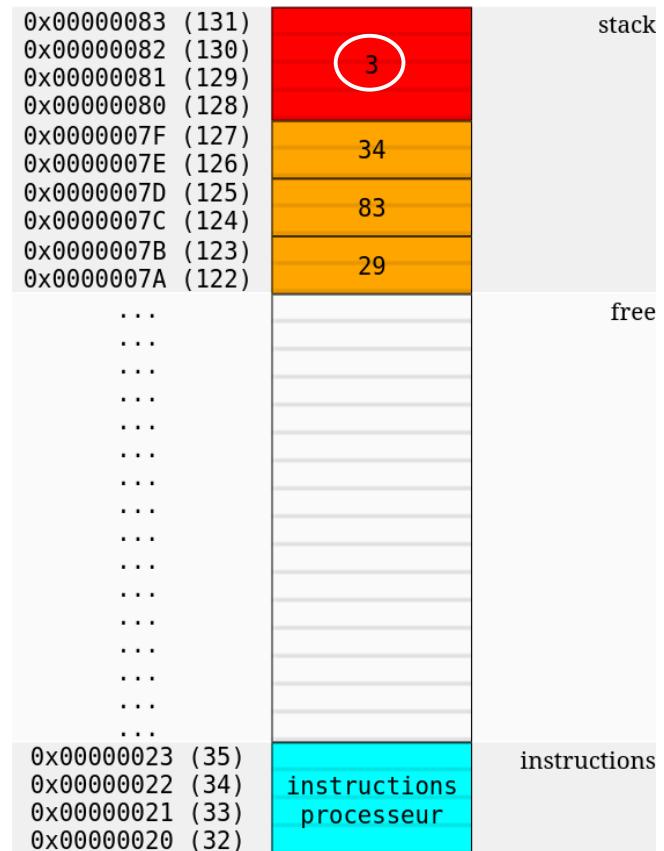
Exécution d'une boucle

```
c loop.c  x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

```
arr[0] : 29
arr[1] : 83
arr[2] : 34
```



Boucles, Tableaux, Chaînes de caractères



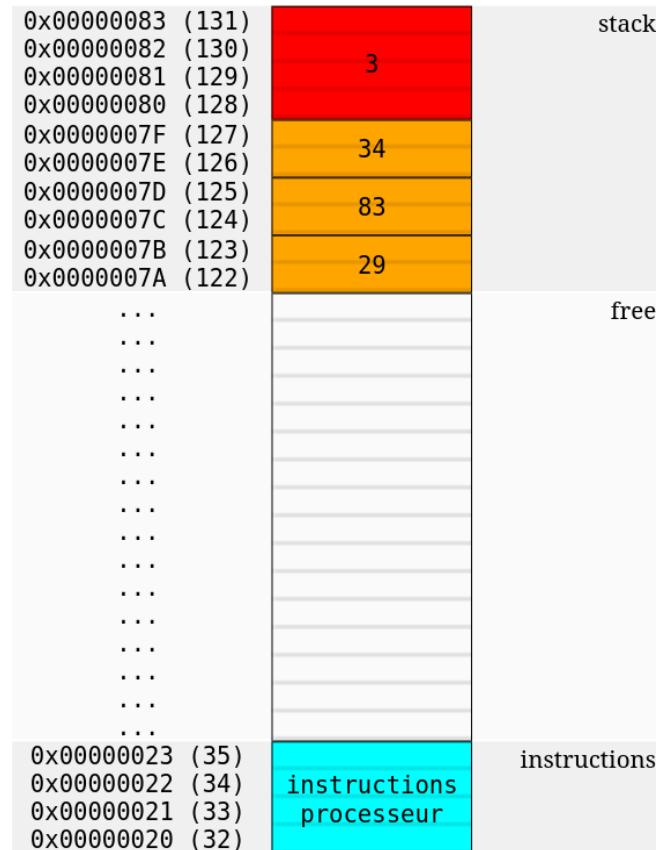
Exécution d'une boucle

```
c loop.c x
c loop.c > main(void)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5
6     short arr[] = {29, 83, 34};
7
8     for (int i = 0; i < 3; i++) {
9         printf("arr[%d] : %hd\n", i, arr[i]);
10    }
11
12    return EXIT_SUCCESS;
13 }
```

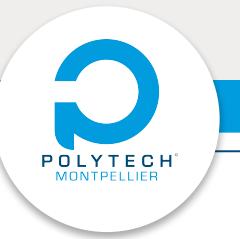
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

```
arr[0] : 29
arr[1] : 83
arr[2] : 34
```



Boucles, Tableaux, Chaînes de caractères



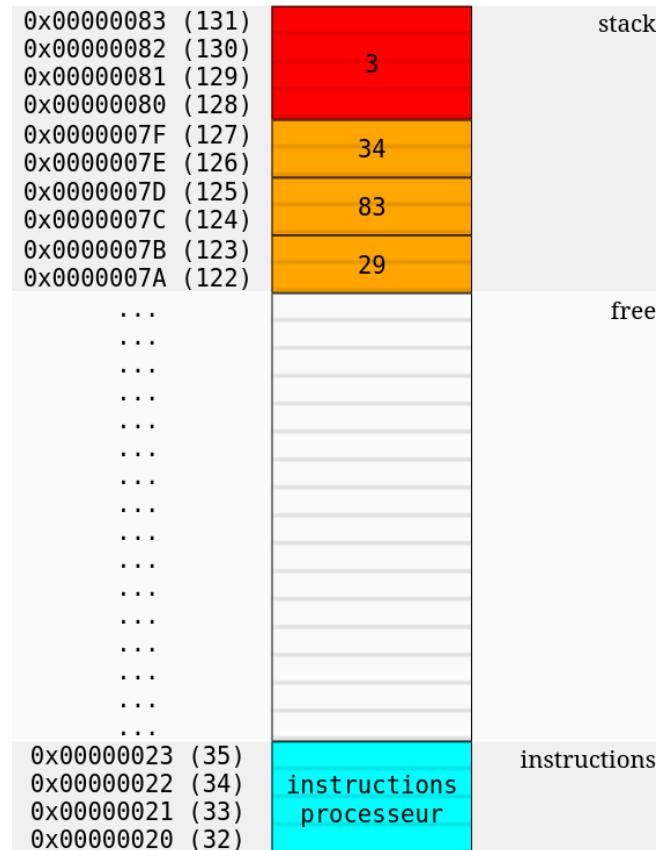
Exécution d'une boucle

```
c loop.c    x
c loop.c > ⌂ main(void)
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5
6       short arr[] = {29, 83, 34};
7
8       for (int i = 0; i < 3; i++) {
9           printf("arr[%d] : %hd\n", i, arr[i]);
10      }
11
12      return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) ➔ examples ./loop

```
arr[0] : 29
arr[1] : 83
arr[2] : 34
```



Boucles, Tableaux, Chaînes de caractères



```
break_continue.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5
6     short arr[] = {29, 83, 34, 77, 13, 35};
7
8     for (int i = 0; i < 6; i++) {
9         // on saute les valeurs paires
10        if (arr[i] % 2 == 0) {
11            continue;
12        }
13        // on s'arrête à la valeur 13
14        if (arr[i] == 13) {
15            break;
16        }
17        printf("arr[%d] : %hd\n", i, arr[i]);
18    }
19
20    return EXIT_SUCCESS;
21 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) > examples ./break_continue
arr[0] : 29
arr[1] : 83
arr[3] : 77
(base) > examples
```

Un contrôle plus fin

Comment **passer** des itérations ou **interrompre** la boucle ?

- Si certaines données ne sont pas éligibles au traitement, on peut directement passer à l'incrémentation à l'aide de l'instruction "**continue**"
- Dans certains cas, la boucle doit être interrompue. Un **return**, mettrait fin à la fonction en cours, on peut interrompre uniquement la boucle avec "**break**"

Boucles, Tableaux, Chaînes de caractères



Les chaînes de caractères (strings) sont des tableaux :

- Les chaînes en C sont par convention "**null-terminated**", elles se terminent par un '\0' (ce qui vaut 0)
- Certaines fonctions de <string.h> ignorent le 0, bien que présent en mémoire !
- Le "**format specifier**" est "%s" pour l'affichage, le saut de ligne est '\n'

```
C strings.c > ⌂ main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(void) {
6      char chaine[] = {'h', 'i', ' ', '!', '\0'};
7      printf("strlen de \"%s\" vaut %ld\n", chaine, strlen(chaine));
8      printf("sizeof \"%s\" vaut %ld\n", chaine, sizeof(chaine));
9      return EXIT_SUCCESS;
10 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ ./strings
strlen de "hi !" vaut 4
sizeof "hi !" vaut 5
○ (base) → examples git:(main) ✘
```

Boucles, Tableaux, Chaînes de caractères

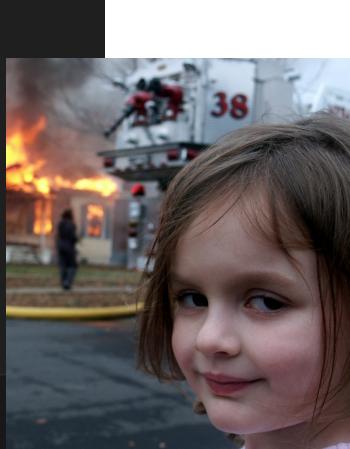
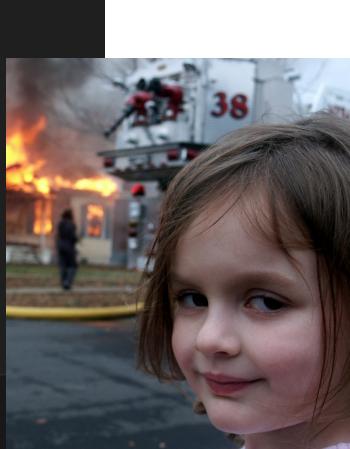
Initialisation des strings :

- Comme des tableaux
- Ou statiquement, avec une variable de type "char *"
- Les chaînes statiques ne peuvent pas être modifiées, en cas d'essai :

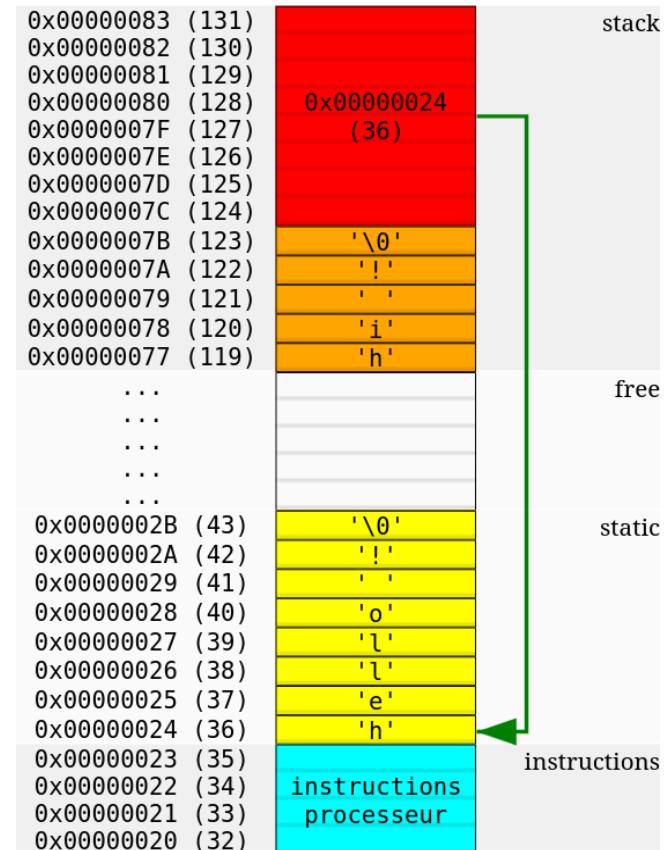
```
C static_strings.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(void) {
6      char chaine[] = {'h', 'i', ' ', '!', '\0'};
7      char *chaine_static = "Hello !";
8      printf("On tente une modification\n");
9      chaine_static[2] = 'e';
10     return EXIT_SUCCESS;
11 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples git:(main) ✘ ./static_strings
On tente une modification
[1] 1586805 segmentation fault (core dumped) ./static_strings
(base) → examples git:(main) ✘
```



Avec l'étoile ligne 7, on a du nouveau...



Boucles, Tableaux, Chaînes de caractères

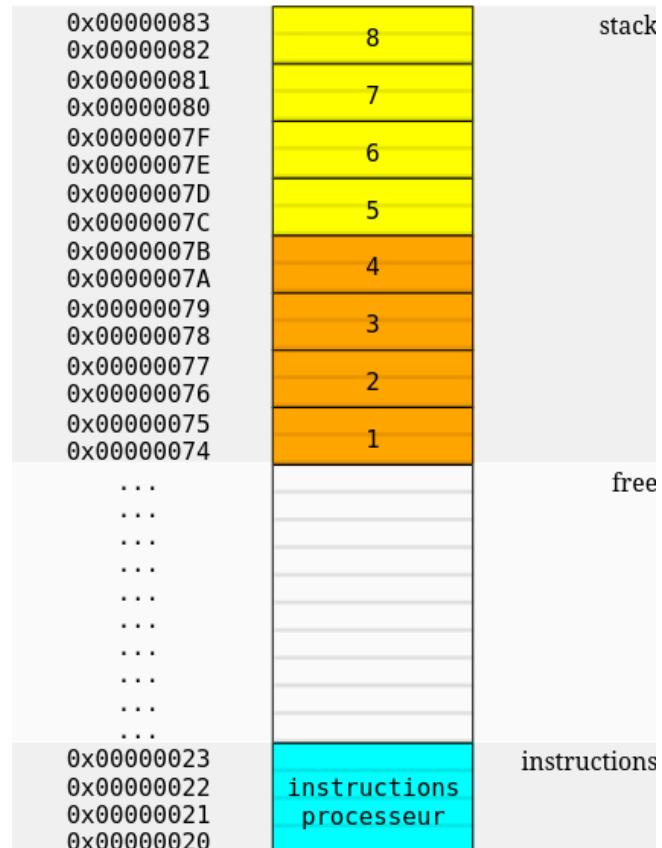


Tableaux multidimensionnels

- Ils sont aussi alignés en mémoire. Pour un **short tab2d[X][Y]**, on aura **X** fois **Y** cases alignées de taille **short** (ici **X = 2**, et **Y = 4**)

```
int main(void) {
    short tab2d[2][4] = { // X = 2 et Y = 4
        {1, 2, 3, 4},
        {5, 6, 7, 8}
};
```

Avec en orange le premier niveau (index 0) et en jaune le deuxième (index 1)



Boucles, Tableaux, Chaînes de caractères



Tableaux multidimensionnels

- Ils sont aussi alignés en mémoire. Pour un **short tab2d[X][Y]**, on aura **X** fois **Y** cases alignées de taille **short** (ici **X = 2**, et **Y = 4**)
- Le compilateur transformera un accès à la case **tab2d[i][j]** par **tab2d[0][i x Y + j]** "comme si" le tableau n'avait qu'une dimension (très grossièrement)
- À la ligne 11, nous "trichons" un peu, on garde le label de notre **int[2][4]**, et on part de la première case du premier niveau avec un overflow : 6 devrait être illégal sur un tableau de taille 4
- En effet, si on considère **vraiment** notre tableau 2D comme un tableau 1D (ligne 13), on peut voir qu'on peut accéder à la même case (comme si notre tableau 1D avait 8 cases)

```
examples > c tab2d_as_1d.c ...
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(void) {
5      short tab2d[2][4] = { // X = 2 et Y = 4
6          {1, 2, 3, 4},
7          {5, 6, 7, 8}
8      };
9      printf("valeur à tab2d[1][2]: %d \n", tab2d[1][2]);
10     // 1 x Y + 2 = 1 x 4 + 2 = 6
11     printf("valeur à tab2d[0][6]: %d \n", tab2d[0][6]);
12     // On détaillera ceci plus tard
13     short *tab1d = &tab2d[0][0];
14     printf("valeur à tab1d[6]: %d \n", tab1d[6]);
15     return EXIT_SUCCESS;
16 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) gcc tab2d_as_1d.c -o tab2d_as_1d
- (base) → examples git:(main) ./tab2d_as_1d
valeur à tab2d[1][2]: 7
valeur à tab2d[0][6]: 7
valeur à tab1d[6]: 7
- (base) → examples git:(main)

Encore une étoile ligne 13... il est temps de parler des pointeurs !



Pointeurs & Structures

Pointeurs & Structures



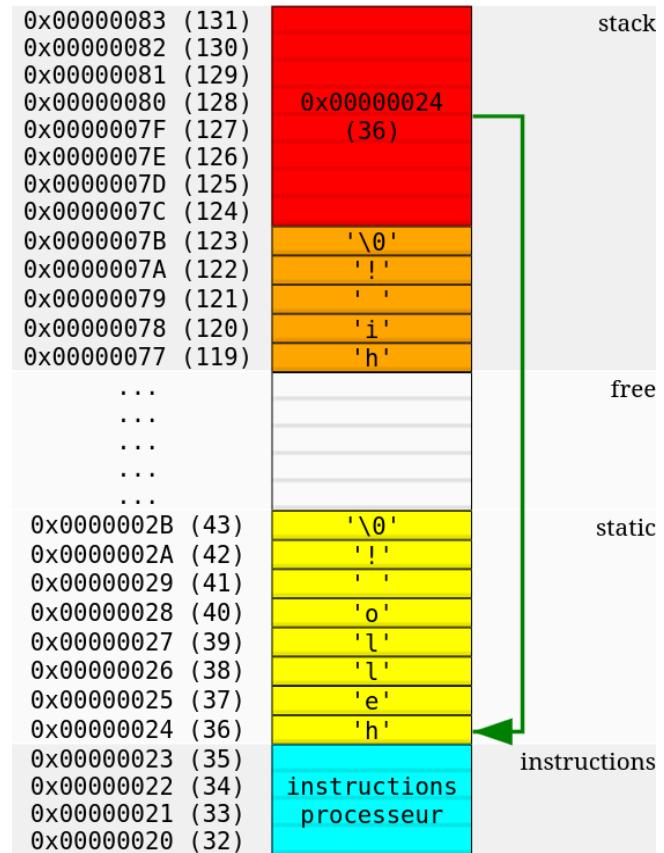
En repartant de l'exemple précédent :

- On observe que la variable **chaine_statique** contient l'**adresse** du premier octet de la chaîne
- Le label est préfixé par une **étoile** et l'adresse est sur 8 octets, **la variable est un pointeur !**

```
C static_strings.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(void) {
6      char chaine[] = {'h', 'i', ' ', '!', '\0'};
7      char *chaine_static = "Hello !";
8      printf("On tente une modification\n");
9      chaine_static[2] = 'e';
10     return EXIT_SUCCESS;
11 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples git:(main) ✘ ./static_strings
On tente une modification
[1] 1586805 segmentation fault (core dumped) ./static_strings
(base) → examples git:(main) ✘
```



Un pointeur est juste une variable contenant une adresse !

Pointeurs & Structures



Syntaxe des pointeurs :

- "*" permet de :
 - Déclarer un pointeur si elle suffit le type d'une initialisation, le type d'un argument de fonction
 - D'accéder à la valeur pointée dans les autres cas. On parle de déréférencement.
- "&" permet de : récupérer l'adresse d'une variable. **N'a pas de sens dans le membre gauche d'une assignation ou dans une liste d'arguments typés** (contrairement à C++ ou Rust)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int yellow_val = 42;
6     int *orange_ptr = &yellow_val;
7     printf("Adresse : %p\n", orange_ptr);
8     printf("Val par deref : %d\n", *orange_ptr);
9     return EXIT_SUCCESS;
10 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./pointeur
Adresse : 0x7ffdb76c472c
Val par deref : 42

Pointeurs & Structures



Syntaxe des pointeurs :

- "*" permet de :
 - Déclarer un pointeur si elle suffit le type d'une initialisation, le type d'un argument de fonction
 - D'accéder à la valeur pointée dans les autres cas. On parle de déréférencement.
- "&" permet de : récupérer l'adresse d'une variable. **N'a pas de sens dans le membre gauche d'une assignation ou dans une liste d'arguments typés** (contrairement à C++ ou Rust)

Les 3 syntaxes suivantes sont valides :

int* var, int *var, int * var

Bien que **int*** semblerait plus logique,
l'étoile à droite nous rappelle que si on
écrit "**int* a, b;**" alors **a** est bien un
pointeur, mais **b** reste un entier.
(Et on verra une autre justification)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int yellow_val = 42;
6     int *orange_ptr = &yellow_val;
7     printf("Adresse : %p\n", orange_ptr);
8     printf("Val par deref : %d\n", *orange_ptr);
9     return EXIT_SUCCESS;
10 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./pointeur
Adresse : 0x7ffdb76c472c
Val par deref : 42

Pointeurs & Structures

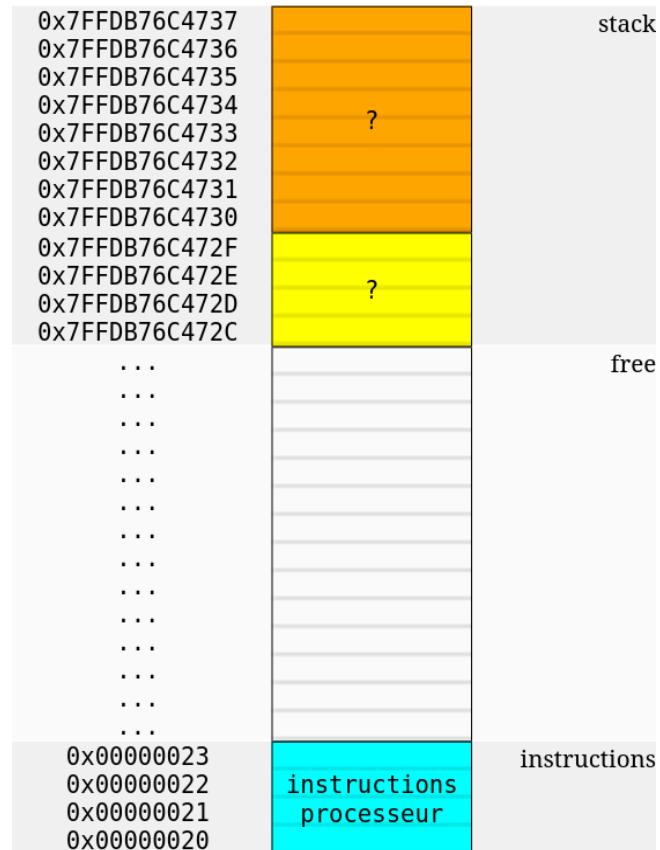


Exemple simple de manipulation des pointeurs

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int yellow_val = 42;
6     int *orange_ptr = &yellow_val;
7     printf("Adresse : %p\n", orange_ptr);
8     printf("Val par deref : %d\n", *orange_ptr);
9     return EXIT_SUCCESS;
10 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./pointeur



Pointeurs & Structures

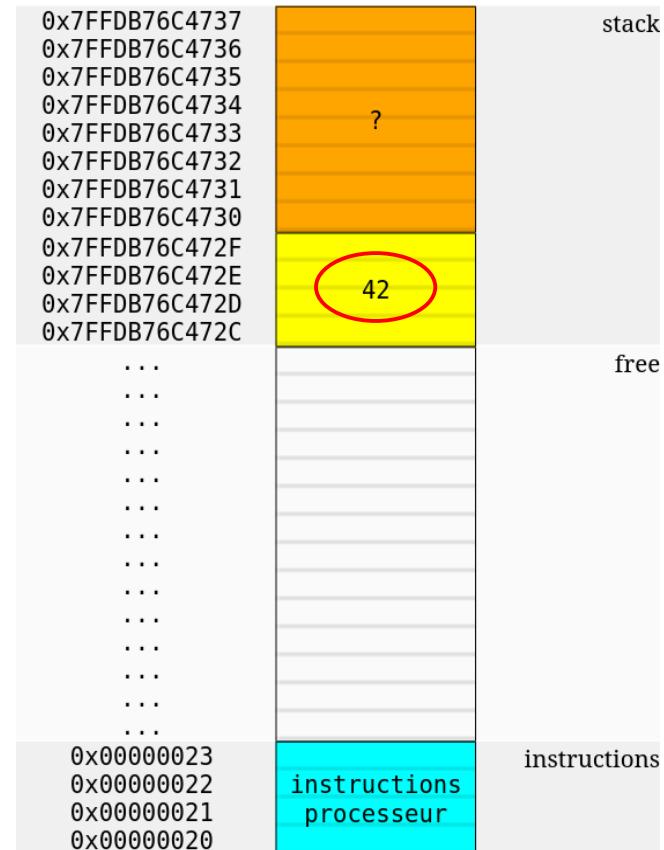


Exemple simple de manipulation des pointeurs

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int yellow_val = 42;
6     int *orange_ptr = &yellow_val;
7     printf("Adresse : %p\n", orange_ptr);
8     printf("Val par deref : %d\n", *orange_ptr);
9     return EXIT_SUCCESS;
10 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./pointeur



Pointeurs & Structures

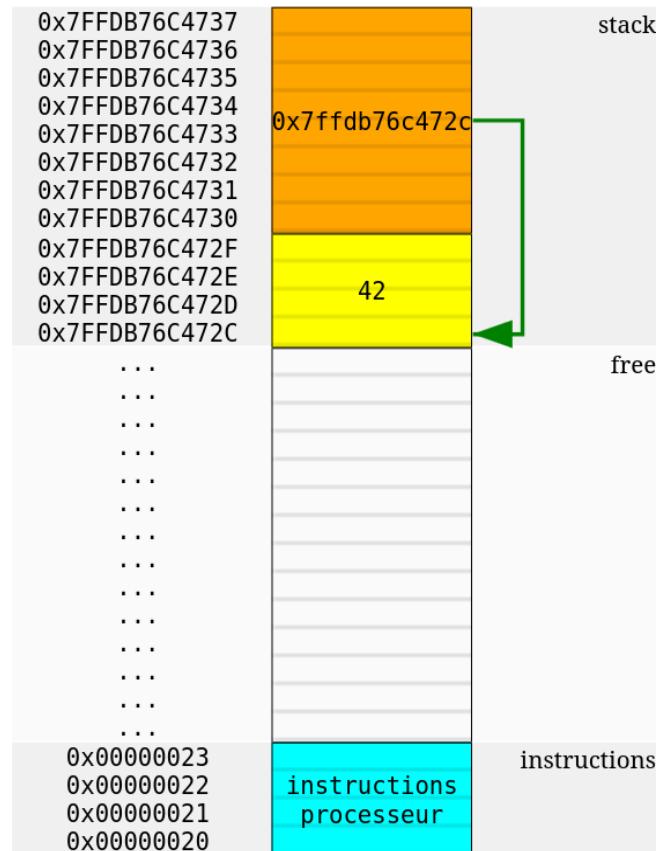


Exemple simple de manipulation des pointeurs

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int yellow_val = 42;
6     int *orange_ptr = &yellow_val;
7     printf("Adresse : %p\n", orange_ptr);
8     printf("Val par deref : %d\n", *orange_ptr);
9     return EXIT_SUCCESS;
10 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./pointeur



Pointeurs & Structures



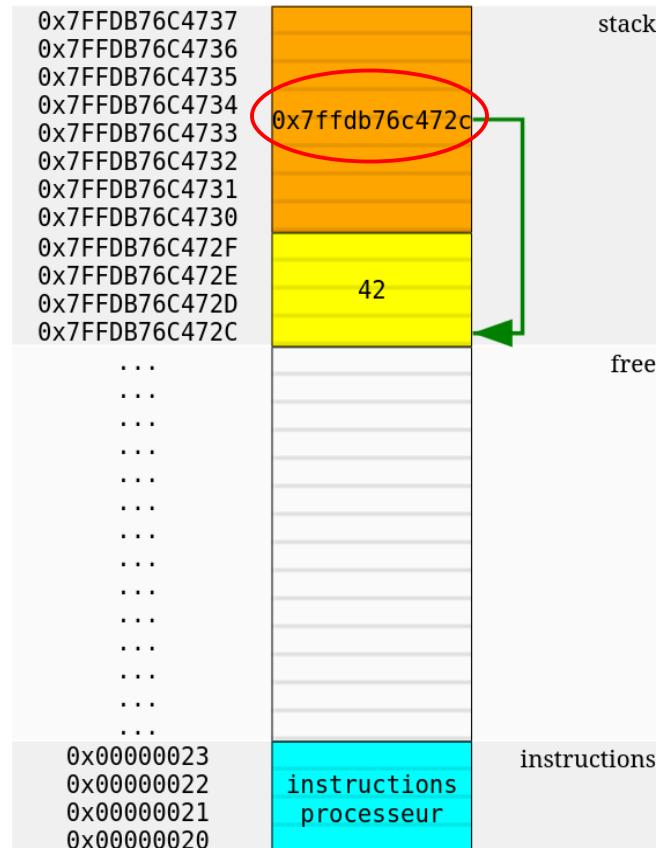
Exemple simple de manipulation des pointeurs

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int yellow_val = 42;
6     int *orange_ptr = &yellow_val;
7     printf("Adresse : %p\n", orange_ptr);
8     printf("Val par deref : %d\n", *orange_ptr);
9     return EXIT_SUCCESS;
10 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ ./pointeur
Adresse : 0x7ffdb76c472c

On affiche l'adresse !



Pointeurs & Structures

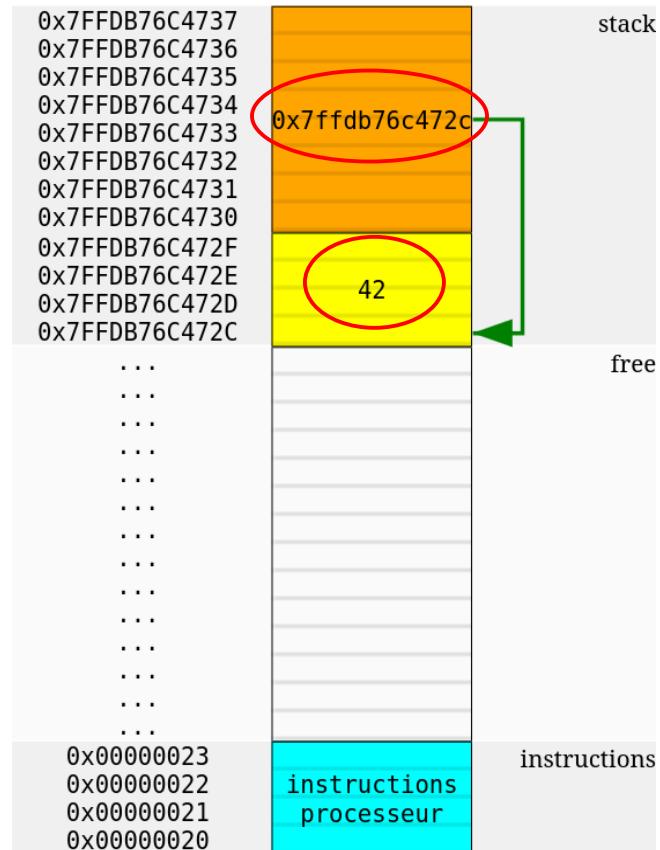


Exemple simple de manipulation des pointeurs

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int yellow_val = 42;
6     int *orange_ptr = &yellow_val;
7     printf("Adresse : %p\n", orange_ptr);
8     printf("Val par deref : %d\n", *orange_ptr); →
9     return EXIT_SUCCESS;
10 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ ./pointeur
Adresse : 0x7ffdb76c472c
Val par deref : 42



On affiche la valeur pointée par
déréférencement !

Pointeurs & Structures



Pointeurs et fonctions :

- On a vu que les valeurs passées en arguments aux fonctions sont **copiées** dans un **frame**
- Une modification des valeurs dans la fonction ne change que la copie dans la **frame**
- **Passer un pointeur** à une fonction **ne présente rien de spécial** : la variable, ici une adresse, est aussi copiée dans la frame
- Cependant, le pointeur original et sa copie ont la même valeur, **ils pointent donc vers la même zone de mémoire**

→ **Le déréférencement de pointeurs permet donc de modifier une zone mémoire en dehors de la frame. Passer un pointeur est une façon d'autoriser une fonction à modifier la valeur d'origine**

Pointeurs & Structures

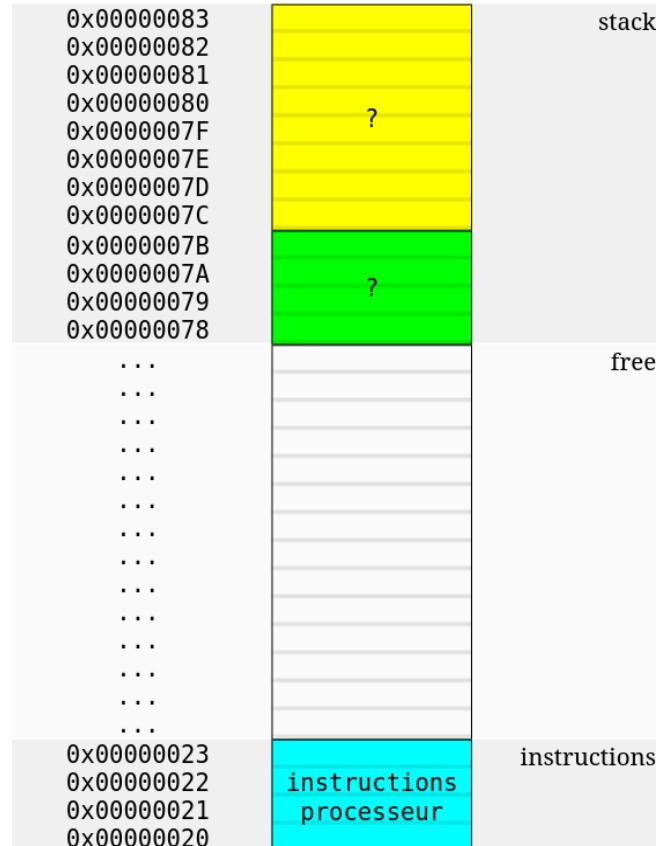


Les pointeurs et fonctions

```
examples > C Fonction_pointeur.c > main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void add_ten(int *orange){
5      *orange += 10;
6  }
7  int main(void) {
8      int green = 5;
9      int *yellow = &green;
10     add_ten(yellow);
11     printf("green : %d\n", green);
12     return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./fonction_pointeur



Pointeurs & Structures

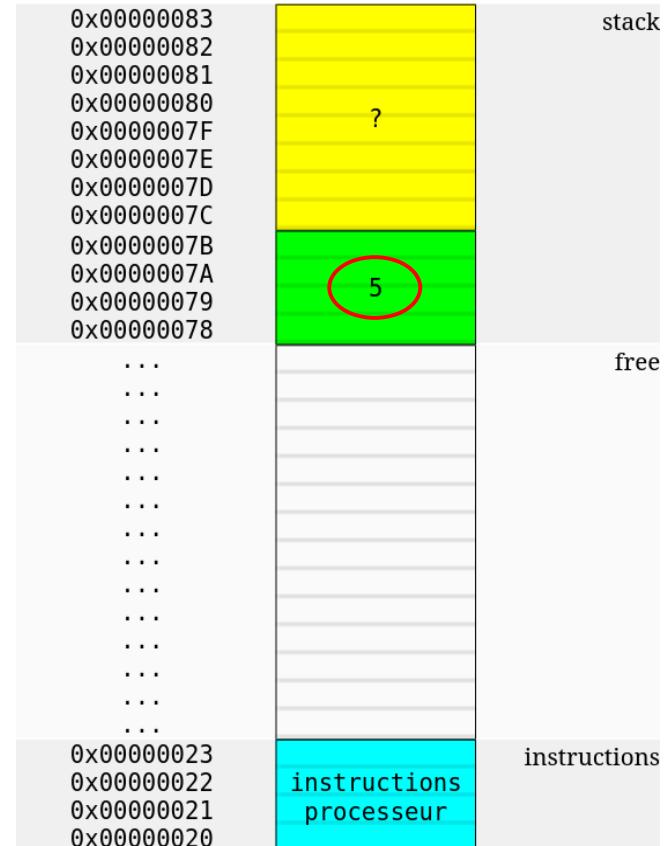


Les pointeurs et fonctions

```
examples > C Fonction_pointeur.c > main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void add_ten(int *orange){
5      *orange += 10;
6  }
7  int main(void) {
8      int green = 5;
9      int *yellow = &green;
10     add_ten(yellow);
11     printf("green : %d\n", green);
12     return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./fonction_pointeur



Pointeurs & Structures

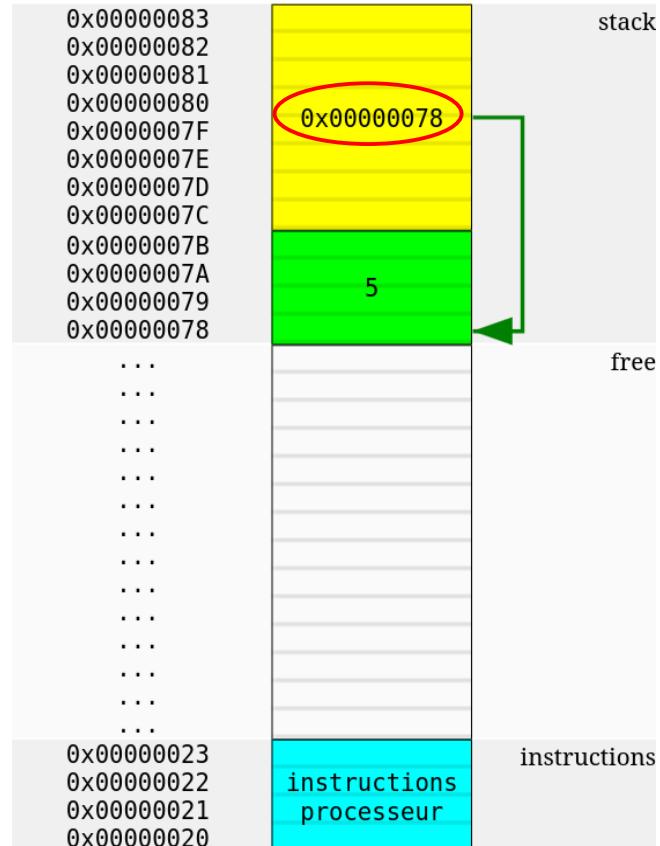


Les pointeurs et fonctions

```
examples > C Fonction_pointeur.c > main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void add_ten(int *orange){
5      *orange += 10;
6  }
7  int main(void) {
8      int green = 5;
9      int *yellow = &green;
10     add_ten(yellow);
11     printf("green : %d\n", green);
12     return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./fonction_pointeur



Pointeurs & Structures



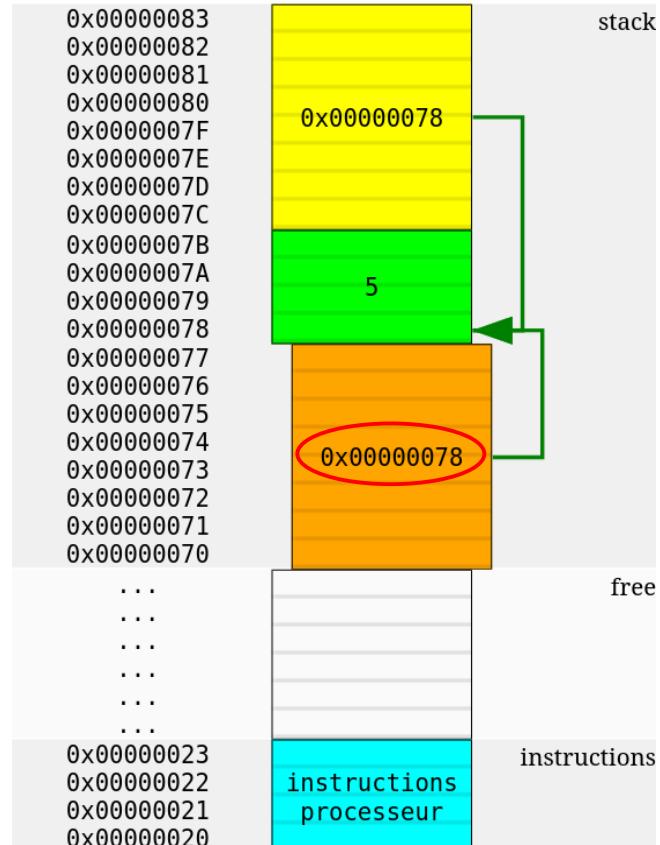
Les pointeurs et fonctions

```
examples > C Fonction_pointeur.c > main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void add_ten(int *orange){
5      *orange += 10;
6  }
7  int main(void) {
8      int green = 5;
9      int *yellow = &green;
10     add_ten(yellow);
11     printf("green : %d\n", green);
12     return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./fonction_pointeur

La copie "orange" dans la frame de la fonction "add_ten" pointe vers la même valeur que le pointeur original "yellow"



Pointeurs & Structures



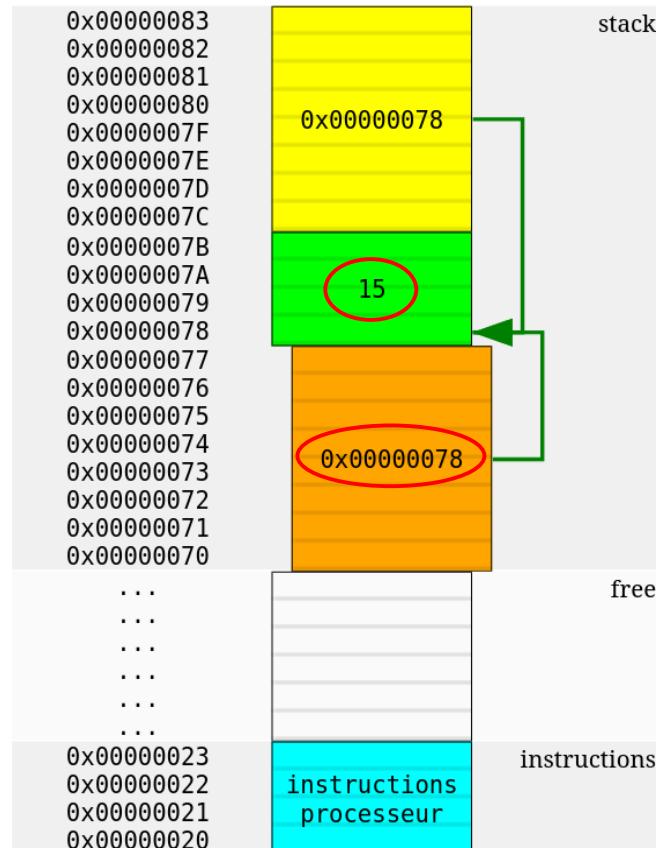
Les pointeurs et fonctions

```
examples > C Fonction_pointeur.c > main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void add_ten(int *orange){
5      *orange += 10;
6  }
7  int main(void) {
8      int green = 5;
9      int *yellow = &green;
10     add_ten(yellow);
11     printf("green : %d\n", green);
12     return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./fonction_pointeur

Modification de "green" par déréférencement
du pointeur "orange"



Pointeurs & Structures

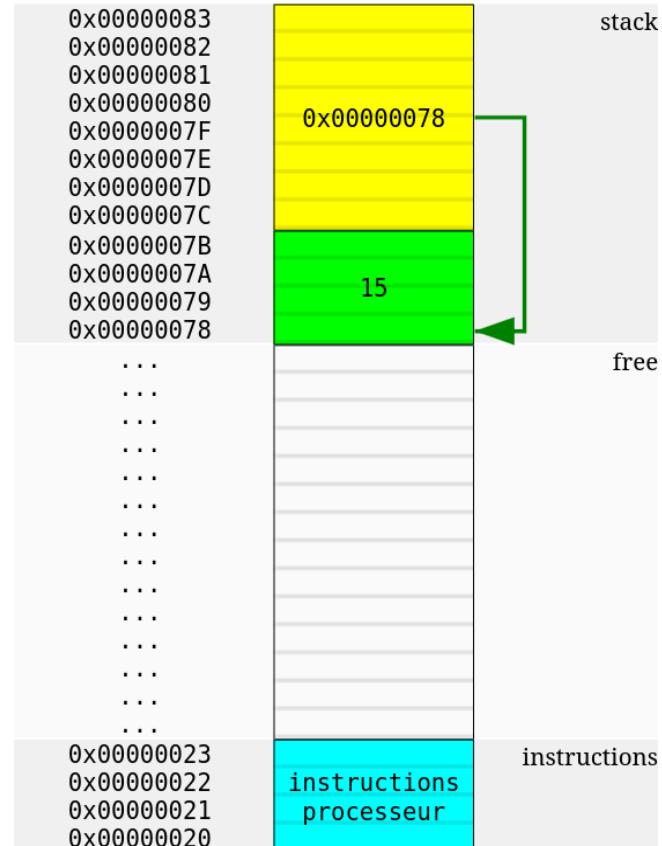


Les pointeurs et fonctions

```
examples > C Fonction_pointeur.c > main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void add_ten(int *orange){
5      *orange += 10;
6  }
7  int main(void) {
8      int green = 5;
9      int *yellow = &green;
10     add_ten(yellow);
11     printf("green : %d\n", green);
12     return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./fonction_pointeur



Pointeurs & Structures



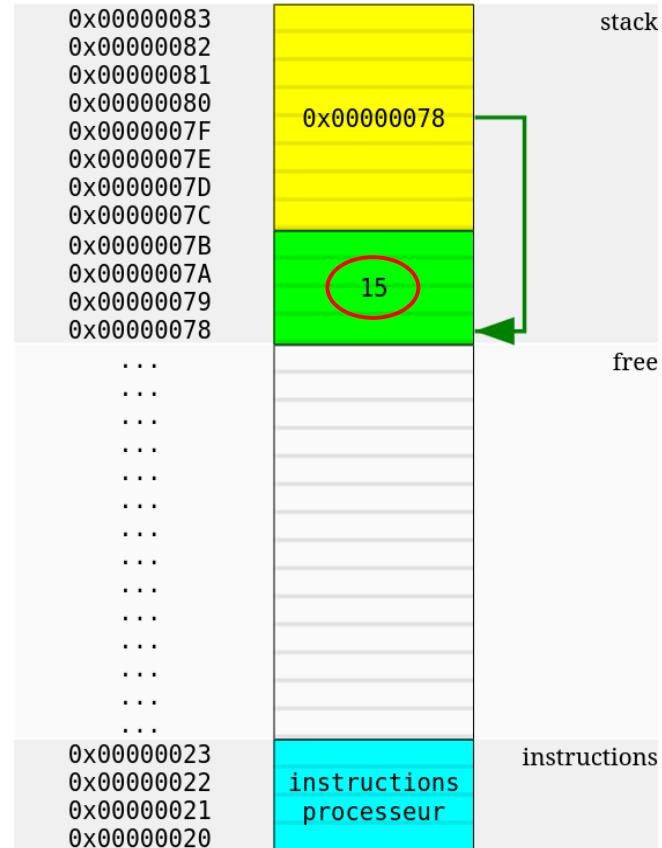
Les pointeurs et fonctions

```
examples > C Fonction_pointeur.c > main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void add_ten(int *orange){
5      *orange += 10;
6  }
7  int main(void) {
8      int green = 5;
9      int *yellow = &green;
10     add_ten(yellow);
11     printf("green : %d\n", green);
12     return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ ./fonction_pointeur
green : 15

La valeur en dehors de la fonction a bien été modifiée



Pointeurs & Structures



Dégradation (decay) en pointeur :

- Les tableaux passés en arguments de fonctions ne sont pas copiés dans la frame
- On observe à la place une conversion implicite en pointeur
- Ce comportement est très "pratique" car il n'est pas souhaitable de copier toutes les données dans les frames

Pointeurs & Structures



```
c degradation.c > ...
1 #include<stdio.h>
2
3 void print(int array[], int longueur) {
4     printf("sizeof(array) donne : %ld, valeurs : ", sizeof(array));
5     for (int i = 0; i < longueur; i++) {
6         printf("%d ", array[i]);
7     }
8     printf("\n");
9 }
10
11 int main(void) {
12     int array_1[] = {1, 2, 3};
13     int array_2[] = {83, 13, 29, 34};
14     printf("sizeof(array_1) : %ld\n", sizeof(array_1));
15     printf("sizeof(array_2) : %ld\n", sizeof(array_2));
16     print(array_1, 3);
17     print(array_2, 4);
18     return 0;
19 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ gcc degradation.c -o degradation -Wno-sizeof-array-argument
- (base) → examples git:(main) ✘ ./degradation

```
sizeof(array_1) : 12
sizeof(array_2) : 16
sizeof(array) donne : 8, valeurs : 1 2 3
sizeof(array) donne : 8, valeurs : 83 13 29 34
```

- (base) → examples git:(main) ✘

La fonction accepte
bien des tableaux
de tailles différentes

Pointeurs & Structures



```
c degradation.c > ...
1 #include<stdio.h>
2
3 void print(int array[], int longueur) {
4     printf("sizeof(array) donne : %ld, valeurs : ", sizeof(array));
5     for (int i = 0; i < longueur; i++) {
6         printf("%d ", array[i]);
7     }
8     printf("\n");
9 }
10
11 int main(void) {
12     int array_1[] = {1, 2, 3};
13     int array_2[] = {83, 13, 29, 34};
14     printf("sizeof(array_1) : %ld\n", sizeof(array_1));
15     printf("sizeof(array_2) : %ld\n", sizeof(array_2));
16     print(array_1, 3);
17     print(array_2, 4);
18     return 0;
19 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ gcc degradation.c -o degradation -Wno-sizeof-array-argument
- (base) → examples git:(main) ✘ ./degradation

```
sizeof(array_1) : 12
sizeof(array_2) : 16
sizeof(array) donne : 8, valeurs : 1 2 3
sizeof(array) donne : 8, valeurs : 83 13 29 34

```

Avant l'appel, **sizeof** donne bien la taille en mémoire du tableau

Pointeurs & Structures



```
c degradation.c > ...
1 #include<stdio.h>
2
3 void print(int array[], int longueur) {
4     printf("sizeof(array) donne : %ld, valeurs : ", sizeof(array));
5     for (int i = 0; i < longueur; i++) {
6         printf("%d ", array[i]);
7     }
8     printf("\n");
9 }
10
11 int main(void) {
12     int array_1[] = {1, 2, 3};
13     int array_2[] = {83, 13, 29, 34};
14     printf("sizeof(array_1) : %ld\n", sizeof(array_1));
15     printf("sizeof(array_2) : %ld\n", sizeof(array_2));
16     print(array_1, 3);
17     print(array_2, 4);
18     return 0;
19 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ gcc degradation.c -o degradation -Wno-sizeof-array-argument
- (base) → examples git:(main) ✘ ./degradation

```
sizeof(array_1) : 12
sizeof(array_2) : 16
sizeof(array) donne : 8, valeurs : 1 2 3
sizeof(array) donne : 8, valeurs : 83 13 29 34
```

- (base) → examples git:(main) ✘

Dans la fonction, le tableau est dégradé en pointeur, et une adresse est ici codée sur 8 octets

Pointeurs & Structures



```
c degradation.c > ...
1 #include<stdio.h>
2
3 void print(int array[], int longueur) {
4     printf("sizeof(array) donne : %ld, valeurs : ", sizeof(array));
5     for (int i = 0; i < longueur; i++) {
6         printf("%d ", array[i]);
7     }
8     printf("\n");
9 }
10
11 int main(void) {
12     int array_1[] = {1, 2, 3};
13     int array_2[] = {83, 13, 29, 34};
14     printf("sizeof(array_1) : %ld\n", sizeof(array_1));
15     printf("sizeof(array_2) : %ld\n", sizeof(array_2));
16     print(array_1, 3);
17     print(array_2, 4);
18     return 0;
19 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ gcc degradation.c -o degradation -Wno-sizeof-array-argument
● (base) → examples git:(main) ✘ ./degradation
sizeof(array_1) : 12
sizeof(array_2) : 16
sizeof(array) donne : 8, valeurs : 1 2 3
sizeof(array) donne : 8, valeurs : 83 13 29 34
○ (base) → examples git:(main) ✘
```

Après dégradation, ce genre de choses n'est plus possible* :

sizeof(array_1) /
sizeof(int)

La longueur du tableau doit être passée en argument

* : à éviter même sans dégradation !

Pointeurs & Structures



```
c degradation.c > ...
1 #include<stdio.h>
2
3 void print(int array[], int longueur) {
4     printf("sizeof(array) donne : %ld, valeurs : ", sizeof(array));
5     for (int i = 0; i < longueur; i++) {
6         printf("%d ", array[i]);
7     }
8     printf("\n");
9 }
10
11 int main(void) {
12     int array_1[] = {1, 2, 3};
13     int array_2[] = {83, 13, 29, 34};
14     printf("sizeof(array_1) : %ld\n", sizeof(array_1));
15     printf("sizeof(array_2) : %ld\n", sizeof(array_2));
16     print(array_1, 3);
17     print(array_2, 4);
18     return 0;
19 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ gcc degradation.c -o degradation -Wno-sizeof-array-argument
● (base) → examples git:(main) ✘ ./degradation
sizeof(array_1) : 12
sizeof(array_2) : 16
sizeof(array) donne : 8, valeurs : 1 2 3
sizeof(array) donne : 8, valeurs : 83 13 29 34
○ (base) → examples git:(main) ✘
```

La dégradation n'affecte pas l'utilisation du tableau, les crochets fonctionnent sur des pointeurs

On peut maintenant voir les crochets (tab[x] ou ptr[x]) comme : "donne moi la donnée à mon adresse avec un décalage de x fois la taille de mon type"

On parle bien de décalage, et pas d'incrémentation un x négatif est possible

Pointeurs & Structures



Dégradation (decay) en pointeur :

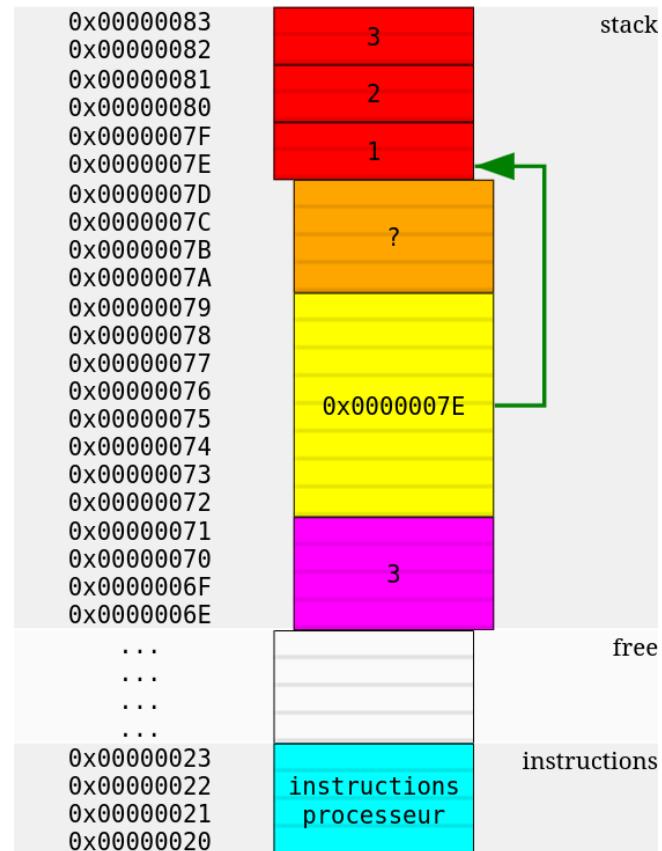
- En orange la zone pour "i"
- Et le layout en mémoire juste après la création de la frame et la copie en mémoire des arguments

```
C array_fonction.c > ...
1 #include<stdio.h>
2
3 void print(short *array, int longueur) {
4     for (int i = 0; i < longueur; i++) {
5         printf("%d ", array[i]);
6     }
7     printf("\n");
8 }
9
10 int main(void) {
11     short array_1[] = {1, 2, 3};
12     print(array_1, 3);
13     return 0;
14 }
```

"int array[]" est du sucre syntaxique pour "int *array"

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ gcc array_fonction.c -o array_fonction
- (base) → examples git:(main) ✘ ./array_fonction
- 1 2 3
- (base) → examples git:(main) ✘



Pointeurs & Structures



```
c tab2d.c > main(void)
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(void) {
5     short tab2d[2][4];
6     printf("Adresse de tab2d: %p \n", &tab2d);
7     printf("Adresse de tab2d[0][0]: %p \n", &tab2d[0][0]);
8     printf("Adresse de tab2d[1][0]: %p \n", &tab2d[1][0]);
9     // Ecart entre ces deux adresses
10    unsigned long ecart = &tab2d[1][0] - &tab2d[0][0];
11    printf("Ecart entre les 2 adresses (raw): %lu\n", ecart);
12    printf("sizeof(tab2d): %ld\n", sizeof(tab2d));
13    printf("sizeof(tab2d[0]): %ld\n", sizeof(tab2d[0]));
14    return EXIT_SUCCESS;
15 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples git:(main)* gcc tab2d.c -o tab2d
(base) → examples git:(main)* ./tab2d
Adresse de tab2d: 0x7ffdab31c210
Adresse de tab2d[0][0]: 0x7ffdab31c210
Adresse de tab2d[1][0]: 0x7ffdab31c218
```

Une adresse est un numéro d'octet, on voit bien un écart de 8 octet



Arithmétique des pointeurs

Bien qu'un pointeur soit un simple entier (une valeur d'adresse), la notion de pointeur est intimement liée à son type et donc, la taille de l'objet pointé

De retour sur les tableaux 2D, on peut essayer de déduire le layout mémoire en affichant les adresses, et calculer un écart par soustraction



Pointeurs & Structures



```
c tab2d.c > main(void)
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(void) {
5     short tab2d[2][4];
6     printf("Adresse de tab2d: %p \n", &tab2d);
7     printf("Adresse de tab2d[0][0]: %p \n", &tab2d[0][0]);
8     printf("Adresse de tab2d[1][0]: %p \n", &tab2d[1][0]);
9     // Ecart entre ces deux adresses
10    unsigned long ecart = &tab2d[1][0] - &tab2d[0][0];
11    printf("Ecart entre les 2 adresses (raw): %lu\n", ecart);
12    printf("sizeof(tab2d): %ld\n", sizeof(tab2d));
13    printf("sizeof(tab2d[0]): %ld\n", sizeof(tab2d[0]));
14    return EXIT_SUCCESS;
15 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples git:(main) ✘ gcc tab2d.c -o tab2d
(base) → examples git:(main) ✘ ./tab2d
Adresse de tab2d: 0x7ffdab31c210
Adresse de tab2d[0][0]: 0x7ffdab31c210
Adresse de tab2d[1][0]: 0x7ffdab31c218
Ecart entre les 2 adresses (raw): 4)
sizeof(tab2d): 16
sizeof(tab2d[0]): 8
(base) → examples git:(main) ✘
```

What ? Le nombre de cases et pas le nombre d'octets ?



Arithmétique des pointeurs

Bien qu'un pointeur soit un simple entier (une valeur d'adresse), la notion de pointeur est intimement liée à son type et donc, la taille de l'objet pointé

De retour sur les tableaux 2D, on peut essayer de déduire le layout mémoire en affichant les adresses, et calculer un écart par soustraction



Pointeurs & Structures



```
C tab2d.c > ...
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(void) {
5     short tab2d[2][4];
6     printf("Adresse de tab2d: %p \n", &tab2d);
7     printf("Adresse de tab2d[0][0]: %p \n", &tab2d[0][0]);
8     printf("Adresse de tab2d[1][0]: %p \n", &tab2d[1][0]);
9     // Ecart entre ces deux adresses
10    unsigned long ecart = (unsigned long)&tab2d[1][0]
11        - (unsigned long)&tab2d[0][0];
12    printf("Ecart entre les 2 adresses (raw): %lu\n", ecart);
13    printf("sizeof(tab2d): %ld\n", sizeof(tab2d));
14    printf("sizeof(tab2d[0]): %ld\n", sizeof(tab2d[0]));
15    return EXIT_SUCCESS;
16 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples git:(main) ✘ gcc tab2d.c -o tab2d
(base) → examples git:(main) ✘ ./tab2d
Adresse de tab2d: 0x7ffcb2b3d050
Adresse de tab2d[0][0]: 0x7ffcb2b3d050
Adresse de tab2d[1][0]: 0x7ffcb2b3d058
Ecart entre les 2 adresses (raw): 8)
sizeof(tab2d): 16
sizeof(tab2d[0]): 8
(base) → examples git:(main) ✘
```

Arithmétique des pointeurs

On est bien sur un comportement propre au pointeurs : on retrouve notre écart de **8** octets entre les deux adresses si on converti le type pointeur **short *** vers un type entier **unsigned long**

Rappel : les entiers longs et les pointeurs sont sur 8 octets (sur cette machine) et une adresse est positive

Pointeurs & Structures



```
C array_pointeur.c > ...
1  #include<stdio.h>
2
3  void print_1(short array[], int longueur) {
4      for (int i = 0; i < longueur; i++) {
5          printf("%d ", array[i]);
6      }
7      printf("\n");
8  }
9
10 void print_2(short *array, int longueur) {
11     short *end = array + longueur;
12     for ( ; array < end; array++) {
13         printf("%d ", *array);
14     }
15     printf("\n");
16 }
17
18 int main(void) {
19     short array_1[] = {1, 2, 3};
20     print_1(array_1, 3);
21     print_2(array_1, 3);
22     return 0;
23 }
```

"end" vaut la valeur de
array + longueur fois
la taille d'un short (et
non d'un int)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ gcc array_pointeur.c -o array_pointeur
● (base) → examples git:(main) ✘ ./array_pointeur
1 2 3
1 2 3
○ (base) → examples git:(main) ✘
```

Arithmétique des pointeurs

Comme on travaille directement avec le pointeur, plus besoin de la variable i,
mais il nous faut une adresse de fin !

Pointeurs & Structures



```
C array_pointeur.c > ...
1  #include<stdio.h>
2
3  void print_1(short array[], int longueur) {
4      for (int i = 0; i < longueur; i++) {
5          printf("%d ", array[i]);
6      }
7      printf("\n");
8  }
9
10 void print_2(short *array, int longueur) {
11     short *end = array + longueur;
12     for ( ; array < end; array++) {
13         printf("%d ", *array);
14     }
15     printf("\n");
16 }
17
18 int main(void) {
19     short array_1[] = {1, 2, 3};
20     print_1(array_1, 3);
21     print_2(array_1, 3);
22     return 0;
23 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ gcc array_pointeur.c -o array_pointeur
● (base) → examples git:(main) ✘ ./array_pointeur
1 2 3
1 2 3
○ (base) → examples git:(main) ✘
```

Arithmétique des pointeurs

Comme on travaille directement avec le pointeur, plus besoin de la variable **i**,
mais il nous faut une adresse de fin !

Comme la dégradation en pointeur
donne une adresse pointant vers la
première case, **plus besoin**
d'initialisation

Pointeurs & Structures



```
C array_pointeur.c > ...
1 #include<stdio.h>
2
3 void print_1(short array[], int longueur) {
4     for (int i = 0; i < longueur; i++) {
5         printf("%d ", array[i]);
6     }
7     printf("\n");
8 }
9
10 void print_2(short *array, int longueur) {
11     short *end = array + longueur;
12     for ( ; array < end, array++) {
13         printf("%d ", *array);
14     }
15     printf("\n");
16 }
17
18 int main(void) {
19     short array_1[] = {1, 2, 3};
20     print_1(array_1, 3);
21     print_2(array_1, 3);
22     return 0;
23 }
```

L'adresse **array** pointerà une case plus loin (donc son adresse + la taille d'un short)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ gcc array_pointeur.c -o array_pointeur
● (base) → examples git:(main) ✘ ./array_pointeur
1 2 3
1 2 3
○ (base) → examples git:(main) ✘
```

Arithmétique des pointeurs

Comme on travaille directement avec le pointeur, plus besoin de la variable **i**,
mais il nous faut une adresse de fin !

Comme la dégradation en pointeur donne une adresse pointant vers la première case, **plus besoin d'initialisation**

Pour passer à la valeur suivante, nous n'avons qu'à **incrémenter le pointeur**

Et dans la boucle, **la syntaxe classique d'accès à la valeur**

Pointeurs & Structures



```
C array_pointeur.c > ...
1 #include<stdio.h>
2
3 void print_1(short array[], int longueur) {
4     for (int i = 0; i < longueur; i++) {
5         printf("%d ", array[i]);
6     }
7     printf("\n");
8 }
9
10 void print_2(short *array, int longueur) {
11     short *end = array + longueur;
12     for ( ; array < end; array++) {
13         printf("%d ", *array);
14     }
15     printf("\n");
16 }
17
18 int main(void) {
19     short array_1[] = {1, 2, 3};
20     print_1(array_1, 3);
21     print_2(array_1, 3);
22     return 0;
23 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ gcc array_pointeur.c -o array_pointeur
● (base) → examples git:(main) ✘ ./array_pointeur
1 2 3
1 2 3
○ (base) → examples git:(main) ✘
```

Arithmétique des pointeurs

Comme on travaille directement avec le pointeur, plus besoin de la variable **i**, **mais il nous faut une adresse de fin !**

Comme la dégradation en pointeur donne une adresse pointant vers la première case, **plus besoin d'initialisation**

Pour passer à la valeur suivante, nous n'avons qu'à **incrémenter le pointeur**

Et dans la boucle, **la syntaxe classique d'accès à la valeur**

Bien sûr, les arguments "**array**" sont des copies de "**array_1**" (pointeur par dégradation) propres aux frames des fonctions "**print_1**" et "**print_2**"

Pointeurs & Structures



```
C array_pointeur.c > ...
1  #include<stdio.h>
2
3  void print_1(short array[], int longueur) {
4      for (int i = 0; i < longueur; i++) {
5          printf("%d ", array[i]);
6      }
7      printf("\n");
8  }
9
10 void print_2(short *array, int longueur) {
11     short *end = array + longueur;
12     for ( ; array < end; array++) {
13         printf("%d ", *array);
14     }
15     printf("\n");
16 }
17
18 int main(void) {
19     short array_1[] = {1, 2, 3};
20     print_1(array_1, 3);
21     print_2(array_1, 3);
22     return 0;
23 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ gcc array_pointeur.c -o array_pointeur
● (base) → examples git:(main) ✘ ./array_pointeur
1 2 3
1 2 3
○ (base) → examples git:(main) ✘
```

Arithmétique des pointeurs

L'**index 0** pour la première valeur d'un tableau prend désormais tout son sens

La où certains langages choisissent **1** pour le premier index (e.g. Lua, Julia, MATLAB), le langage C est plus **proche de la logique du compilateur**, car un tableau peut être traité comme un pointeur par dégradation (automatique avec un appel de fonction)

Et effectivement, le tableau devient une adresse vers le premier élément et l'accès à un élément se fait avec **adresse + index x taille d'un élément**, et ce, gratuitement avec l'arithmétique des pointeurs

Pointeurs & Structures



Grouper des données :

- Pour de la **programmation plus haut niveau**, les types primitifs ne suffisent plus. Traiter, modifier, copier, et passer en argument les **différentes caractéristiques d'un concept** comme un tout plutôt d'individuellement peut être enviable.
- Une structure permet donc de définir un groupement de données (types primitifs ou autres structures). Elle se déclare de la manière suivante :
 - 1) Le mot clef "**struct**"
 - 2) Son **nom**
 - 3) Un **bloc** contenant des déclarations de variables : ses **membres**, appelés aussi **champs** ou **attributs**
 - 4) Un **point virgule**
- Une fois déclarée, "**struct <nom>**" peut être utilisé comme un type.
- Une structure obéit aux mêmes règles que les types primitifs concernant la mémoire, e.g. **copies des valeurs dans les frames pour les appels de fonctions**, mais **contrairement aux tableaux**, les valeurs des membres ne sont **pas forcément contiguës en mémoire** (pour des raisons architecturales, de performance)

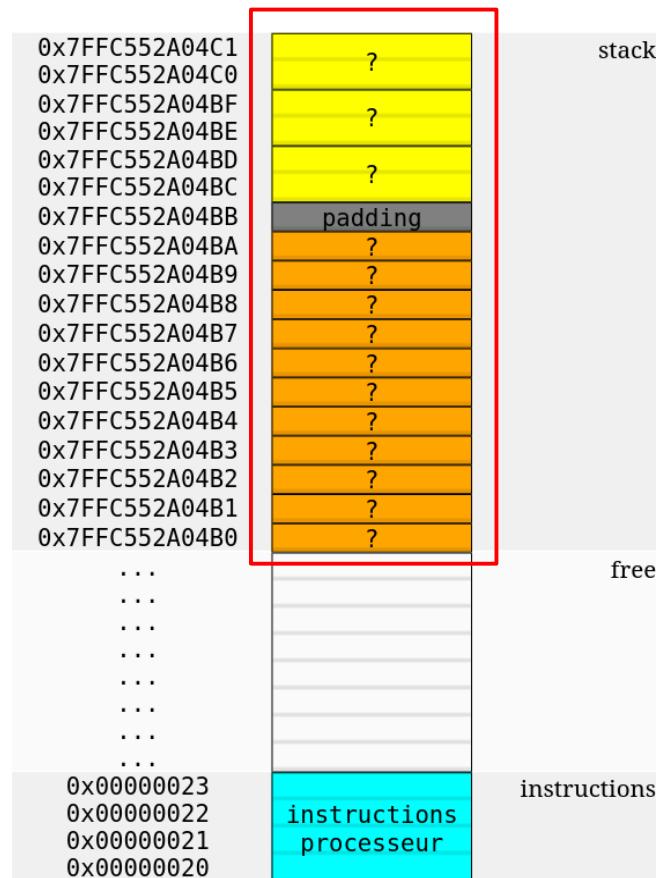
Pointeurs & Structures



Grouper des données :

- Avec l'appel à la fonction main, une **frame** est créée avec une zone mémoire pour notre variable **p** ayant un layout mémoire correspondant à celui déclaré par **struct personne**
- Une **struct personne** contient également une **struct date**

```
examples > C structure.c > ...
1  struct date {
2    short jour;
3    short mois;
4    short annee;
5  };
6
7  struct personne {
8    char prenom[11];
9    struct date naissance;
10 };
11
12 int main() {
13   struct personne p;
14   return 0;
15 }
```



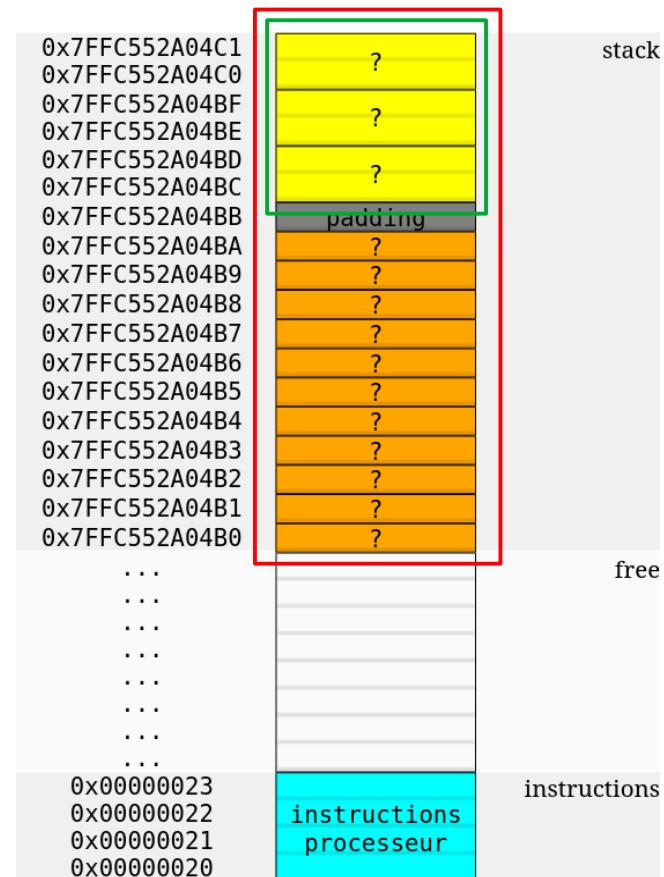
Pointeurs & Structures



Grouper des données :

- Si une structure contient une autre structure, les données sont juste **à la suite en mémoire**
- Dans cet exemple, le compilateur ajoute un octet de **padding** pour avoir le prochain membre, ici **naissance** (une **struct date**), sur une adresse "qui l'arrange".

```
examples > C structure.c > ...
1  struct date {
2      short jour;
3      short mois;
4      short annee;
5  };
6
7  struct personne {
8      char prenom[11];
9      struct date naissance;
10 };
11
12 int main() {
13     struct personne p;
14     return 0;
15 }
```



Pointeurs & Structures



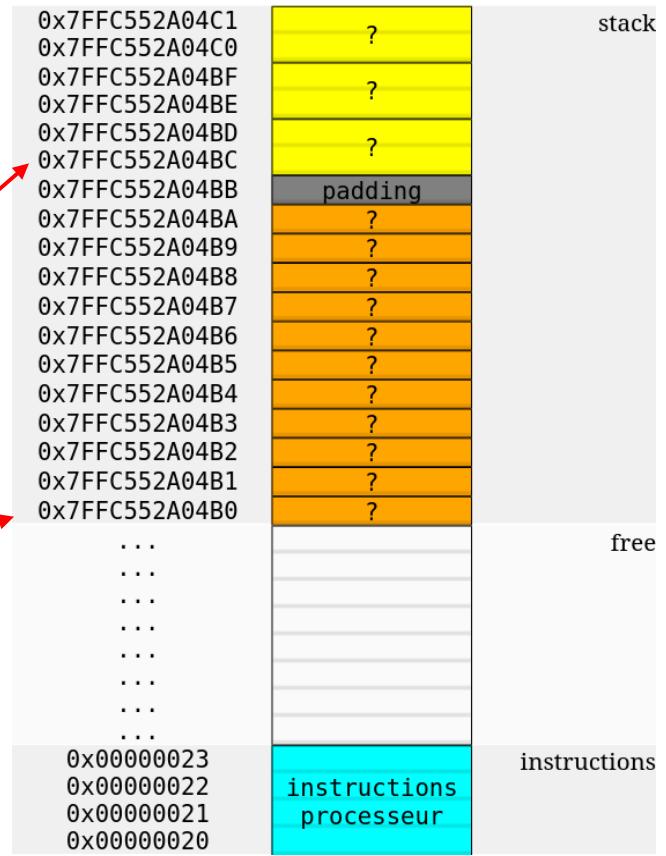
Grouper des données :

- La structure est effectivement **un octet plus grande**
- L'emplacement du padding est bien avant "naissance"

```
examples > c structure.c > main()
1 #include<stdio.h>
2
3 struct date {
4     short jour;
5     short mois;
6     short annee;
7 };
8
9 struct personne {
10    char prenom[11];
11    struct date naissance;
12 };
13
14 int main() {
15    struct personne p;
16    printf("size : %ld\n", sizeof(p));
17    printf("addr p %p\n", &p);
18    printf("addr p.prenom %p\n", &p.prenom);
19    printf("addr p.naissance %p\n", &p.naissance);
20    return 0;
21 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples git:(main) ✘ ./structure
size : 18
addr p 0x7ffc552a04b0
addr p.prenom 0x7ffc552a04b0
addr p.naissance 0x7ffc552a04bc
(base) → examples git:(main) ✘
```



Pointeurs & Structures



Un autre sucre syntaxique lié aux pointeurs :

- Dans l'exemple suivant, on va ajouter membre à membre les valeurs d'une structure dans une autre structure
- On a besoin d'accéder à la mémoire des valeurs de la structure à modifier, et juste les valeurs de la seconde
- Pour modifier une structure avec son adresse, il faut déréférencer, puis accéder aux membres. Des parenthèses sont donc nécessaires, il faudrait écrire :

(*struct_ptr).val = 42

- Le langage C propose une syntaxe alternative plus pratique avec "->", directement applicable à une adresse vers une structure. La flèche opère le déréférencement :

struct_ptr->val = 42

Pointeurs & Structures

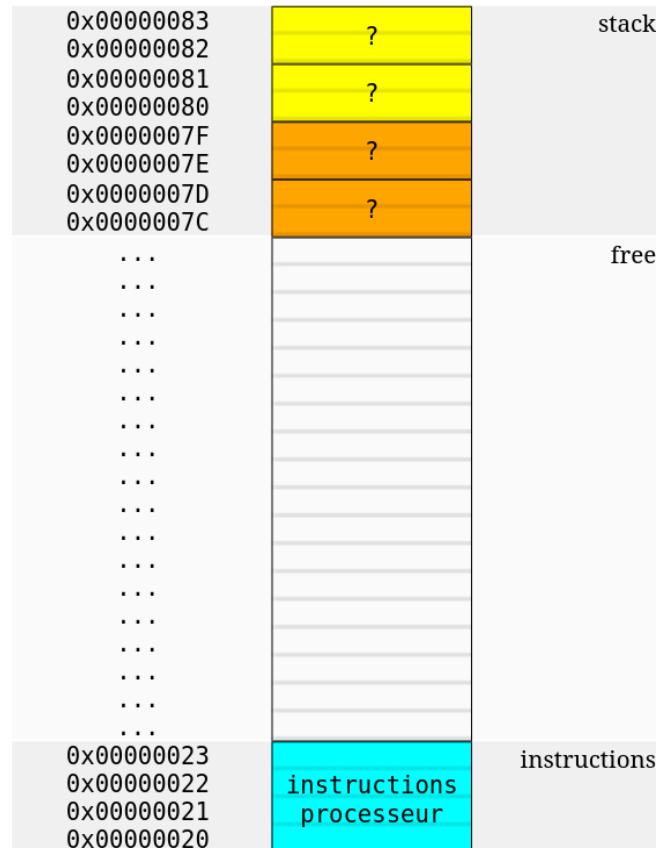


Un autre succèe syntaxique lié aux pointeurs :

```
C struct_ptr.c x
examples > C struct_ptr.c > data > b
1 #include <stdio.h>
2
3 struct data {
4     short a;
5     short b;
6 };
7
8 void add_into(struct data *dest, struct data with){
9     (*dest).a += with.a;
10    dest->b += with.b; // préférable
11 }
12
13 int main(void) {
14     struct data orange = {-4, 5};
15     struct data yellow = {20, -8};
16     add_into(&orange, yellow);
17     printf("a: %d, b: %d\n", orange.a, orange.b);
18     return 0;
19 }
20
```

PROBLEMS OUTPUT TERMINAL ... zsh - examples + -

(base) → examples git:(main) ✘ ./struct_ptr



Pointeurs & Structures

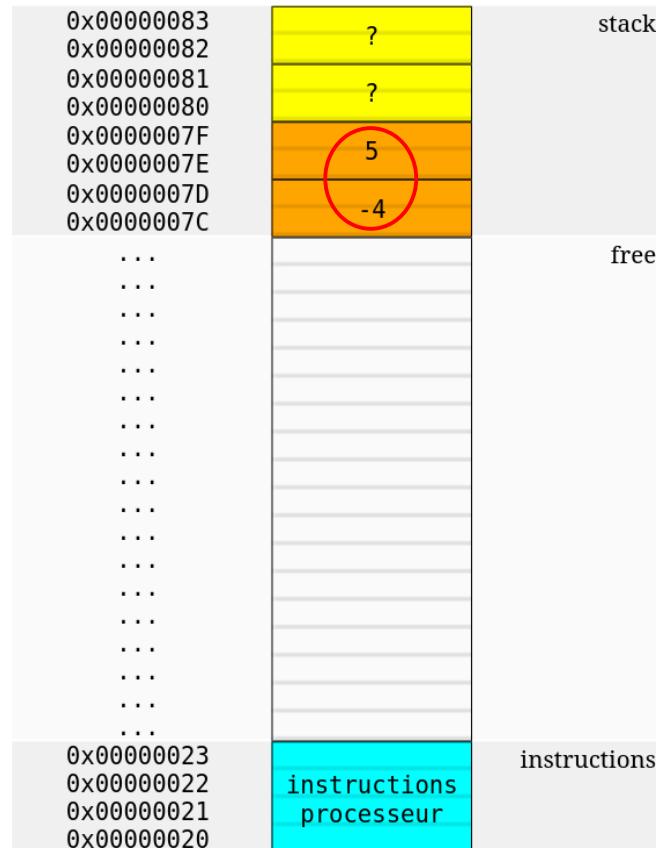


Un autre successeur syntaxique lié aux pointeurs :

A screenshot of a terminal window showing a C program named `struct_ptr.c`. The code defines a `struct data` with `a` and `b` fields, and a `void add_into` function that adds the `b` field of one struct to another. In the `main` function, two `struct data` objects are created: `orange` with values `{-4, 5}` and `yellow` with values `{20, -8}`. The `add_into` function is called with `&orange` and `yellow` as arguments. The `printf` statement at the end prints the values of `orange.a` and `orange.b`. A red arrow points to the line where `orange` is defined.

```
C struct_ptr.c x
examples > C struct_ptr.c > data > b
1  #include <stdio.h>
2
3  struct data {
4      short a;
5      short b;
6  };
7
8  void add_into(struct data *dest, struct data with){
9      (*dest).a += with.a;
10     dest->b += with.b; // préférable
11 }
12
13 int main(void) {
14     struct data orange = {-4, 5};
15     struct data yellow = {20, -8};
16     add_into(&orange, yellow);
17     printf("a: %d, b: %d\n", orange.a, orange.b);
18     return 0;
19 }
20

PROBLEMS    OUTPUT    TERMINAL ...      zsh - examples + - x
(base) → examples git:(main) ✘ ./struct_ptr
```



Pointeurs & Structures



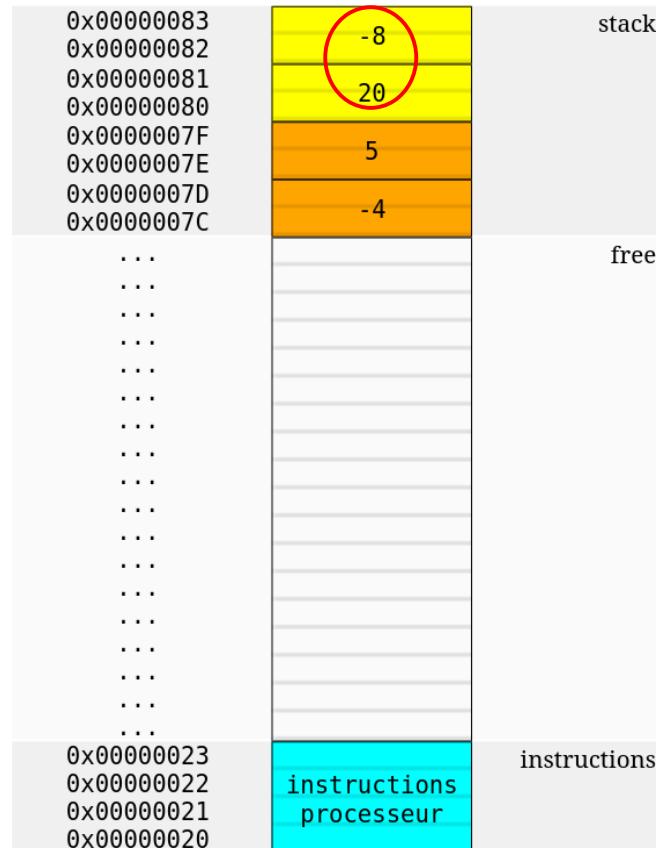
Un autre successeur syntaxique lié aux pointeurs :

```
C struct_ptr.c x
examples > C struct_ptr.c > data > b
1  #include <stdio.h>
2
3  struct data {
4      short a;
5      short b;
6  };
7
8  void add_into(struct data *dest, struct data with){
9      (*dest).a += with.a;
10     dest->b += with.b; // préférable
11 }
12
13 int main(void) {
14     struct data orange = {-4, 5};
15     struct data yellow = {20, -8};
16     add_into(&orange, yellow);
17     printf("a: %d, b: %d\n", orange.a, orange.b);
18     return 0;
19 }
20
```

The code shows a C program with a struct definition, a function add_into that adds two structures, and a main function that creates two structures, calls add_into on them, and prints the result. A red arrow points to the line 'add_into(&orange, yellow);'.

PROBLEMS OUTPUT TERMINAL ... zsh - examples + -

(base) → examples git:(main) ✘ ./struct_ptr



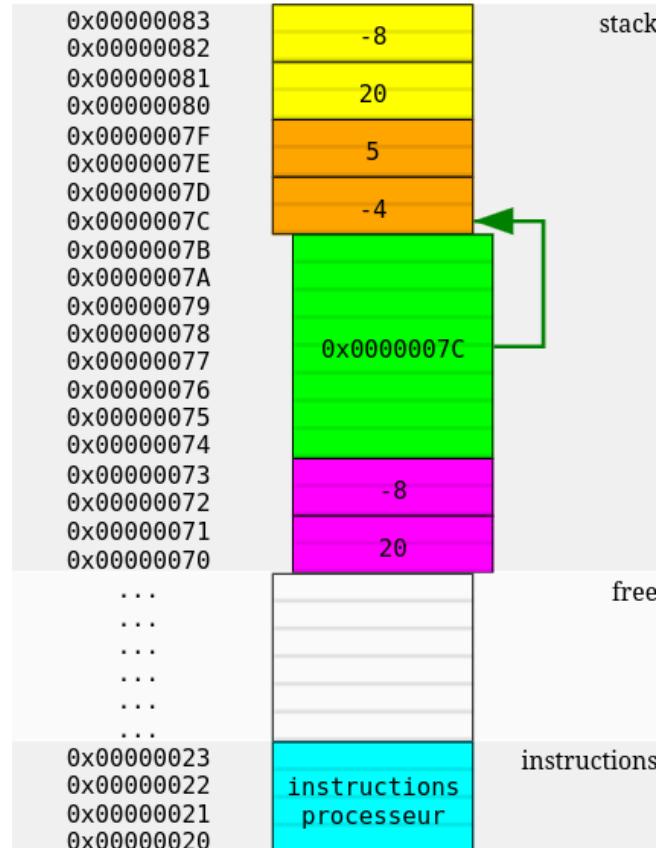
Pointeurs & Structures



Un autre successeur syntaxique lié aux pointeurs :

```
C struct_ptr.c x
examples > C struct_ptr.c > data > b
1  #include <stdio.h>
2
3  struct data {
4      short a;
5      short b;
6  };
7
8  void add_into(struct data *dest, struct data with){
9      (*dest).a += with.a;
10     dest->b += with.b; // préférable
11 }
12
13 int main(void) {
14     struct data orange = {-4, 5};
15     struct data yellow = {20, -8};
16     add_into(&orange, yellow);
17     printf("a: %d, b: %d\n", orange.a, orange.b);
18     return 0;
19 }
20

PROBLEMS    OUTPUT    TERMINAL    ...
zsh - examples + ▾ □ ▫
(base) → examples git:(main) ✘ ./struct_ptr
```



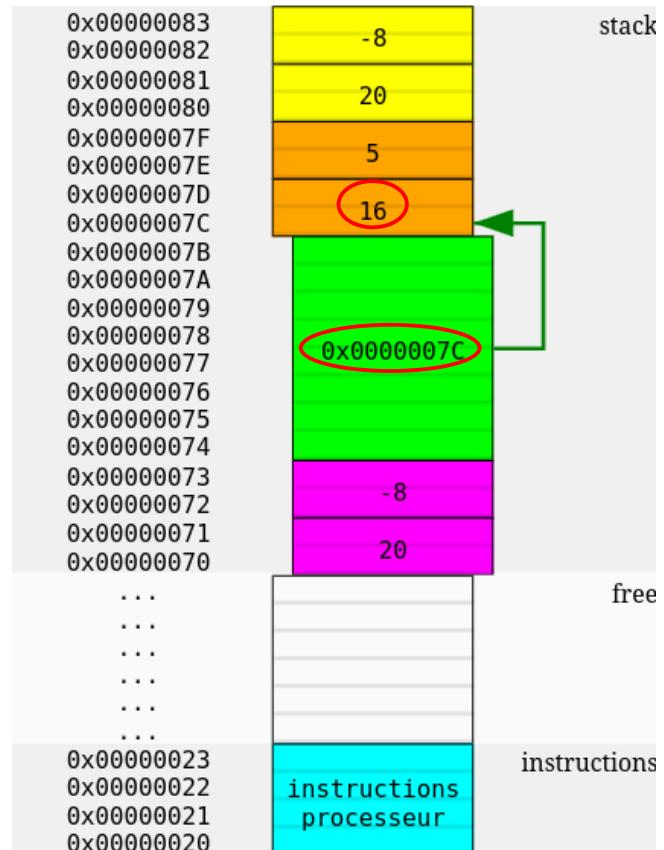
Pointeurs & Structures



Un autre succèe syntaxique lié aux pointeurs :

```
C struct_ptr.c x
examples > C struct_ptr.c > data > b
1  #include <stdio.h>
2
3  struct data {
4      short a;
5      short b;
6  };
7
8  void add_into(struct data *dest, struct data with){
9      (*dest).a += with.a;
10     dest->b += with.b; // préférable
11 }
12
13 int main(void) {
14     struct data orange = {-4, 5};
15     struct data yellow = {20, -8};
16     add_into(&orange, yellow);
17     printf("a: %d, b: %d\n", orange.a, orange.b);
18     return 0;
19 }
20

PROBLEMS    OUTPUT    TERMINAL ...      zsh - examples + ▾ □ ▫
(base) → examples git:(main) ✘ ./struct_ptr
```



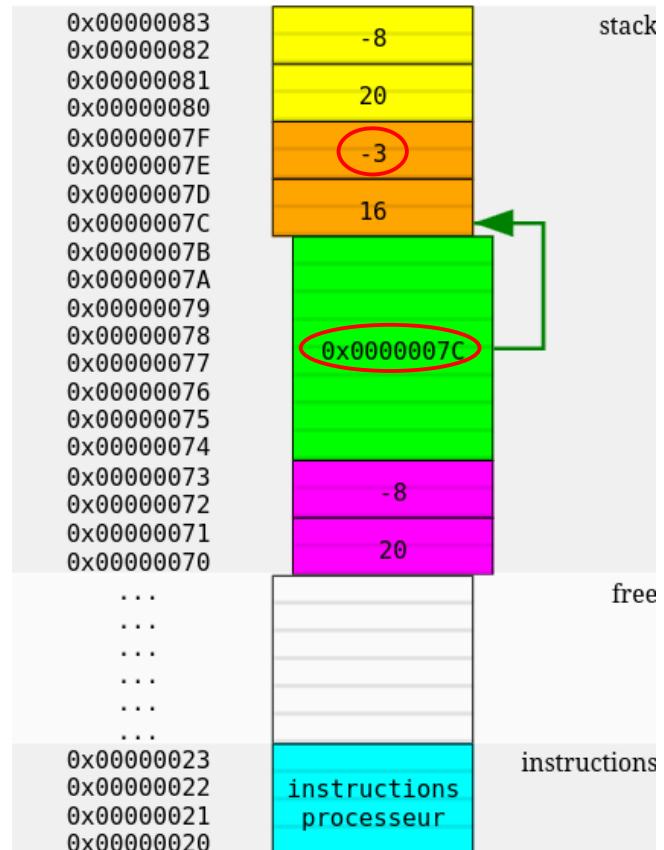
Pointeurs & Structures



Un autre successeur syntaxique lié aux pointeurs :

```
C struct_ptr.c x
examples > C struct_ptr.c > data > b
1  #include <stdio.h>
2
3  struct data {
4      short a;
5      short b;
6  };
7
8  void add_into(struct data *dest, struct data with){
9      (*dest).a += with.a;
10     dest->b += with.b; // préférable
11 }
12
13 int main(void) {
14     struct data orange = {-4, 5};
15     struct data yellow = {20, -8};
16     add_into(&orange, yellow);
17     printf("a: %d, b: %d\n", orange.a, orange.b);
18     return 0;
19 }
20

PROBLEMS    OUTPUT    TERMINAL    ...
zsh - examples + ▾ □ ▫
(base) → examples git:(main) ✘ ./struct_ptr
```



Pointeurs & Structures



Un autre succèe syntaxique lié aux pointeurs :

A screenshot of a terminal window showing a C program named `struct_ptr.c`. The code defines a `struct data` with two short integers `a` and `b`. It includes a `void add_into` function that adds the values of `a` and `b` from one struct to another. The `main` function creates two `struct data` objects, `orange` and `yellow`, initializes them, calls `add_into` to sum their values, and then prints the result. A red arrow points to the `printf` statement in the `main` function.

```
C struct_ptr.c x
examples > C struct_ptr.c > data > b
1 #include <stdio.h>
2
3 struct data {
4     short a;
5     short b;
6 };
7
8 void add_into(struct data *dest, struct data with){
9     (*dest).a += with.a;
10    dest->b += with.b; // préférable
11 }
12
13 int main(void) {
14     struct data orange = {-4, 5};
15     struct data yellow = {20, -8};
16     add_into(&orange, yellow);
17     printf("a: %d, b: %d\n", orange.a, orange.b);
18     return 0;
19 }
20

PROBLEMS OUTPUT TERMINAL ...
zsh - examples + -
```

(base) → examples git:(main) ✘ ./struct_ptr

0x00000083	-8	stack
0x00000082	20	
0x00000081	-3	
0x00000080	16	
0x0000007F		
0x0000007E		
0x0000007D		
0x0000007C		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
...		
0x00000023	instructions	
0x00000022		
0x00000021		
0x00000020		
	processeur	

Pointeurs & Structures

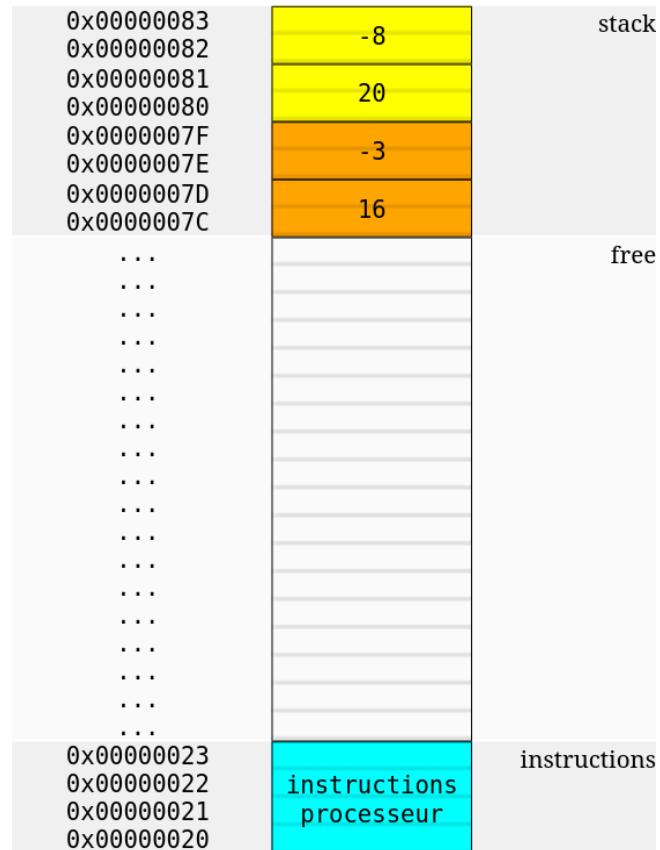


Un autre successeur syntaxique lié aux pointeurs :

A screenshot of a code editor showing a C program named `struct_ptr.c`. The code defines a `struct data` with two short integers `a` and `b`. It includes a `void add_into` function that adds the values of `a` and `b` from one struct to another. The `main` function creates two `struct data` objects, `orange` and `yellow`, initializes them, calls `add_into`, and then prints the values of `orange.a` and `orange.b`. The terminal output shows the result: `a: 16, b: -3`.

```
C struct_ptr.c x
examples > C struct_ptr.c > data > b
1 #include <stdio.h>
2
3 struct data {
4     short a;
5     short b;
6 };
7
8 void add_into(struct data *dest, struct data with){
9     (*dest).a += with.a;
10    dest->b += with.b; // préférable
11 }
12
13 int main(void) {
14     struct data orange = {-4, 5};
15     struct data yellow = {20, -8};
16     add_into(&orange, yellow);
17     printf("a: %d, b: %d\n", orange.a, orange.b);
18     return 0;
19 }
20

PROBLEMS OUTPUT TERMINAL ...
(base) ➜ examples git:(main) ✘ ./struct_ptr
a: 16, b: -3
(base) ➜ examples git:(main) ✘
```



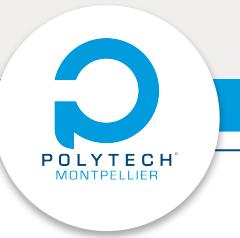
Pointeurs & Structures



L'opérateur d'assignation/affectation

On peut directement copier tous les membres d'une structure avec une assignation !

```
c struct_assign.c > ...
1  struct data {
2  |     short a;
3  |     short b;
4  };
5
6  // Beaucoup plus simple :
7  void example_1(struct data *dest, struct data src){
8  |     *dest = src;
9  }
10
11 // On peut assigner avec/sans (double) indirection
12 // On rappelle qu'une assignation est une simple copie
13
14 void example_2(struct data *dest, struct data *src){
15 |     *dest = *src;
16 }
17
18 void example_3(struct data dest, struct data *src){
19 |     dest = *src;
20 }
21
22 void example_4(struct data dest, struct data src){
23 |     dest = src;
24 }
25
26 void example_5(struct data src){
27 |     struct data tab[10];
28 |     tab[3] = src;
29 }
```



Allocation dynamique & Casting

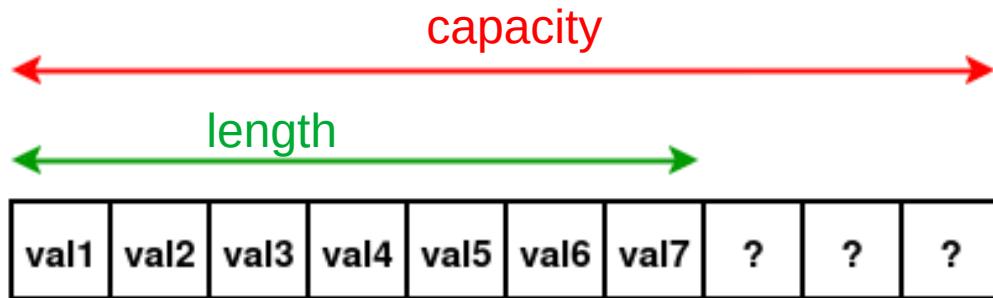
Allocation dynamique & Casting



Vocabulaire : Capacity and length

Si l'on souhaite utiliser un tableau comme une liste, on différencie :

- La taille du tableau (à l'initialisation), qu'on appellera **capacity**
- Le nombre d'éléments dans le tableau, qu'on appellera **length** (donc 0 à l'initialisation d'une liste vide)



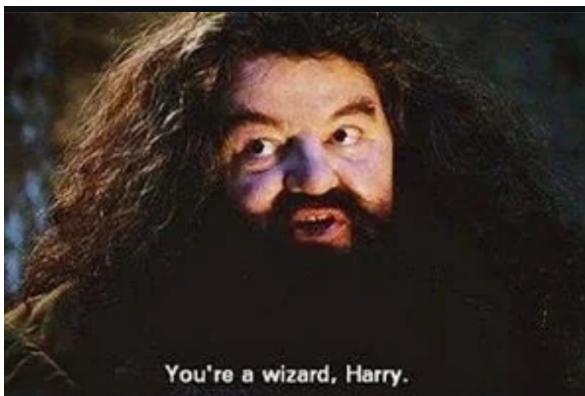
⚠ En français, éviter le mot "taille" pour lever les ambiguïtés est bienvenu dans ce contexte

Allocation dynamique & Casting



Une fois n'est pas coutume ..

- Nous avons ici un petit script **Python** qui stocke des chaînes de caractères dans une liste
- La capacité de la liste n'est pas renseignée, on peut donc ajouter des éléments à l'infini magiquement ? Est-ce que ce resize est systématique (à chaque ajout) ?



```
examples > ➜ list.py > ...
1  # a list
2  word_list = []
3  # make it grow..
4  while True:
5      word = input("Next word (or \"stop\"): ")
6      # ..until a given marker
7      if word == "stop":
8          break
9      word_list.append(word)
10 print(word_list)
11
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) ➜ examples git:(main) ✘ python list.py
Next word (or "stop"): C
Next word (or "stop"): is
Next word (or "stop"): better
Next word (or "stop"): stop
['C', 'is', 'better']
(base) ➜ examples git:(main) ✘
```

Allocation dynamique & Casting



Regardons de plus près

- La fonction "**list_capacity**" nous donne la **capacité** du tableau où sont stockées les données
- On observe que la **capacité** augmente quand elle devient insuffisante. Par exemple, la capacité passe de 4 à 8 quand on ajoute le 5ème élément
- La taille en mémoire est donc définie à chaque instant, mais est-ce que ce resize est automatique ? (non)

Il est temps de parler d'allocation dynamique

```
examples > ⚡ list_size.py > ...
1  import ctypes
2
3  def list_capacity(lst):
4      return ctypes.cast(id(lst), ctypes.POINTER(ctypes.c_size_t))[4]
5
6  lst = []
7  capacity = list_capacity(lst)
8  print(f"capacity: {capacity}, length: {len(lst)}")
9  for i in range(50):
10     lst.append(i)
11     new_capacity = list_capacity(lst)
12     if capacity != new_capacity:
13         capacity = new_capacity
14         print(f"capacity: {capacity}, length: {len(lst)}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

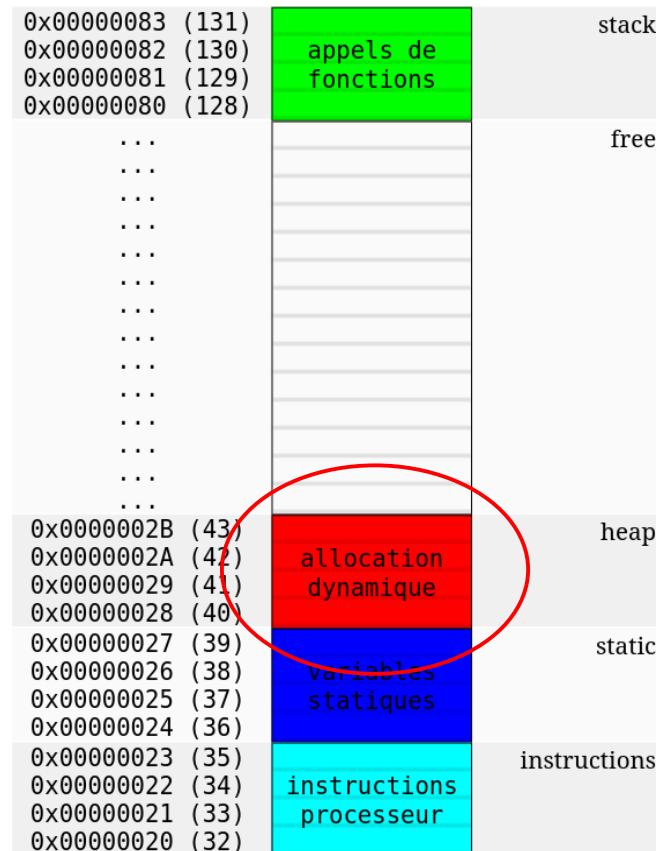
```
(base) → examples git:(main) ✘ python list_size.py
capacity: 0, length: 0
capacity: 4, length: 1
capacity: 8, length: 5
capacity: 16, length: 9
capacity: 24, length: 17
capacity: 32, length: 25
capacity: 40, length: 33
capacity: 52, length: 41
(base) → examples git:(main) ✘
```

Allocation dynamique & Casting



Qu'est-ce que l'allocation dynamique ?

- Il s'agit d'une méthode d'allocation de mémoire exposée par le système d'exploitation via des **syscalls**
- Cela permet, pendant l'exécution, de demander un **pointeur** vers une zone mémoire allouée dans le **tas (heap)**
- Le programme n'utilise pas directement les appels système (mmap, brk, sbrk), il passe par des fonctions intermédiaires comme **malloc, calloc, realloc, free**
- **Le programmeur reste responsable de l'allocation**, il doit :
 - **Vérifier** le retour de l'allocation : cela peut échouer
 - **Gérer** correctement les tailles allouées : e.g. capacité
 - **Réallouer** si nécessaire : c.f. l'exemple en Python
 - **Libérer** la mémoire : ce n'est pas automatique

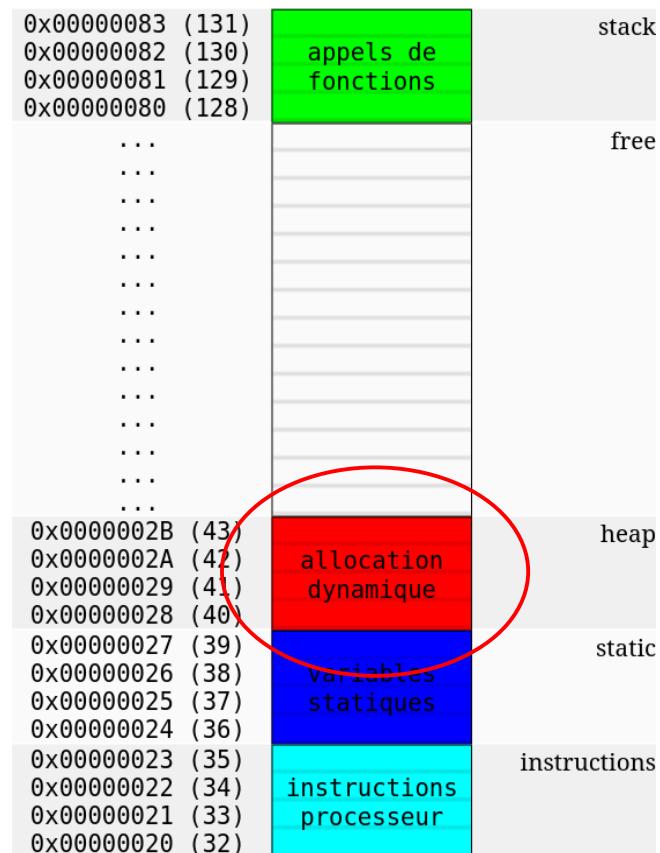


Allocation dynamique & Casting



Fonctions principales

- **malloc** : allocation "brute" de **n** d'octets. La zone est non initialisée
- **calloc** : allocation de **n** éléments de taille **s** chacun, la zone est initialisée avec des zéros
- **realloc** : redimensionne la zone pointée par **p pour** atteindre **n** octet. Possible copie de données, en cas d'agrandissement, la zone supplémentaire n'est pas initialisée.
- **free** : libérer la zone pointée par **p**, elle peut donc être réallouée ensuite par un malloc / calloc / realloc



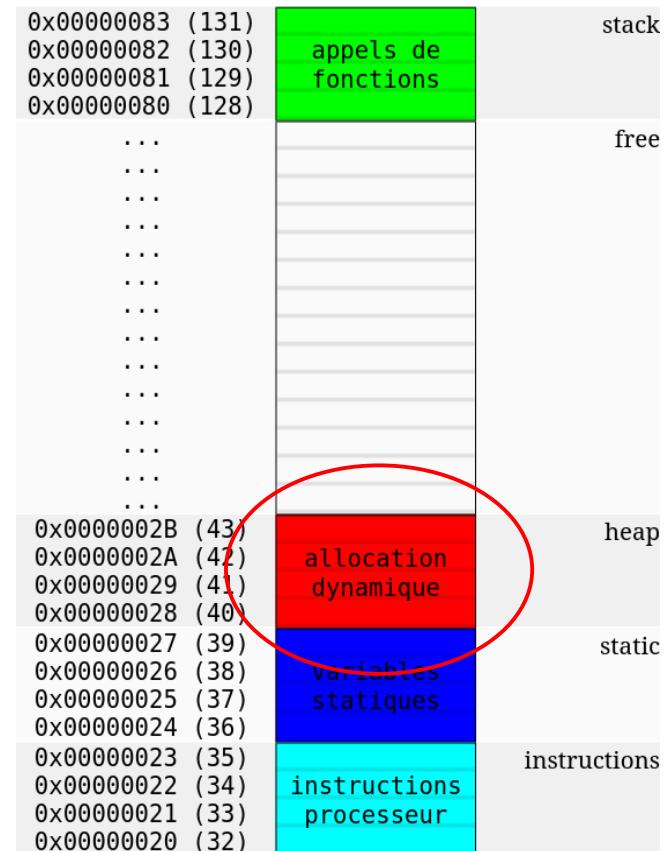
Allocation dynamique & Casting



Casting

Les fonctions **malloc** / **calloc** / **realloc** rendent un pointeur en cas de succès, le pointeur vaudra NULL (0) en cas d'erreur, le pointeur est de type **void *** :

- En C il n'est pas nécessaire de cast le pointeur de retour vers le type cible, la conversion est automatique (mais ce n'est pas le cas en C++)
- En fonction du compilateur, de la plateforme, des compatibilités recherchées et des imports, ajouter le cast peut être préférable (ou au contraire risqué)
- Dans le cours, on utilisera le cast uniquement pour rappeler la conversion implicite (il y a donc redondance)



Allocation dynamique & Casting

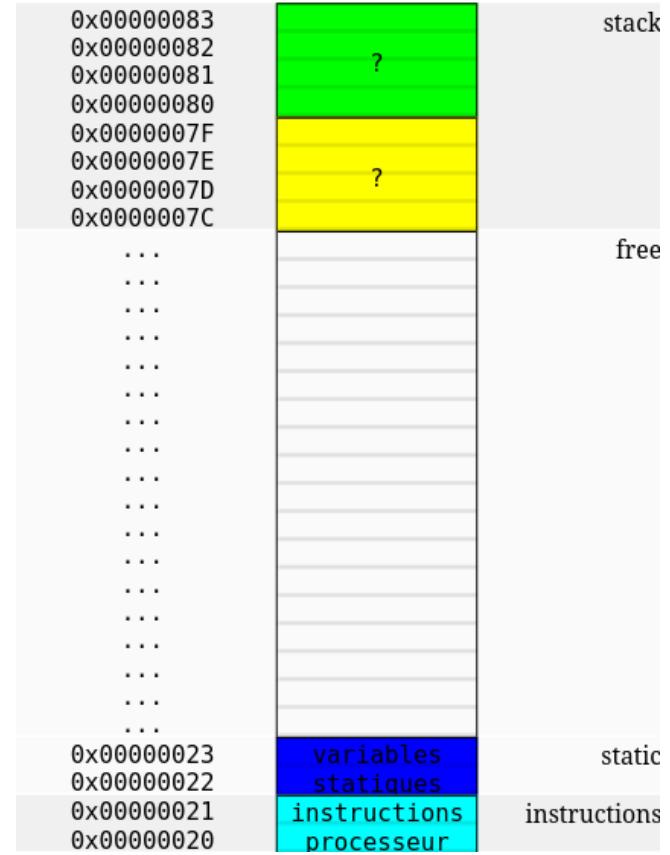


Exemple complet

Création de la frame (rien de spécial)

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

⚠ Ce code présente des problèmes, saurez-vous les trouver ?!



Allocation dynamique & Casting

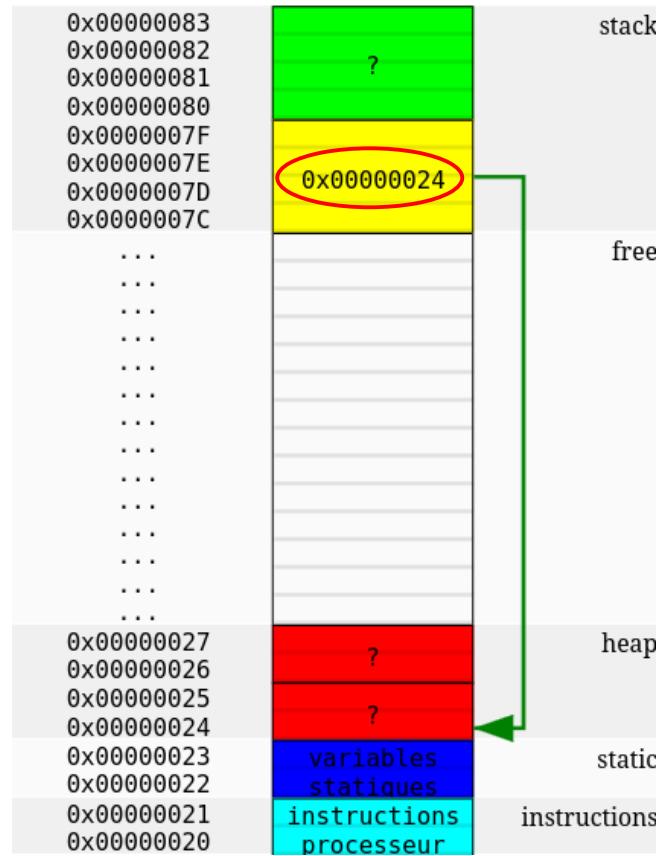


Exemple complet

Première allocation dans le tas

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      → char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

⚠ Ce code présente des problèmes, saurez-vous les trouver ?!



Allocation dynamique & Casting

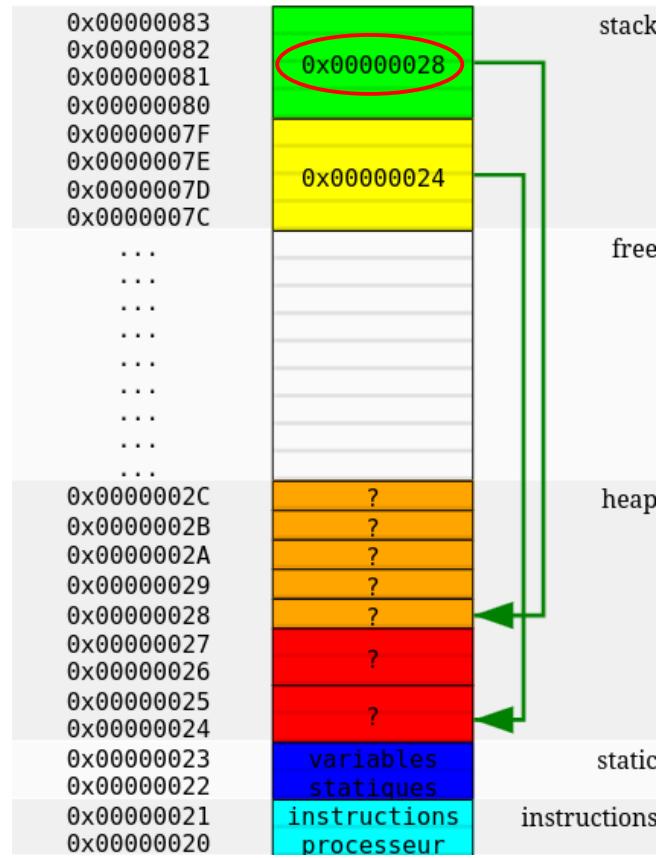


Exemple complet

Seconde allocation dans le tas

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

⚠ Ce code présente des problèmes, saurez-vous les trouver ?!



Allocation dynamique & Casting

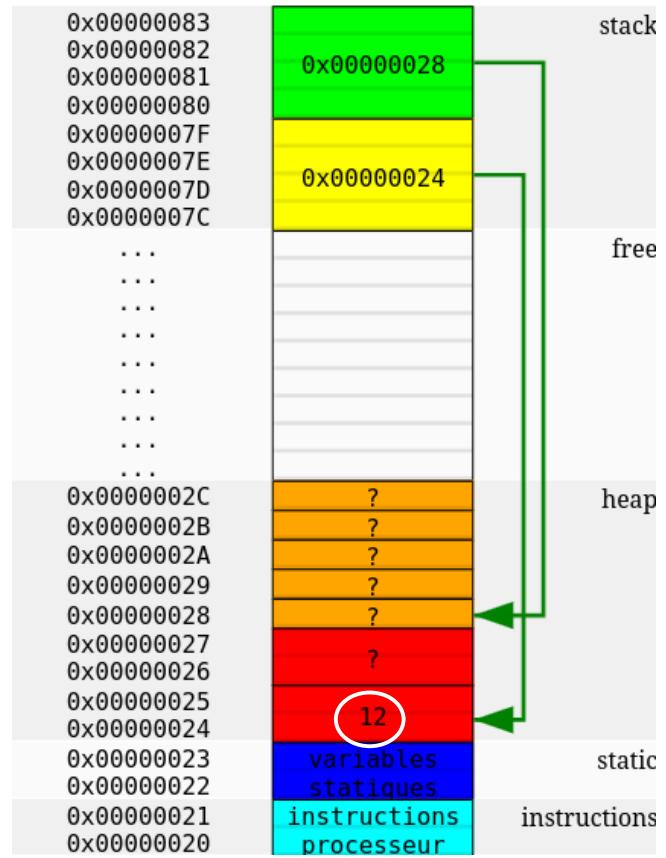


Exemple complet

Attribution de données

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12; →
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

⚠ Ce code présente des problèmes, saurez-vous les trouver ?!



Allocation dynamique & Casting

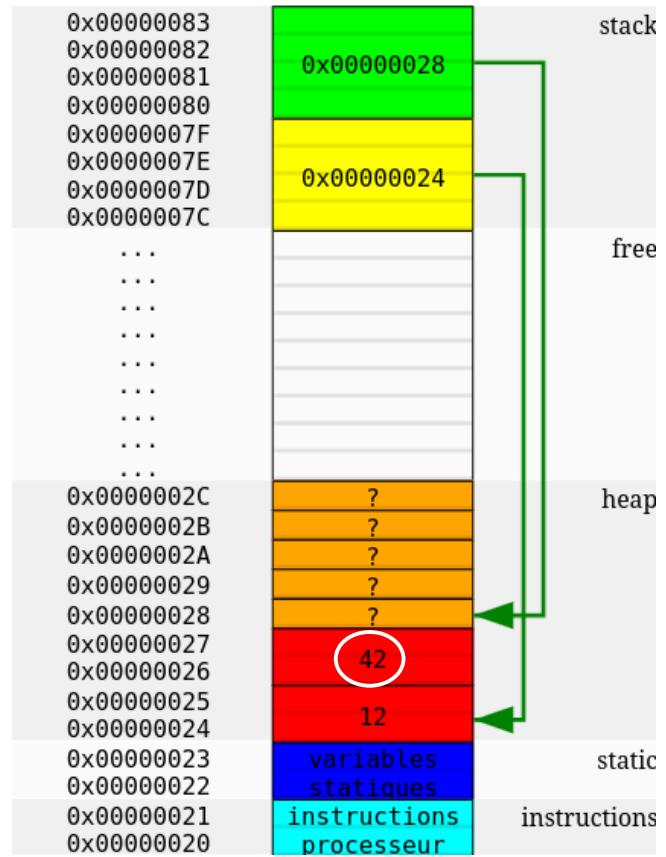


Exemple complet

Attribution de données

```
examples > C malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

⚠ Ce code présente des problèmes, saurez-vous les trouver ?!



Allocation dynamique & Casting

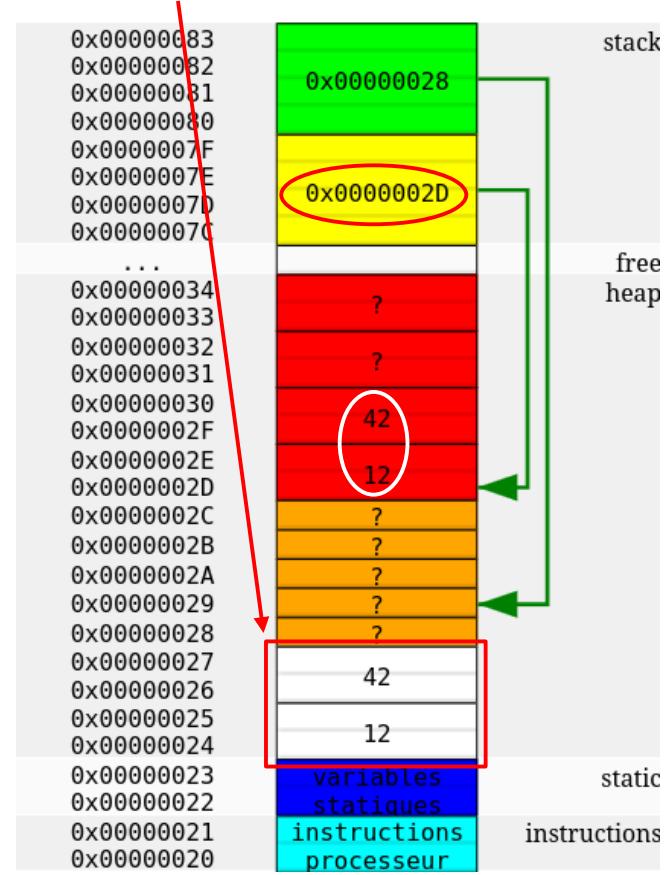


Exemple complet

Realloc de array_1, array_2 est juste derrière array_1,
impossible sans copie (et on stocke le nouveau pointeur)

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {                                ! Ce code présente
17         return EXIT_FAILURE;                            des problèmes,
18     }                                                 saurez-vous les
19     // free memory                                 trouver ?!
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

Zone éligible pour de futures allocations (mais les données ne sont pas nettoyées)



Allocation dynamique & Casting

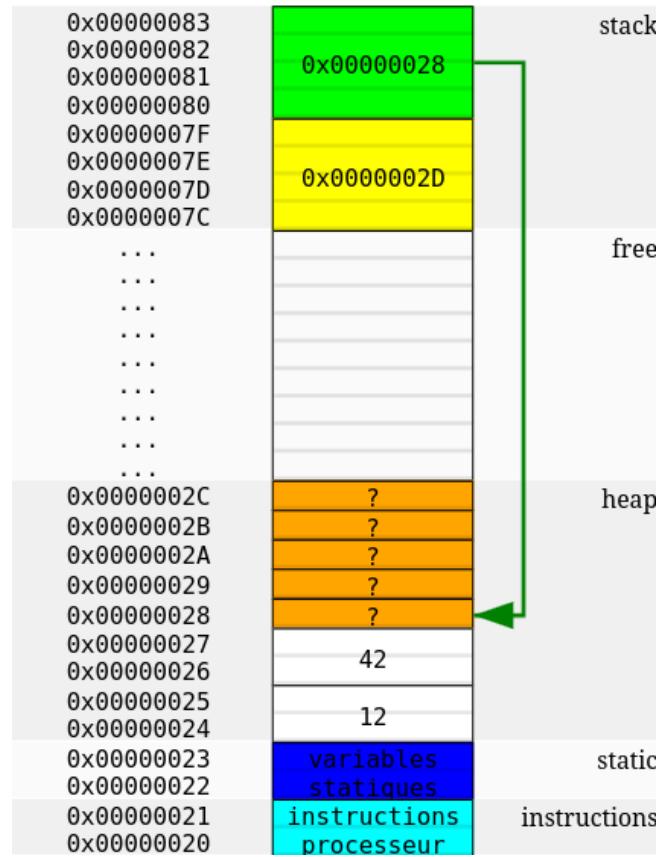


Exemple complet

Libération de la mémoire pointée par array_1

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

⚠ Ce code présente des problèmes, saurez-vous les trouver ?!



Allocation dynamique & Casting

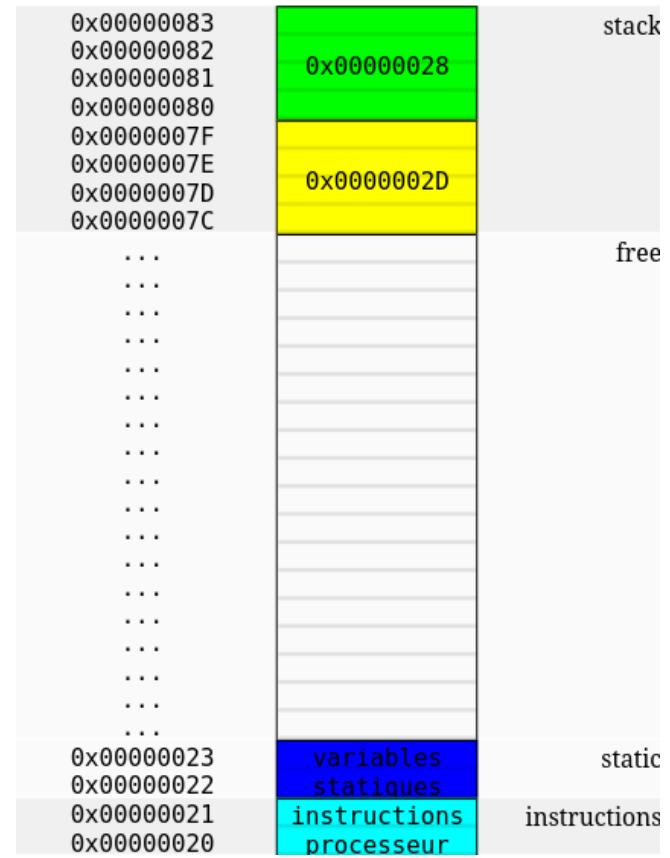


Exemple complet

Libération de la mémoire pointée par array_2

```
examples > C malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

⚠ Ce code présente des problèmes, saurez-vous les trouver ?!



Allocation dynamique & Casting



```
examples > C malloc.c > main(void)
1  #include<stdlib.h>
2  #include<stdio.h>
3  int main(void) {
4
5      int * array_1 = (int *) malloc(10* sizeof(int));
6      char * array_2 = (char *) malloc(10* sizeof(char));
7      if (array_1 == NULL || array_2 == NULL) {
8          return EXIT_FAILURE;
9      }
10     printf("Address of array_1: %p\n", array_1);
11     printf("Address of array_2: %p\n", array_2);
12     array_1 = (int *) realloc(array_1, 100 * sizeof(int));
13     if (array_1 == NULL) {
14         return EXIT_FAILURE;
15     }
16     printf("Address of array_1 after realloc: %p\n", array_1);
17     free(array_1);
18     free(array_2);
19     return EXIT_SUCCESS;
20 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./malloc
Address of array_1: 0x5e5e13b4d2a0
Address of array_2: 0x5e5e13b4d2d0
Address of array_1 after realloc: 0x5e5e13b4d700
(base) → examples git:(main) ✘

⚠ Ce code présente des problèmes, saurez-vous les trouver ?!

Exemple réaliste

Pour observer ce comportement, il faut augmenter un peu la taille des zones à allouer

Les zones ont généralement un padding (sur cette machine, 32 octets), et ne sont pas contiguës dans le tas

Dans l'exemple précédent, libérer une zone en haut de tas fait passer cette zone dans la section "free" (donc utilisable par la stack), ceci est OS-dépendant

Allocation dynamique & Casting



Fuite mémoire

En pratique ce bout de code n'est pas dangereux **sur un système moderne**, et à condition que cette fonction reste la fonction main

Que peut-il se passer si cette fonction n'est plus la fonction principale ?

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

Allocation dynamique & Casting



Fuite mémoire

En pratique ce bout de code n'est pas dangereux sur un système moderne, et à condition que cette fonction reste la fonction main

Que peut-il se passer si cette fonction n'est plus la fonction principale ?

- Si l'allocation pour **array_1** réussit et celle pour **array_2** échoue, on sort de la fonction sans libérer **array_1**

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2 * sizeof(short));
6      char *array_2 = (char *) malloc(5 * sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          | return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         | return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

Allocation dynamique & Casting



Fuite mémoire

En pratique ce bout de code n'est pas dangereux sur un système moderne, et à condition que cette fonction reste la fonction main

Que peut-il se passer si cette fonction n'est plus la fonction principale ?

- Si l'allocation pour **array_1** réussit et celle pour **array_2** échoue, on sort de la fonction sans libérer **array_1**
- Si le realloc échoue, le pointeur vers **array_1** est perdu, **array_2** n'est pas libéré, et la libération de **array_1** n'est même plus possible

```
examples > c malloc.c > main(void)
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      char *array_2 = (char *) malloc(5* sizeof(char));
7      // Check alloc error
8      if (array_1 == NULL || array_2 == NULL) {
9          return EXIT_FAILURE;
10     }
11
12     array_1[0] = 12;
13     array_1[1] = 42;
14     // Check alloc error
15     array_1 = (short *) realloc(array_1, 4 * sizeof(short));
16     if (array_1 == NULL) {
17         return EXIT_FAILURE;
18     }
19     // free memory
20     free(array_1);
21     free(array_2);
22     return EXIT_SUCCESS;
23 }
```

Allocation dynamique & Casting



Version corrigée

Pour éviter les erreurs :

- Traiter les cas d'erreur après chaque allocation
- En cas de réussite partielle, ne pas oublier de libérer la mémoire déjà allouée jusqu'à présent avant de retourner une erreur
- Toujours utiliser un pointeur temporaire pour stocker le retour de realloc
- Utiliser des outils de qualité logicielle (e.g. Valgrind)

```
examples > c malloc_corrigé.c > ...
1  #include<stdlib.h>
2
3  int main(void) {
4
5      short *array_1 = (short *) malloc(2* sizeof(short));
6      // Check each allocation separately
7      if (array_1 == NULL) {
8          return EXIT_FAILURE;
9      }
10     char *array_2 = (char *) malloc(5* sizeof(char));
11
12     if (array_2 == NULL) {
13         // Free each allocated array
14         free(array_1);
15         return EXIT_FAILURE;
16     }
17
18     array_1[0] = 12;
19     array_1[1] = 42;
20     // Use a temporary pointer to check if realloc failed
21     short* tmp_1 = (short *) realloc(array_1, 4 * sizeof(short));
22     if (tmp_1 == NULL) {
23         free(array_1);
24         free(array_2);
25         return EXIT_FAILURE;
26     }
27     // If it succeed, realloc returns the new pointer, replace
28     array_1 = tmp_1;
29
30     free(array_1);
31     free(array_2);
32     return EXIT_SUCCESS;
33 }
```

Allocation dynamique & Casting



Fonctions + pointeurs + allocations

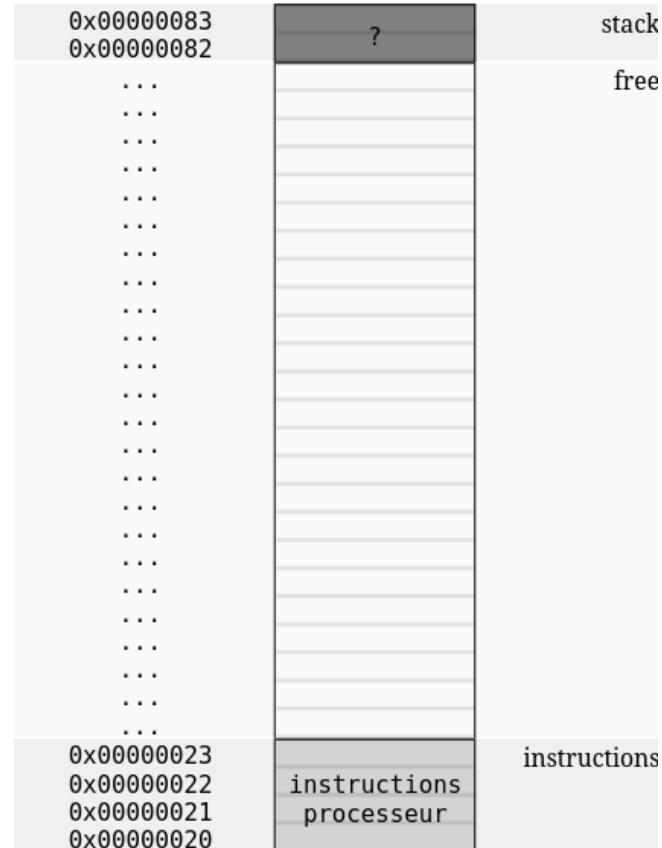
```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf ("The result is %d\n", magenta);
6  }
7
8 // multiply by 2 and store in the heap
9 short *multiply(short orange) {
10    short *yellow = malloc(sizeof(short));
11    *yellow = orange * 2;
12    return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Pour gagner de la place, on passe sur un adressage sur 4 octets, on utilise des shorts et on ne teste pas le retour de malloc

On assignera à "grey" la valeur de retour de "add"



Allocation dynamique & Casting



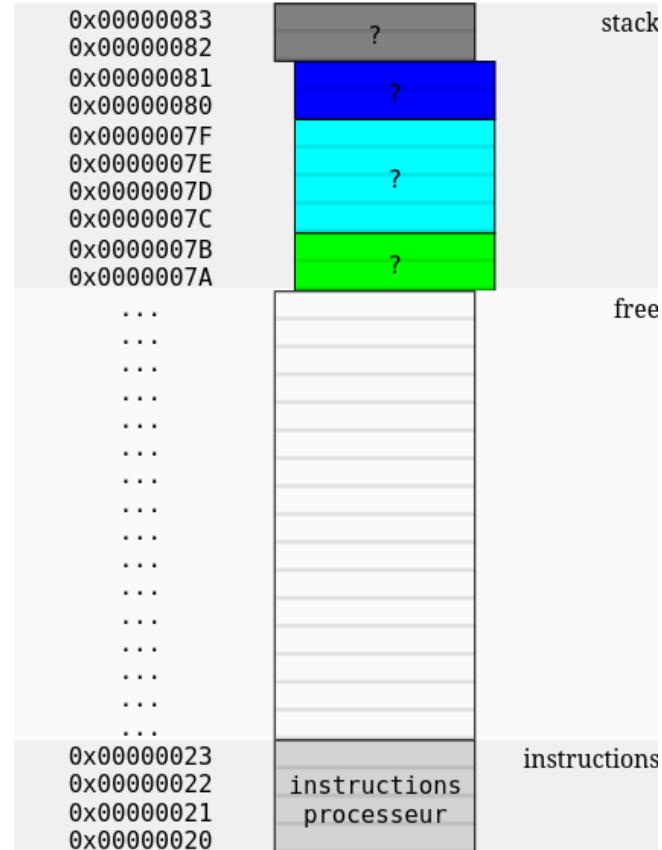
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Création de la frame pour "add"



Allocation dynamique & Casting



Fonctions + pointeurs + allocations

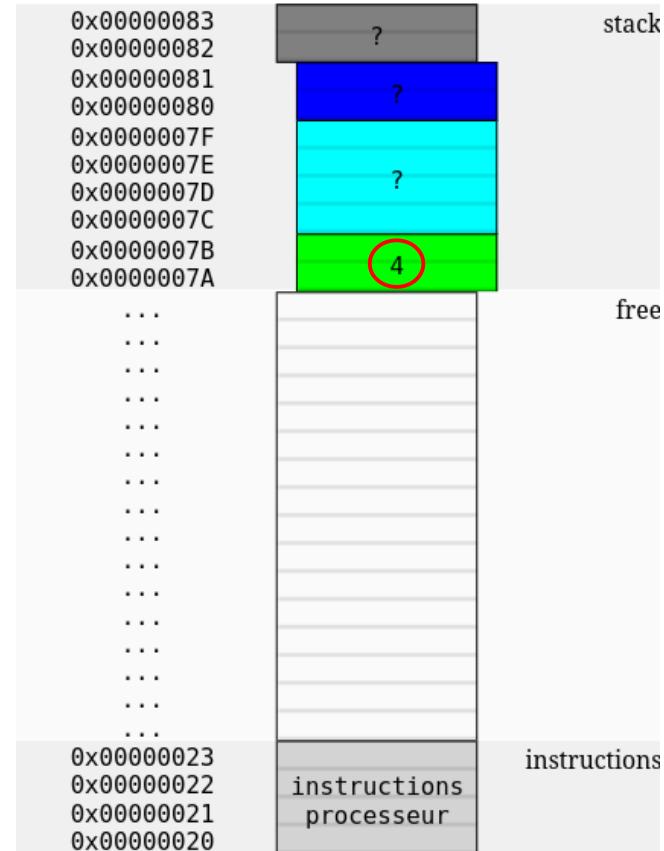
```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Copie des valeurs connues, ici 4 pour "green"

On assignera le retour de "multiply" au pointeur "cyan"



Allocation dynamique & Casting



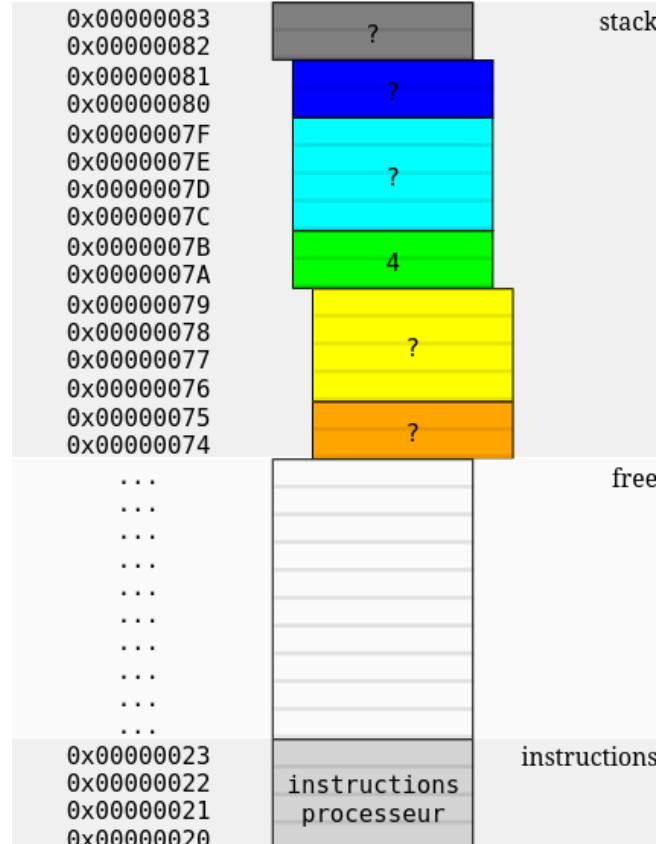
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8 // multiply by 2 and store in the heap
9 short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Création de la frame pour "multiply"



Allocation dynamique & Casting



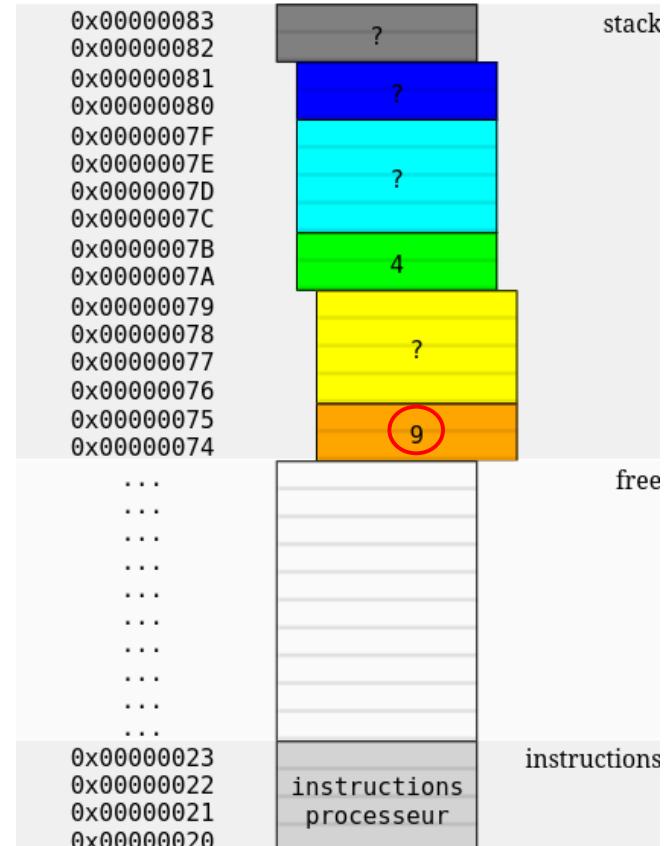
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Copie des valeurs connues, ici 9 pour "orange" ("green" + 5)



Allocation dynamique & Casting



Fonctions + pointeurs + allocations

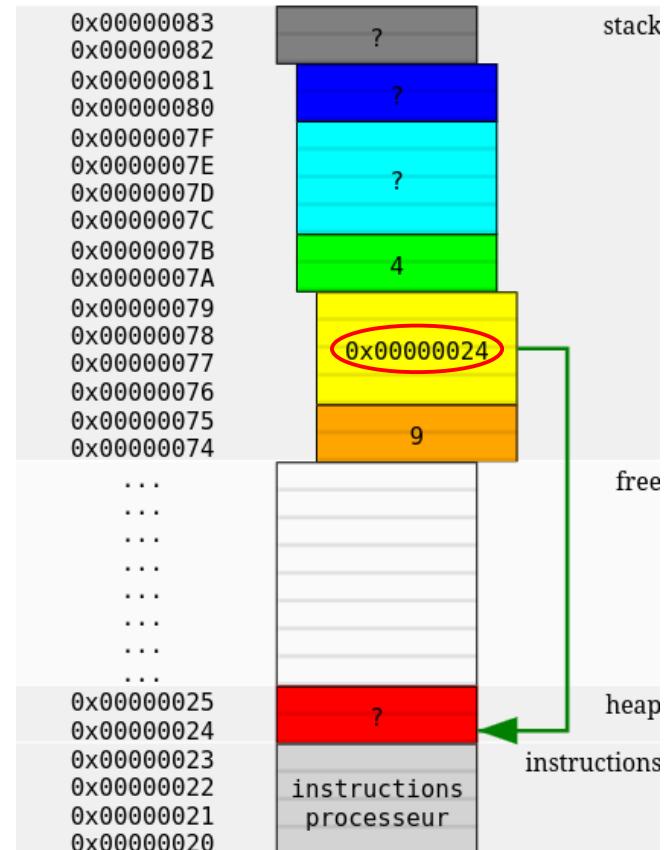
```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Appel à malloc (on ne représente pas la frame associée) et on assigne au pointeur "yellow" la valeur de retour

⚠ ici on ne teste pas le retour de malloc (il faut normalement le tester)



Allocation dynamique & Casting



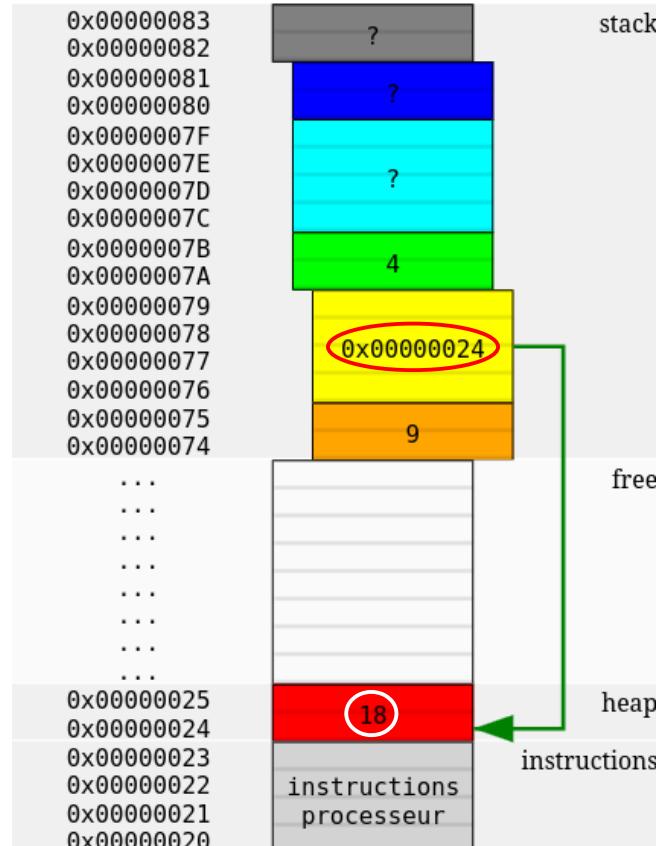
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

On assigne la valeur
"orange" * 2 à la
zone pointée par
yellow



Allocation dynamique & Casting



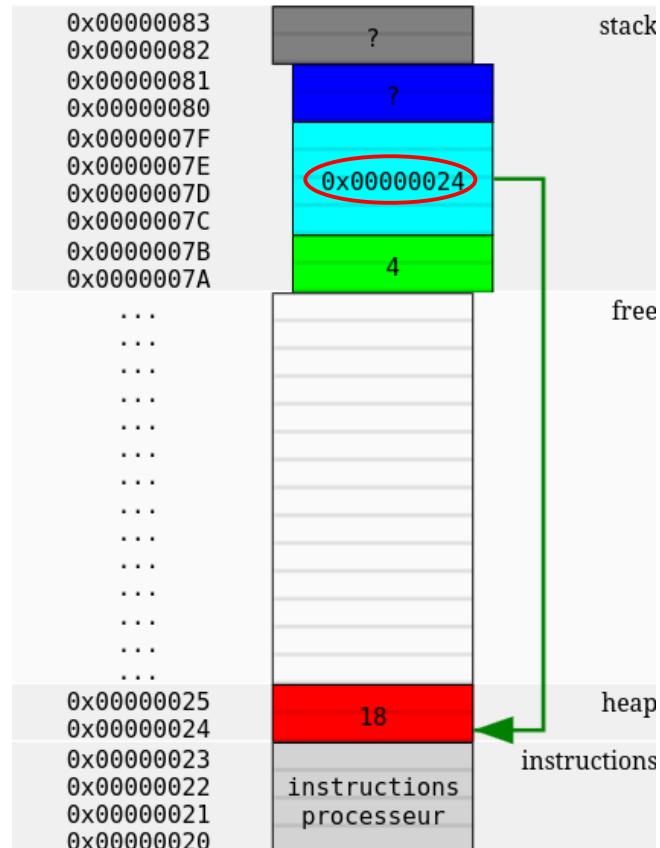
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Le pointeur
retourné par
"multiply" (yellow)
est assigné (copier)
dans "cyan"



Allocation dynamique & Casting

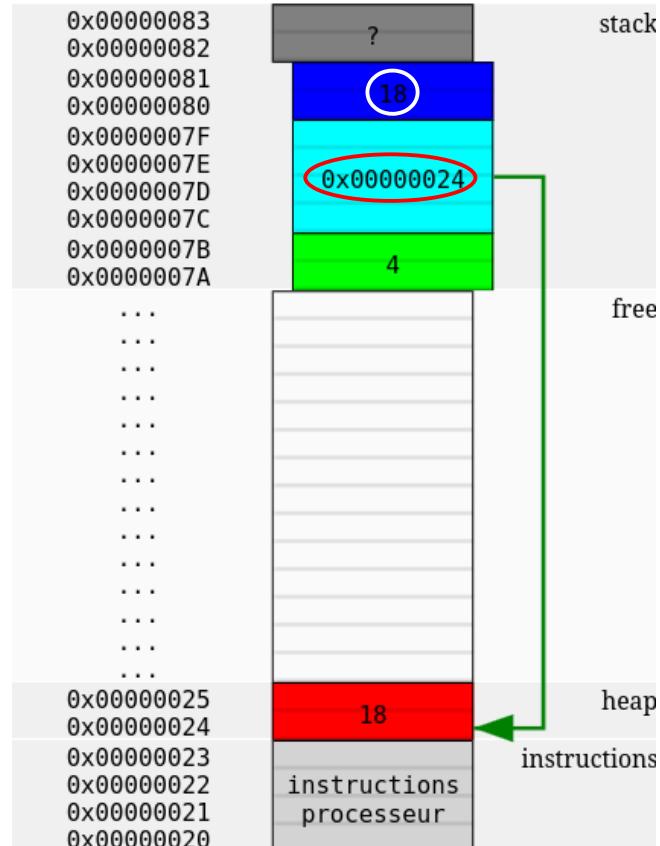


Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
● (base) → examples git:(main) ✘ ./calcul
```

On assigne/copie la valeur pointée par cyan (18) à "blue"



Allocation dynamique & Casting



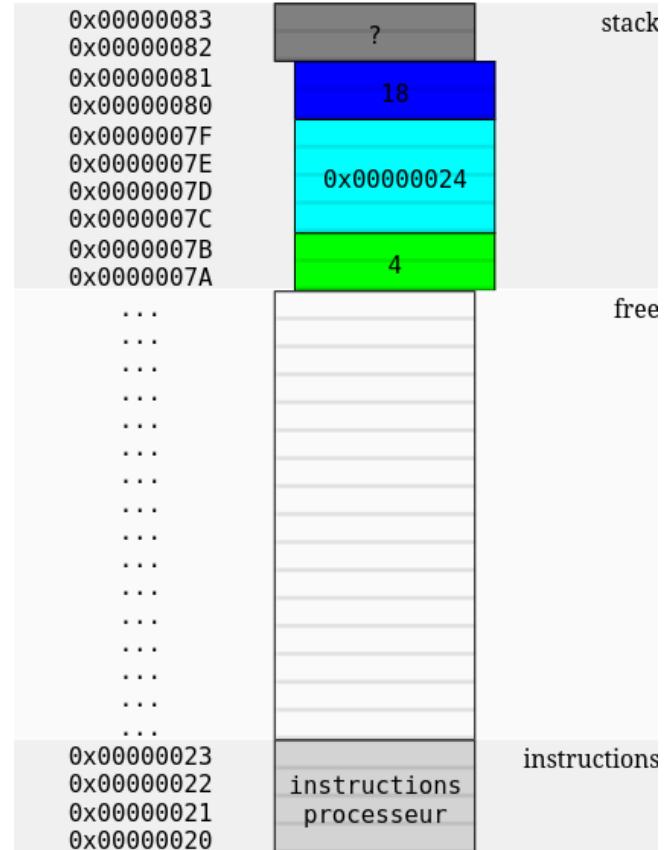
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Libération de la
mémoire pointée
par "cyan"



Allocation dynamique & Casting



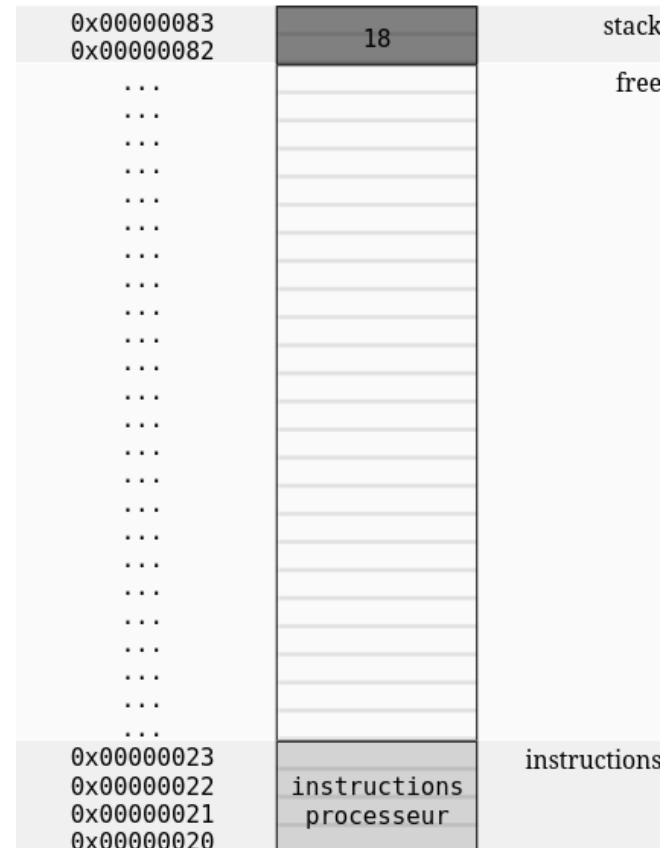
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● (base) → examples git:(main) ✘ ./calcul

La valeur de retour
de "add" ("blue") est
assignée/copiée
dans "grey"



Allocation dynamique & Casting



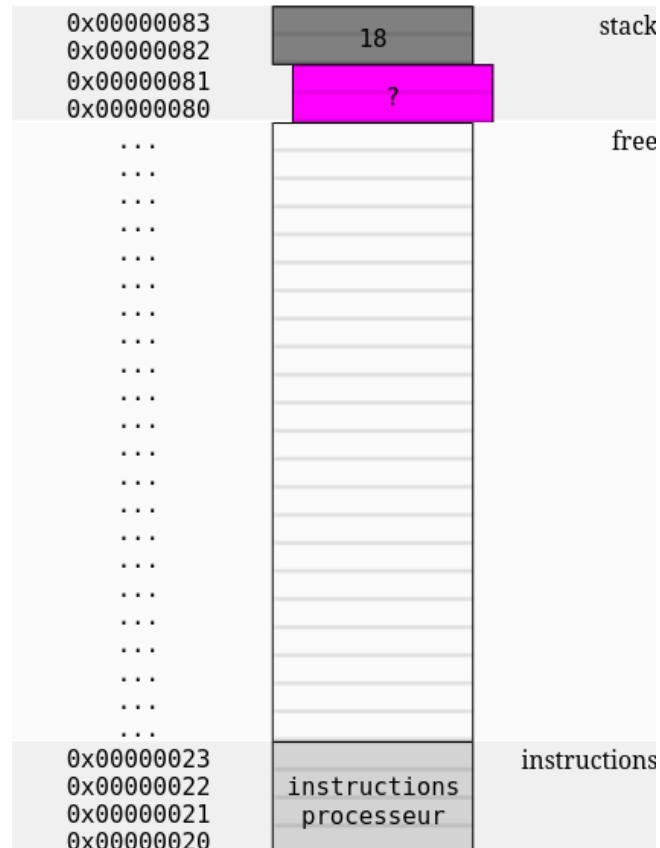
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Création de la frame pour la fonction d'affichage



Allocation dynamique & Casting



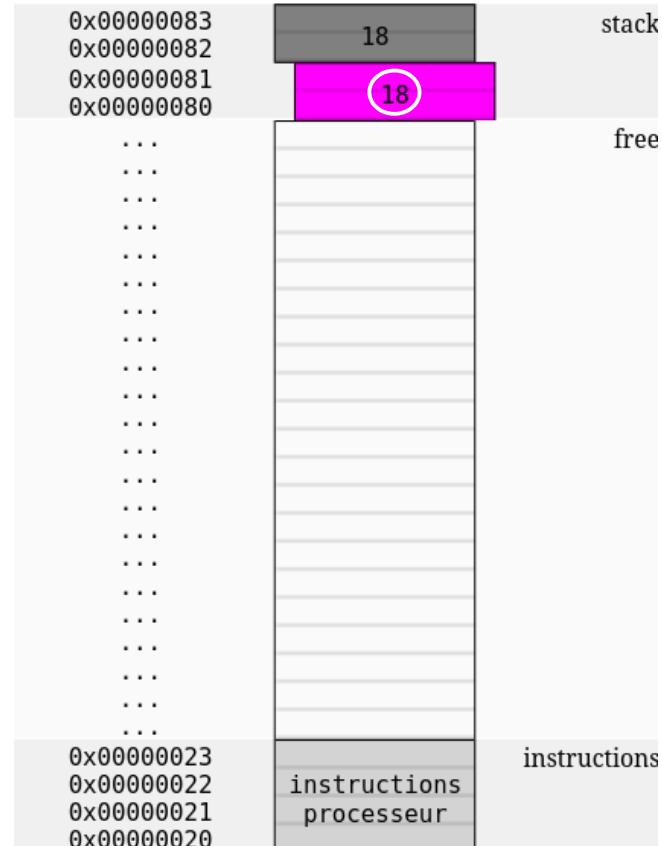
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul

Copie des valeurs connues, "18" pour "magenta" ("grey")



Allocation dynamique & Casting



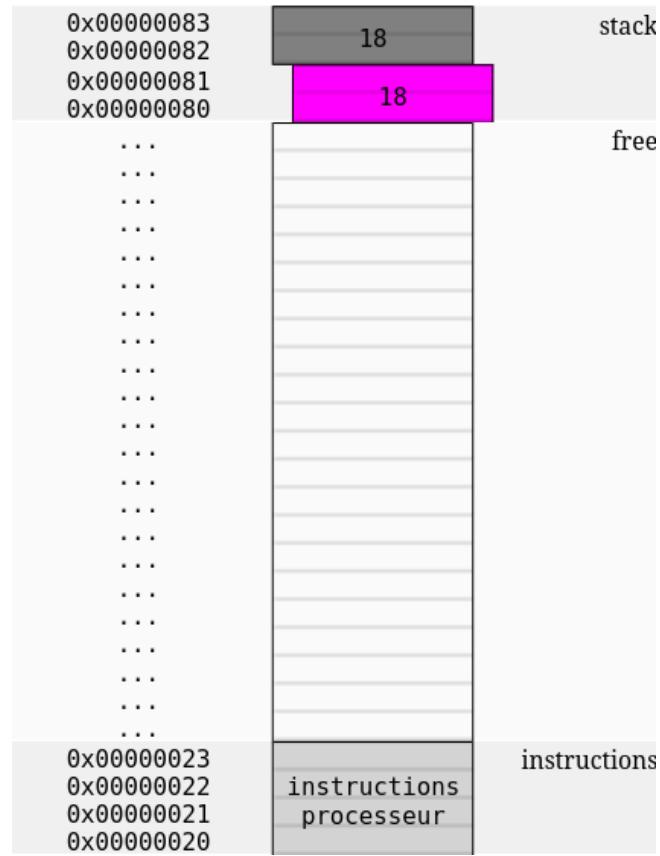
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void display(short magenta) {
5     printf("The result is %d\n", magenta);
6 }
7
8 // multiply by 2 and store in the heap
9 short *multiply(short orange) {
10    short *yellow = malloc(sizeof(short));
11    *yellow = orange * 2;
12    return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17    short *cyan = multiply(green + 5);
18    short blue = *cyan;
19    free(cyan);
20    return blue;
21 }
22
23 int main() {
24    short grey = add(4);
25    display(grey);
26    return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul
The result is 18

Appel à printf (on ne représente pas la frame associée)



Allocation dynamique & Casting



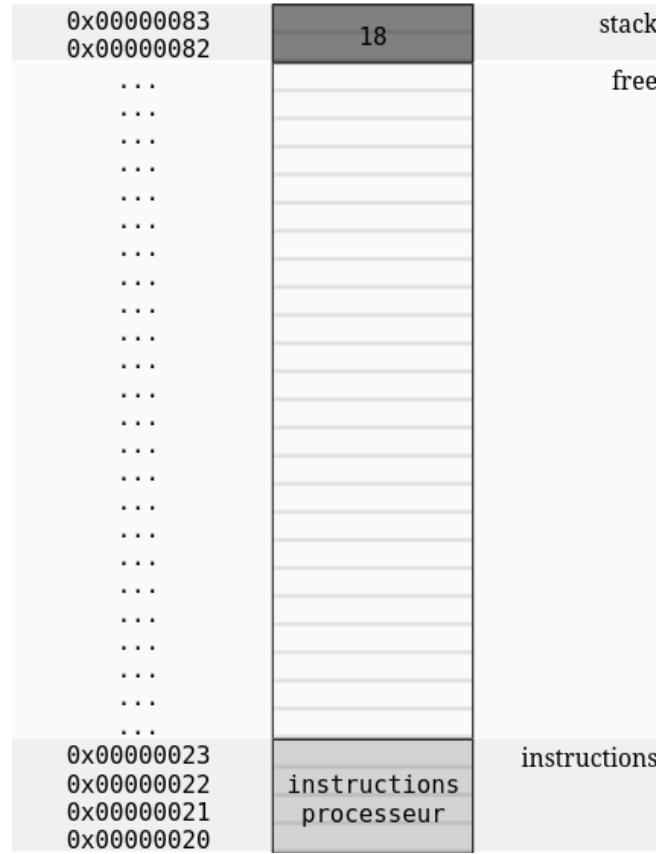
Fonctions + pointeurs + allocations

```
C calcul.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void display(short magenta) {
5      printf("The result is %d\n", magenta);
6  }
7
8  // multiply by 2 and store in the heap
9  short *multiply(short orange) {
10     short *yellow = malloc(sizeof(short));
11     *yellow = orange * 2;
12     return yellow;
13 }
14
15 // add 5
16 short add(short green) {
17     short *cyan = multiply(green + 5);
18     short blue = *cyan;
19     free(cyan);
20     return blue;
21 }
22
23 int main() {
24     short grey = add(4);
25     display(grey);
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(base) → examples git:(main) ✘ ./calcul
The result is 18

Retour de "display"



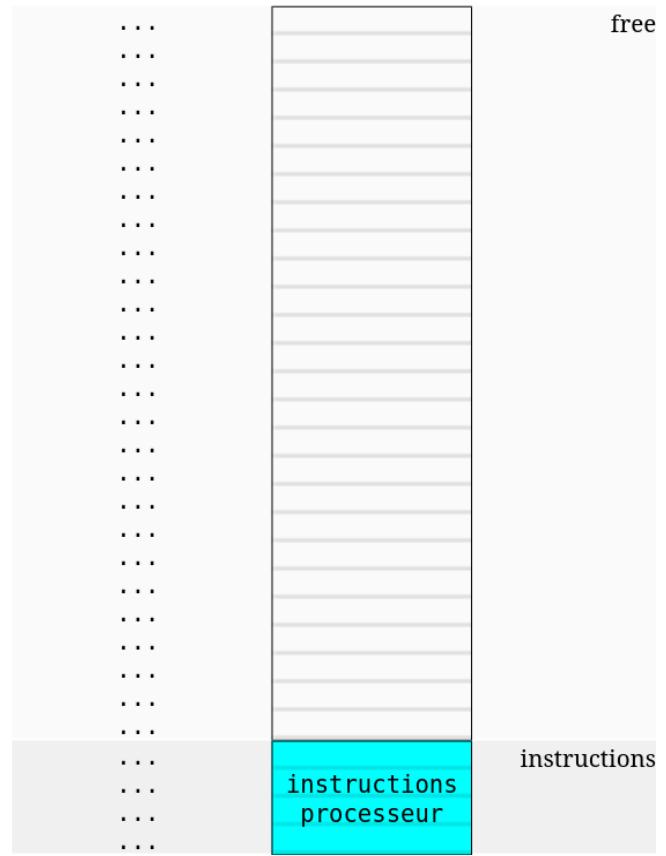
Allocation dynamique & Casting



Pointeur de pointeur

Dans cet exemple, on cherche à créer une fonction qui permet de prendre une adresse en argument et la modifier, ici car elle utilise realloc

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



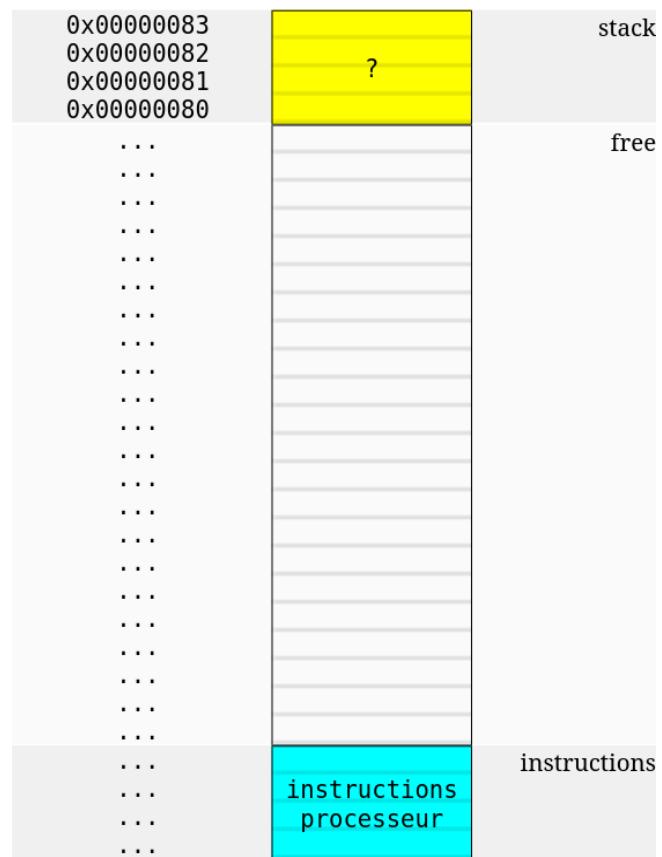
Allocation dynamique & Casting



Pointeur de pointeur

Création de la frame pour "main"

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



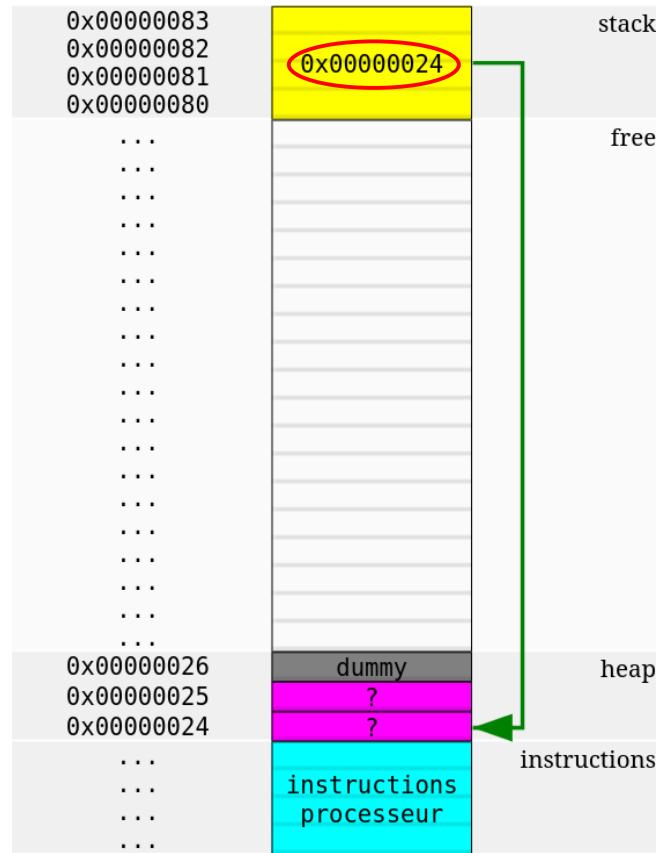
Allocation dynamique & Casting



Pointeur de pointeur

Allocation d'un tableau de 2 caractères

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



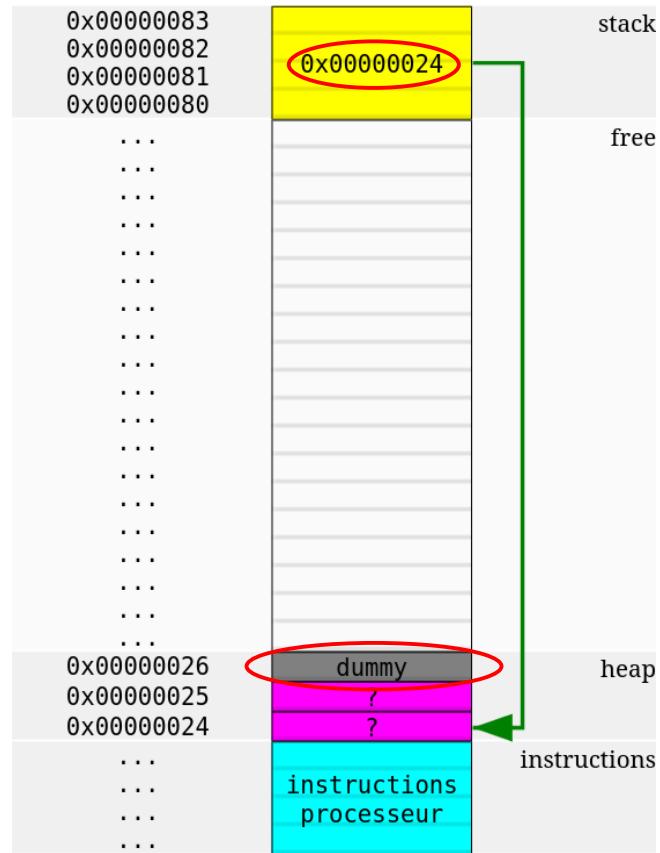
Allocation dynamique & Casting



Pointeur de pointeur

Pour forcer un changement d'adresse par realloc, (ce qui est tout à fait possible), on ajoute une zone factice (dummy area) qui n'existe normalement pas

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



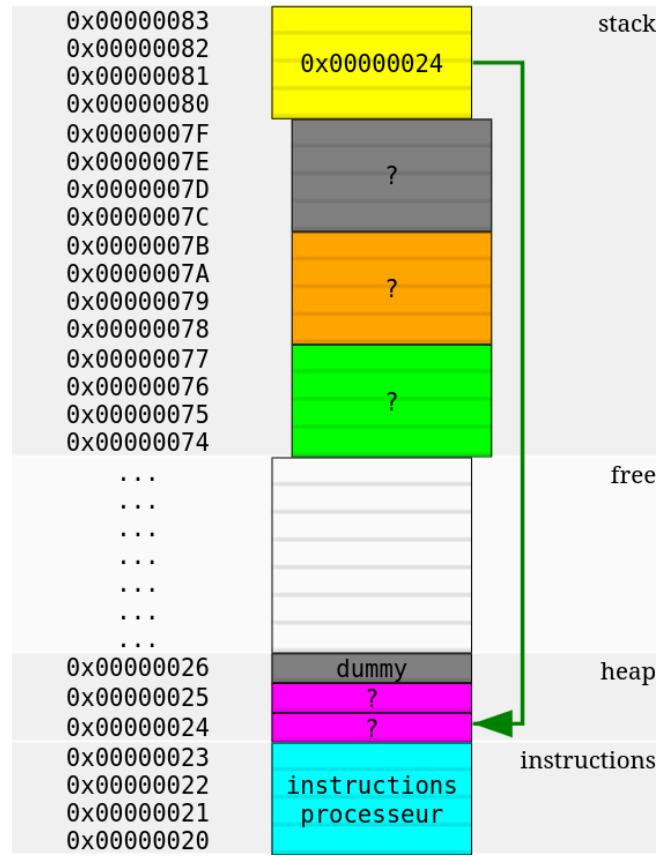
Allocation dynamique & Casting



Pointeur de pointeur

Appel à my_realloc

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



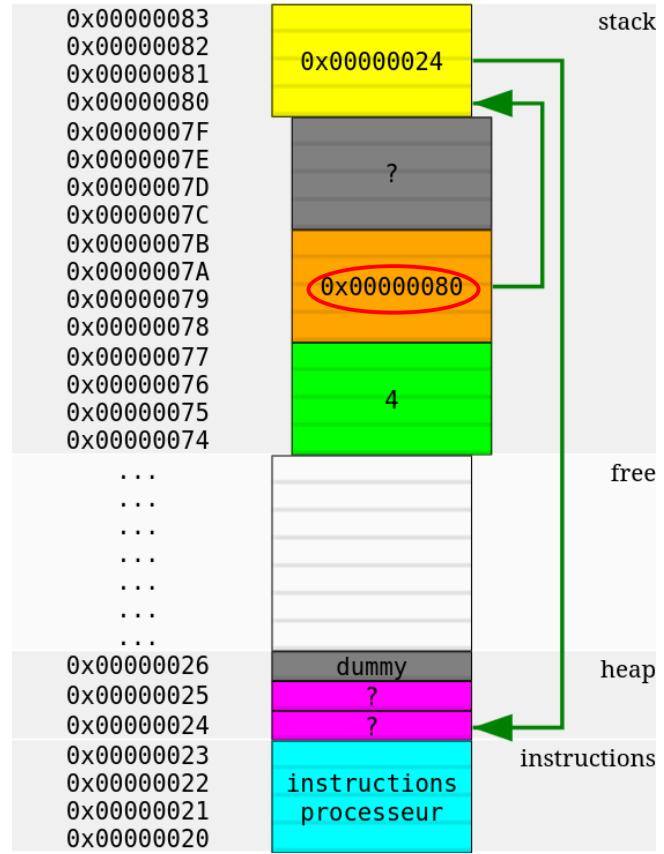
Allocation dynamique & Casting



Pointeur de pointeur

On passe l'adresse du "tableau", i.e. l'adresse de l'adresse, par dégradation

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



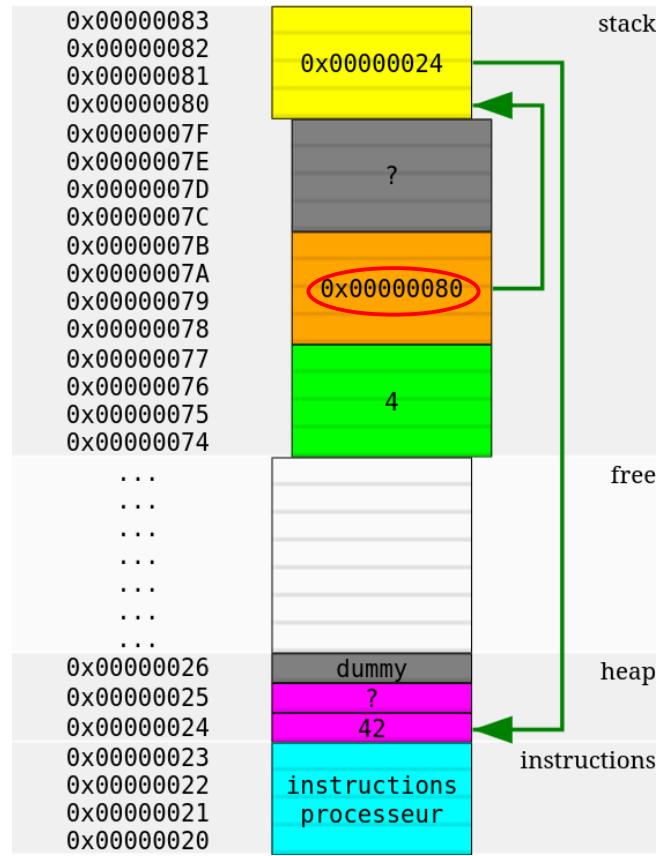
Allocation dynamique & Casting



Pointeur de pointeur

On passe l'adresse du "tableau", i.e. l'adresse de l'adresse, par dégradation

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



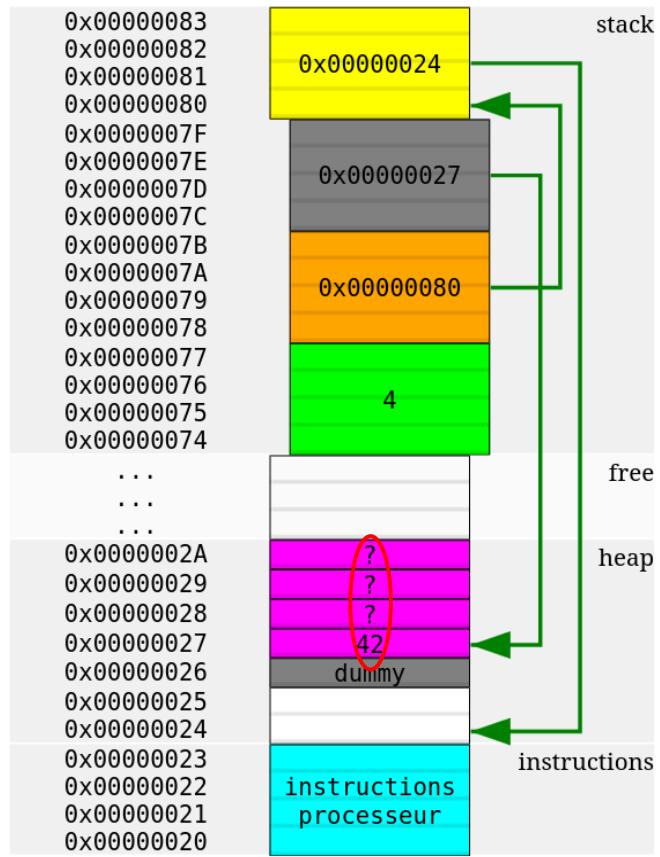
Allocation dynamique & Casting



Pointeur de pointeur

A cause de la zone factice, realloc ne peut pas juste étendre la zone. Realloc copie les données vers une nouvelles zone !

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      → char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



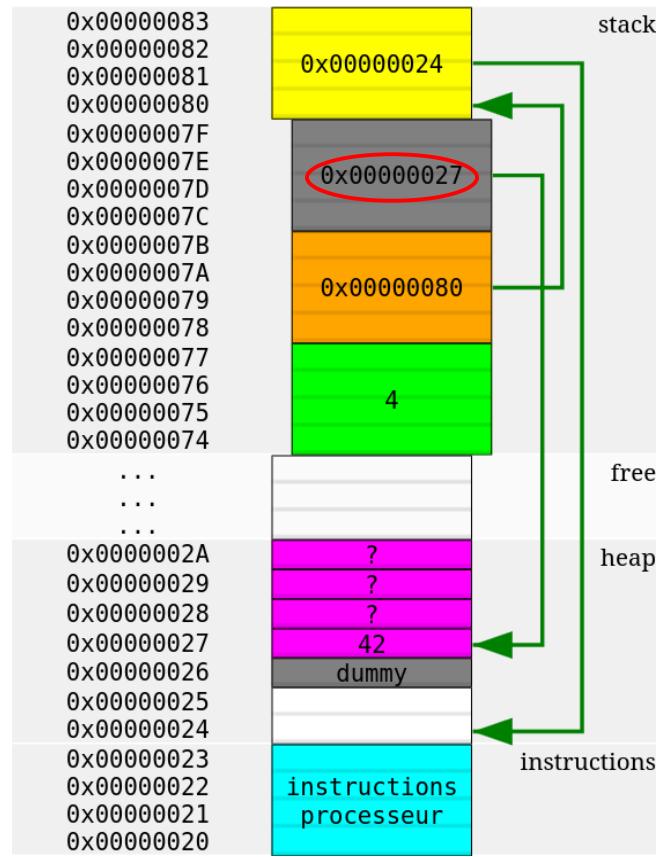
Allocation dynamique & Casting



Pointeur de pointeur

Realloc : on stocke la nouvelle adresse dans un pointeur temporaire. Il faut tester le retour de realloc, sans "tmp" on perdrait autrement l'ancienne adresse (fuite mémoire)

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      → char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



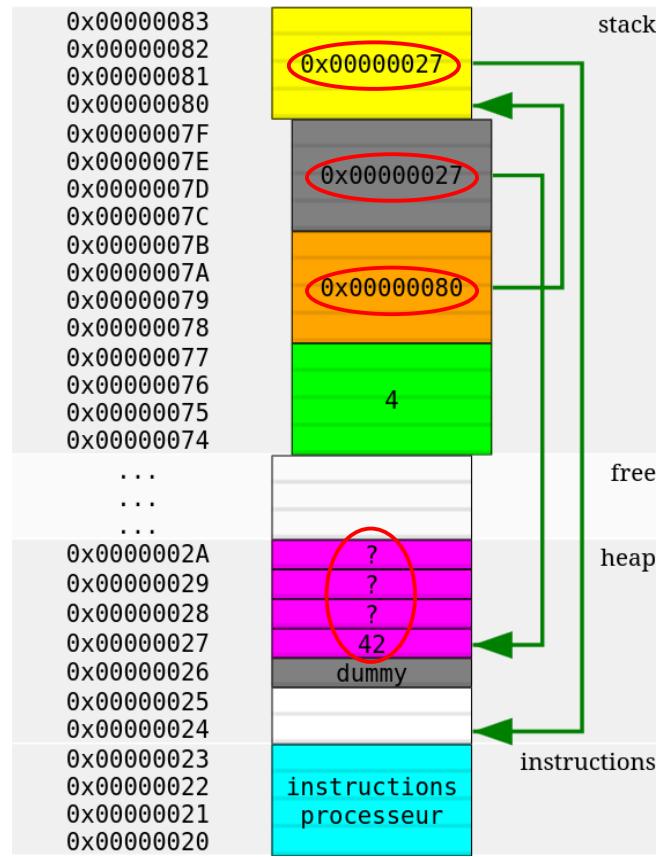
Allocation dynamique & Casting



Pointeur de pointeur

On remplace la valeur à l'adresse pointée par "ptr_addr" par "tmp" (on remplace l'ancienne adresse par la nouvelle).

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



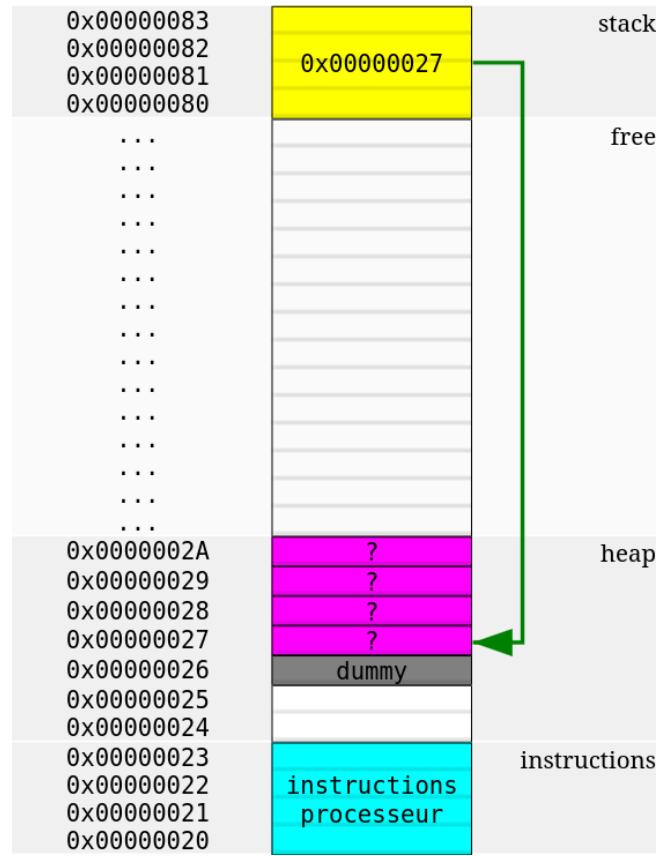
Allocation dynamique & Casting



Pointeur de pointeur

On a donc bien en sortie de fonction le pointeur d'origine modifié si le realloc n'a pas échoué

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



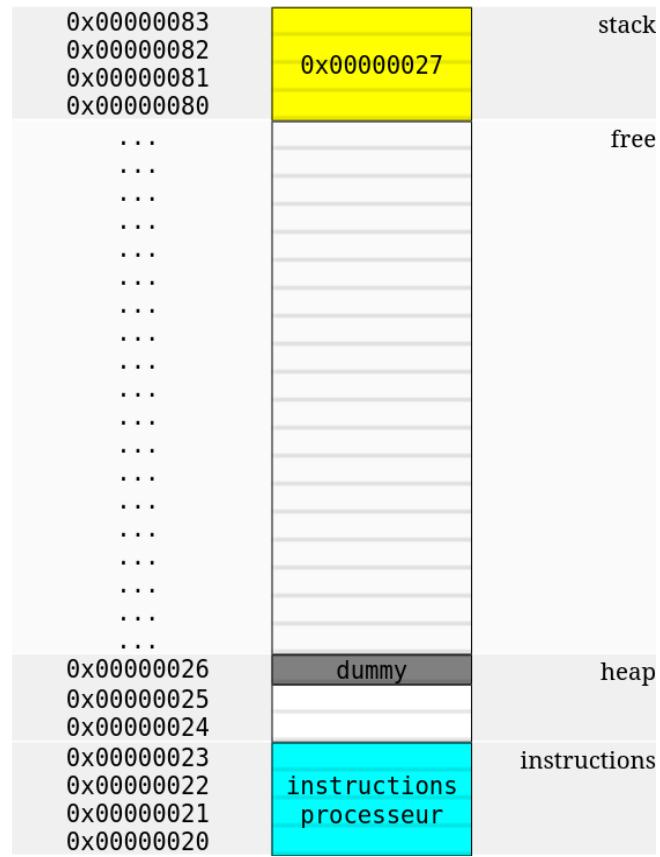
Allocation dynamique & Casting



Pointeur de pointeur

Comme toujours, on libère la mémoire. On note que "tab" pointerait toujours sur 0x00000024 si le realloc avait échoué, le free fonctionnerait également

```
c realloc.c > ...
1  #include <stdlib.h>
2
3  void my_realloc(char **ptr_addr, int new_size) {
4      *ptr_addr[0] = 42;
5      char *tmp = realloc(*ptr_addr, new_size);
6      if (tmp != NULL) {
7          *ptr_addr = tmp;
8      }
9  }
10
11 int main(void) {
12     char *tab = (char *) malloc(sizeof(char) * 2);
13     if (tab == NULL) {
14         return EXIT_FAILURE;
15     }
16     my_realloc(&tab, 4);
17     free(tab);
18     return EXIT_SUCCESS;
19 }
```



Allocation dynamique & Casting



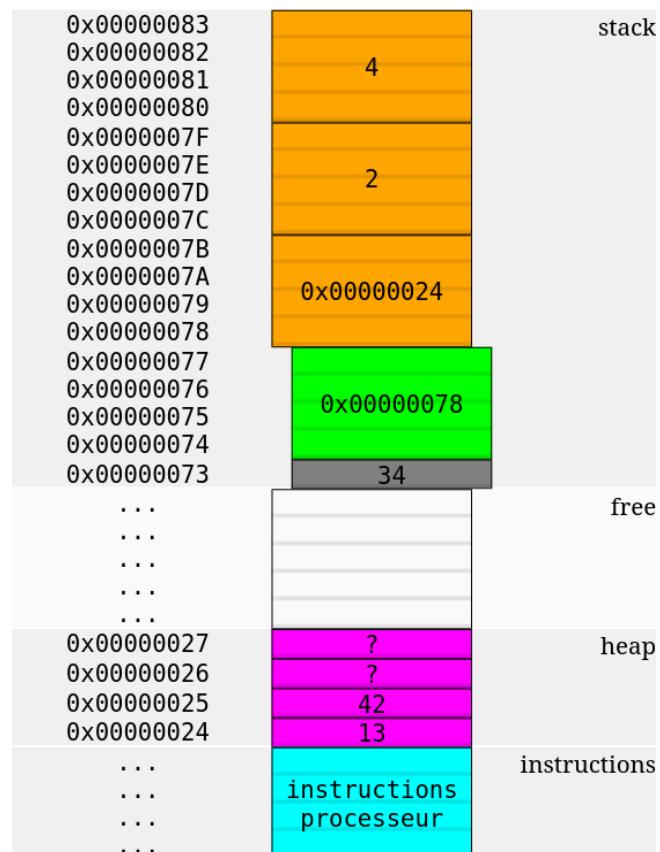
Le cas des vecteurs

Les vecteurs (**Vec** en Rust, **list** en Python) sont classiquement implémentés avec un tableau dynamique

On imagine la création d'une liste avec 2 éléments dans la fonction principale (main), en Python [13, 42]

On aura donc deux frames, la fonction principale, et la fonction d'ajout

On traitera le cas d'ajout au début, (`insert(0, val)` ou `push_front(val)`), et d'ajout à la fin (`append(val)` ou `push_back(val)`)



Allocation dynamique & Casting



Vectors : push back

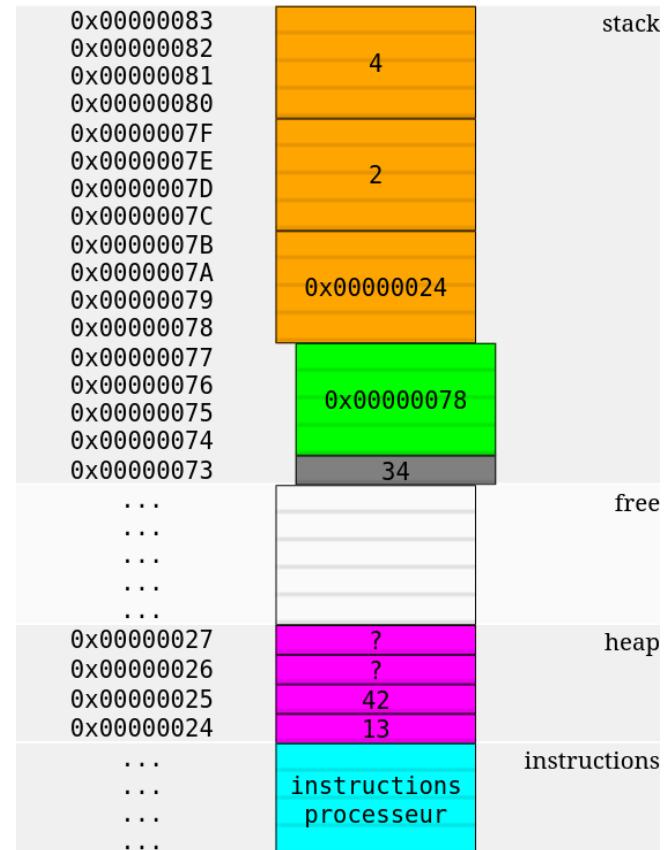
En orange : la liste

En vert : son adresse passée à "list_push_back_char"

En gris : elem (34)

En violet : la zone des données

```
examples > vector > C vector_push_back.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back_char(list_char *list, char elem)
9  int FUNC_NAME(list_push_back_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     list->array[list->length] = elem;
15     list->length += 1;
16     return 1;
17 }
```



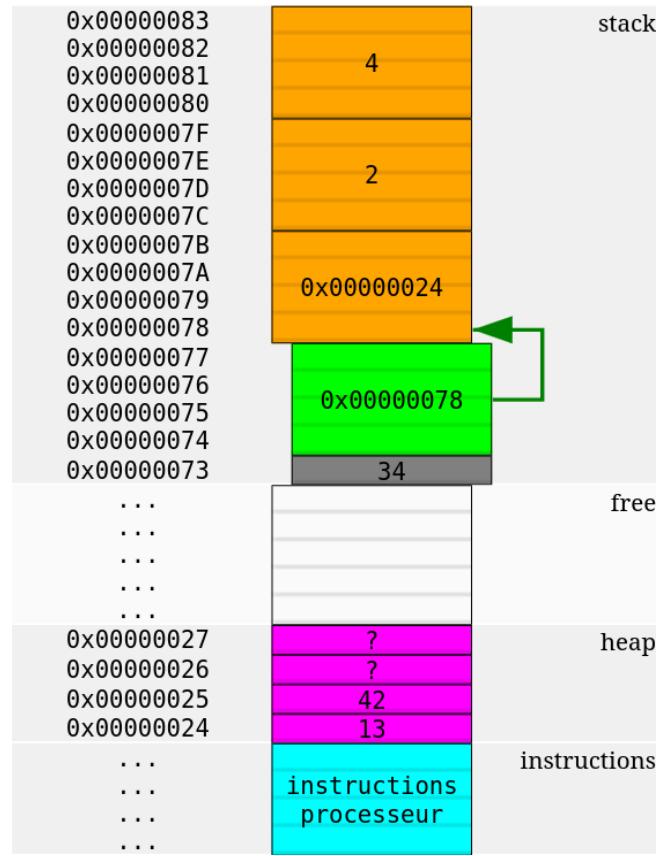
Allocation dynamique & Casting



Vectors : push back

On déréférence l'adresse de la liste pour l'atteindre

```
examples > vector > C vector_push_back.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back_char(list_char *list, char elem)
9  int FUNC_NAME(list_push_back_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     list->array[list->length] = elem;
15     list->length += 1;
16     return 1;
17 }
```



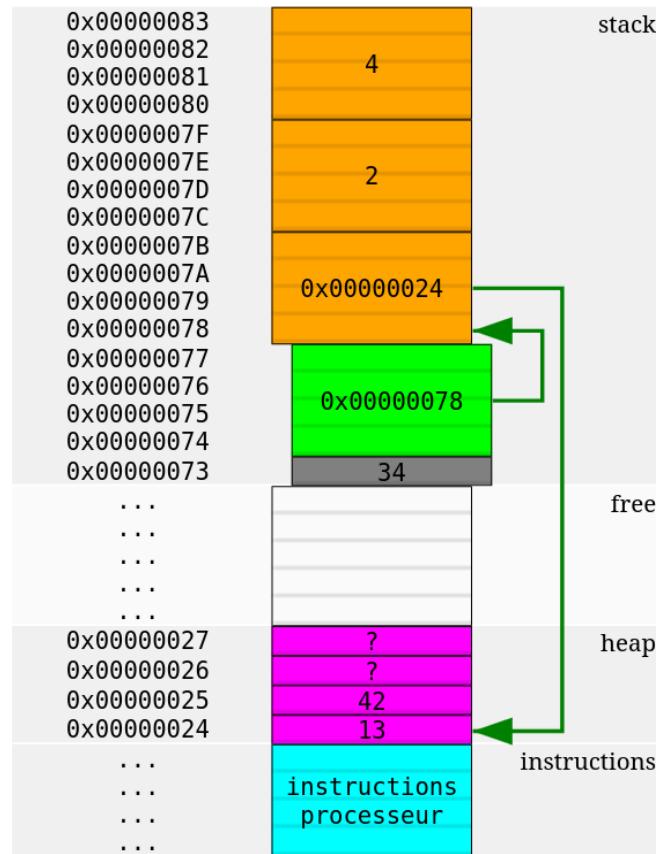
Allocation dynamique & Casting



Vectors : push back

On **déréference** l'adresse de la liste pour l'atteindre, puis on utilise l'opérateur de souscription (les crochets), pour accéder à la case ayant l'index "**list→length**".

```
examples > vector > C vector_push_back.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back_char(list_char *list, char elem)
9  int FUNC_NAME(list_push_back_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     list->array[list->length] = elem;
15     list->length += 1;
16     return 1;
17 }
```



Allocation dynamique & Casting

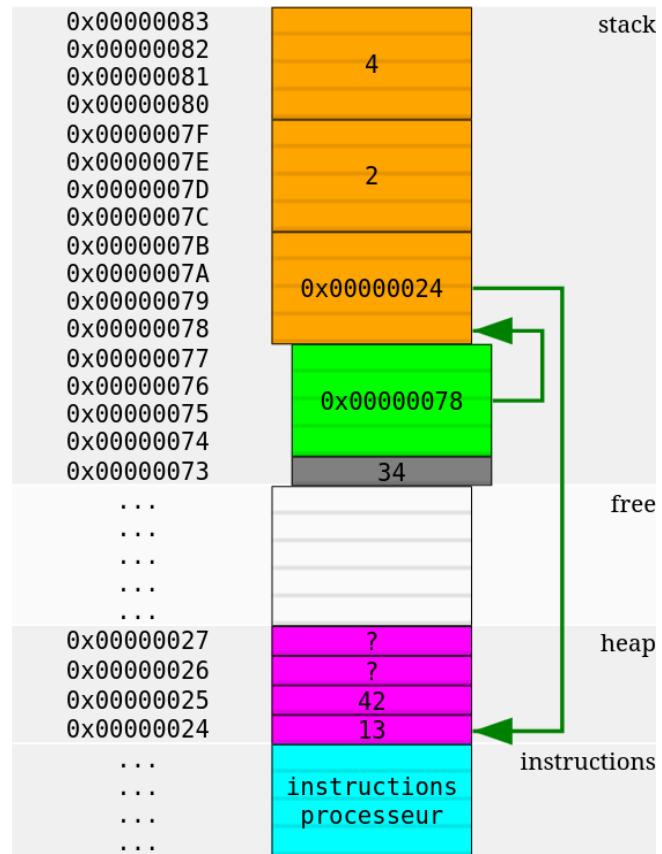


Vectors : push back

Le " \rightarrow length" ajoute un offset à l'adresse "list" pour l'accès (ici 4 octets).

La souscription agit comme une déréférence, la valeur ajoute un offset (ici 2 fois la taille d'un char, donc 2×1 octets)

```
examples > vector > C vector_push_back.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back_char(list_char *list, char elem)
9  int FUNC_NAME(list_push_back_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     list->array[list->length] = elem;
15     list->length += 1;
16     return 1;
17 }
```



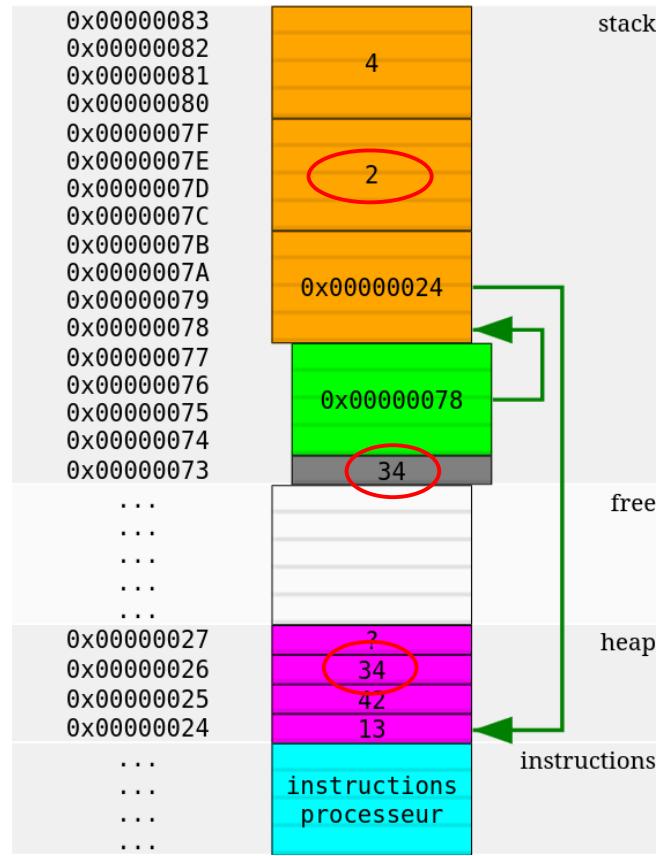
Allocation dynamique & Casting



Vectors : push back

On modifie donc la valeur à la 3 ème position ! Une assignation est une copie !

```
examples > vector > C vector_push_back.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back_char(list_char *list, char elem)
9  int FUNC_NAME(list_push_back_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     list->array[list->length] = elem; // Redirection
15     list->length += 1;
16     return 1;
17 }
```



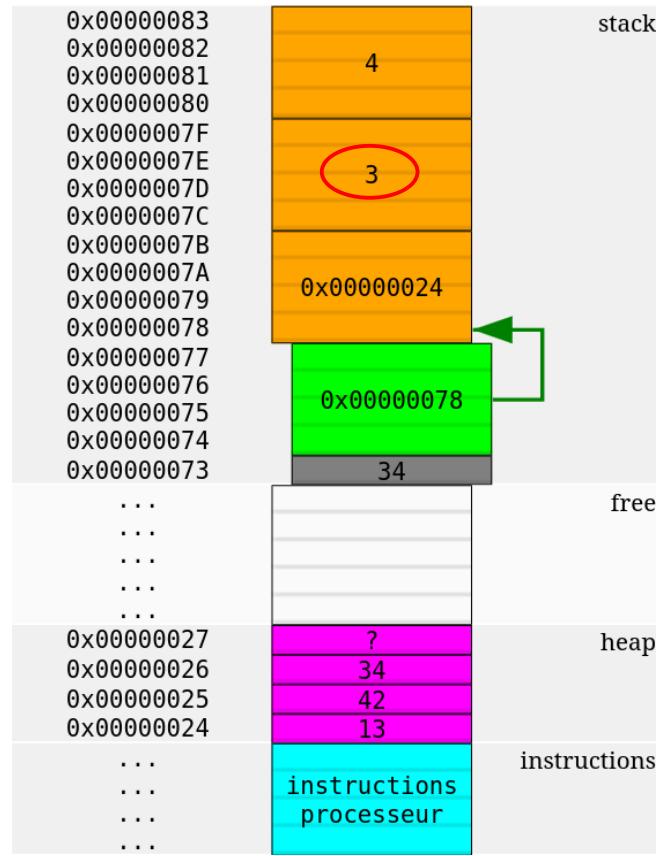
Allocation dynamique & Casting



Vectors : push back

On modifie donc la valeur à la 3 ème position !

```
examples > vector > C vector_push_back.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back_char(list_char *list, char elem)
9  int FUNC_NAME(list_push_back_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     list->array[list->length] = elem;
15     list->length += 1; // Red arrow points here
16     return 1;
17 }
```



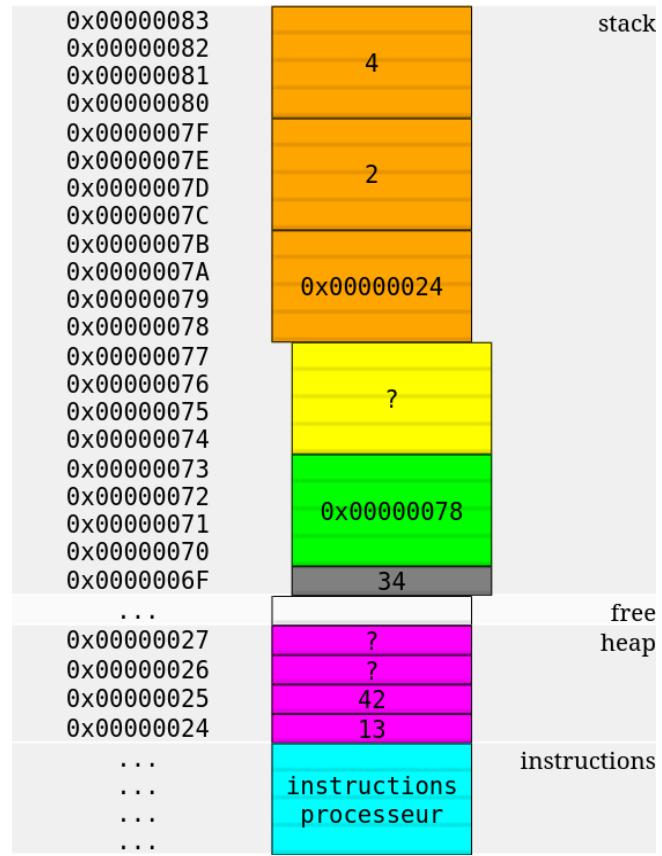
Allocation dynamique & Casting



Vectors : push front

On veut maintenant mettre la valeur en premier position, il faut donc faire de la place !

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i = list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



Allocation dynamique & Casting

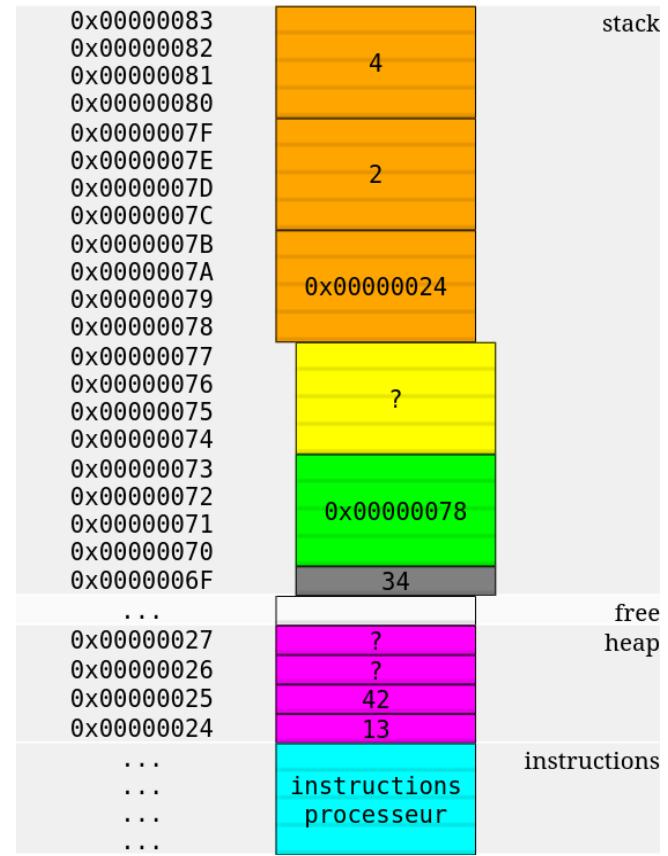


Vectors : push front

La frame est très similaire, on observe seulement une variable supplémentaire:

En jaune : la valeur de i

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list push back char(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i= list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



Allocation dynamique & Casting

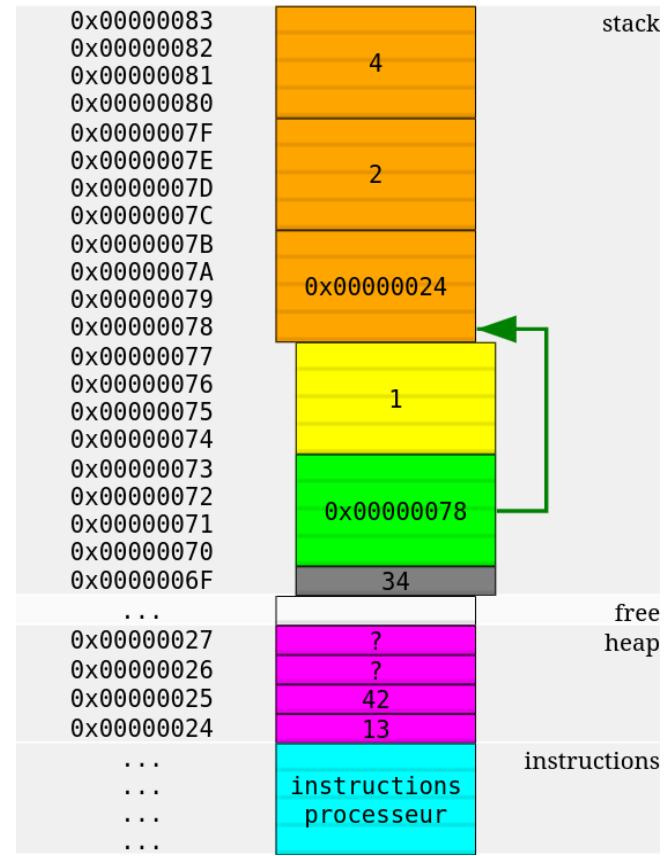


Vectors : push front

La frame est très similaire, on observe seulement une variable supplémentaire:

En jaune : la valeur de i qu'on initialise pour la boucle

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list push back char(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i = list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



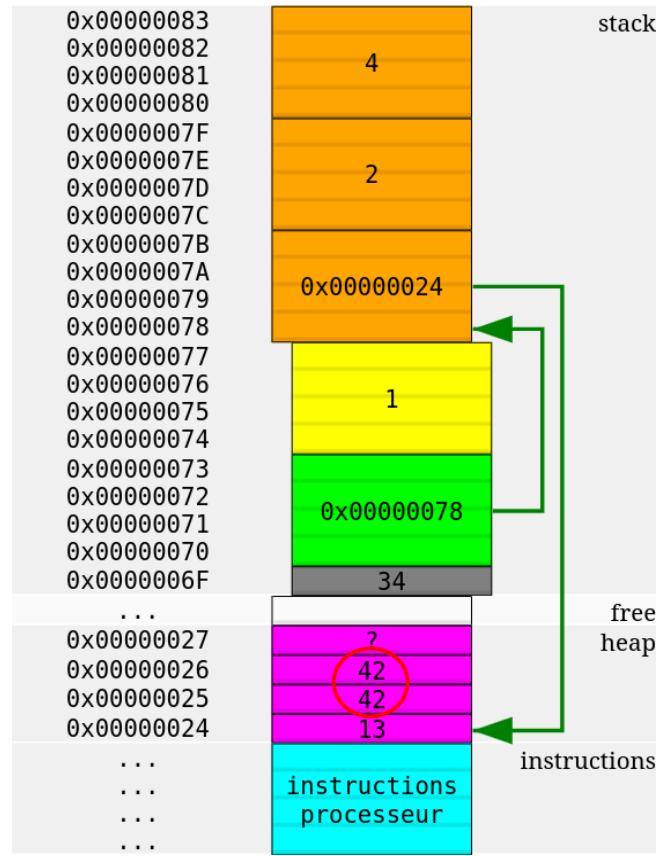
Allocation dynamique & Casting



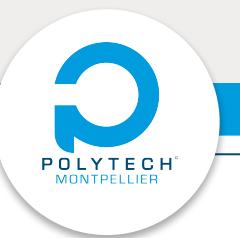
Vectors : push front

On assigne (copie) la valeur de **array[1]** dans **array[2]**

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i = list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



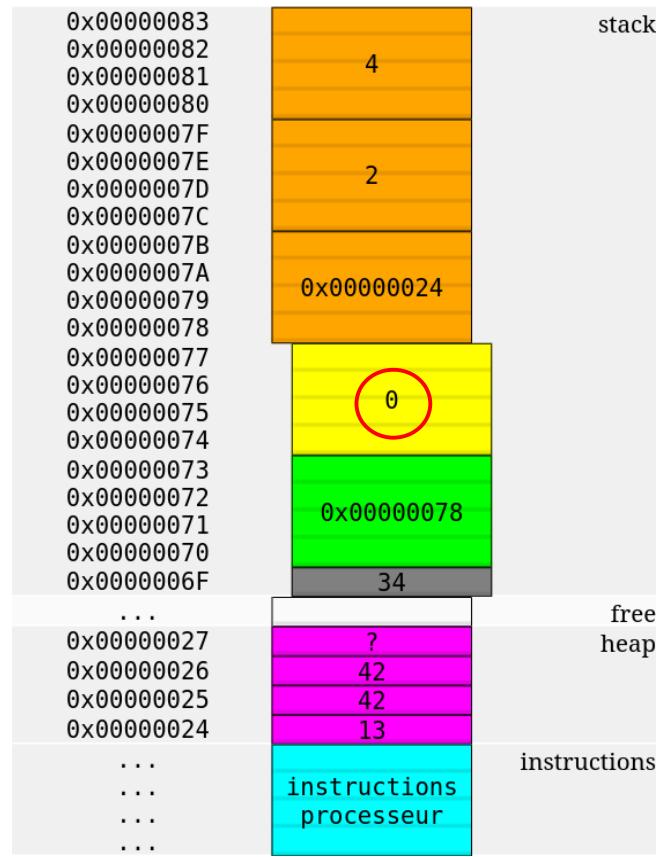
Allocation dynamique & Casting



Vectors : push back

Fin de tour, on passe à l'incrément (ici décrément), et on test la condition, qui est toujours vraie, **on continue !**

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i = list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



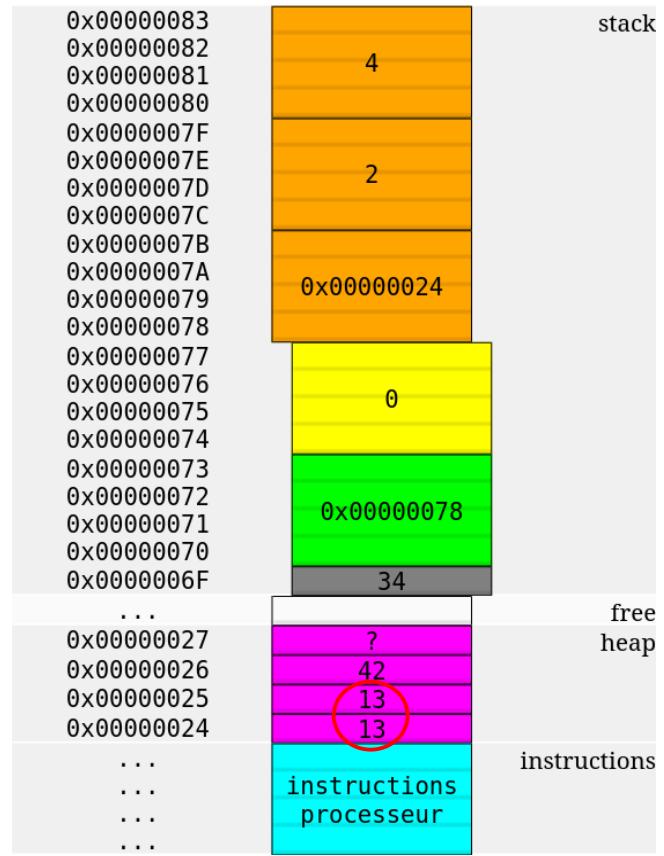
Allocation dynamique & Casting



Vectors : push front

On assigne (copie) la valeur de **array[0]** dans **array[1]**

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list_push_back(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i = list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



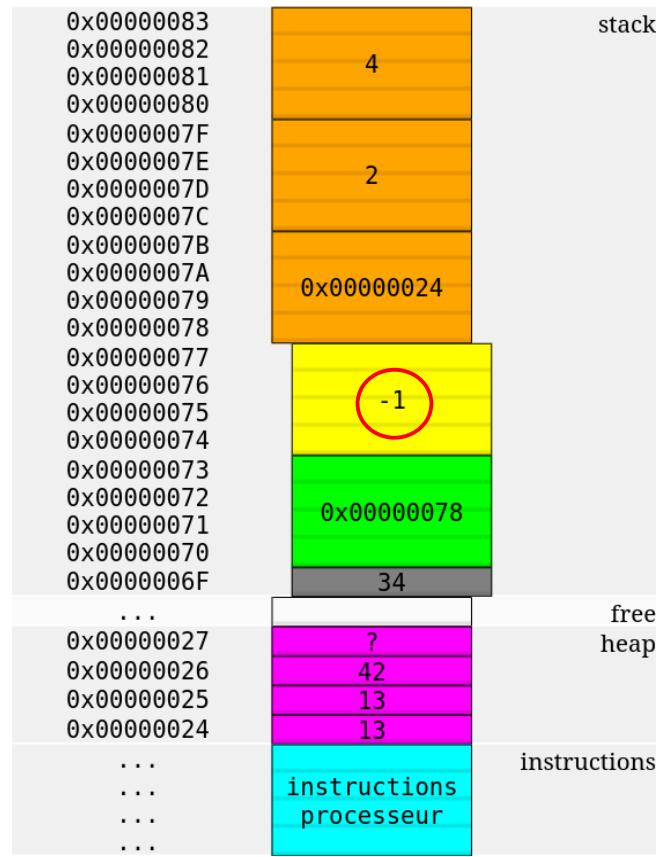
Allocation dynamique & Casting



Vectors : push front

Fin de tour, on passe à l'incrément (ici décrément), et on test la condition, qui est maintenant fausse, **on stop !**

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list push back char(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i = list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



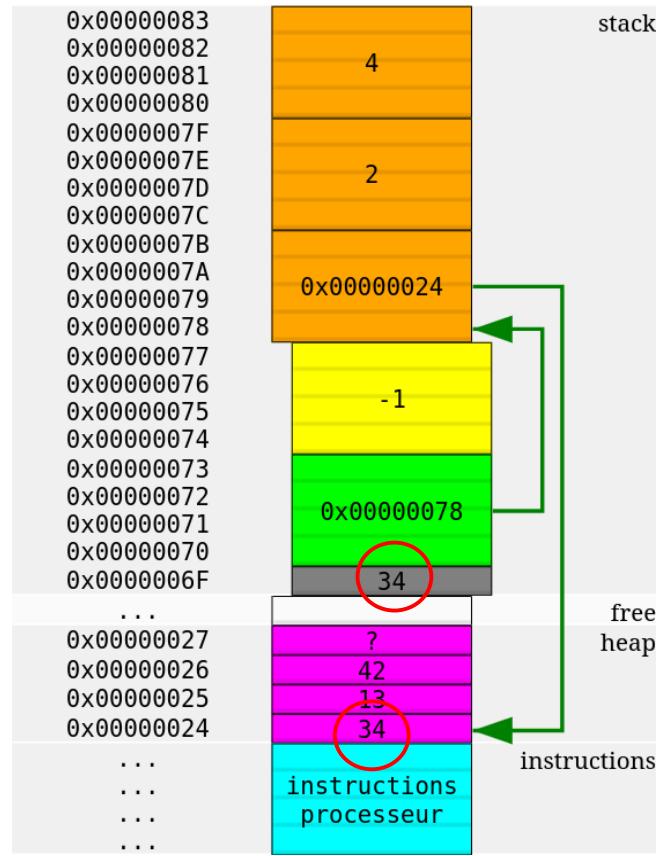
Allocation dynamique & Casting



Vectors : push front

On remplace la première valeur par **elem** (par copie) !

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list push back char(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i = list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



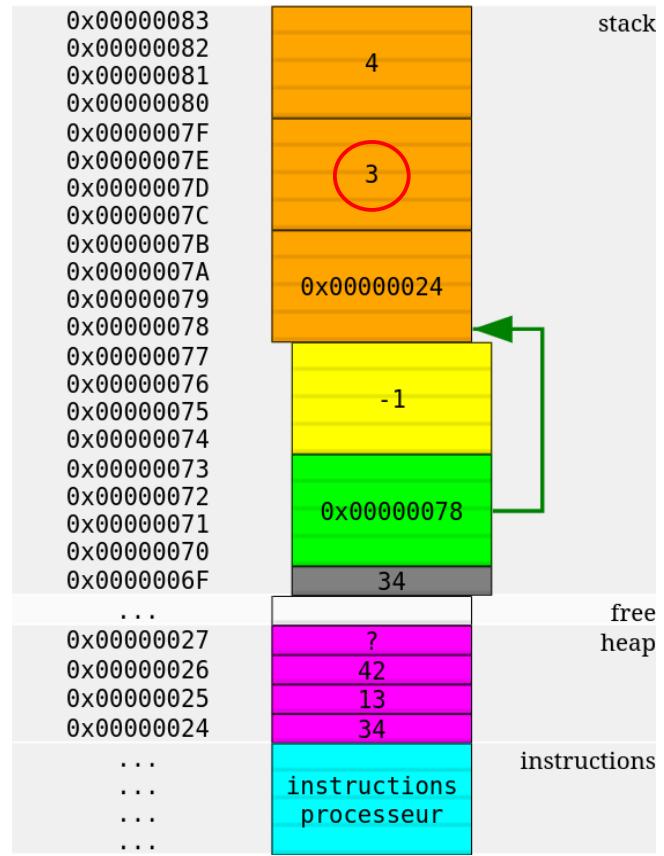
Allocation dynamique & Casting



Vectors : push front

On met à jour la longueur de notre liste / vecteur !

```
examples > vector > C vector_push_front.c > ...
1  typedef struct {
2      TYPE *array;
3      size_t length;
4      size_t capacity;
5  } LIST_NAME(TYPE);
6
7  // Implémentation générique e.g.:
8  // list push back char(list_char *list, char elem)
9  int FUNC_NAME(list_push_front_, TYPE)(LIST_NAME(TYPE) *list, TYPE elem){
10     // Agrandir la zone dynamique si besoin
11     if (FUNC_NAME(_resize_, TYPE)(list) == 0){
12         return 0;
13     }
14     // Faire de la place !
15     for(int i = list->length - 1; i >= 0 ; i--){
16         list->array[i+1] = list->array[i];
17     }
18     list->array[0] = elem;
19     list->length += 1;
20     return 1;
21 }
```



Allocation dynamique & Casting



Pointeur de pointeur, dégradation, et tableaux 2D

Attention : même si le pointeur peut être manipulé comme un tableau (avec `ptr[i]` par exemple), et le tableau comme un pointeur (après dégradation), les types sont différents, et supposer qu'ils sont identiques de par notre expérience avec les tableaux 1D est une erreur. Par exemple :

- Un `int array[10][8]` sera dégradé en `int (*)[8]` et non `int **array`

```
int fonction(int array[10][8]) {
```

- Avec `int(*)[8]`, on a bien un pointeur vers **8 int**, et donc par arithmétique des pointeurs, avec `int[2][8]` on sera bien à l'adresse `array + 2 * sizeof(int [8])`, **les deux niveaux de données sont alignés**
- Avec un `int **array`, c'est différent, on a une adresse vers une adresse. La taille d'un élément à `array[x]` sera donc `sizeof(int *)`. Le contenu étant une adresse, on peut tout de même enchaîner avec `array[x][y]`, mais les **deux niveaux de données de ce type ne sont pas alignés il y a une indirection**

Allocation dynamique & Casting



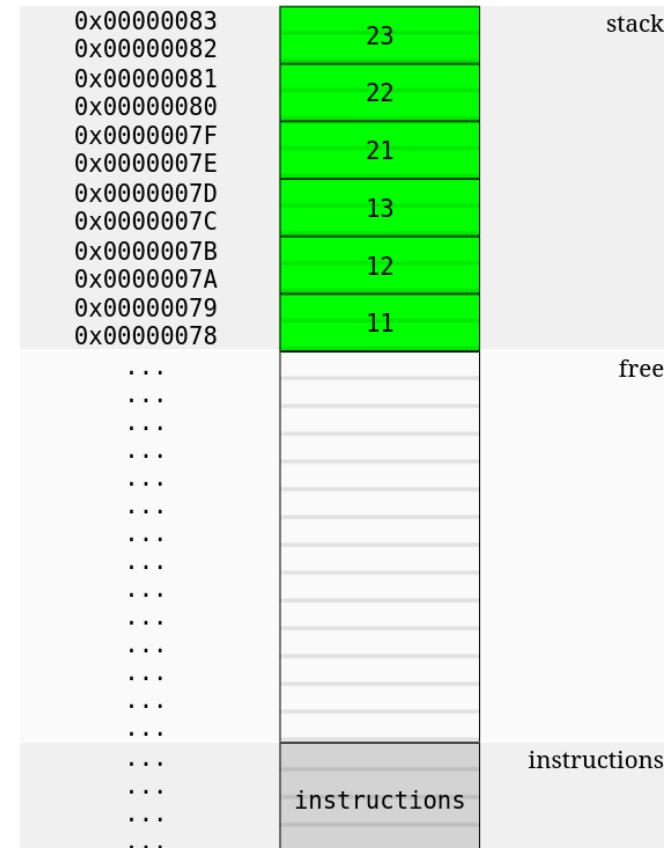
Récapitatif tableaux multidimensionnels

```
examples > C polytab.c > ...
```

```
1 int main (void) {
2     short tab[2][3] = {{11, 12, 13}, {21, 22, 23}};
3     return 0;
4 }
```

Ici **tab** n'est pas vraiment un pointeur, il s'agit d'un label pour accès à un tableau dans la pile. Pas d'indirection, le compilateur transformera l'accès `tab[x][y]`, en un accès à la case $Y * x + y$ (ici avec $X = 2$ et $Y = 3$).

Il peut être dégradé en **short (*)[3]**.



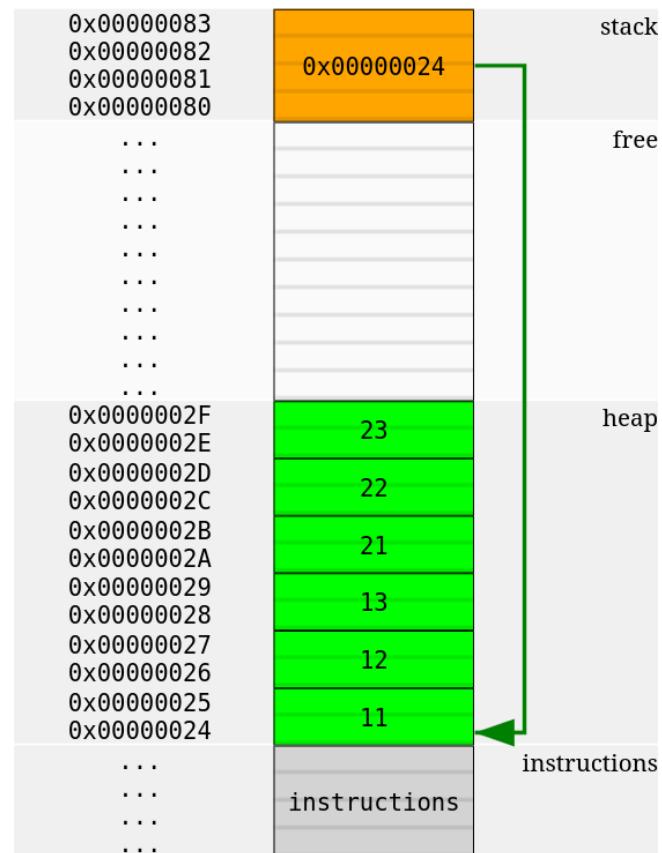
Allocation dynamique & Casting



Récapitatif tableaux multidimensionnels

```
examples > C polymalloctab.c > ...
1  #include<stdlib.h>
2
3  int main (void) {
4      short (*tab)[3] = malloc(sizeof(short[3]) * 2 );
5      tab[0][0] = 11;
6      tab[0][1] = 12;
7      tab[0][2] = 13;
8      tab[1][0] = 21;
9      tab[1][1] = 22;
10     tab[1][2] = 23;
11     return 0;
12 }
```

Ici **tab** est directement déclaré comme un **short (*)[3]**.
Un pointeur vers un **short [3]**, ou plusieurs tableaux par arithmétique des pointeurs.



⚠ Layout avec adressage sur 4 octets pour avoir assez de place

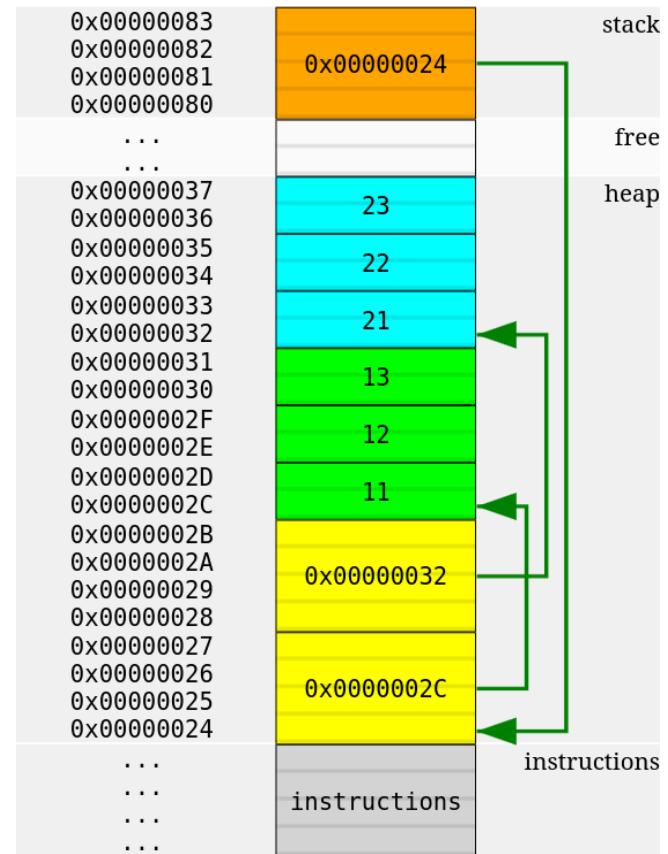
Allocation dynamique & Casting



Récapitulatif tableaux multidimensionnels

```
examples > C polymallocmalloc.c > ...
1  #include<stdlib.h>
2
3  int main (void) {
4      short **tab = malloc(sizeof(short *) * 2);
5      tab[0] = malloc(sizeof(short) * 3);
6      tab[0][0] = 11;
7      tab[0][1] = 12;
8      tab[0][2] = 13;
9      tab[1] = malloc(sizeof(short) * 3);
10     tab[1][0] = 21;
11     tab[1][1] = 22;
12     tab[1][2] = 23;
13     return 0;
14 }
```

Ici nous avons un tableau de pointeurs (adresse **0x24**), implémenté par un pointeur de pointeur (**short****) utilisé comme tableau 2D par arithmétique des pointeurs.



⚠ Layout avec adressage sur 4 octets pour avoir assez de place



Interpréteur Python

Compilation vs Interprétation



Les langages comme C :

- Compilés et la mémoire est gérée manuellement
- Les allocations et libérations de mémoire sont explicites dans le code (copies, **malloc**, **free**)
- Avantage : rapide. Inconvénient : la compilation ne garantit pas l'absence de fuites mémoires

Les langages comme Python :

- Interprétés et la mémoire est gérée "automatiquement"
- Mais rien n'est magique, la mémoire est gérée par un "garbage collector" (ramasse-miettes) qui scanne les objets pour libérer la mémoire de ceux n'étant plus **référencés**
- Avantage : simple pour l'utilisateur. Inconvénient : plus lent, moins flexible

Compilation vs Interprétation



Et le processus dans tout ça ?

En C :

- Le programme compilé pour être traduit en code machine, une suite d'instruction exécutées par le processeur.
- **La création du processus se fait en chargeant en mémoire ces instructions.**

En Python :

- **Le programme exécuté par le processus n'est pas le script python mais l'interpréteur python, écrit en C !**
- **L'exécution du script par l'interpréteur suit des principes analogues, i.e. : pile d'exécution, frames, etc.**

Et maintenant Python



Ce que fait l'interpréteur Python

- En C on utilise la pile matérielle, **Python utilise une pile logique**
- En C les frames sont des zones bien définies dans la zone mémoire de la pile, l'interpréteur Python crée **des frames logiques**, des objets qu'il empile dans sa pile logique

Particularités

- En python on ne manipule que des références : des pointeurs
- Les types primitifs comme les entiers ne peuvent PAS être modifiés (ils sont copiés)
- Toutes les données sont stockées dans la zone d'allocation dynamique
- Contrairement à C, les frames sont entièrement dynamiques. Comprendre "stockées dans le tas, et la taille d'une frame change pendant l'exécution de manière moins contrainte qu'en C"

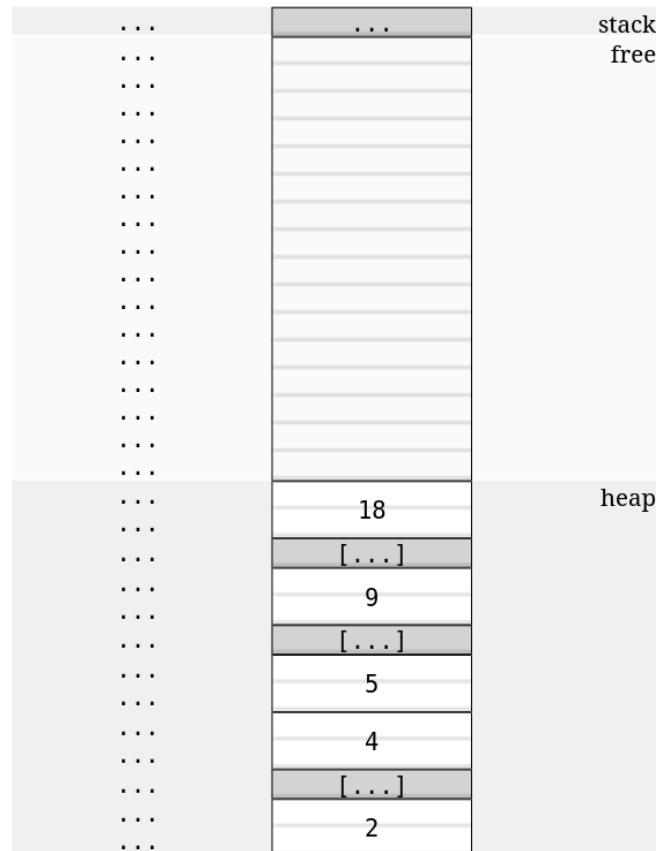
Exemple de processus Interpréte (Python)



⚠ REPRÉSENTATION

Le layout à droite n'est plus le layout "matérielle" du processus, on représente maintenant **le layout logique maintenu par l'interpréteur**

"stack" devient la pile logique, et "heap" la zone des valeurs



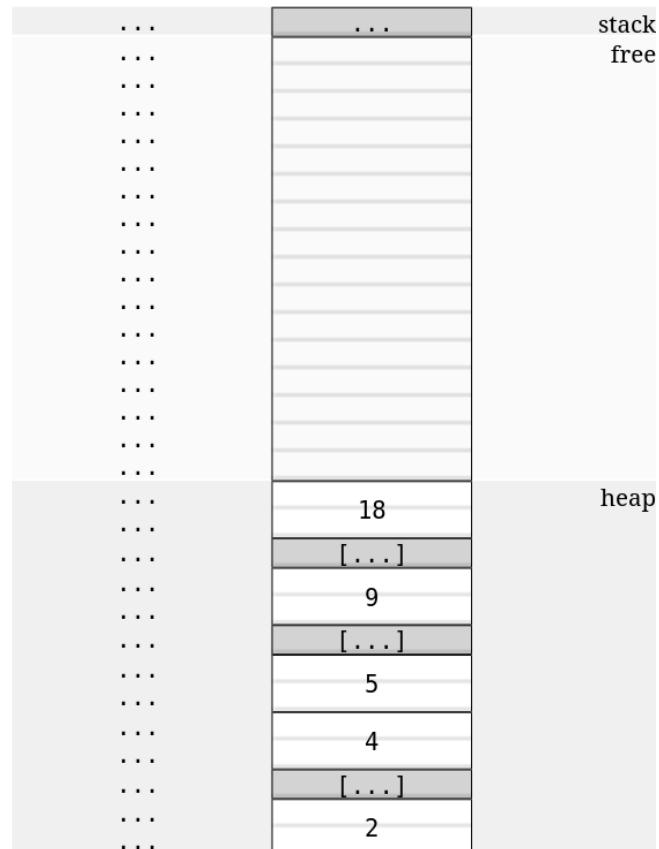
Exemple de processus Interprétré (Python)



⚠ Small Object Caching

On observe que des valeurs sont déjà présentes dans la mémoire.

Par souci d'optimisation, l'interpréteur python pré-alloue certaines valeurs pour ne pas faire d'allocation dynamique pendant l'exécution (pour les entiers les valeurs entre -5 et 256)

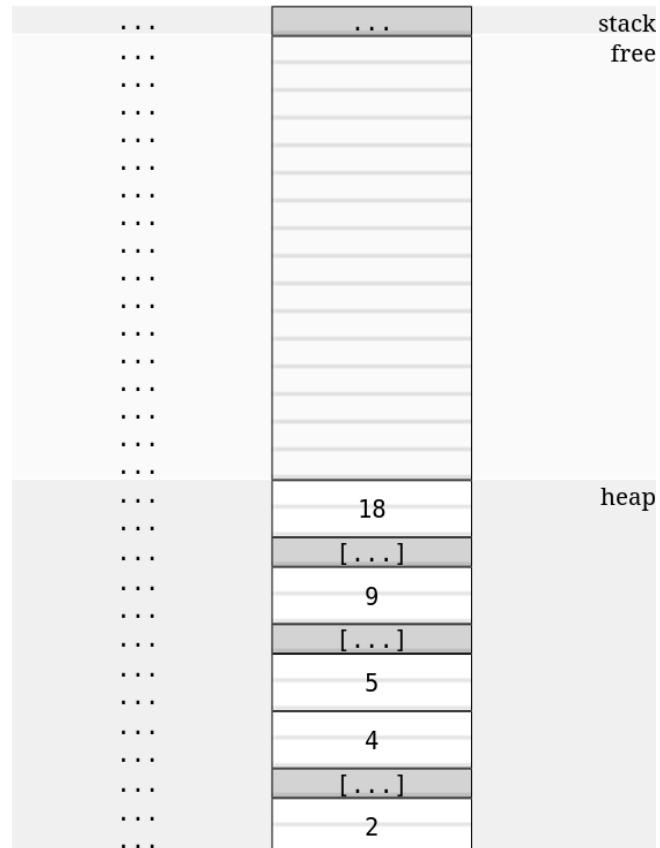


Exemple de processus Interpréte (Python)



```
1 # multiply by 2
2 def multiply(orange):
3     result = orange * 2
4     return result
5
6 # add 5, then multiply by 2
7 def add(green):
8     result = multiply(green + 5)
9     return result
10
11 def main():
12     grey = add(4)
13     print(grey) # let's ignore that call
14
15 if __name__ == "__main__":
16     main()
```

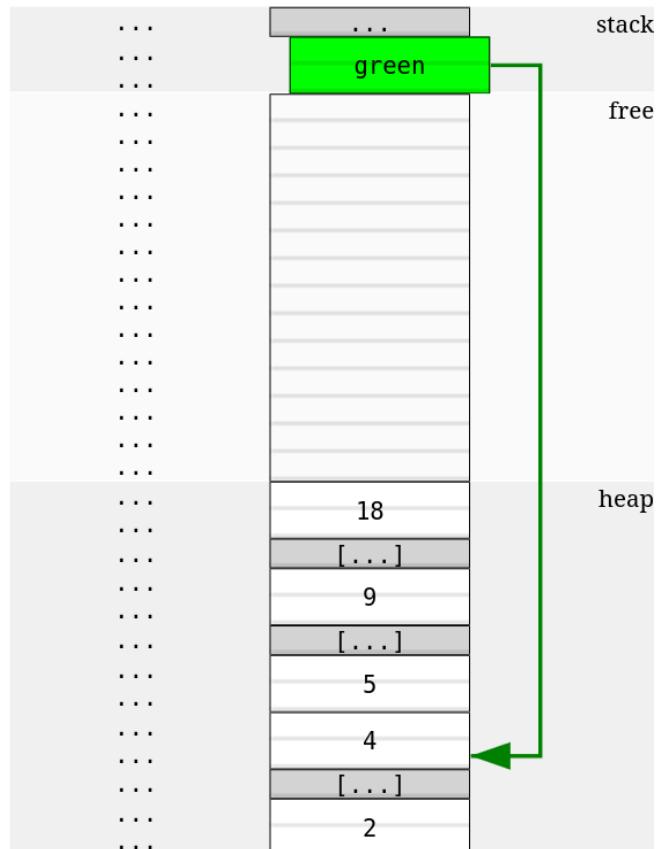
Création de la frame pour **main**, contrairement à C, elle ne contient pas encore de zone réservée pour "grey"



Exemple de processus Interpréte (Python)



```
1 # multiply by 2
2 def multiply(orange):
3     result = orange * 2
4     return result
5
6 # add 5, then multiply by 2
7 def add(green):
8     result = multiply(green + 5)
9     return result
10
11 def main():
12     grey = add(4)
13     print(grey) # let's ignore that call
14
15 if __name__ == "__main__":
16     main()
```



Pour créer **grey** il faut d'abord évaluer l'appel à
add : création de la frame pour **add**

Python nous ne manipulons que des références
vers des objets, **green** pointe donc vers un
PyLongObject valant 4 ("Long" pour entier long)

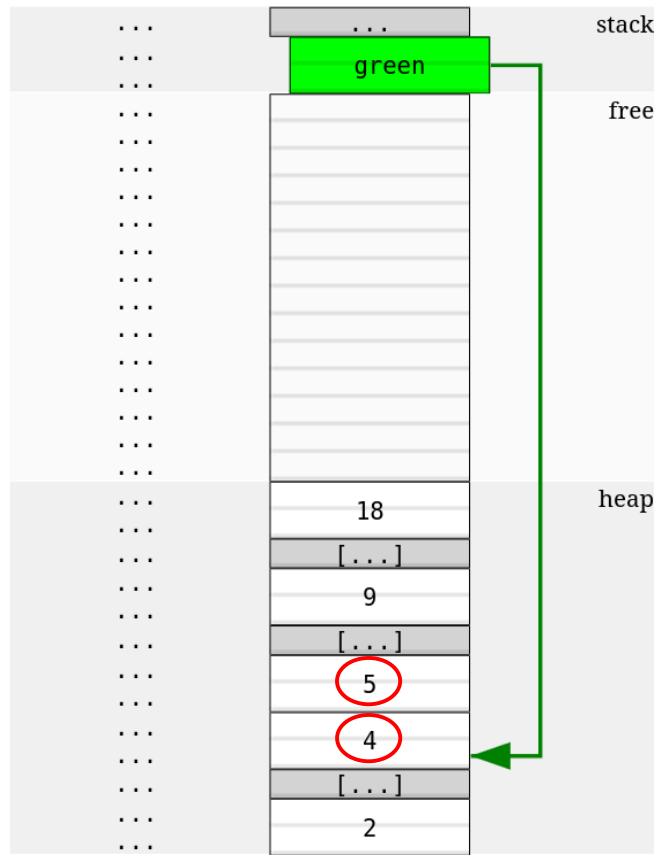
Exemple de processus Interprété (Python)



```
1 # multiply by 2
2 def multiply(orange):
3     result = orange * 2
4     return result
5
6 # add 5, then multiply by 2
7 def add(green):
8     result = multiply(green + 5)
9     return result
10
11 def main():
12     grey = add(4)
13     print(grey) # let's ignore that call
14
15 if __name__ == "__main__":
16     main()
```

Pour créer **result** dans **add** il faut d'abord évaluer l'appel à **multiply** : création de la frame pour **multiply**

L'interpréteur doit faire l'addition (par deref) entre deux **PyLongObject**. Il récupère une référence vers le **PyLongObject** valant 5



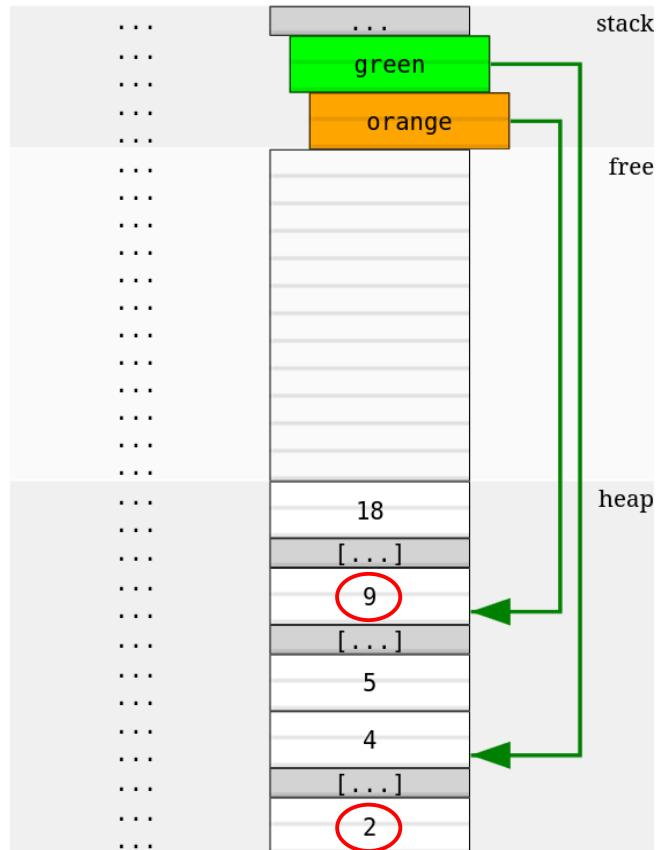
Exemple de processus Interpréte (Python)



```
1 # multiply by 2
2 def multiply(orange):
3     result = orange * 2
4     return result
5
6 # add 5, then multiply by 2
7 def add(green):
8     result = multiply(green + 5)
9     return result
10
11 def main():
12     grey = add(4)
13     print(grey) # let's ignore that call
14
15 if __name__ == "__main__":
16     main()
```

Pour créer **result** dans **multiply** il faut d'abord évaluer l'opération

L'interpréteur fait la multiplication (par deref) entre deux **PyLongObject**. Il récupère une référence vers le **PyLongObject** valant 2

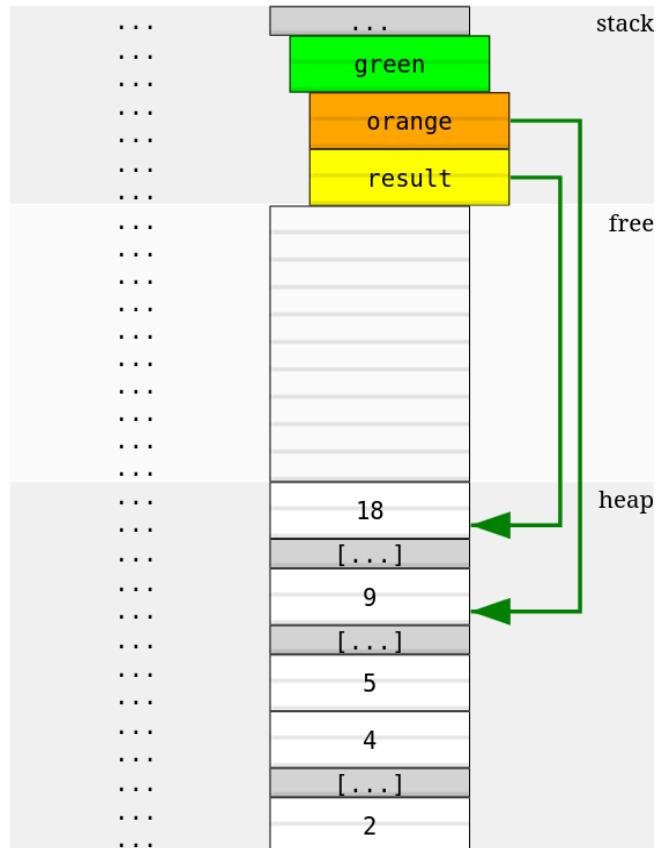


Exemple de processus Interpréte (Python)



```
1 # multiply by 2
2 def multiply(orange):
3     result = orange * 2
4     return result
5
6 # add 5, then multiply by 2
7 def add(green):
8     result = multiply(green + 5)
9     return result
10
11 def main():
12     grey = add(4)
13     print(grey) # let's ignore that call
14
15 if __name__ == "__main__":
16     main()
```

Mise à jour de la frame de l'appel à `multiply` avec la création de la variable `result`

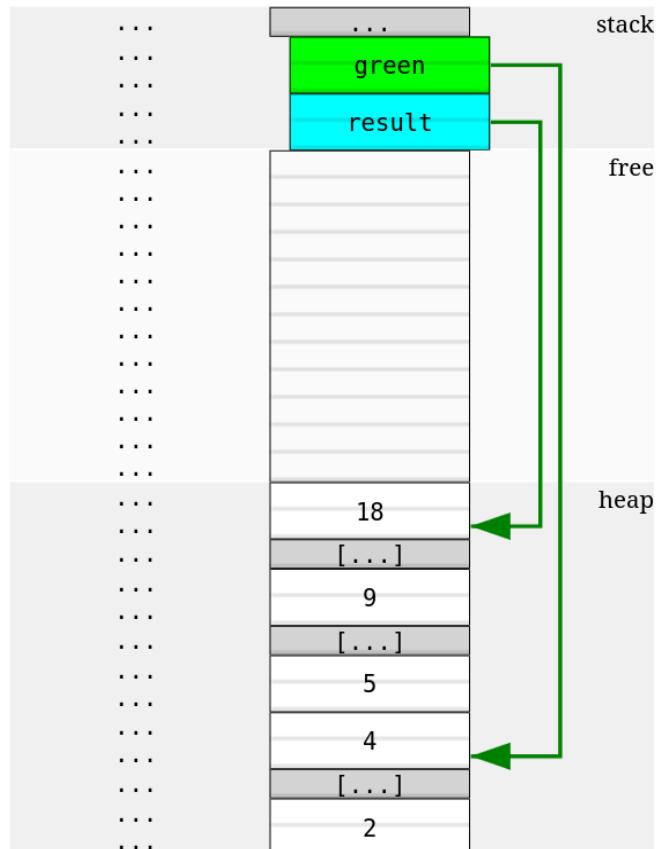


Exemple de processus Interpréte (Python)



```
1 # multiply by 2
2 def multiply(orange):
3     result = orange * 2
4     return result
5
6 # add 5, then multiply by 2
7 def add(green):
8     result = multiply(green + 5)
9     return result
10
11 def main():
12     grey = add(4)
13     print(grey) # let's ignore that call
14
15 if __name__ == "__main__":
16     main()
```

multiply donne en retour la référence vers le **PyLongObject** valant 18, ce qui permet de mettre à jour la frame de **add** avec la variable **result**



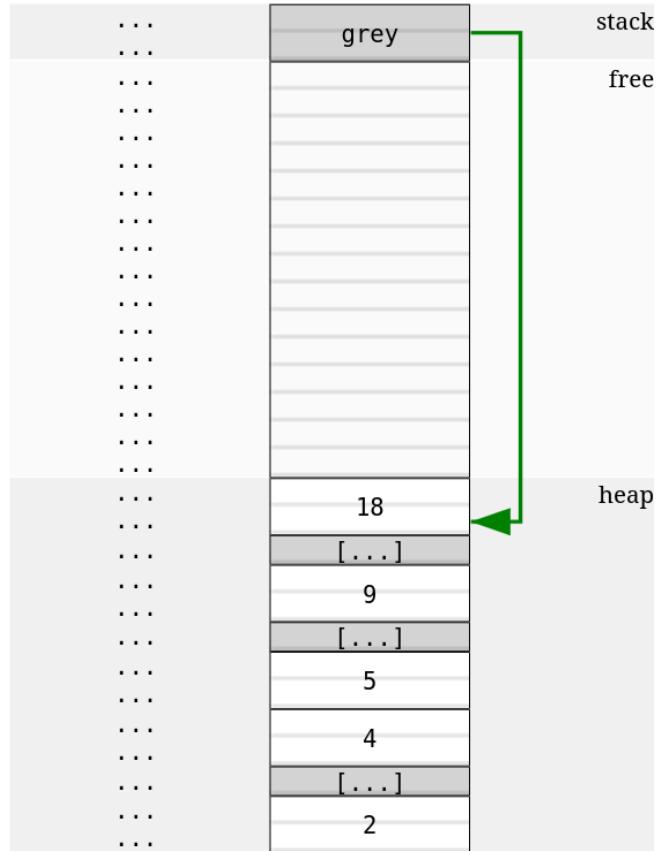
Exemple de processus Interpréte (Python)



```
1 # multiply by 2
2 def multiply(orange):
3     result = orange * 2
4     return result
5
6 # add 5, then multiply by 2
7 def add(green):
8     result = multiply(green + 5)
9     return result
10
11 def main():
12     grey = add(4)
13     print(grey) # let's ignore that call
14
15 if __name__ == "__main__":
16     main()
```

add donne en retour la référence vers le **PyLongObject** valant 18, ce qui permet de mettre à jour la frame de **main** avec la variable **grey**

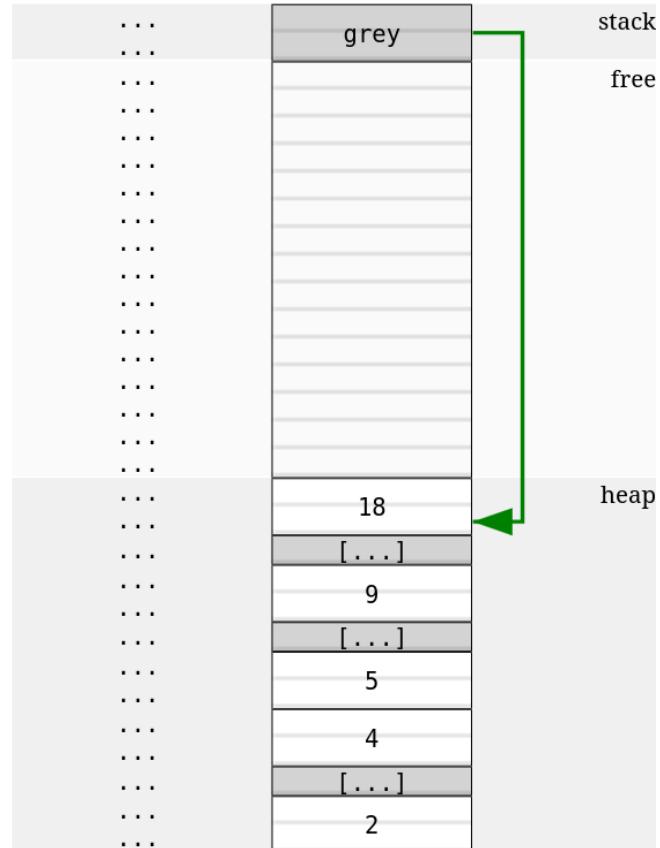
Ensuite on affiche (ligne 13) et le processus de l'interpréteur se termine peu après



Exemple de processus Interpréte (Python)



```
1 # multiply by 2
2 def multiply(orange):
3     result = orange * 2
4     return result
5
6 # add 5, then multiply by 2
7 def add(green):
8     result = multiply(green + 5)
9     return result
10
11 def main():
12     grey = add(4)
13     print(grey) # let's ignore that call
14
15 if __name__ == "__main__":
16     main()
```



Et si un entier est très grand ? (e.g. val = 625)

On ne peut pas se contenter d'une référence vers le cache, il y a donc allocation dynamique pour créer un PyLongObject

Exemple de processus Interpréte (Python)



```
(base) → ~ ipython
Python 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:12:24) [GCC 11.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.28.0 -- An enhanced Interactive Python. Type '?' for help.
```

```
In [1]: v1 = 42
```

**Id() donne l'identifiant de l'objet pointé
(l'adresse mémoire)**

```
In [3]: v3 = 2077
```

**42 est petit et fait partie du cache. Comme les
entiers sont non mutable, deux références
vers 42 n'est pas un problème.**

```
In [4]: v4 = 2077
```

```
In [5]: id(v1)
Out[5]: 9773128
```

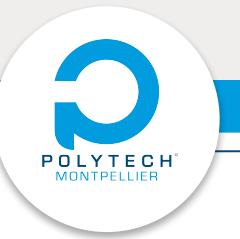
```
In [6]: id(v2)
Out[6]: 9773128
```

**2077 est supérieur à 256, python créera un
PyLongObject pour les deux variables**

```
In [7]: id(v3)
Out[7]: 124003829513968
```

**Les objets v3 et v4 seront libérés (avec free()) par
le ramasse-miettes quand ils ne seront plus
référencés)**

```
In [8]: id(v4)
Out[8]: 124003829513264
```



Préprocesseur & VLAs

Préprocesseur & VLAs



Statique ou dynamique ?

Nous avons pour l'instant couvert 2 paradigmes, on peut définir des tableaux de taille fixe (dans la pile) :

(1) : int array[10];

Et des tableaux de taille variable (dans le tas) :

(2) : int *array = malloc(10 * sizeof(int)); // Peut être realloc

Dans certains cas, on ne connaît pas la taille du tableau assez tôt pour utiliser un tableau statique **(1)**, mais, sachant qu'elle restera fixe, est-on malgré tout obligé de recourir à une allocation dynamique **(2)** ?

Préprocesseur & VLAs



Option 1 : le préprocesseur

Nous l'avons déjà utilisé le préprocesseur sans le nommer. Effectivement **#include<stdio.h>** et **#include<stdlib.h>** sont des **directives** du préprocesseur permettant d'ajouter les fichiers de headers (ici pour dire au compilateur quelles sont les fonctions disponibles), et ce **avant la compilation**

Un cas particulier de directive est l'utilisation de **macros**, cela permet, avant la compilation, de définir des constantes (e.g. **MY_CONST**) ou de générer du code (e.g. **SQUARE**).

⚠ Le préprocesseur est très peu évolué et s'apparente à du rechercher/remplacer !

La génération de code ressemble à la syntaxe des fonctions, mais on reste sur du rechercher/remplacer !

```
examples > c preprocess.c > main()
1 #include <stdio.h>
2
3 #define MY_CONST 42
4 #define SQUARE(x) x * x
5
6 int main () {
7     int a = 3;
8
9     int b = SQUARE(a + 1); // a + 1 * a + 1
10    printf("b vaut %d\n", b);
11
12    int c = SQUARE((a + 1)); // (a + 1) * (a + 1)
13    printf("c vaut %d\n", c);
14
15    // défini dans le fichier
16    printf("MY_CONST vaut %d\n", MY_CONST);
17    // passé en argument de gcc
18    printf("OTHER_CONST vaut %d\n", OTHER_CONST);
19    return 0;
20 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) ➔ examples git:(main) ✘ gcc preprocess.c -DOTHER_CONST=34 -o preprocessor
- (base) ➔ examples git:(main) ✘ ./preprocessor

```
b vaut 7
c vaut 16
MY_CONST vaut 42
OTHER_CONST vaut 34
○ (base) ➔ examples git:(main) ✘
```

Préprocesseur & VLAs



Option 1 : le préprocesseur

- Les directives du préprocesseur offrent un certain degré de logique conditionnelle avec :
- **#if / #else / #elif / #endif** (la condition est vraie si différente de 0)
- **#ifdef / #ifndef** pour "if (not) defined" sont des conditionnelles permettant de vérifier si une macro est définie ou non
- **#undef** permet de supprimer une macro

Ces directives permettent un contrôle plus fin du comportement avec l'option -D de gcc

A screenshot of a code editor showing a C program named 'preproc.c'. The code defines a function 'main' that prints the value of 'VAL' based on its sign. It uses '#ifndef' to check if 'VAL' is defined, and '#if' and '#else' blocks to handle positive and negative values. The code editor interface includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. Below the code, a terminal window shows the execution of the program with different '-DVAL' arguments, demonstrating the conditional compilation logic.

```
examples > c preproc.c > main()
1  #include <stdio.h>
2
3  int main () {
4
5      #ifndef VAL
6          printf("-DVAL=<val> required\n");
7      #else
8          #if VAL > 0
9              printf("VAL est positif\n");
10         #elif VAL < 0
11             printf("VAL est négatif\n");
12         #else
13             printf("VAL vaut 0\n");
14         #endif
15     #endif
16
17     return 0;
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ gcc preproc.c -o preproc
- (base) → examples git:(main) ✘ ./preproc
-DVAL=<val> required
- (base) → examples git:(main) ✘ gcc preproc.c -DVAL=2 -o preproc
- (base) → examples git:(main) ✘ ./preproc
VAL est positif
- (base) → examples git:(main) ✘ gcc preproc.c -DVAL=-4 -o preproc
- (base) → examples git:(main) ✘ ./preproc
VAL est négatif
- (base) → examples git:(main) ✘ gcc preproc.c -DVAL=0 -o preproc
- (base) → examples git:(main) ✘ ./preproc
VAL vaut 0
- (base) → examples git:(main) ✘

Préprocesseur & VLAs



Option 1 : le préprocesseur

De retour sur notre problème de tableau

- Une macro définissant une constante est tout à fait éligible pour définir une taille de tableau
- Il faut donc recompiler pour changer la taille
- La directive **#ifndef** peut nous permettre de définir une taille par défaut

→ Ici on remplace COUNT par 10 (ou 3 dans l'exemple avec -D), il n'y a rien de plus à dire concernant le layout du processus.

```
examples > C scanf.c > ...
1  #include <stdio.h>
2
3  // C preprocessor
4  #ifndef COUNT
5  #define COUNT 10
6  #endif
7
8  int main(void) {
9
10    int array[COUNT] = {0};
11    for (int i = 0; i < COUNT ; i++){
12      scanf("%d", &array[i]);
13    }
14    for (int i = 0; i < COUNT ; i++){
15      printf("%d ", array[i]);
16    }
17    printf("\n");
18  }
19 }
```

PROBLEMS OUTPUT TERMINAL ... zsh - examples + -

- (base) → examples git:(main) ✘ gcc -DCOUNT=3 scanf.c -o scanf
- (base) → examples git:(main) ✘ ./scanf

```
123
-2
9
123 -2 9
```

- (base) → examples git:(main) ✘

Préprocesseur & VLAs



Option 2 : VLAs (Variable-Length Arrays)

- La taille d'un array peut être définie par variable, la taille de la frame est adaptée à l'exécution, mais **la taille du tableau reste fixe**. Il s'agit d'un Variable-Length Array (VLA)
- Dans l'exemple à droite, on utilise le nombre d'arguments (**argc**) passé en ligne de commande comme taille du tableau. **Ceci est juste pour l'exemple** pour éviter de coder la gestion des erreurs si on veut récupérer la taille dans **argv**.
- L'option **-Wvla** informe sur l'utilisation des VLAs.

→ **Avec cette méthode, la taille de frame doit être adaptée, cela peut expliquer la position des VLAs, à savoir après les arguments (au sommet de pile)**

```
examples > C scanf_argv.c > ...
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4      // don't do that
5      int array[argc];
6      for (int i = 0; i < argc ; i++){
7          scanf("%d", &array[i]);
8      }
9      printf(" ---\ninputs :\n");
10     for (int i = 0; i < argc ; i++){
11         printf("%d ", array[i]);
12     }
13     printf("\n");
14 }
15 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ ./scanf_args dummy dummy dummy
123
321
-23
42

inputs :
123 321 -23 42
- (base) → examples git:(main) ✘

Préprocesseur & VLAs

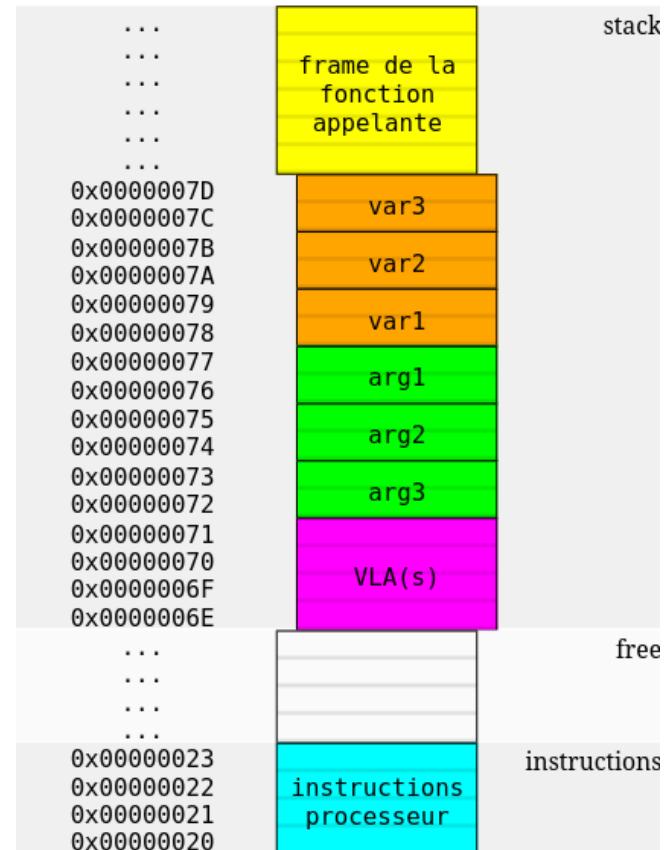


Option 2 : VLAs (Variable-Length Arrays)

```
examples > c vla_layout.c > function(short, short, short)
1  #include<stdio.h>
2
3  void function(short arg1, short arg2, short arg3) {
4      short var1;
5      short var2;
6      char vla[arg1]; // position in code won't affect the layout
7      short var3;
8
9
10     printf("var3: %p (%ld)\n", &var3, (long) &var3);
11     printf("var2: %p (%ld)\n", &var2, (long) &var2);
12     printf("var1: %p (%ld)\n", &var1, (long) &var1);
13     printf("arg1: %p (%ld)\n", &arg1, (long) &arg1);
14     printf("arg2: %p (%ld)\n", &arg2, (long) &arg2);
15     printf("arg3: %p (%ld)\n", &arg3, (long) &arg3);
16     printf("vla: %p (%ld)\n", &vla[0], (long) &vla[0]);
17 }
18
19 void main(void){
20     function(1, 2, 3);
21 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ ./vla_layout
var3: 0x7ffd8787e776 (140726877284214)
var2: 0x7ffd8787e774 (140726877284212)
var1: 0x7ffd8787e772 (140726877284210)
arg1: 0x7ffd8787e76c (140726877284204)
arg2: 0x7ffd8787e768 (140726877284200)
arg3: 0x7ffd8787e764 (140726877284196)
vla: 0x7ffd8787e750 (140726877284176)





Enums, Labels, Switches & Unions

Enums, Labels, Switchs & Unions



```
examples > C enum.c > ⌂ compute(int)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  enum status {
5      ALLOCERROR,
6      PROCESSERROR,
7      SUCCESS = 42,
8  };
9
10 enum status compute(int index) {
11     int *tab = malloc(100 * sizeof(int));
12     if (tab == 0) {
13         return ALLOCERROR;
14     }
15     if (index >= 100) {
16         return PROCESSERROR;
17     }
18     // < Do fancy stuff >
19     free(tab);
20     return SUCCESS;
21 }
22
23 int main(void){
24     enum status res = compute(32);
25     printf("status : %d\n", res);
26     if (res == SUCCESS){
27         return EXIT_SUCCESS;
28     }
29     return EXIT_FAILURE;
30 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

- (base) → examples git:(main) ✘ gcc enum.c -o enum
- (base) → examples git:(main) ✘ ./enum
- status : 42
- (base) → examples git:(main) ✘

Énumérations

Il est très classique d'associer un sens à des valeurs numériques. Exemple classique avec **EAST**, **WEST**, **NORTH**, **SOUTH**. Plutôt que multiplier les macros, on peut mettre en place des énumérations. **Une enum est un alias de int signé**

La syntaxe de déclaration est similaire aux structures, mais le corps présente une liste de membres non typés (généralement en majuscule) séparés par des virgules. Les valeurs sont entières et signées (ici sur 4 octets), et peuvent être indiquées à la déclaration.

Si la valeur du premier membre n'est pas renseignée, le compilateur lui assignera la valeur **0**. Si les suivantes ne sont pas renseignées, elles prendront la valeur de la précédente + 1.

Deux membres peuvent avoir la même valeur.

L'utilité des enums en C est limitée. Elles sont plus évoluées en Swift ou Rust

Enums, Labels, Switchs & Unions



```
examples > C switch.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  enum direction {
5      NORTH,
6      SOUTH,
7      EAST,
8      WEST
9  };
10
11 int main(void) {
12     enum direction dir = EAST;
13
14     switch (dir) {
15         case NORTH:
16             printf("Moving NORTH\n");
17             break;
18         case SOUTH:
19             printf("Moving SOUTH\n");
20             break;
21         case EAST:
22             printf("Moving EAST\n");
23             break;
24         case WEST:
25             printf("Moving WEST\n");
26             break;
27         default:
28             printf("Invalid direction!\n");
29             return EXIT_FAILURE;
30     }
31     return EXIT_SUCCESS;
32 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

```
● (base) → examples git:(main) ✘ gcc switch.c -o switch
● (base) → examples git:(main) ✘ ./switch
Moving EAST
○ (base) → examples git:(main) ✘
```

Switch case

Maintenant que nous avons des énumérations pouvant traduire des états, ou des options d'un même type, il est important de pouvoir les traiter, et le **switch case** permet d'éviter les enchaînements de conditionnelles :

- On commence avec le mot clef **switch** et la valeur à traiter entre parenthèses
- On enchaîne avec une série de labels commençant par **case** suivi d'une option possible
- Les instructions suivantes seront exécutées si la valeur testée matche l'option.
- On peut ajouter un traitement par defaut si la valeur n'est pas matché avec le label **default** (c'est optionnel)



Particularité du switch case :

- On "sauve" au **case** correspondant, il ne faut pas voir un **case** comme un test, mais plutôt comme un **label** (see goto). Par exemple, en l'absence de **break** ligne 23, "Moving WEST" serait aussi affiché, **break** set à stopper la propagation.

Enums, Labels, Switchs & Unions



```
1 #include <stdlib.h>
2
3 struct engine{};
4 struct wheels{};
5 struct car {
6     struct engine *engine;
7     struct wheels *wheels;
8 };
9
10 struct car *new_car() {
11     struct car *car = malloc(sizeof(struct car));
12     if (car == NULL) {
13         goto car_failed;
14     }
15     struct engine *engine = malloc(sizeof(struct engine));
16     if (engine == NULL) {
17         goto engine_failed;
18     }
19     struct wheels *wheels = malloc(sizeof(struct wheels));
20     if (wheels == NULL) {
21         goto wheels_failed;
22     }
23     goto build;
24 }
```

Plus on échoue tôt, plus on saute loin (car moins de choses à nettoyer), les labels, comme lors des switch, ne bloque pas la propagation.

Mais peut-on faire plus simple ? (oui)

Return et label

On peut parfois entendre que c'est une mauvaise pratique d'avoir plusieurs **return** dans une fonction, qu'il ne devrait en avoir qu'un à la fin du bloc. Ceci peut être justifié par des oubli de libération de mémoire. On peut donc créer des labels et sauter à ces labels avec **goto**

```
25
26     wheels_failed:
27         free(engine);
28     engine_failed:
29         free(car);
30     car_failed:
31         return NULL;
32     build:
33         car->engine = engine;
34         car->wheels = wheels;
35         return car;
36     }
37
38     int main(void) {
39         struct car *car = new_car();
40         if (car) {
41             free(car->engine);
42             free(car->wheels);
43             free(car);
44             return EXIT_SUCCESS;
45         }
46         return EXIT_FAILURE;
47     }
```

Enums, Labels, Switchs & Unions



```
10 struct car *new_car() {
11     struct car *car = NULL;
12     struct engine *engine = NULL;
13     struct wheels *wheels = NULL;
14
15     car = malloc(sizeof(*car));
16     if (!car) goto fail;
17
18     engine = malloc(sizeof(*engine));
19     if (!engine) goto fail;
20
21     wheels = malloc(sizeof(*wheels));
22     if (!wheels) goto fail;
23
24     car->engine = engine;
25     car->wheels = wheels;
26     return car;
27
28 fail:
29     free(wheels);
30     free(engine);
31     free(car);
32     return NULL;
33 }
```

Initialisation des variables

Certains langages forcent le développeur à initialiser les variables (comme Rust), ceci permet d'éviter certaines erreurs

Dans le cas de notre gestion d'erreur, initialiser les variables peut nous permettre de limiter le nombre de labels. En effet, **le free sur un pointeur NULL est bien défini : il n'a aucun effet**

Pointeurs comme valeurs, il est bon de toujours initialiser vos variables

En cas d'erreur de programmation, un pointeur **NULL** fera crash votre programme plus rapidement, et vous permettra de détecter le bug plus rapidement

Enums, Labels, Switchs & Unions



```
examples > C loop_goto.c ...
1  #include <stdlib.h>
2  #include <stdio.h>
3  #define SIZE 10
4
5  int main(void) {
6      int tab[SIZE] = {12, 7, 32, 13, 45, 123, 545, 23, 18, 8};
7      int i = 0;
8
9      cond:
10     if ( ! (i < SIZE)) {
11         goto fin; // fin de boucle
12     }
13     if (tab[i] % 2 == 1) {
14         goto ite; // continue
15     }
16     printf("%d ", tab[i]);
17     if (tab[i] == 23) {
18         goto fin; // break
19     }
20     ite:
21     i++;
22     goto cond; // fin d'itération
23
24     fin:
25     printf("\n");
26     return EXIT_SUCCESS;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
• (base) → examples git:(main) ✘ gcc loop_goto.c -o loop_goto
• (base) → examples git:(main) ✘ ./loop_goto
12 32 18 8
• (base) → examples git:(main) ✘
```

Pourquoi éviter le goto ?

Le goto est très bas niveau. Un bon exemple est l'implémentation d'une boucle avec **goto** : le code assembleur correspondant restera très proche de sa version en C

L'utilisation de **goto**, sauf dans des cas très particuliers, **nuit à la lisibilité de code**, il a aussi mauvaise pub depuis l'incident "goto fail" d'Apple

Par exemple : il est déconseillé d'implémenter une boucle selon le schéma présenté à gauche. Bien que l'exercice conserve un intérêt pédagogique

→ Cet exemple doit faire écho à la partie assembleur du cours de système

Enums, Labels, Switchs & Unions



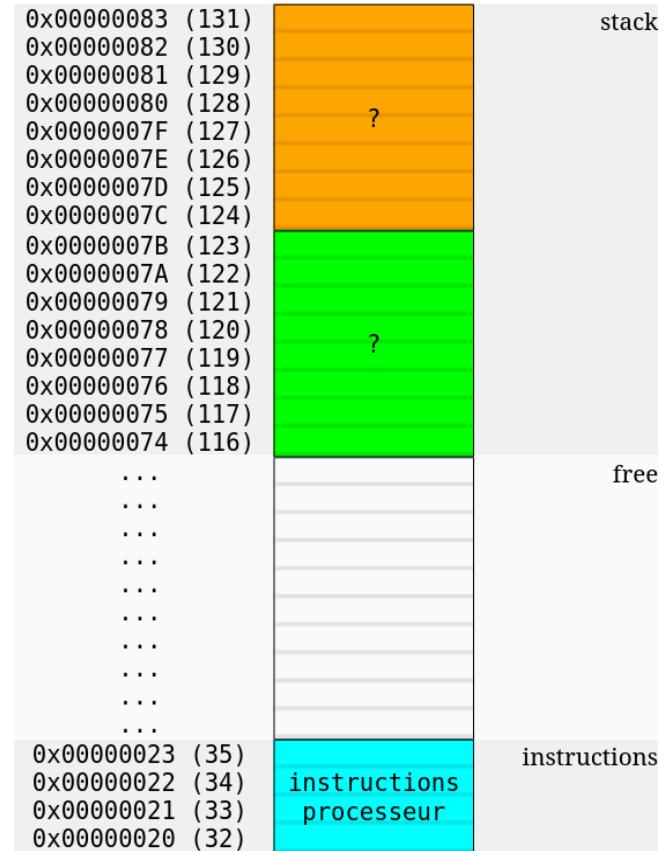
Qu'est-ce qu'une union ?

1) Une union est un **type** de données qui permet de stocker plusieurs types dans le **même** espace mémoire

```
examples > C union.c > ...
1  #include <stdio.h>
2
3  union double_or_uint {
4      double doubl;
5      unsigned int uint;
6  };
7
8  int main (void){
9      union double_or_uint green = { .doubl = 3.14 };
10     printf("green.d = %f\n", green.doubl);
11
12     union double_or_uint orange = { .uint = 2077 };
13     printf("orange.d = %u\n", orange.uint);
14
15 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) gcc union.c -o union
- (base) → examples git:(main) ./union



Enums, Labels, Switchs & Unions



Qu'est-ce qu'une union ?

2) Logiquement, une union ne stocke qu'un **seul** de ses membres à la fois

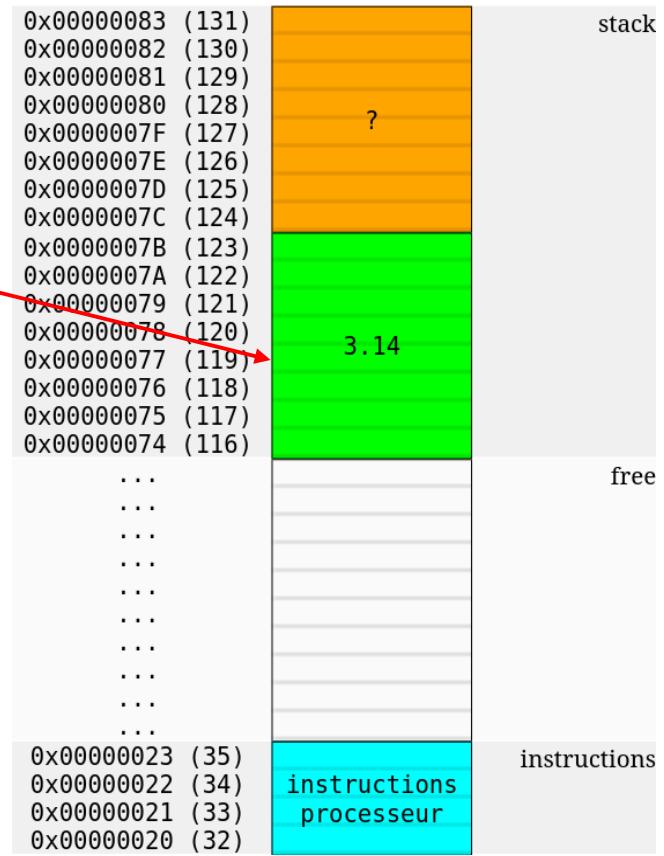
```
examples > C union.c > ...
1  #include <stdio.h>
2
3  union double_or_uint {
4      double doubl;
5      unsigned int uint;
6  };
7
8  int main (void){
9     union double_or_uint green = { .doubl = 3.14 };
10    printf("green.d = %f\n", green.doubl);
11    →
12    union double_or_uint orange = { .uint = 2077 };
13    printf("orange.d = %u\n", orange.uint);
14    return 0;
15 }
```

Il n'y a plus de place pour "uint"

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) gcc union.c -o union
- (base) → examples git:(main) ./union

green.d = 3.140000



Enums, Labels, Switchs & Unions

Qu'est-ce qu'une union ?

2) La **taille** de l'union est la taille du membre **le plus grand**, ici double (8 octets) > unsigned int (4 octets)

```
examples > C union.c > ...
1  #include <stdio.h>
2
3  union double_or_uint {
4      double doubl;
5      unsigned int uint;
6  };
7
8  int main (void){
9      union double_or_uint green = { .doubl = 3.14 };
10     printf("green.d = %f\n", green.doubl);
11
12     union double_or_uint orange = { .uint = 2077 };
13     printf("orange.d = %u\n", orange.uint);
14     return 0;
15 }
```

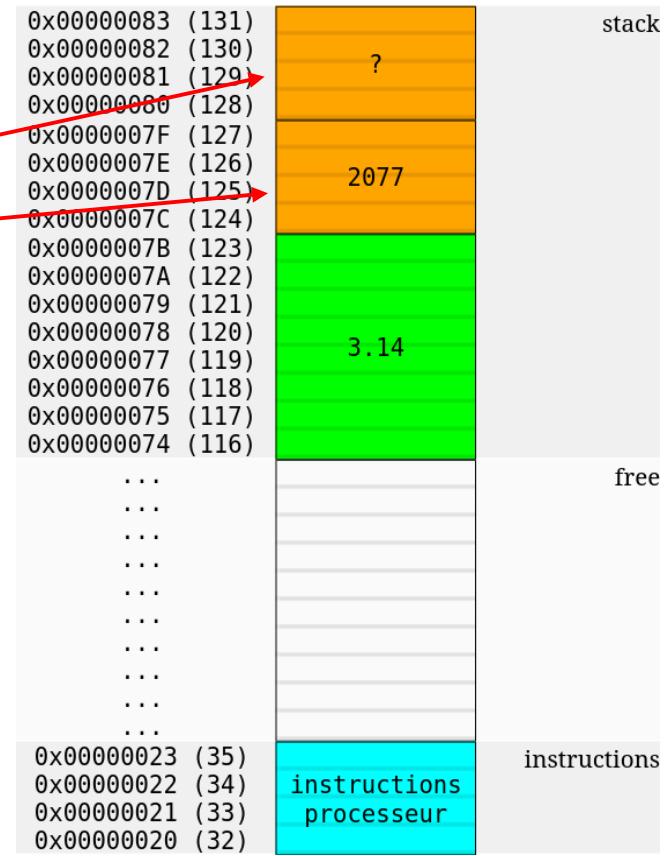
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) gcc union.c -o union
- (base) → examples git:(main) ./union

green.d = 3.140000
orange.d = 2077

- (base) → examples git:(main)

Si on stocke un
unsigned int, la taille
reste celle d'un
double (certains
octets restent
inutilisés)



Enums, Labels, Switchs & Unions



```
examples > C option.c > ...
1  #include <stdio.h>
2
3  enum Label { None, Double, Integer };
4
5  struct OptionValue {
6      enum Label label;
7      union { // anonymous to bypass union indirection
8          double d;
9          unsigned int u;
10     };
11 }
12
13 void afficher(struct OptionValue opt){
14     switch (opt.label) {
15     case None:
16         printf("The option is empty\n");
17         break;
18     case Double:
19         printf("Float in option : %f\n", opt.d);
20         break;
21     case Integer:
22         printf("Integer in option : %u\n", opt.u);
23         break;
24     default:
25         break;
26     }
27 }
28
29 int main(void) {
30     struct OptionValue opt1 = { .label = None };
31     struct OptionValue opt2 = { .label = Double, .d = 3.14 };
32     struct OptionValue opt3 = { .label = Integer, .u = 42 };
33     afficher(opt1);
34     afficher(opt2);
35     afficher(opt3);
36
37     return 0;
38 }
```

Gérer des options et une utilisation classique des unions

On stocke le type de l'option dans une structure d'encapsulation, et on définit l'union comme **anonyme** pour traiter ses différents membres comme s'ils étaient **directement** membres de la structure

0x00000083	(131)
0x00000082	(130)
0x00000081	(129)
0x00000080	(128)
0x0000007F	(127)
0x0000007E	(126)
0x0000007D	(125)
0x0000007C	(124)
0x0000007B	(123)
0x0000007A	(122)
0x00000079	(121)
0x00000078	(120)

...

...

...

...

...

...

...

...

...

...

...

...

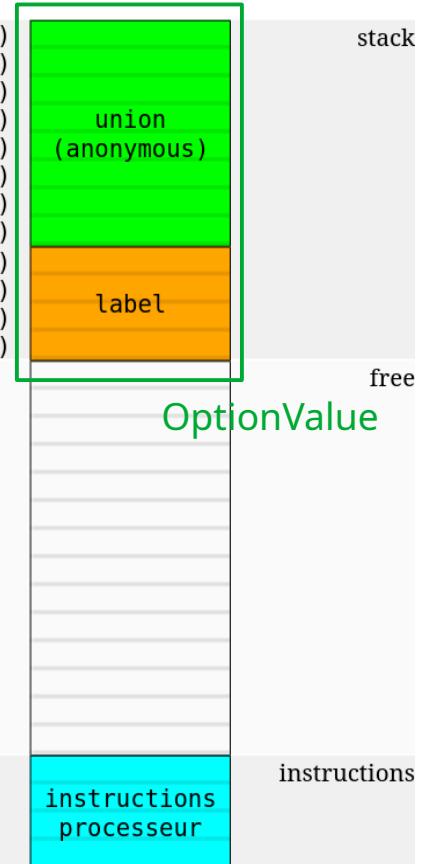
...

...

...

...

0x00000023	(35)
0x00000022	(34)
0x00000021	(33)
0x00000020	(32)





Pointeurs génériques & pointeurs de fonction

Pointeurs génériques & pointeurs de fonction



Polymorphisme

"Object-oriented programming is an exceptionally bad idea which could only have originated in California" -
Edsger Dijkstra



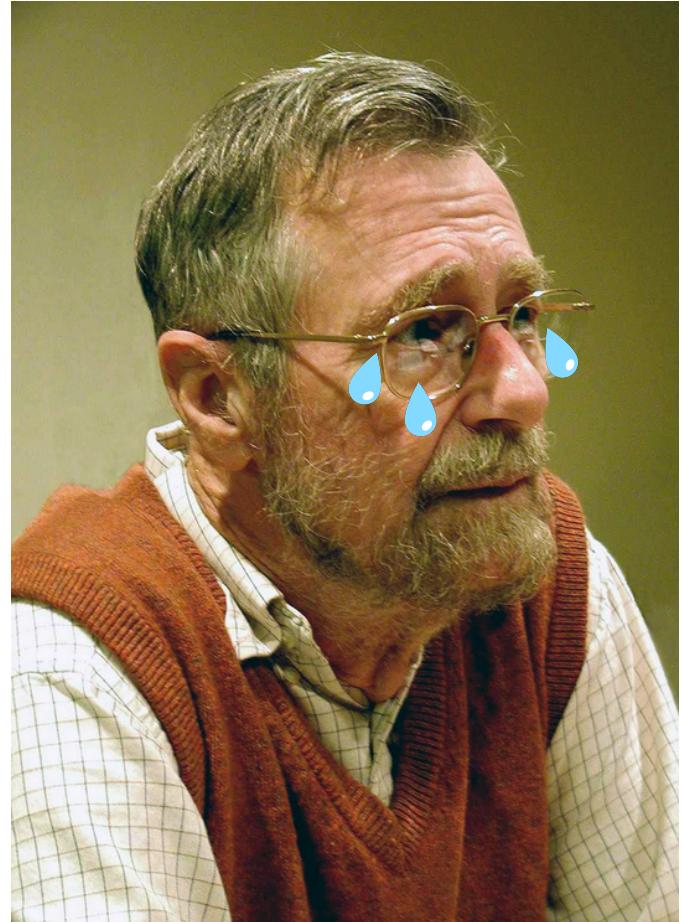
Pointeurs génériques & pointeurs de fonction



Polymorphisme

"Object-oriented programming is an exceptionally bad idea which could only have originated in California" -
Edsger Dijkstra

- Or, la programmation orientée objet (POO) est un paradigme **exceptionnellement populaire** en informatique, même si certains aspects sont critiqués (pas sa meilleure take, donc)



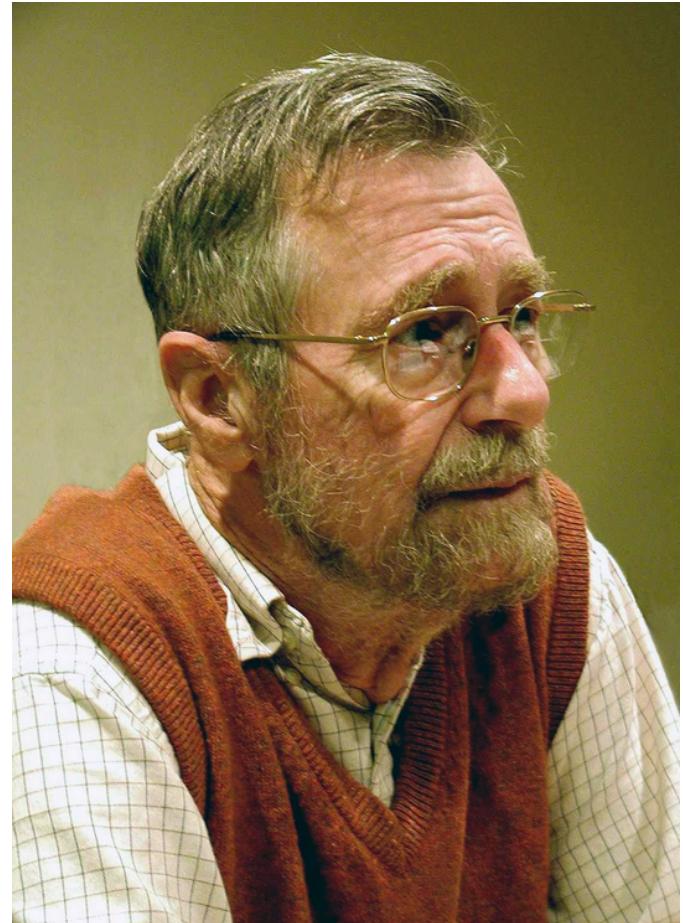
Pointeurs génériques & pointeurs de fonction



Polymorphisme

"Object-oriented programming is an exceptionally bad idea which could only have originated in California" - Edsger Dijkstra

- Or, la programmation orientée objet (POO) est un paradigme **exceptionnellement populaire** en informatique, même si certains aspects sont critiqués (pas sa meilleure take, donc)
- Un des concepts centraux de la POO est le **polymorphisme**
- Nous n'avons pas besoin d'OOP pour comprendre ou implémenter un certain degré de polymorphisme → l'idée est d'avoir une même "interface" (contrat) pour des implémentations différentes
- L'idée est de comprendre que ce mécanisme n'est pas gratuit, et d'autant plus d'apprécier les langages de haut niveau qui en masquent le coût.

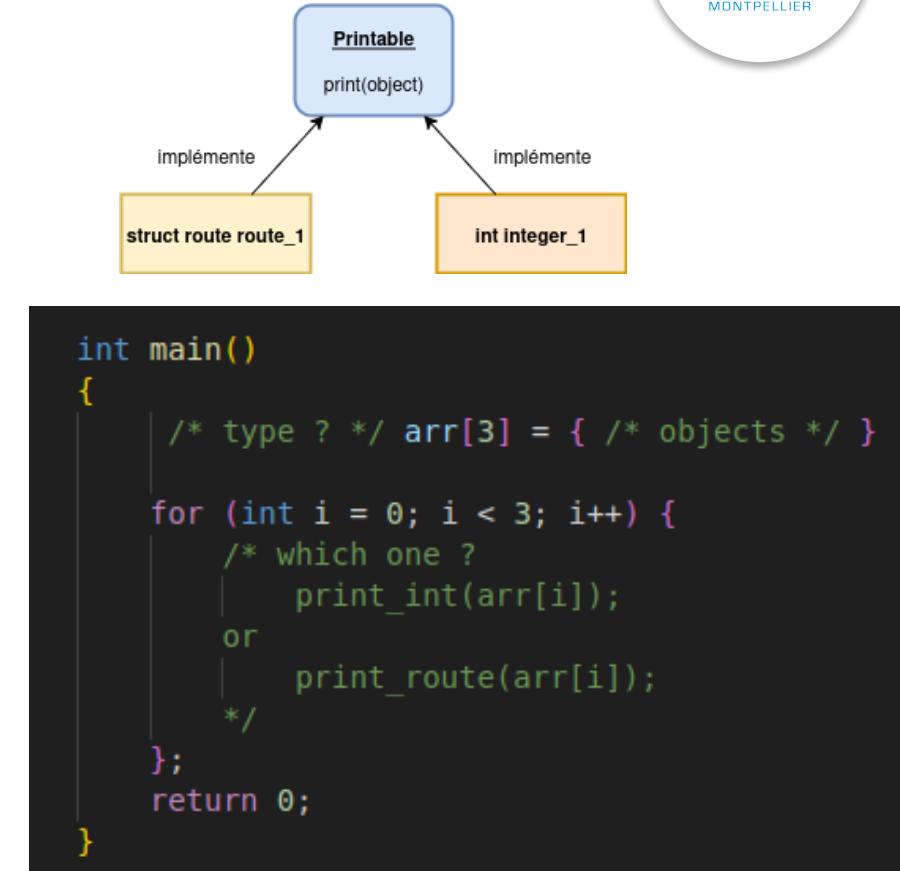


Pointeurs génériques & pointeurs de fonction



Polymorphisme

- "Facile, je vais écrire deux fonctions" :
 - **print_int(int object)** { /* code */}
 - **print_route(struct route object)** { /* code */}
- **Pas si simple**, les deux fonctions sont distinctes, et les objects sont toujours traités comme une **struct route** ou un **int**, alors qu'on souhaite les traiter comme des objets **Printable**.
- Si je souhaite créer un tableau de printable : je suis bloqué, quel est le type du tableau ? Struct route arr[] ou int[] ? La taille d'un **int** est différente d'une **struct route** cela ne peut pas marcher (on sent le besoin d'avoir des pointeurs)
- En imaginant que le problème de la liste est réglé, je boucle sur mon tableau, comment différencier les **int** des **struct route** ?



Et "**Printable**"
n'apparaît même pas, il nous
manque des concepts

Pointeurs génériques & pointeurs de fonction



Étape 1 : pointeurs génériques

- On va avoir besoin d'un pointeur → on veut des éléments de même taille pour nos tableaux, et on sait que la taille d'un pointeur dépend de la range d'adressage, et non du type pointé, donc :
sizeof(struct route *) == sizeof(int *)
- Le type "pointeur vers quelque chose" ou **pointeur générique**, est le **void ***. **Et on connaît ce type : c'est le type de retour de malloc**
- Il est important de comprendre que l'accès à la valeur pointée par le **void * n'est pas possible sans cast**, en effet, le type (et donc sa taille) de la donnée pointée est renseigné par le type du pointeur

A screenshot of a code editor showing a C file named void.c. The code defines a variable 'a' of type int and a void pointer 'a_ptr' pointing to 'a'. It then prints the value of 'a' plus 5. A red circle highlights the 'void' keyword in the declaration of 'a_ptr'. The code editor interface shows tabs for PROBLEMS (1), OUTPUT, DEBUG CONSOLE, and TERMINAL. In the terminal tab, a gcc compilation of void.c results in a warning about dereferencing a void pointer at line 8, column 33, and an error about a void value not being ignored at the same location.

```
C void.c > ...
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 3;
6     void *a_ptr = &a;
7     printf("a + 5 = %d \n", 5 + *a_ptr);
8     return 0;
9 }
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples git:(main) ✘ gcc void.c -o void
void.c: In function 'main':
void.c:8:33: warning: dereferencing 'void *' pointer
  8 |     printf("a + 5 = %d \n", 5 + *a_ptr);
               ^
void.c:8:33: error: void value not ignored as it ought to be
(base) → examples git:(main) ✘
```

Pointeurs génériques & pointeurs de fonction



Étape 1 : pointeurs génériques

- On va avoir besoin d'un pointeur → on veut des éléments de même taille pour nos tableaux, et on sait que la taille d'un pointeur dépend de la range d'adressage, et non du type pointé, donc :
sizeof(struct route *) == sizeof(int *)
- Le type "pointeur vers quelque chose" ou **pointeur générique**, est le **void ***. **Et on connaît ce type : c'est le type de retour de malloc**
- Il est important de comprendre que l'accès à la valeur pointée par le **void *** **n'est pas possible sans cast**, en effet, le type (et donc sa taille) de la donnée pointée est renseigné par le type du pointeur

```
C void.c > ...
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 3;
6     void *a_ptr = &a;
7     printf("a + 5 = %d \n", 5 + *(int *) a_ptr);
8     return 0;
9 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) → examples git:(main) ✘ gcc void.c -o void
(base) → examples git:(main) ✘ ./void
a + 5 = 8
(base) → examples git:(main) ✘
```

Dans cet exemple, on cast le void * en int * pour déréférencer un int

Pointeurs génériques & pointeurs de fonction



Est-ce que cela suffit ?

```
#include <stdio.h>

void print(void * object) {
    // euh.. quel est le type d'object ?
}

int main()
{
    int a = 3;
    double b = 4;
    print(&a);
    print(&b);
    return 0;
}
```

Nouvel essai...

Pointeurs génériques & pointeurs de fonction



Est-ce que cela suffit ? (non)

```
1 #include <stdio.h>
2
3 void print(void * object, int type) {
4     if (type == 0) {
5         printf("int: %d\n", *(int *) object);
6         return;
7     }
8     if (type == 1) {
9         printf("double: %f\n", *(double *) object);
10        return;
11    }
12 }
13
14 int main()
15 {
16     int a = 3;
17     double b = 4;
18     print(&a, 0);
19     print(&b, 1);
20     return 0;
21 }
```

C'est du bricolage :

- Le contrat n'est pas respecté, l'argument "type" ne faisait pas partie du "print" du contrat
- Le début des problèmes :
 - **Printable** n'est toujours pas présent dans le code. Le développeur tente d'implémenter un contrat non défini dans le code. Or, ça devrait être au contrat (et son code) de contraindre le développeur.
 - **On a juste déplacé le problème**, le control flow lignes 4 à 11, pourrait être autour des lignes 18 et 19
 - Échec de la séparation des préoccupations : si print fait partie du contrat, l'adhésion au contrat demande modification du contrat : il faut ajouter du control flow pour ajouter un type Printable

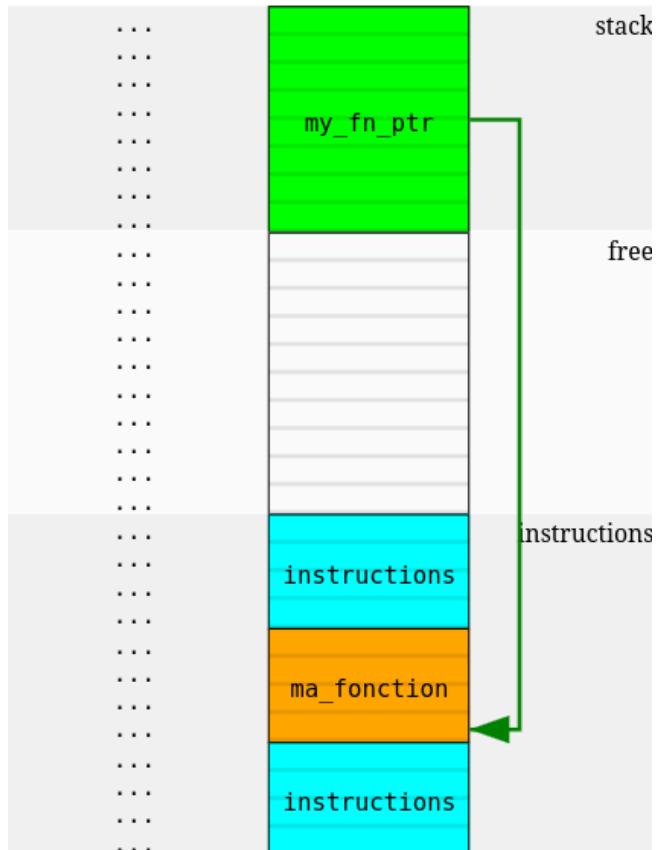
Pointeurs génériques & pointeurs de fonction



Étape 2 : pointeurs de fonctions

- On a vu des pointeurs vers la pile, des pointeurs vers le tas, des pointeurs vers la zone statique (pour les chaînes de caractères), **un pointeur de fonction pointe vers la zone des instructions** aussi appelé "code segment" ou "text"
- Le détail sera vu en cours système : la première instruction n'est pas forcément l'adresse de la première instruction C, on peut avoir besoin d'instructions pour préparer la frame
- Très grossièrement, un appel de fonction, c'est connaître l'adresse de la première instruction de celle-ci, et changer l'adresse de la prochaine instruction par cette adresse

On ne rentrera pas davantage dans les détails, ce n'est pas forcément approprié sans parler d'assembleur



Pointeurs génériques & pointeurs de fonction



Étape 2 : pointeurs de fonctions

- Un cas classique d'utilisation de pointeur de fonction est par utilisation de la fonction **qsort** (de stdlib) pour trier les éléments d'un tableau
- Avoir à trier des données est très courant, et la plupart des langages proposent dans leurs librairies standard des algorithmes de tri.
- Par contre, cela presuppose que les éléments doivent pouvoir être comparés
- **qsort** demande la fonction de comparaison en argument : **un pointeur de fonction**

Dans l'exemple, une route à une heure d'arrivée et un nombre de sauts (e.g. nombre de correspondances) → on souhaite arriver le plus tôt possible, et si deux options ont la même heure d'arrivée, on souhaite le moins de saut possible.

```
c fn_ptr.c > cmp(const void *, const void *)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct route {
5     int arrival;
6     int hops;
7 };
8 // négatif si a < b, positif si a > b, 0 si a == b
9 int cmp(const void *a, const void *b) {
10    const struct route *x = a;
11    const struct route *y = b;
12    // Cela fonctionne car arrival et hops sont des int
13    if (x->arrival != y->arrival)
14        return (x->arrival - y->arrival);
15    return (x->hops - y->hops);
16 }
17
18 int main()
19 {
20    struct route arr[3] = { { 6, 3 }, { 7, 2 }, { 6, 2 } };
21    qsort(arr, 3, sizeof(struct route), cmp); // trier
22    for (int i = 0; i < 3; i++) {
23        printf("Route %d: arrival at %d with %d hops\n", i,
24               arr[i].arrival, arr[i].hops
25        );
26    }
27    return 0;
28 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ ./fn_ptr
Route 0: arrival at 6 with 2 hops
Route 1: arrival at 6 with 3 hops
Route 2: arrival at 7 with 2 hops
- (base) → examples git:(main) ✘

Pointeurs génériques & pointeurs de fonction



Étape 2 : pointeurs de fonctions

- Un cas classique d'utilisation de pointeur de fonction est par utilisation de la fonction **qsort** (de stdlib) pour trier les éléments d'un tableau
- Avoir à trier des données est très courant, et la plupart des langages proposent dans leurs librairies standard des algorithmes de tri.
- Par contre, cela presuppose que les éléments doivent pouvoir être comparés
- **qsort** demande la fonction de comparaison en argument : **un pointeur de fonction**

Dans l'exemple, une route à une heure d'arrivée et un nombre de sauts (e.g. nombre de correspondances) → on souhaite arriver le plus tôt possible, et si deux options ont la même heure d'arrivée, on souhaite le moins de saut possible.

```
9  int cmp(const void *a, const void *b) {  
10    const struct route *x = a;  
11    const struct route *y = b;  
12  
13    if (x->arrival < y->arrival) {  
14      return -1;  
15    }  
16    if (x->arrival > y->arrival) {  
17      return 1;  
18    }  
19    // donc x->arrival == y->arrival  
20    if (x->hops < y->hops) {  
21      return -1;  
22    }  
23    if (x->hops > y->hops) {  
24      return 1;  
25    }  
26    // donc x->hops == y->hops  
27    return 0;  
28  }  
29 }
```

Version équivalente
(mais moins rapide)



Dynamique dispatch

Pointeurs génériques & pointeurs de fonction



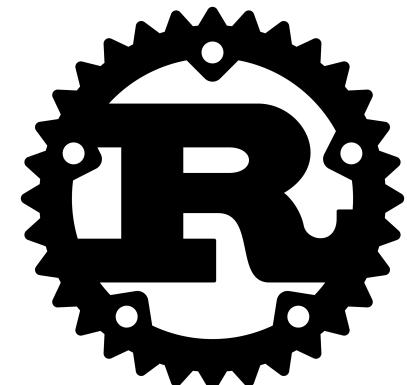
Polymorphisme simulé

Dans cette section, nous verrons deux méthodes :

- La première se rapproche de l'implémentation en C++, par **ajouts de champs** dans la structure implementant la ou les interfaces
- La seconde se rapproche de l'implémentation en Rust, avec le concept des **fat pointers**

Dans les deux cas, on écrit du code afin de produire un code compilé ressemblant à celui de ces deux langages **à des fins pédagogiques**

→ **Le C n'est pas une étape intermédiaire pour C++ et Rust**



Pointeurs génériques & pointeurs de fonction

```
examples > C polymorphism.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Printable {
5      void (*print)(void*);
5 };
7
8  struct route {
9      struct Printable methods_route;
10     int arrival;
11     int hops;
12 };
13
14 void print_route(void *ptr_route) {
15     printf("Route: arrival at %d with %d hops\n",
16           ((struct route *)ptr_route)->arrival,
17           ((struct route *)ptr_route)->hops
18     );
19 }
20
21 struct str {
22     struct Printable methods_str;
23     char * s;
24 };
25
26 void print_str(void *ptr_str){
27     printf("String: %s\n", (char *) ((struct str *) ptr_str)->s);
28 }
29
```

Polymorphisme simulé (C++)

- Ici, nous avons une entité programmée pour notre interface "Printable", et comme dans notre diagramme, elle ne possède que la fonction du contrat



Étoile collée à la variable, 2ème justification :

Essayez de coller l'étoile au type de gauche..

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Printable {
5      void (*print)(void*); // Method pointer
6  };
7
8  struct route {
9      struct Printable methods_route; // Method pointer
10     int arrival;
11     int hops;
12 };
13
14 void print_route(void *ptr_route) {
15     printf("Route: arrival at %d with %d hops\n",
16           ((struct route *)ptr_route)->arrival,
17           ((struct route *)ptr_route)->hops
18     );
19 }
20
21 struct str {
22     struct Printable methods_str; // Method pointer
23     char * s;
24 };
25
26 void print_str(void *ptr_str){
27     printf("String: %s\n", (char *) ((struct str *) ptr_str)->s);
28 }
29
```

En OOP, on appelle "méthode" une fonction associée à une classes ou interface

on garde ici cette terminologie

Polymorphisme simulé (C++)

- Ici, nous avons une entité programmée pour notre interface "Printable", et comme dans notre diagramme, elle ne possède que la fonction du contrat
- La signature du contrat (l'implémentation de l'interface) se fait à la création de la structure "signataire", les objets sont donc "Printable" sans "signature différée" par encapsulation

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Printable {
5      void (*print)(void*);
6  };
7
8  struct route {
9      struct Printable methods_route;
10     int arrival;
11     int hops;
12 };
13
14 void print_route(void *ptr_route) {
15     printf("Route: arrival at %d with %d hops\n",
16           ((struct route *)ptr_route)->arrival,
17           ((struct route *)ptr_route)->hops
18     );
19 }
20
21 struct str {
22     struct Printable methods_str;
23     char * s;
24 };
25
26 void print_str(void *ptr_str){
27     printf("String: %s\n", (char *) ((struct str *) ptr_str)->s);
28 }
29
30 int main()
31 {
32     struct str my_str = { { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] = {
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };

```

Polymorphisme simulé (C++)

- Ici, nous avons une entité programmée pour notre interface "Printable", et comme dans notre diagramme, elle ne possède que la fonction du contrat
- La signature du contrat (l'implémentation de l'interface) se fait à la création de la structure "signataire", les objets sont donc "Printable" sans "signature différée" par encapsulation
- Avec ce nouveau type "**struct Printable**" pour notre interface "Printable", nous pouvons donc avoir une liste d'objets printables, non encapsulés, mais par indirection (pointeurs), car ils n'ont pas forcément la même taille, et ne peuvent donc pas être alignés directement dans la mémoire associée au tableau

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Printable {
5      void (*print)(void*);
6  };
7
8  struct route {
9      struct Printable methods_route;
10     int arrival;
11     int hops;
12 };
13
14 void print_route(void *ptr_route) {
15     printf("Route: arrival at %d with %d hops\n",
16           ((struct route *)ptr_route)->arrival,
17           ((struct route *)ptr_route)->hops
18     );
19 }
20
21 struct str {
22     struct Printable methods_str;
23     char * s;
24 };
25
26 void print_str(void *ptr_str){
27     printf("String: %s\n", (char *) ((struct str *) ptr_str)->s);
28 }
29
30 int main()
31 {
32     struct str my_str = { { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] = {
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };
}
```

Polymorphisme simulé (C++)

- Ici, nous avons une entité programmée pour notre interface "Printable", et comme dans notre diagramme, elle ne possède que la fonction du contrat
- La signature du contrat (l'implémentation de l'interface) se fait à la création de la structure "signataire", les objets sont donc "Printable" sans "signature différée" par encapsulation
- Avec ce nouveau type "**struct Printable**" pour notre interface "Printable", nous pouvons donc avoir une liste d'objets printables, non encapsulés, mais par indirection (pointeurs), car ils n'ont pas forcément la même taille, et ne peuvent donc pas être alignés directement dans la mémoire associée au tableau
- Et donc à partir de cette liste d'adresses d'objets "Printable", on peut donc accéder à "methods" et donc la fonction "print" et..

Wait a minute..

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Printable {
5     void (*print)(void*);
6 };
7
8 struct route {
9     struct Printable methods_route;
10    int arrival,
11    int hops;
12 };
13
14 void print_route(void *ptr_route) {
15     printf("Route: arrival at %d with %d hops\n",
16           ((struct route *)ptr_route)->arrival,
17           ((struct route *)ptr_route)->hops
18     );
19 }
20
21 struct str {
22     struct Printable methods_str;
23     char * s,
24 };
25
26 void print_str(void *ptr_str){
27     printf("String: %s\n", (char *) ((struct str *) ptr_str)->s);
28 }
29
30 int main()
31 {
32     struct str my_str = { { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] = {
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };
40     for (int i = 0; i < 3; i++) {
41         arr[i]->print(arr[i]);
42     }
43     return 0;
44 }
```

Polymorphisme simulé (C++)



Comment pourrait-on accéder à "print" sans passer par les membres "**methods_<X>**" de nos objets de type Printable? On va clairement sauter une étape..

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Printable {
5     void (*print)(void*);
6 };
7
8 struct route {
9     struct Printable methods_route;
10    int arrival,
11    int hops;
12 };
13
14 void print_route(void *ptr_route) {
15     printf("Route: arrival at %d with %d hops\n",
16           ((struct route *)ptr_route)->arrival,
17           ((struct route *)ptr_route)->hops
18     );
19 }
20
21 struct str {
22     struct Printable methods_str;
23     char * s,
24 };
25
26 void print_str(void *ptr_str){
27     printf("String: %s\n", (char *) ((struct str *) ptr_str)->s);
28 }
29
30 int main()
31 {
32     struct str my_str = { { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] = {
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };
40     for (int i = 0; i < 3; i++) {
41         arr[i]->print(arr[i]);
42     }
43     return 0;
44 }
```

Polymorphisme simulé (C++)



Comment pourrait-on accéder à "print" sans passer par les membres "**methods_<X>**" de nos objets de type Printable? On va clairement sauter une étape..

Et ça fonctionne !

```
● (base) → examples git:(main) ✘ ./polymorphism
String: Hello !
Route: arrival at 6 with 3 hops
Route: arrival at 7 with 2 hops
○ (base) → examples git:(main) ✘
```



Pointeurs génériques & pointeurs de fonction



```
30 int main()
31 {
32     struct str my_str = { { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] = {
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };
}
```

Polymorphisme simulé (C++)

Ça fonctionne, la question sera pourquoi, mais pouvait-on de toute façon accéder "proprement" à "**methods**" ?

Pointeurs génériques & pointeurs de fonction



```
30 int main()
31 {
32     struct str my_str = { { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] = {
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };
}
```

Polymorphisme simulé (C++)

Ça fonctionne, la question sera pourquoi, mais pouvait-on de toute façon accéder "proprement" à "**methods**" ?

Mais on n'a pas de type générique pour le cast les pointeurs, **on ne va quand même pas cast vers "struct route"** par défaut ?



Pointeurs génériques & pointeurs de fonction

```
30 int main()
31 {
32     struct str my_str = { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] = {
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };
40     for (int i = 0; i < 3; i++) {
41         struct route * ptr = (struct route *) arr[i];
42         ptr->methods_route.print(arr[i]);
43     };
44     return 0;
45 }
```

Polymorphisme simulé (C++)

Ça fonctionne, la question sera pourquoi, mais pouvait-on de toute façon accéder "proprement" à "**methods**" ?

Mais on n'a pas de type générique pour le cast les pointeurs, on ne va quand même pas cast vers "**struct route**" par défaut ? **On va tenter..**



Pointeurs génériques & pointeurs de fonction



```
30 int main()
31 {
32     struct str my_str = { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] =
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };
40     for (int i = 0; i < 3; i++) {
41         struct route * ptr = (struct route *) arr[i];
42         ptr->methods_route.print(arr[i]);
43     }
44     return 0;
45 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

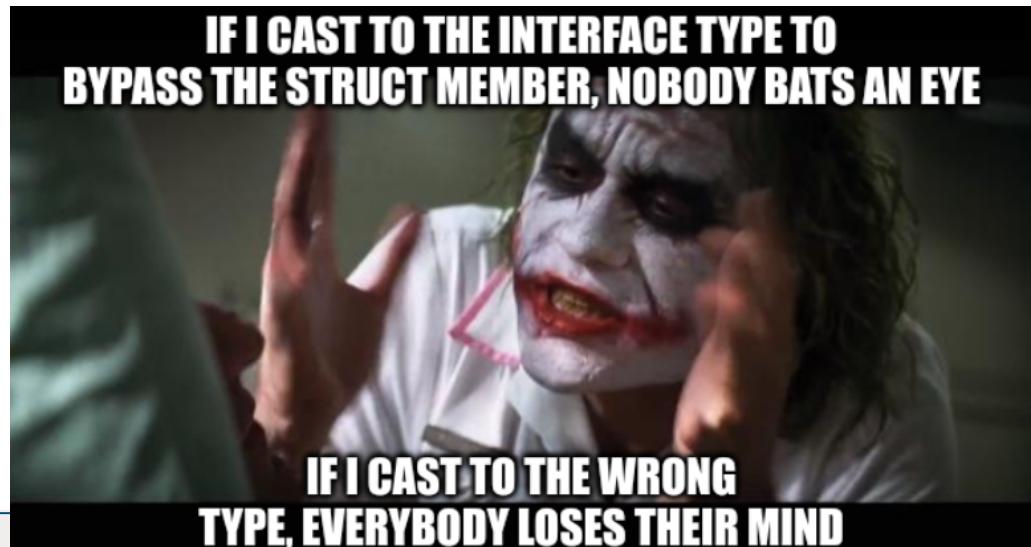
```
● (base) → examples git:(main) ✘ gcc -o polymorphism_2 polymorphism_2.c
● (base) → examples git:(main) ✘ ./polymorphism_2
String: Hello !
Route: arrival at 6 with 3 hops
Route: arrival at 7 with 2 hops
● (base) → examples git:(main) ✘
```

Polymorphisme simulé (C++)

Ça fonctionne, la question sera pourquoi, mais pouvait-on de toute façon accéder "proprement" à "**methods**" ?

Mais on n'a pas de type générique pour le cast les pointeurs, **on ne va quand même pas cast vers "struct route"** par défaut ? **On va tenter..**

Et ça marche.. alors que notre **struct str a été cast vers **struct route**. En "réglant" un problème, on en a créé un nouveau**



Pointeurs génériques & pointeurs de fonction



```
30 int main()
31 {
32     struct str my_str = { print_str }, "Hello !" };
33     struct route route_1 = { { print_route }, 6, 3 };
34     struct route route_2 = { { print_route }, 7, 2 };
35     struct Printable *arr[3] =
36         (struct Printable *) &my_str,
37         (struct Printable *) &route_1,
38         (struct Printable *) &route_2
39     };
40     for (int i = 0; i < 3; i++) {
41         struct route * ptr = (struct route *) arr[i];
42         ptr->methods_route.print(arr[i]);
43     }
44     return 0;
45 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) + examples git:(main) ✘ gcc -o polymorphism_2 polymorphism_2.c
● (base) + examples git:(main) ✘ ./polymorphism_2
String: Hello !
Route: arrival at 6 with 3 hops
Route: arrival at 7 with 2 hops
● (base) + examples git:(main) ✘
```

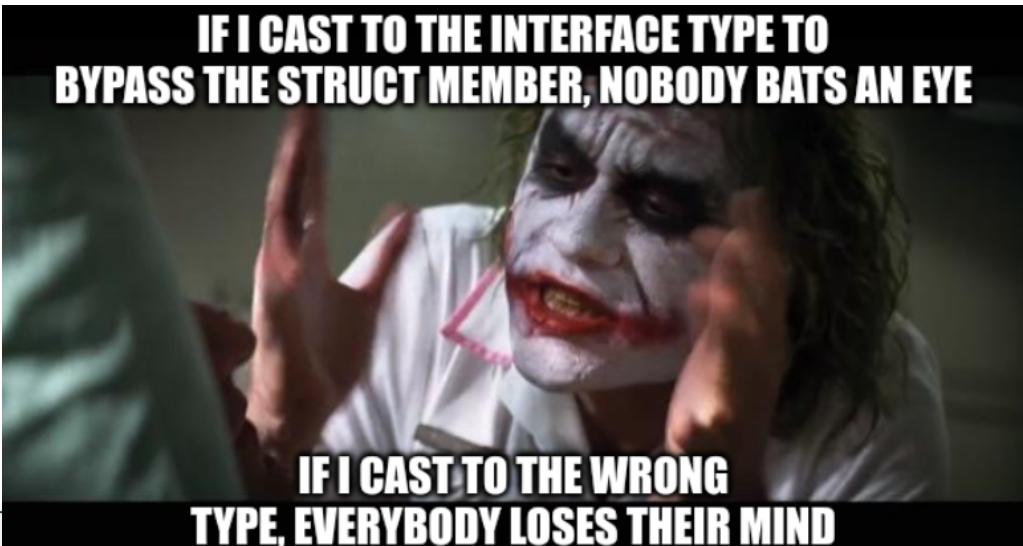
Clairement il nous manque une information pour comprendre pourquoi cela fonctionne également

Polymorphisme simulé (C++)

Ça fonctionne, la question sera pourquoi, mais pouvait-on de toute façon accéder "proprement" à "**methods**" ?

Mais on n'a pas de type générique pour le cast les pointeurs, **on ne va quand même pas cast vers "struct route"** par défaut ? **On va tenter..**

Et ça marche.. alors que notre **struct str a été cast vers **struct route**. En "réglant" un problème, on en a créé un nouveau**



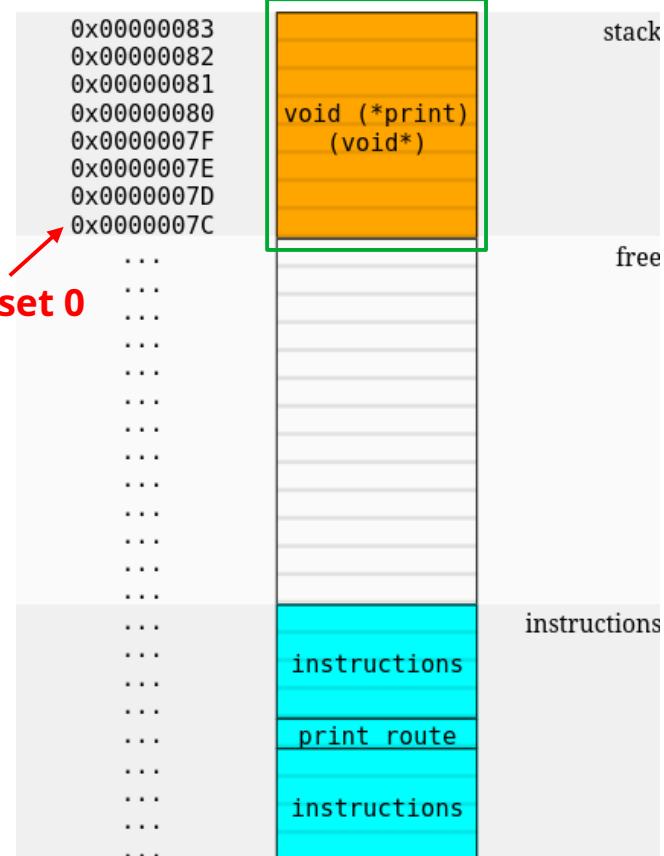
Pointeurs génériques & pointeurs de fonction



Pourquoi cela fonctionne ?

- Le langage C propose certaines garanties concernant le layout des structures :
 - 1) Le premier membre d'une structure ne doit pas avoir de padding avec le début de la structure, i.e. l'adresse de la structure est l'adresse du premier membre
 - 2) L'ordre de déclaration des membres doit être respecté (dans la mémoire)
 - 3) Les structures imbriquées sont alignées en mémoire
- Les structures **struct str**, **struct route** et **struct Printable** ont chacun un premier membre partageant le même type, le cast permet de dire au compilateur d'accéder à cette zone mémoire comme s'il s'agissait d'une **struct Printable**.
- "**print**" est donc bien à l'offset 0 grâce à 1) pour une **struct Printable** et à l'offset 0 également pour **struct route** ou **struct str** grâce à 1) et 3)

Une **struct Printable** en mémoire



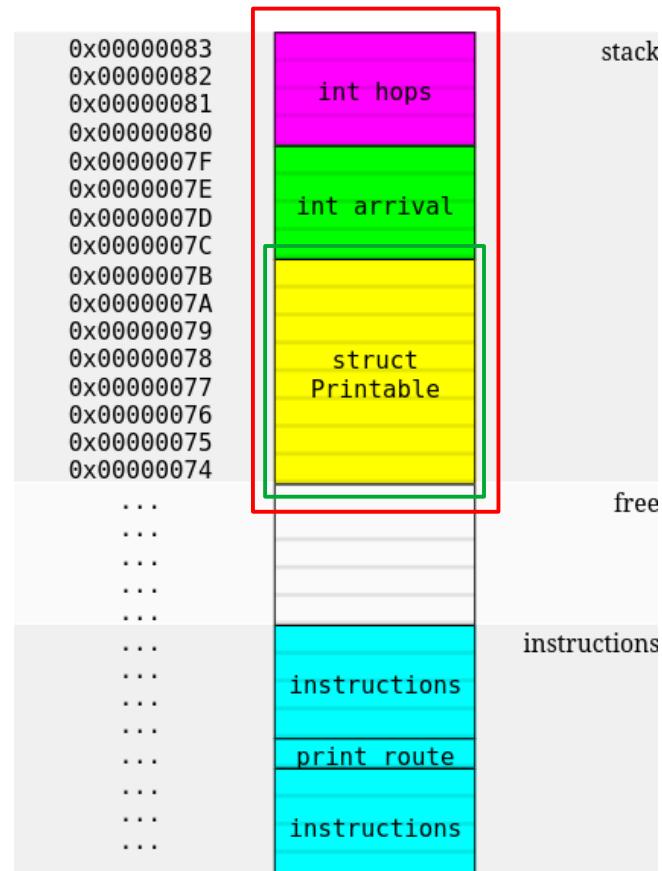
Pointeurs génériques & pointeurs de fonction



Pourquoi cela fonctionne ?

- Le langage C propose certaines garanties concernant le layout des structures :
 - 1) Le premier membre d'une structure ne doit pas avoir de padding avec le début de la structure, i.e. l'adresse de la structure est l'adresse du premier membre
 - 2) L'ordre de déclaration des membres doit être respecté (dans la mémoire)
 - 3) Les structures imbriquées sont alignées en mémoire
- Les structures **struct str**, **struct route** et **struct Printable** ont chacun un premier membre partageant le même type, le cast permet de dire au compilateur d'accéder à cette zone mémoire comme s'il s'agissait d'une **struct Printable**.
- "print" est donc bien à l'offset 0 grâce à 1) pour une **struct Printable** et à l'offset 0 également pour **struct route** ou **struct str** grâce à 1) et 3)

Une **struct route** présente une **struct Printable** en premier membre



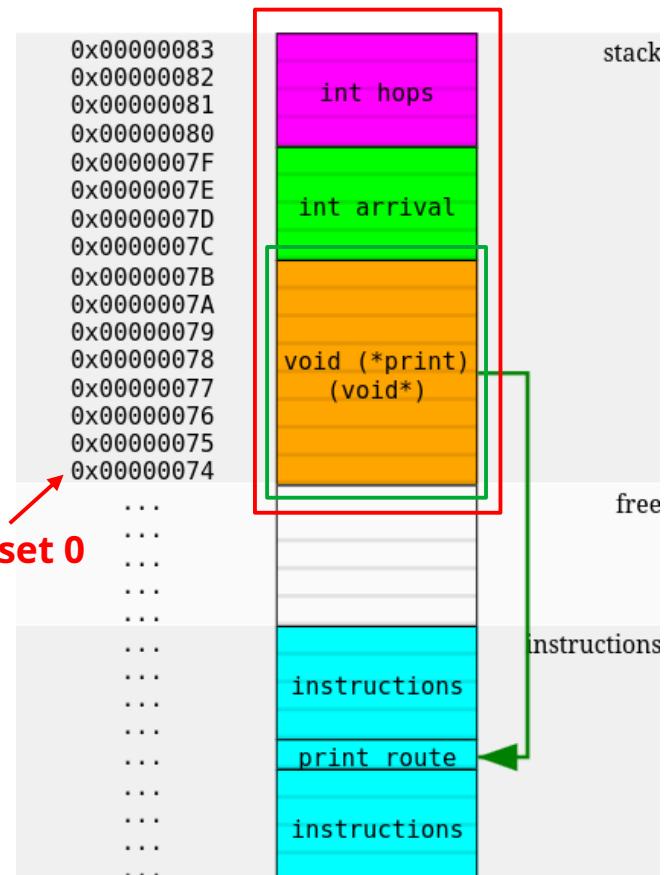
Pointeurs génériques & pointeurs de fonction



Pourquoi cela fonctionne ?

- Le langage C propose certaines garanties concernant le layout des structures :
 - Le premier membre d'une structure ne doit pas avoir de padding avec le début de la structure, i.e. l'adresse de la structure est l'adresse du premier membre
 - L'ordre de déclaration des membres doit être respecté (dans la mémoire)
 - Les structures imbriquées sont alignées en mémoire
- Les structures **struct str**, **struct route** et **struct Printable** ont chacun un premier membre partageant le même type, le cast permet de dire au compilateur d'accéder à cette zone mémoire comme s'il s'agissait d'une **struct Printable**.
- "**print**" est donc bien à l'offset 0 grâce à 1) pour une **struct Printable** et à l'offset 0 également pour **struct route** ou **struct str** grâce à 1) et 3)

Nous avons bien "**print**" au même offset dans les 2 cas (0)



Pointeurs génériques & pointeurs de fonction



Pour aller plus loin

- Bien que le code présenté fonctionnera sur la plupart des compilateurs, le code a été simplifié et ne respecte pas la "**strict aliasing rule**" : Sans rentrer dans les détails, il faudrait plutôt utiliser un pointeur générique (**void ***) plutôt que sa version casté en **struct Printable ***
- Pour respecter cette règle (certes un peu rigide), il faudrait stocker des **void *** dans le tableau, et recast depuis **void ***. Il faut en effet cast vers des types compatibles (**void *** est compatible avec tout) là où cast de **struct route** vers **struct Printable** inquiète le compilateur

```
41 int main() {
42     struct str my_str = { { print_str }, "Hello !" };
43     struct route route_1 = { { print_route }, 6, 3 };
44     struct route route_2 = { { print_route }, 7, 2 };
45
46     // Use void* array - no aliasing issues
47     void *arr[3] = {
48         &my_str,
49         &route_1,
50         &route_2
51     };
52
53     for (int i = 0; i < 3; i++) {
54         ((struct Printable *) arr[i])->print(arr[i]);
55     }
56
57     return 0;
58 }
```

Pointeurs génériques & pointeurs de fonction



Pour aller plus loin

- Bien que le code présenté fonctionnera sur la plupart des compilateurs, le code a été simplifié et ne respecte pas la "**strict aliasing rule**" : Sans rentrer dans les détails, il faudrait plutôt un pointeur générique (**void ***) plutôt que sa version casté en **struct Printable ***
- Mais on peut faire plus **compliqué, juste pour le sport**, avec une fonction qui rend un pointeur de fonction



Et non ceci (retour souligné) :

`void (*)(void*) get_print_function_2(void* obj);`

La "grammaire" (cours automates et langages) du C lit le code de "l'intérieur vers l'extérieur". Voir la "Clockwise/spiral rule" :

<https://c-faq.com/decl/spiral.anderson.html>

```
30 void (*get_print_function(void *obj))(void*) {
31     return ((struct Printable *)obj)->print;
32 }
33
34 int main() {
35     struct str my_str = { { print_str }, "Hello !" };
36     struct route route_1 = { { print_route }, 6, 3 };
37     struct route route_2 = { { print_route }, 7, 2 };
38
39     // Use void* array - no aliasing issues
40     void *arr[3] = {
41         &my_str,
42         &route_1,
43         &route_2
44     };
45
46     for (int i = 0; i < 3; i++) {
47         void (*print_fn)(void*) = get_print_function(arr[i]);
48         print_fn(arr[i]);
49     }
50
51     return 0;
52 }
```

Pointeurs génériques & pointeurs de fonction



Pour aller plus loin

- Bien que le code présenté fonctionnera sur la plupart des compilateurs, le code a été simplifié et ne respecte pas la "**strict aliasing rule**" : Sans rentrer dans les détails, il faudrait plutôt un pointeur générique (**void ***) plutôt que sa version casté en **struct Printable ***
- Mais on peut faire plus **compliqué, juste pour le sport**, avec une fonction qui rend un pointeur de fonction

On peut simplifier cette syntaxe avec une déclaration en **2 étapes** :

- **Création d'un alias** pour le type de retour avec **typedef**
- **Utilisation de l'alias** pour une déclaration "classique" de **get_print_function**

```
35 typedef void (*print_fn_t)(void*);  
36  
37 print_fn_t get_print_function(void *obj) {  
38     return ((struct Printable *)obj)->print;  
39 }  
40  
41 int main() {  
42     struct str my_str = { { print_str }, "Hello !" };  
43     struct route route_1 = { { print_route }, 6, 3 };  
44     struct route route_2 = { { print_route }, 7, 2 };  
45  
46     // Use void* array - no aliasing issues  
47     void *arr[3] = {  
48         &my_str,  
49         &route_1,  
50         &route_2  
51     };  
52  
53     for (int i = 0; i < 3; i++) {  
54         void (*print_fn)(void*) = get_print_function(arr[i]);  
55         print_fn(arr[i]);  
56     }  
57  
58     return 0;  
59 }
```

Pointeurs génériques & pointeurs de fonction



Pour aller plus loin

- Bien que le code présenté fonctionnera sur la plupart des compilateurs, le code a été simplifié et ne respecte pas la "**strict aliasing rule**" : Sans rentrer dans les détails, il faudrait plutôt un pointeur générique (**void ***) plutôt que sa version casté en **struct Printable ***
- Mais on peut faire plus **compliqué, juste pour le sport**, avec une fonction qui rend un pointeur de fonction

En pratique, il y a très peu de chance d'observer des problèmes dans ce cas précis, mais au cas où :

gcc -O2 -Wall -Wstrict-aliasing=2
pour détecter des violations de cette règle

On peut simplifier cette syntaxe avec une déclaration en 2 étapes :

- Création d'un **alias** pour le type de retour avec **typedef**
- Utilisation de l'**alias** pour une déclaration "classique" de **get_print_function**

```
35 typedef void (*print_fn_t)(void*);  
36  
37 print_fn_t get_print_function(void *obj) {  
38     return ((struct Printable *)obj)->print;  
39 }  
40  
41 int main() {  
42     struct str my_str = { { print_str }, "Hello !" };  
43     struct route route_1 = { { print_route }, 6, 3 };  
44     struct route route_2 = { { print_route }, 7, 2 };  
45  
46     // Use void* array - no aliasing issues  
47     void *arr[3] = {  
48         &my_str,  
49         &route_1,  
50         &route_2  
51     };  
52  
53     for (int i = 0; i < 3; i++) {  
54         void (*print_fn)(void*) = get_print_function(arr[i]);  
55         print_fn(arr[i]);  
56     }  
57  
58     return 0;  
59 }
```

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism_fat.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Printable {
5     void (*print)(void*);
6 };
7
8 struct route {
9     int arrival;
10    int hops;
11 };
12
13 void print_route(void *ptr_route) {
14     printf("Route: arrival at %d with %d hops\n",
15            ((struct route *)ptr_route)->arrival,
16            ((struct route *)ptr_route)->hops
17        );
18 }
19
20 void print_str(void *ptr_str){
21     printf("String: %s\n", (char *) ptr_str);
22 }
23
24 struct fat_ptr_printable {
25     void *data;
26     struct Printable methods;
27 };
28
29 int main()
30 {
31     char *my_str = "Hello !";
32     struct route route_1 = { 6, 3 };
33     struct route route_2 = { 7, 2 };
34     struct fat_ptr_printable arr[3] = {
35         { my_str, { &print_str } },
36         { &route_1, { &print_route } },
37         { &route_2, { &print_route } }
38     };
39     for (int i = 0; i < 3; i++) {
40         arr[i].methods.print(arr[i].data);
41     };
42     return 0;
43 }
```

Polymorphisme simulé (Rust)

- Plutôt que de stocker le pointeur vers l'interface dans la structure, nous allons le stocker à l'extérieur

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism_fat.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Printable {
5     void (*print)(void*);
6 };
7
8 struct route {
9     int arrival;
10    int hops;
11 };
12
13 void print_route(void *ptr_route) {
14     printf("Route: arrival at %d with %d hops\n",
15            ((struct route *)ptr_route)->arrival,
16            ((struct route *)ptr_route)->hops
17        );
18 }
19
20 void print_str(void *ptr_str){
21     printf("String: %s\n", (char *) ptr_str);
22 }
23
24 struct fat_ptr_printable {
25     void *data;
26     struct Printable methods;
27 };
28
29 int main()
30 {
31     char *my_str = "Hello !";
32     struct route route_1 = { 6, 3 };
33     struct route route_2 = { 7, 2 };
34     struct fat_ptr_printable arr[3] = {
35         { my_str, { &print_str } },
36         { &route_1, { &print_route } },
37         { &route_2, { &print_route } }
38     };
39     for (int i = 0; i < 3; i++) {
40         arr[i].methods.print(arr[i].data);
41     };
42     return 0;
43 }
```

Polymorphisme simulé (Rust)

- Plutôt que de stocker le pointeur vers l'interface dans la structure, nous allons le stocker à l'extérieur
- Nous utiliserons directement **char *** pour les chaîne de caractère, plus besoin d'une structure

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism_fat.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Printable {
5     void (*print)(void*);
6 };
7
8 struct route {
9     int arrival;
10    int hops;
11 };
12
13 void print_route(void *ptr_route) {
14     printf("Route: arrival at %d with %d hops\n",
15            ((struct route *)ptr_route)->arrival,
16            ((struct route *)ptr_route)->hops
17        );
18 }
19
20 void print_str(void *ptr_str){
21     printf("String: %s\n", (char *) ptr_str);
22 }
23
24 struct fat_ptr_printable {
25     void *data;
26     struct Printable methods;
27 };
28
29 int main()
30 {
31     char *my_str = "Hello !";
32     struct route route_1 = { 6, 3 };
33     struct route route_2 = { 7, 2 };
34     struct fat_ptr_printable arr[3] = {
35         { my_str, { &print_str } },
36         { &route_1, { &print_route } },
37         { &route_2, { &print_route } }
38     };
39     for (int i = 0; i < 3; i++) {
40         arr[i].methods.print(arr[i].data);
41     };
42     return 0;
43 }
```

Polymorphisme simulé (Rust)

- Plutôt que de stocker le pointeur vers l'interface dans la structure, nous allons le stocker à l'extérieur
- Nous utiliserons directement **char *** pour les chaîne de caractère, plus besoin d'une structure
- Nous allons implémenter un "**fat pointer**" afin de "wrapper" (encapsuler) un pointeur générique (void *) vers les données et un "**struct Printable**" portant l'adresse de la fonction pour le type pointé

Pointeurs génériques & pointeurs de fonction



```
examples > C polymorphism_fat.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Printable {
5     void (*print)(void*);
6 };
7
8 struct route {
9     int arrival;
10    int hops;
11 };
12
13 void print_route(void *ptr_route) {
14     printf("Route: arrival at %d with %d hops\n",
15            ((struct route *)ptr_route)->arrival,
16            ((struct route *)ptr_route)->hops
17        );
18 }
19
20 void print_str(void *ptr_str){
21     printf("String: %s\n", (char *) ptr_str);
22 }
23
24 struct fat_ptr_printable {
25     void *data;
26     struct Printable methods;
27 };
28
29 int main()
30 {
31     char *my_str = "Hello !";
32     struct route route_1 = { 6, 3 };
33     struct route route_2 = { 7, 2 };
34     struct fat_ptr_printable arr[3] = {
35         { my_str, { &print_str } },
36         { &route_1, { &print_route } },
37         { &route_2, { &print_route } }
38     };
39     for (int i = 0; i < 3; i++) {
40         arr[i].methods.print(arr[i].data);
41     };
42     return 0;
43 }
```

Polymorphisme simulé (Rust)

- Plutôt que de stocker le pointeur vers l'interface dans la structure, nous allons le stocker à l'extérieur
- Nous utiliserons directement **char *** pour les chaîne de caractère, plus besoin d'une structure
- Nous allons implémenter un "**fat pointer**" afin de "wrapper" (encapsuler) un pointeur générique (void *) vers les données et un "**struct Printable**" portant l'adresse de la fonction pour le type pointé
- "**Magie**" : on obtient le même résultat, avec le même degré d'indirection, le même overhead, mais cette fois **sans cast**

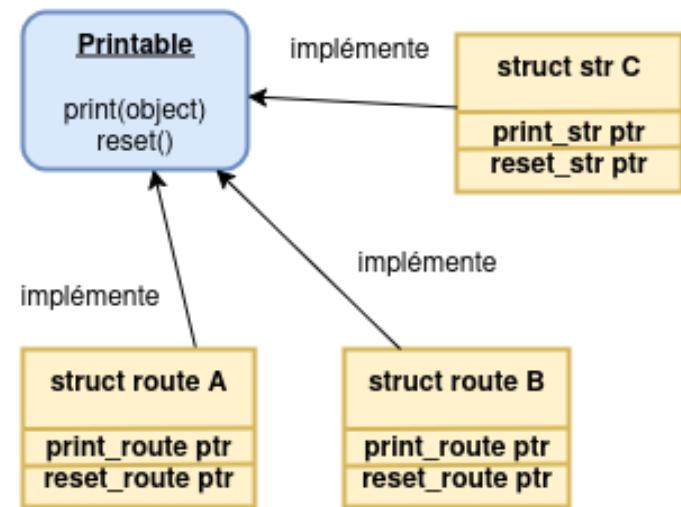
```
(base) → examples git:(main) ✘ gcc ./polymorphism_fat.c -o polymorphism_fat
(base) → examples git:(main) ✘ ./polymorphism_fat
String: Hello !
Route: arrival at 6 with 3 hops
Route: arrival at 7 with 2 hops
(base) → examples git:(main) ✘
```

Pointeurs génériques & pointeurs de fonction



Pour conclure

- Attention, nous avons vu ici une méthode parmi d'autres pour simuler le polymorphisme à l'exécution.
- Nous avons un problème de mémoire : si l'on ajoute des méthodes à notre **struct Printable**, chaque structure implémentant ce contrat va augmenter en taille !

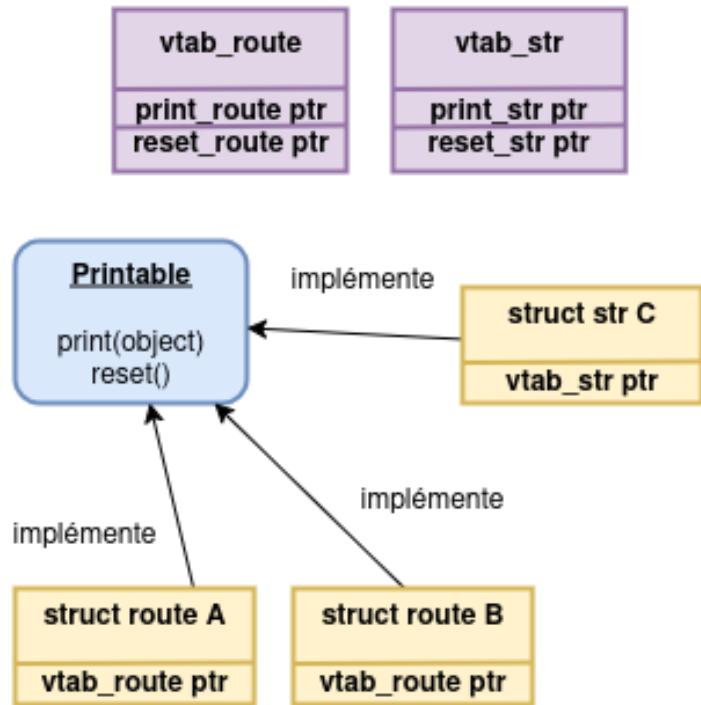


Pointeurs génériques & pointeurs de fonction

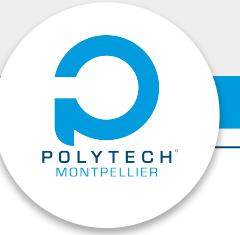


Pour conclure

- Attention, nous avons vu ici une méthode parmi d'autres pour simuler le polymorphisme à l'exécution.
- Nous avons un problème de mémoire : si l'on ajoute des méthodes à notre **struct Printable**, chaque structure implémentant ce contrat va augmenter en taille !
- Nous pouvons pallier ce problème avec une **vtable**. Pour ce faire, nous devons créer une table (ou structure) par type ou « classe » : une pour toutes les **struct route**, une autre pour toutes les **struct str**, etc. Les objets portent seulement une seule adresse : celle de la table associée à leur type.
- En C++ : peu importe le nombre de méthodes de notre interface, chaque objet d'un type l'implémentant ne verra sa taille en mémoire augmenter que de **la taille d'une adresse vers la table**.
- En Rust : pareil, le fat pointeur ne grossit pas en fonction d'une nombre de méthodes, **on stock aussi une adresse vers la table**.

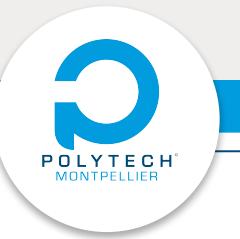


Makefile



On peut supposer que la méthode Rust
est plus propre, pourquoi ?

→ Pour les vtables, exemple complet sur moodle !



Qualité logicielle & Ownership

Qualité logicielle & Ownership



La qualité logicielle, qu'est-ce que c'est ?

Dans cette partie, nous mettons de côté les aspects liés au respect des spécifications fonctionnelles (le logiciel est-il efficace ?) et à la performance (est-il efficient ?), autrement dit : répond-il correctement au besoin métier, et le fait-il avec un usage optimal des ressources.

Nous n'abordons pas non plus l'utilisation d'outils tiers (e.g. Valgrind).

Ici, nous considérons la qualité logicielle sous l'angle de la maintenabilité, de la robustesse, et surtout du respect des **bonnes pratiques de programmation**. C'est sur ce dernier point que nous allons nous concentrer

Qualité logicielle & Ownership



Nommage

"There are only two hard things in Computer Science: cache invalidation and naming things."
(citation célèbre attribuée à Phil Karlton)

Il faut nommer les variables et fonctions avec des noms **signifiants** (et si possible avec plus d'une lettre) On tolérera des exceptions si l'ambiguïté n'est pas possible. Par exemple **i** pour une itération ou **x** et **y** dans un repère.

On cherche à trouver la position relative d'un point (struct pos) par rapport à un autre. E.g. si on veut la position relative de {1, 3} par rapport à {2, 4}, alors on obtient {-1, -1}

Quelle définition préférez vous ?

```
struct pos my_function(struct pos p1, struct pos p2);  
  
struct pos get_relative_pos(struct pos target, struct pos origin);
```

La deuxième → Nom signifiant et une position relative dépend d'une origine

Qualité logicielle & Ownership



Commentaires

Pas de commentaires pour faire beau → pas de paraphrase (exception : en cas d'utilisation d'outils de génération de documentation).

Exemple : cette fonction n'a probablement pas besoin de commentaires :

```
struct pos get_relative_pos(struct pos target, struct pos origin);
```

Un commentaire doit ajouter de l'information à la simple lecture du prototype. Si votre fonction renvoi un **int** il peut être intéressant de spécifier sa nature si elle ne peut être déduite, et ce avec des commentaires (e.g. code d'erreurs, code de succès, etc.). Si la fonction est complexe, spécifier son fonctionnement est bienvenu.

Exemple : la définition de cette fonction, seule, ne serait pas suffisant

```
// Implementation of Dijkstra that finds the shortest route from "src" to "dst" for a bundle "msg" injected
// at "time". Pathfinding first minimizes the hop count, and then the arrival time ("Hop"). The exploration
// is based on Contact Graph Routing, jumping from contact to contact as if vertices were contacts and edges
// periods of retention ("ContactParenting"). If "msg" isn't NULL, the msg characteristics are taken into
// account for filtering.
struct route HopDijkstraContactParenting(struct graph grph, int src, int dst, int time, struct bundle *msg);
```

Qualité logicielle & Ownership



Allocation avec principe d'ownership

Pour bien traiter l'allocation dynamique (mais pas que), il faut introduire le principe **d'ownership**, ou **propriété**, et **respecter ces quelques règles** :

- Une fonction qui **alloue** de la mémoire devient **propriétaire** de cette mémoire
- Le pointeur vers cette mémoire peut être **borrowed**, ou **prêté**, à une autre fonction, si le pointeur est **prêté**, la fonction appelante garde la propriété, la **fonction appelée n'aura pas le droit de libérer la mémoire**
- **La propriété peut être cédée** de la fonction appelante à la fonction appelée, dans ce cas, **la fonction appelante n'a plus le droit d'utiliser le pointeur après l'appel**
- **Une fonction qui possède la propriété** d'une zone mémoire doit ou **céder la propriété, ou la libérer avant son retour**
- **malloc transfère la propriété à l'appelant, free récupère la propriété**

Cette partie sert à préparer le cours sur Rust, certaines terminologies/techniques ne reflètent pas nécessairement les standards de l'industrie (adaptez-vous à votre entreprise)

Qualité logicielle & Ownership



Allocation avec principe d'ownership

En Rust, l'ownership est intégrée au langage, et vérifiée à la compilation. En C, il faut trouver d'autres moyens. Une méthode simple est d'adapter le nommage, pour intégrer ces mots clefs (par exemple) : **new**, **take**, **borrow**. Exemple :

```
struct route *new_route(int src, int dst, struct path *take_path, struct context *borrow_ctx);
```

Et à l'utilisation :

```
// malloc transfers ownership (as if "new_malloc")
struct path *found_path = malloc(sizeof(struct path));

// < Here I init found_path >

// before call: ctx borrowed, found_path is owned
struct route *rte = new_route(1, 4, found_path, ctx)
// after call
// ctx's ownership was just borrowed, I can still use it
// found_path's ownership was taken by new_route, I am not allowed to reuse
// I got ownership of rte, I need to free it, or transfer ownership

// < here I do fancy stuff here >

// free takes ownership (as if "take_free")
free(rte);
// I do not own anything anymore, the function can safely exit,
return 0;
```

Qualité logicielle & Ownership



Allocation avec principe d'ownership

```
examples > c ownership.c > ownership(path **)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct path{
5     // <some members >
6 };
7
8 void ownership_test(struct path **take_var) {
9     struct path *var = *take_var;
10    *take_var = NULL;
11    printf("Dans la fonction %p\n", var);
12    // < do something with var >
13    free(var);
14 }
15
16 int main()
17 {
18     struct path *my_path =(struct path *) malloc(sizeof(struct path));
19     printf("Avant transfert %p\n", my_path);
20     ownership_test(&my_path);
21     printf("Après transfert %p\n", my_path);
22     return 0;
23 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

```
● (base) → examples git:(main) ✘ gcc ownership.c -o ownership
● (base) → examples git:(main) ✘ ./ownership
Avant transfert 0x5e2ede5ed2a0
Dans la fonction 0x5e2ede5ed2a0
Après transfert (nil)
○ (base) → examples git:(main) ✘
```

On peut également "forcer" le transfert de propriété programmatiquement :

- La fonction **main** possède la propriété de **my_path**

Qualité logicielle & Ownership



Allocation avec principe d'ownership

```
examples > c ownership.c > ownership(path **)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct path{
5     // <some members >
6 };
7
8 void ownership_test(struct path**take_var) {
9     struct path *var = *take_var;
10    *take_var = NULL;
11    printf("Dans la fonction %p\n", var);
12    // < do something with var >
13    free(var);
14 }
15
16 int main()
17 {
18     struct path *my_path =(struct path *) malloc(sizeof(struct path));
19     printf("Avant transfert %p\n", my_path);
20     ownership_test(&my_path);
21     printf("Après transfert %p\n", my_path);
22     return 0;
23 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

```
● (base) → examples git:(main) ✘ gcc ownership.c -o ownership
● (base) → examples git:(main) ✘ ./ownership
Avant transfert 0x5e2ede5ed2a0
Dans la fonction 0x5e2ede5ed2a0
Après transfert (nil)
○ (base) → examples git:(main) ✘
```

On peut également "forcer" le transfert de propriété programmatiquement :

- La fonction **main** possède la propriété de **my_path**
- La propriété doit être transférée (**take_var**) lors d'un appel à **ownership_test**

Qualité logicielle & Ownership



Allocation avec principe d'ownership

```
examples > c ownership.c > ownership(path **)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct path{
5     // <some members >
6 };
7
8 void ownership_test(struct path **take_var) {
9     struct path *var = *take_var;
10    *take_var = NULL;
11    printf("Dans la fonction %p\n", var);
12    // < do something with var >
13    free(var); ←
14 }
15
16 int main()
17 {
18     struct path *my_path =(struct path *) malloc(sizeof(struct path));
19     printf("Avant transfert %p\n", my_path);
20     ownership_test(&my_path);
21     printf("Après transfert %p\n", my_path);
22     return 0;
23 }
```

free ne set **pas** le pointeur à 0

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

```
● (base) → examples git:(main) ✘ gcc ownership.c -o ownership
● (base) → examples git:(main) ✘ ./ownership
Avant transfert 0x5e2ede5ed2a0
Dans la fonction 0x5e2ede5ed2a0
Après transfert (nil)
○ (base) → examples git:(main) ✘
```

On peut également "**forcer**" le transfert de propriété programmatiquement :

- La fonction **main** possède la propriété de **my_path**
- La propriété doit être transférée (**take_var**) lors d'un appel à **ownership_test**
- En passant l'**adresse du pointeur**, on peut "forcer" le transfert en passant à **NULL** la valeur du pointeur dans **main**

Qualité logicielle & Ownership



Allocation avec principe d'ownership

```
examples > c ownership.c > ownership(path **)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct path{
5     // <some members >
6 };
7
8 void ownership_test(struct path **take_var) {
9     struct path *var = *take_var;
10    *take_var = NULL;
11    printf("Dans la fonction %p\n", var);
12    // < do something with var >
13    free(var);
14 }
15
16 int main()
17 {
18     struct path *my_path =(struct path *) malloc(sizeof(struct path));
19     printf("Avant transfert %p\n", my_path);
20     ownership_test(&my_path);
21     printf("Après transfert %p\n", my_path);
22     return 0;
23 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

```
● (base) → examples git:(main) ✘ gcc ownership.c -o ownership
● (base) → examples git:(main) ✘ ./ownership
Avant transfert 0x5e2ede5ed2a0
Dans la fonction 0x5e2ede5ed2a0
Après transfert (nil)
○ (base) → examples git:(main) ✘
```

On peut également "**forcer**" le transfert de propriété programmatiquement :

- La fonction **main** possède la propriété de **my_path**
- La propriété doit être transférée (**take_var**) lors d'un appel à **ownership_test**
- En passant l'**adresse du pointeur**, on peut "forcer" le transfert en passant à **NULL** la valeur du pointeur dans **main**
- Même si elle le voulait, **main** ne peut plus utiliser **my_path**

Qualité logicielle & Ownership



Ownership et modification

Autoriser une fonction à borrow un objet qui nous appartient ne propose aucun contrôle concernant la gestion du droit de modification

Le langage C propose le mot-clef **const**.

Pour retenir la syntaxe, on dira que le mot clef **const** s'associe au **type** et non à la **variable** (ou au mot suivant)

En effet, on peut lire "**(const int) var1**", et "**(const int) *ptr**", c'est le **int** qui est constant, et non la variable (la valeur d'adresse) dans le cas du pointeur

```
1 #include <stdlib.h>
2
3 int main(void) {
4     const int var1 = 2;
5     int var2 = 32;
6     // Attention ce n'est pas le pointeur qui est const
7     // mais la valeur pointé si on deref
8     const int *ptr = &var2;
9     var2 = 21; // Valide
10    ptr = NULL; // Valide
11    *ptr = 42; // Invalide, modif de var2 par deref
12    return EXIT_SUCCESS;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

✖ (base) → examples git:(main) ✘ gcc const.c
const.c: In function 'main':
const.c:12:10: error: assignment of read-only location '*ptr'
12 | *ptr = 42; // Invalide, modif de var2 par deref
| ^

○ (base) → examples git:(main) ✘

Qualité logicielle & Ownership



Ownership et modification

Définir une adresse constante est possible, dans ce cas, on place **const** avant la variable, après l'étoile

Et pour les adresses d'adresses ?

```
examples > C const_2.c > ...
1  #include <stdlib.h>
2
3  int main(void) {
4      int var = 32;
5      int * const ptr1 = &var;
6      *ptr1 = 42; // ici c'est l'adresse qui est constante
7      ptr1 = NULL;
8      return EXIT_SUCCESS;
9  }

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    REFACTOR PREVIEW

✖ (base) → examples git:(main) ✘ gcc const_2.c
const_2.c: In function 'main':
const_2.c:7:10: error: assignment of read-only variable 'ptr1'
    7 |         ptr1 = NULL;
          ^
```

Qualité logicielle & Ownership



Ownership et modification

Oui c'est possible



"où placer le const ?"

```
examples > c const_3.c > ...
1  #include <stdlib.h>
2
3  int main(void) {
4      int var = 32;
5      int *ptr1 = &var;
6
7      int ** const ptr2 = &ptr1;
8      // const est devant ptr2
9      // c'est le pointeur de 2ème niveau qui est constant
10     *ptr2 = NULL; // set ptr1 à NULL
11     **ptr2 = 42; // valide aussi, le int n'est pas const
12     ptr2 = NULL;
13     return EXIT_SUCCESS;
14 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

```
® (base) → examples git:(main) ✘ gcc const_3.c
const_3.c: In function 'main':
const_3.c:12:10: error: assignment of read-only variable 'ptr2'
    12 |         ptr2 = NULL;
          ^
○ (base) → examples git:(main) ✘
```

Qualité logicielle & Ownership



Ownership et modification

Ici le pointeur de premier niveau est constant : la valeur pointée par ptr2, donc *ptr2.

Et effectivement on ne peut pas modifier *ptr2

Mais si je peux dire que l'adresse *ptr est constant sur du pointage à 2 niveaux, puis-je dire que le int est constant sur un simple pointeur avec la même syntaxe ?

```
examples > c const_4.c > ...
1  #include <stdlib.h>
2
3  int main(void) {
4      int var = 32;
5      int *ptr1 = &var;
6      // ce n'est plus ptr2 qui est const, mais le pointeur pointé
7      int * const *ptr2 = &ptr1;
8      ptr2 = NULL;
9      *ptr2 = NULL; // et effectivement...
10
11     return EXIT_SUCCESS;
12 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL REFACTOR PREVIEW

- ✖ (base) → examples git:(main) ✘ gcc const_4.c
const_4.c: In function 'main':
const_4.c:9:11: error: assignment of read-only location '*ptr2'
9 | *ptr2 = NULL; // et effectivement...
| ^
- (base) → examples git:(main) ✘

Qualité logicielle & Ownership



Ownership et modification

Oui !

et même sans pointeur :

"int const var" est aussi
une syntaxe valide !

```
examples > C const_5.c > ...
1  #include <stdlib.h>
2
3  int main(void) {
4      int var = 32;
5      // équivalent à "const int *ptr1" ??
6      int const* ptr1 = &var;
7      *ptr1 = 42;
8      return EXIT_SUCCESS;
9  }
10

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    REFACTOR PREVIEW

① (base) → examples git:(main) ✘ gcc const_5.c
const_5.c: In function 'main':
const_5.c:7:11: error: assignment of read-only location '*ptr1'
      7 |         *ptr1 = 42;
                  ^
○ (base) → examples git:(main) ✘
```

C'est l'heure du cheat code !

Qualité logicielle & Ownership

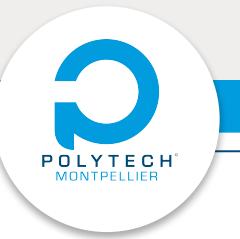


Ownership et modification

Cheat code : Toujours mettre
le const après le type



- int const **var** :
var = val → illégal
- int * const **ptr** :
ptr = addr → illégal
- int const ***ptr** :
***ptr** = var → illégal
- int const ****ptr_of_ptr** :
****ptr_of_ptr** = var → illégal
- int * const ***ptr_of_ptr** :
***ptr_of_ptr** = addr → illégal
- int ** const **ptr_of_ptr** :
ptr_of_ptr = addr_of_addr → illégal
- Etc.



Prototypes, Headers, Compilation

Prototypes, Headers, Compilation



The elephant in the room



Allons-y, étape par étape..

Prototypes, Headers, Compilation



Première exposition au comportement de gcc

- Le compilateur traite le code du haut vers le bas
- Il est parfois juste impossible de simplement ordonner les déclarations (comme dans l'exemple)
- Si une fonction n'est pas définie avant son appel, le compilateur produira un warning.
Intéressant, pourquoi pas une erreur ?
- Pour aider le compilateur, on doit renseigner la **signature de la fonction** avec un **prototype** : son nom, les types des arguments et du retour

```
home > olivier > Documents > Lectures > C > examples > c prototype.c > ...
1  #include <stdio.h>
2
3  int is_odd(int n); prototype
4
5  int is_even(int n) {
6      if (n == 0)
7          return 1;
8      else
9          return is_odd(n - 1); Premier appel avant la définition
10 }
11
12 int is_odd(int n) { Definition
13     if (n == 0)
14         return 0;
15     else
16         return is_even(n - 1);
17 }
18
19 int main() {
20     int num = 7;
21     // opérateur ternaire :
22     // <cond> ? <expr> : <alt>; (si <cond> alors <expr> sinon <alt>)
23     printf("%d is even? %s\n", num, is_even(num) ? "Yes" : "No");
24     printf("%d is odd? %s\n", num, is_odd(num) ? "Yes" : "No");
25     return 0;
26 }
27
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ gcc prototype.c -o prototype
- (base) → examples git:(main) ✘ ./prototype

```
7 is even? No
7 is odd? Yes
```

- (base) → examples git:(main) ✘

Prototypes, Headers, Compilation



```
examples > C prototype_2.c > ...
1  #include <stdio.h>
2
3 // pas de proto
4
5 int is_even(int n) {
6     if (n == 0)
7         return 1;
8     else
9         return is_odd(n - 1); // 1 seul argument
10 }
11
12 int is_odd(int n, int dummy[100]) {
13     if (n == 0)
14         return 0;
15     else
16         return is_even(n - 1);
17 }
18
19 int main() {
20     int num = 7;
21     int dummy[100];
22     // opérateur ternaire :
23     // <cond> ? <expr> : <alt>; (si <cond> alors <expr> sinon <alt>)
24     printf ("%d is even? %s\n", num, is_even(num) ? "Yes" : "No");
25     printf ("%d is odd? %s\n", num, is_odd(num, dummy) ? "Yes" : "No");
26     return 0;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) ➔ examples git:(main) ✘ gcc prototype_2.c -o prototype_2
prototype_2.c: In function 'is_even':
prototype_2.c:9:16: warning: implicit declaration of function 'is_odd' [-Wimplicit-function-declaration]
 9 | return is_odd(n - 1); // 1 seul argument
 ^~~~~~
- (base) ➔ examples git:(main) ✘ ./prototype_2
7 is even? No
7 is odd? Yes
- (base) ➔ examples git:(main) ✘

Sans proto, cela fonctionne même avec un changement de la signature !



Même si faire des pronostiques sur du code faux est discutable :
Savoir que ce code ne fonctionne plus si on échange de position n et dummy dans la définition, ainsi que nos connaissances du fonctionnement de la mémoire nous donne un indice sur ce qu'il ce passe..

Prototypes, Headers, Compilation



```
examples > C prototype_3.c > is_even(int)
1  #include <stdio.h>
2
3 // pas de proto
4
5 int is_even(int n) {
6     if (n == 0)
7         return 1;
8     else
9         return is_odd(n - 1); // 1 seul argument
10 }
11
12 int is_odd(int n, int dummy[100]) {
13     if (n == 0)
14         return 0;
15     else
16         return is_even(n - 1);
17 }
18
19 int main() {
20     int num = 7;
21     int dummy[100];
22     // opérateur ternaire :
23     // <cond> ? <expr> : <alt>; (si <cond> alors <expr> sinon <alt>)
24     printf("%d is even? %s\n", num, is_even(num) ? "Yes" : "No");
25     printf("%d is odd? %s\n", num, is_odd(num) ? "Yes" : "No");
26     return 0;
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(base) ~ examples git:(main) ✘ gcc prototype_3.c -o prototype_3
prototype_3.c: In function 'is_even':
prototype_3.c:9:16: warning: implicit declaration of function 'is_odd' [-Wimplicit-f
    9 |         return is_odd(n - 1); // 1 seul argument
          ^~~~~~
prototype_3.c: In function 'main'.
prototype_3.c:25:36: error: too few arguments to function 'is_odd'
    25 |     printf("%d is odd? %s\n", num, is_odd(num) ? "Yes" : "No");
          ^~~~~~
prototype_3.c:12:5: note: declared here
    12 | int is_odd(int n, int dummy[100]) {
          ^~~~~~
```

La fonction main est une fonction comme une autre

Au moment où main est compilée, is_odd est définie, donc si on supprime le deuxième argument dummy dans l'appel..

Prototypes, Headers, Compilation



```
examples > C prototype_4.c > is_even(int)
1  #include <stdio.h>
2
3 // pas de proto
4
5 int main() {
6     int num = 7;
7     int dummy[100];
8     // opérateur ternaire :
9     // <cond> ? <expr> : <alt>; (si <cond> alors <expr> sinon <alt>)
10    printf("%d is even? %s\n", num, is_even(num) ? "Yes" : "No");
11    printf("%d is odd? %s\n", num, is_odd(num) ? "Yes" : "No");
12    return 0;
13 }
14
15 int is_even(int n) {
16     if (n == 0)
17         return 1;
18     else
19         return is_odd(n - 1); // 1 seul argument
20 }
21
22 int is_odd(int n, int dummy[100]) {
23     if (n == 0)
24         return 0;
25     else
26         return is_even(n - 1);
27 }
28
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● (base) → examples git:(main) ✘ gcc prototype_4.c -o prototype_4
prototype_4.c: In function ‘main’:
prototype_4.c:10:37: warning: implicit declaration of function ‘is_even’ [-Wimplicit-
 10 |     printf("%d is even? %s\n", num, is_even(num) ? "Yes" : "No");
                  ^~~~~~
prototype_4.c:11:36: warning: implicit declaration of function ‘is_odd’ [-Wimplicit-
 11 |     printf("%d is odd? %s\n", num, is_odd(num) ? "Yes" : "No");
                  ^~~~~~
● (base) → examples git:(main) ✘ ./prototype_4
7 is even? No
7 is odd? Yes
● (base) → examples git:(main) ✘
```

La fonction main est une fonction comme une autre

Mais si on déplace main au dessus de **is_odd**, on devrait avoir le même comportement que l'appel à **is_odd** dans **is_even**, et effectivement..

Ça "fonctionne" (ne faites pas ça)

Prototypes, Headers, Compilation



Compilation : prenons le problème à l'envers

La compilation (simplifiée) comprend au moins 3 étapes :

- 1) **Preprocessing** : suppression des commentaires, développement des macros, les directives #include sont remplacées par le contenu des fichiers correspondants (par quoi ? où sont-ils stockés ?) → toujours du code c : **extension .c**
- 2) **Compilation & Assembly** : traduction en assembleur puis code machine : **extension .o**
- 3) **Linking** : liaison entre les appels de fonction et leurs définitions : **pas d'extension**

Réponse à pourquoi seulement un warning : Le compilateur peut toujours produire le binaire des fonctions appelantes pendant la phase de compilation, même en l'absence de prototype ou de définition préalable de la fonction appelée.

Si la fonction est effectivement définie dans un autre fichier ou plus tard dans le même fichier, la phase de linking (étape 3) parvient à résoudre l'appel, et aucune erreur de liaison ne se produit.

Prototypes, Headers, Compilation



Linking : libraries

Ok donc on comprend le rôle du **linking** (**édition de liens**)... testons un nouvel import

On utilise maintenant la librairie donnant accès à certaines fonctions pour les mathématiques, qu'on **importe comme les autres librairies**

Et ça ne fonctionne pas !

```
examples > C sqrt.c > ...
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h> // for sqrt
4
5  int main (int argc, char *argv[]) {
6      if (argc < 2) {
7          printf ("Usage: $ %s <float>\n", argv[0]);
8          return EXIT_FAILURE;
9      }
10
11     // Pour l'exemple, "atof" marche aussi mais
12     // ne détecte pas les erreurs
13     char * res = NULL;
14     float f = strtod(argv[1], &res);
15     if (res == argv[1]) {
16         printf ("%s is not a float\n", argv[1]);
17         return EXIT_FAILURE;
18     }
19     printf ("sqrt(%f) = %f\n", f, sqrt(f));
20
21 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
✖ (base) → examples git:(main) ✘ gcc sqrt.c -o sqrt
/usr/bin/ld: /tmp/cc90tLW3.o: in function `main':
sqrt.c:(.text+0xca): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
✖ (base) → examples git:(main) ✘
```

Prototypes, Headers, Compilation



Linking : libraries

Ok donc on comprend le rôle du **linking** (**édition de liens**)... testons un nouvel import

On utilise maintenant la librairie donnant accès à certaines fonctions pour les mathématiques, qu'on **importe comme les autres librairies**

Et ça ne fonctionne pas !

Ok testons avec **-l m** (le **l** pour library et **m** pour maths) → **ça fonctionne**

Ok mais pourquoi pas d'options particulières pour **stdlib** et **stdio** ? **Le compilateur est sympa et l'ajoute par défaut : -lc pour libc. Option reconnue !**

```
examples > C sqrt.c > ...
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h> // for sqrt
4
5  int main (int argc, char *argv[]) {
6      if (argc < 2) {
7          printf ("Usage: $ %s <float>\n", argv[0]);
8          return EXIT_FAILURE;
9      }
10
11     // Pour l'exemple, "atof" marche aussi mais
12     // ne détecte pas les erreurs
13     char * res = NULL;
14     float f = strtod(argv[1], &res);
15     if (res == argv[1]) {
16         printf ("%s is not a float\n", argv[1]);
17         return EXIT_FAILURE;
18     }
19     printf("sqrt(%f) = %f\n", f, sqrt(f));
20     return EXIT_SUCCESS;
21 }
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
● (base) → examples git:(main) ✘ gcc sqrt.c -o sqrt -lm -lc			
● (base) → examples git:(main) ✘ ./sqrt 50.25			
	sqrt(50.250000) = 7.088723		
○ (base) → examples git:(main) ✘			

Prototypes, Headers, Compilation



Linking : libraries

Ok on touche au but, si on ne met pas le **-lm**, ça ne compile pas (fonction non définie), mais si on laisse le **-lm** et qu'on enlève l'import, se pourraut-il que..

```
examples > C sqrt.c > ...
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h> // for sqrt
4
5  int main (int argc, char *argv[]) {
6      if (argc < 2) {
7          printf ("Usage: $ %s <float>\n", argv[0]);
8          return EXIT_FAILURE;
9      }
10
11     // Pour l'exemple, "atof" marche aussi mais
12     // ne détecte pas les erreurs
13     char * res = NULL;
14     float f = strtod(argv[1], &res);
15     if (res == argv[1]) {
16         printf ("%s is not a float\n", argv[1]);
17         return EXIT_FAILURE;
18     }
19     printf("sqrt(%f) = %f\n", f, sqrt(f));
20     return EXIT_SUCCESS;
21 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ gcc sqrt.c -o sqrt -lm -lc
- (base) → examples git:(main) ✘ ./sqrt 50.25
sqrt(50.250000) = 7.088723
- (base) → examples git:(main) ✘

Prototypes, Headers, Compilation



Linking : libraries

Ok on touche au but, si on ne met pas le **-lm**, ça ne compile pas (fonction non définie), mais si on laisse le **-lm** et qu'on enlève l'import, se pourraut-il que..

BINGO !

```
examples > c sqrt_2.c > ...
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main (int argc, char *argv[])
5  {
6      if (argc < 2)
7          printf("Usage: $ %s <float>\n", argv[0]);
8      return EXIT_FAILURE;
9
10     // Pour l'exemple, "atof" marche aussi mais
11     // ne détecte pas les erreurs
12     char * res = NULL;
13     float f = strtod(argv[1], &res);
14     if (res == argv[1])
15         printf("%s is not a float\n", argv[1]);
16     return EXIT_FAILURE;
17 }
18 printf("sqrt(%f) = %f\n", f, sqrt(f));
19 return EXIT_SUCCESS;
20 }
21

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
● (base) → examples git:(main) ✘ gcc sqrt_2.c -o sqrt_2 -lm
sqrt_2.c: In function 'main'.
sqrt_2.c:18:34: warning: implicit declaration of function 'sqrt' [-Wimplicit]
18 |     printf("sqrt(%f) = %f\n", f, sqrt(f));
|               ^
sqrt_2.c:3:1: note: include '<math.h>' or provide a declaration of 'sqrt'
  2 | #include <stdio.h>
  +++ |+#include <math.h>
  3 |
sqrt_2.c:18:34: warning: incompatible implicit declaration of built-in function 'sqrt'
18 |     printf("sqrt(%f) = %f\n", f, sqrt(f));
|               ^
sqrt_2.c:18:34: note: include '<math.h>' or provide a declaration of 'sqrt'
(base) → examples git:(main) ✘ ./sqrt_2 50.25
sqrt(50.250000) = 7.088723
(base) → examples git:(main) ✘
```

Prototypes, Headers, Compilation



Linking : libraries

Ok on touche au but, si on ne met pas le **-lm**, ça ne compile pas (fonction non définie), mais si on laisse le **-lm** et qu'on enlève l'import, se pourrait-il que..

BINGO !

On observe que le **-lm** suffit à permettre au compilateur de résoudre l'édition de liens

L'absence de l'include ne provoque qu'un warning de déclaration implicite, **le code n'est pas dans le .h : il est ailleurs !**

D'où le ".h", pour "**header**", un fichier d'entête ne contient généralement que les signatures : des **prototypes** !

```
examples > c sqrt_2.c > ...
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main (int argc, char *argv[])
5  {
6      if (argc < 2)
7          printf("Usage: $ %s <float>\n", argv[0]);
8      return EXIT_FAILURE;
9  }
10
11 // Pour l'exemple, "atof" marche aussi mais
12 // ne détecte pas les erreurs
13 char * res = NULL;
14 float f = strtod(argv[1], &res);
15 if (res == argv[1])
16     printf("%s is not a float\n", argv[1]);
17     return EXIT_FAILURE;
18 }
19 printf("sqrt(%f) = %f\n", f, sqrt(f));
20 return EXIT_SUCCESS;
21 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- (base) → examples git:(main) ✘ gcc sqrt_2.c -o sqrt_2 -lm
sqrt_2.c: In function 'main':
sqrt_2.c:18:34: warning: implicit declaration of function 'sqrt' [-Wimplicit-function-decl]
18 | printf("sqrt(%f) = %f\n", f, sqrt(f));
| ^~~~
- sqrt_2.c:3:1: note: include '<math.h>' or provide a declaration of 'sqrt'
2 | #include <stdio.h>
+++ |+#include <math.h>
3 |
sqrt_2.c:18:34: warning: incompatible implicit declaration of built-in function 'sqrt' [-Wbuiltin-decl-diff]
18 | printf("sqrt(%f) = %f\n", f, sqrt(f));
| ^~~~
- sqrt_2.c:18:34: note: include '<math.h>' or provide a declaration of 'sqrt'
(base) → examples git:(main) ✘ ./sqrt_2 50.25
sqrt(50.250000) = 7.088723
- (base) → examples git:(main) ✘

Prototypes, Headers, Compilation



Notre compréhension

- Les directives de preprocessing **#include** permettent d'aller chercher "**quelque part**" les prototypes des fonctions que l'on souhaite utiliser en remplaçant la directive par le contenu du fichier de header
- Les options **-I** de **gcc** permettent d'aller chercher "**quelque part**" le code compilé et assemblé des fonctions que l'on souhaite utiliser
- Pour la **librairie standard**, le code correspondant aux prototypes des fonctions dans **stdio.h**, **stdlib.h**, **string.h**, etc. est inclus par défaut. Pour vous en convaincre, compiler avec **-nodefaultlibs**, puis rajouter **-lc**

Prototypes, Headers, Compilation



Notre compréhension

- Les directives de preprocessing **#include** permettent d'aller chercher "**quelque part**" les prototypes des fonctions que l'on souhaite utiliser en remplaçant la directive par le contenu du fichier de header
- Les options **-I** de **gcc** permettent d'aller chercher "**quelque part**" le code compilé et assemblé des fonctions que l'on souhaite utiliser
- Pour la **librairie standard**, le code correspondant aux prototypes des fonctions dans **stdio.h**, **stdlib.h**, **string.h**, etc. est inclus par défaut. Pour vous en convaincre, compiler avec **-nodefaultlibs**, puis rajouter **-lc**

On ne va pas se satisfaire d'un "quelque part"

Prototypes, Headers, Compilation



"Quelque part" → Analysons le preprocessing

```
(base) ➔ ~ echo "" | gcc -E -v -
```

Prototypes, Headers, Compilation



"Quelque part" → Analysons le preprocessing

```
(base) ➔ ~ echo "" | gcc -E -v -
```



```
#include "..." search starts here:  
#include <...> search starts here:  
 /usr/lib/gcc/x86_64-linux-gnu/13/include  
 /usr/local/include  
 /usr/include/x86_64-linux-gnu  
 /usr/include
```

Prototypes, Headers, Compilation



"Quelque part" → Analysons le preprocessing

```
(base) ➔ ~ echo "" | gcc -E -v -
```



```
#include "... search starts here:  
#include <...> search starts here:  
/usr/lib/gcc/x86_64-linux-gnu/13/include  
/usr/local/include  
/usr/include/x86_64-linux-gnu  
/usr/include  
etc...
```

↓ Cherchons le header
de stdlib.h par exemple

```
(base) ➔ ~ ls /usr/include/stdlib.h  
/usr/include/stdlib.h
```

Et le .h contient bien
des prototypes

Prototypes, Headers, Compilation



"Quelque part" → Analysons le preprocessing

```
(base) ➔ ~ echo "" | gcc -E -v -
```



```
#include "..." search starts here:  
#include <...> search starts here:  
 /usr/lib/gcc/x86_64-linux-gnu/13/include  
 /usr/local/include  
 /usr/include/x86_64-linux-gnu  
 /usr/include  
 ...
```

```
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/13:/usr/lib/gcc/x86_64-li  
nux-gnu/13/.../.../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/1  
3/.../.../.../lib/:/lib/x86_64-linux-gnu/:/lib/:/usr/lib/x86_6  
4-linux-gnu/:/usr/lib/:/lib/:/usr/lib/gcc/x86_64-linux-gnu/13/.../  
...:/lib/:/usr/lib/  
COLLECT_GCC_OPTIONS='-E' '-v' '-mtune=generic' '-march=x86-64'
```

Cherchons le header
de stdlib.h par exemple

```
(base) ➔ ~ ls /usr/include/stdlib.h  
/usr/include/stdlib.h
```

Et le .h contient bien
des prototypes

Prototypes, Headers, Compilation



"Quelque part" → Analysons le preprocessing

```
(base) ➔ ~ echo "" | gcc -E -v -
```



```
#include "..." search starts here:  
#include <...> search starts here:  
 /usr/lib/gcc/x86_64-linux-gnu/13/include  
 /usr/local/include  
 /usr/include/x86_64-linux-gnu  
 /usr/include  
 ...  
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/13:/usr/lib/gcc/x86_64-li  
nux-gnu/13/.../.../x86_64-linux-gnu:/usr/lib/gcc/x86_64-linux-gnu/1  
3/.../.../.../lib:/lib/x86_64-linux-gnu:/lib/..:/lib:/usr/lib/x86_6  
4-linux-gnu:/usr/lib/..:/lib:/usr/lib/gcc/x86_64-linux-gnu/13/.../  
...:/lib:/usr/lib/  
COLLECT_GCC_OPTIONS='-E' '-v' '-mtune=generic' '-march=x86-64'
```



Cherchons le header
de stdlib.h par exemple

```
(base) ➔ ~ ls /usr/include/stdlib.h  
/usr/include/stdlib.h
```

Les librairies commencent par "**lib**",
cherchons **libc** par exemple

```
(base) ➔ ~ ldconfig -p | grep -E "^/s*libc\."  
 libc.so.6 (libc6,x86-64) => /lib/x86_64-linux-gnu/libc.so.6
```

Et le .h contient bien
des prototypes

Prototypes, Headers, Compilation



"Quelque part" → Analysons le preprocessing

```
(base) ➔ ~ echo "" | gcc -E -v -
```



```
#include "... search starts here:  
#include <...> search starts here:  
 /usr/lib/gcc/x86_64-linux-gnu/13/include  
 /usr/local/include  
 /usr/include/x86_64-linux-gnu  
 /usr/include  
 ...
```



```
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/13:/usr/lib/gcc/x86_64-li  
nux-gnu/13/.../.../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/1  
3/.../.../.../lib/:/lib/x86_64-linux-gnu/:/lib/..:/lib:/usr/lib/x86_6  
4-linux-gnu:/usr/lib/..:/lib:/usr/lib/gcc/x86_64-linux-gnu/13/.../  
...:/lib:/usr/lib/  
COLLECT_GCC_OPTIONS='-E' '-v' '-mtune=generic' '-march=x86-64'
```

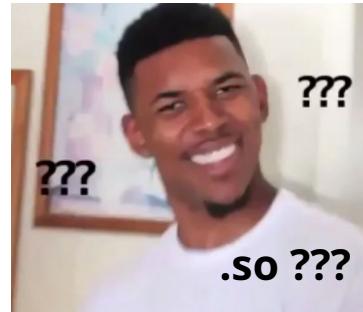


Cherchons le header
de stdlib.h par exemple

Les librairies commencent par "**lib**",
cherchons **libc** par exemple

```
(base) ➔ ~ ls /usr/include/stdlib.h  
/usr/include/stdlib.h
```

```
(base) ➔ ~ ldconfig -p | grep -E "^/s*libc\."  
 libc.so.6 (libc6,x86-64) => /lib/x86_64-linux-gnu/libc.so.6
```



Et le .h contient bien
des prototypes

Prototypes, Headers, Compilation



"Quelque part" → Analysons le preprocessing

```
(base) ➔ ~ echo "" | gcc -E -v -
```



```
#include "... search starts here:  
#include <...> search starts here:  
 /usr/lib/gcc/x86_64-linux-gnu/13/include  
 /usr/local/include  
 /usr/include/x86_64-linux-gnu  
 /usr/include  
 ...
```

```
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/13:/usr/lib/gcc/x86_64-li  
nux-gnu/13/.../.../x86_64-linux-gnu:/usr/lib/gcc/x86_64-linux-gnu/1  
3/.../.../.../lib:/lib/x86_64-linux-gnu:/lib/..:/lib:/usr/lib/x86_6  
4-linux-gnu:/usr/lib/..:/lib:/usr/lib/gcc/x86_64-linux-gnu/13/.../  
...:/lib:/usr/lib/  
COLLECT_GCC_OPTIONS='-E' '-v' '-mtune=generic' '-march=x86-64'
```

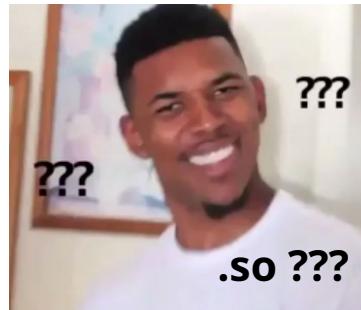


Cherchons le header
de stdlib.h par exemple

Les librairies commencent par "**lib**",
cherchons **libc** par exemple

```
(base) ➔ ~ ls /usr/include/stdlib.h  
/usr/include/stdlib.h
```

```
(base) ➔ ~ ldconfig -p | grep -E "^/s*libc\."  
 libc.so.6 (libc6,x86-64) => /lib/x86_64-linux-gnu/libc.so.6
```



Et le .h contient bien
des prototypes

Note : l'extension
attendue était .a pour
Archive. Une librairie
est une archive de .o

Prototypes, Headers, Compilation



Static Linking vs dynamic linking

L'édition de liens associe les `.o` pour créer l'exécutable. **Problème :**

Beaucoup de programmes utilisent `printf`. Le code associé à `printf` dans `libc.o` serait ajouté au binaire final, et ce pour **chaque programme !**



Prototypes, Headers, Compilation



Static Linking vs dynamic linking

L'édition de liens associe les **.o** pour créer l'exécutable. **Problème :**

Beaucoup de programmes utilisent **printf**. Le code associé à **printf** dans **libc.o** serait ajouté au binaire final, et ce pour **chaque programme !**

Solution : les librairies partagées !

- Appelées **Shared Objects** sur linux (**.so**)
- Appelées **Dynamic-Link Libraries** sur windows (**.dll**)



Prototypes, Headers, Compilation



Résumé de la compilation :

- Le binaire est construit à partir de plusieurs **.o** si le projet à plusieurs fichiers/modules (classiquement 1 module = 1 fichier), et à partir de librairies externes (e.g. **libm.so**, **libc.so**)
- Les librairies sont "ajoutées" à l'étape d'édition de liens
- Avec les librairies statiques **.a** le code utilisé est copié dans l'exécutable à créer
- Avec les librairies dynamique **.so** les adresses des fonctions utilisées sont résolues à l'exécution par le **dynamic loader** et chargées en mémoire. Le binaire obtenu avec gcc n'inclut que des "références" (on n'entre pas dans les détails) vers les librairies dynamiques à utiliser. Ces références ne sont **PAS** des adresses.
- On utilise la même option **-I** pour les librairies statiques et dynamiques : **gcc** cherchera libdummy.a ou libdummy.so avec l'option -ldummy (-l dummy fonctionne aussi)
- Si on produit notre propre librairie (statique ou dynamique), nous ne sommes pas obligés de la copier dans **/lib/** ou **/usr/lib/** pour l'édition de lien. On peut par exemple indiquer à **gcc** de chercher également dans un répertoire particulier, par exemple dans le répertoire courant avec l'option "**-L.**"

Prototypes, Headers, Compilation



Exécution et dynamic loading des .so :

Fournir le **-L** lors de la compilation permet de résoudre les symboles et de **valider** l'édition de lien, mais le dynamic loader a toujours besoin de trouver la librairie au moment de l'exécution, on a au moins 3 options :

- **Option 1** : Si la librairie se trouve dans **/lib/** ou **/usr/lib/**, tout fonctionne sans configuration supplémentaire : gcc la trouve à l'édition de lien et le loader la trouve à l'exécution. Cependant, ce n'est pas optimal, **car ces dossiers sont réservés aux librairies système.**
- **Option 2** : On peut **inclure le chemin du .so lors de l'édition de lien** avec **-Wl,-rpath=<chemin>**. Le chemin est alors enregistré dans l'exécutable et le loader le récupère automatiquement. Limitation : si la librairie est déplacée, **le chemin dans l'exécutable devient invalide, et il faut recompiler pour corriger.**
- **Option 3** : Le dynamic loader peut chercher les librairies dans : les répertoires par défaut, le chemin indiqué par rpath dans l'exécutable, ou les chemins de la variable d'environnement **LD_LIBRARY_PATH**. Cette dernière méthode est **flexible** et **peu intrusive** pour le système : si la librairie change de place, **il suffit de mettre à jour LD_LIBRARY_PATH sans recompiler l'exécutable.**

Prototypes, Headers, Compilation

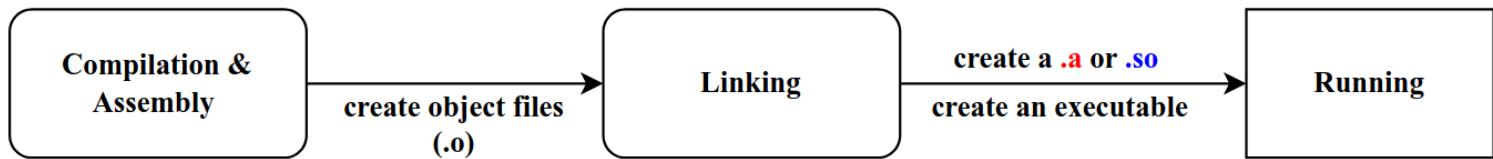


Cartographie des commandes

CREATING
A LIBRARY

Static	gcc -c	ar -rcs	N/A
--------	--------	---------	-----

Dynamic	gcc -c -fPIC	gcc -shared	N/A
---------	--------------	-------------	-----



USING A
LIBRARY

Static	gcc -c -I<lib headers>	gcc -l<lib name> -L<lib path> -o <exe>	./<exe>
--------	------------------------	--	---------

Dynamic	gcc -c -I<lib headers>	gcc -l<lib name> -L<lib path> -Wl,-rpath=<lib path> -o <exe>	./<exe>
---------	------------------------	--	---------

Dynamic	gcc -c -I<lib headers>	gcc -l<lib name> -L<lib path> -o <exe>	LD_LIBRARY_PATH=<lib path> ./<exe>
---------	------------------------	--	---------------------------------------



Makefile

Makefile



Build systems :

Le C est un langage très intéressant pédagogiquement et prédominant dans certains domaines, mais son principal défaut est peut-être **l'absence d'un système de build intégré**

Les langages modernes, comme Rust, offrent des outils puissants tels que Cargo, qui permettent de compiler et de gérer les dépendances avec beaucoup de flexibilité

En C, tout doit encore être fait manuellement : créer les fichiers objets (**.o**) avec l'option **-c**, puis effectuer l'édition de liens en ajoutant les librairies nécessaires avec **-l** et **-L**

Il faut aussi gérer soi-même l'organisation des dossiers : il est courant de séparer les fichiers **.h** et **.c**, et pour les gros modules, de créer des sous-dossiers contenant plusieurs fichiers **.c**

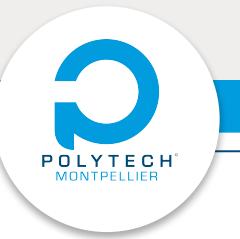
Pour faciliter cette tâche, l'outil historique est **Makefile**. L'essentiel est de comprendre le fonctionnement de la compilation. Pour notre projet, nous nous limiterons aux bases de **Makefile**

N.B. : CMake est plus récent mais il peut être plus complexe à prendre en main et présente ses propres problèmes...

Makefile

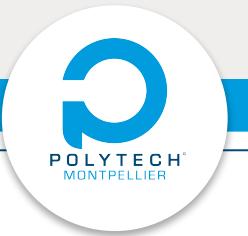


Aller à l'exemple complet sur moodle !



Templates & Generics simulés

Templates



Des vectors... de quoi ?

Avoir des collections d'objets du même type est bien évidemment très classique. Avec le Vector pour illustrer, on peut avoir besoin de **vector de int**, de **vector de double**, etc. etc.

Problème : comment implémenter le vector ?

- Option 1 : on implémente une version par type ? **Ok et pour les types non primitifs comme les structures ?**
- Option 2 : on ne stocke que des pointeurs ? **Ok et l'overhead d'indirection ?**

Solution : métaprogrammation

- On sait qu'une collection d'objets identiques permet de les aligner en mémoire, et **on sait le coder mais on ne veut pas coder une version par objet**
- **Il nous faudrait du code pour générer du code, pour bénéficier de la spécialisation de type à la compilation**

Makefile



On peut simuler le templating de haut niveau en C

→ exemple complet sur moodle !



FIN