Atelier programmation
Premiers pas en C avec GCC sous Linux x64

CFPT – Michaël Mathieu

26 janvier 2026

1 Hello, world

1.1 Code source – helloWorld.c

```
#include <stdio.h>

int main() {
    printf("Hello, world");
    return 0;
}
```

- `stdio.h` pour *standard input output* : inclure la dépendance permettant d'afficher à l'écran
- `main` : point d'entrée du programme
- `printf` : afficher du texte à l'écran
- `return 0` : informe l'appelant du programme que tout s'est bien déroulé (chiffre ≥ 0)

1.2 Compiler le code source avec GCC

```
> gcc -c helloWorld.c
```

- `-c` pour lancer la compilation du code source
- les éventuels messages d'erreurs et avertissements sont affichés à l'écran
- s'il n'y a pas d'erreurs, le fichier `helloWorld.o` sera créé

1.3 Construire le programme avec GCC (linker)

```
> gcc helloWorld.o -o helloWorld
```

- `-o` pour spécifier le nom du programme qui sera construit
- le fichier `helloWorld` est créé et peut être exécuté :

```
> ./helloWorld
```

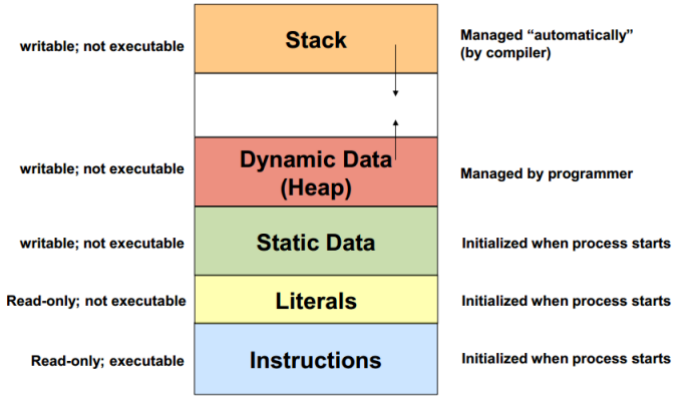
2 Utilisation de la mémoire

Les variables (type natif), les tableaux dont la taille est connue à la **compilation**, les appels de fonctions et leurs paramètres sont stockés sur la stack. Les tableaux dont la taille est dynamique (`malloc`) sont stockés sur la heap.

Sur la stack, après utilisation, la mémoire est libérée automatiquement. Sur la heap, il faut le faire explicitement (`free`).

⚠sous Windows la mémoire n'est pas organisée de la même manière

Observons les adresses mémoire attribuées le système d'exploitation (Linux) au programme suivant :




```c
#include <stdio.h>

int sum(int a, int b) {
    printf("&a => %p\r\n", &a);
    printf("&b => %p\r\n", &b);

    return a + b;
}

int main() {
    int c = 2;
    int d = 3;
    int result;

    printf("&c => %p\r\n", &c);
    printf("&d => %p\r\n", &d);
    printf("&result => %p\r\n", &result);

    result = sum(c, d);

    printf("&result => %p\r\n", &result);
    printf("result => %d\r\n", result);

    return 0;
}
```

Sortie

```
&c => 0x7ffd6a5bffec
&d => 0x7ffd6a5bfff0
&result => 0x7ffd6a5bfff4
&a => 0x7ffd6a5bffcc
&b => 0x7ffd6a5bffc8
&result => 0x7ffd6a5bfff4
result => 5
```

3 Gestion de la stack

3.1 Elements

— RSP (Stack Pointer) : pointe toujours vers le sommet de la stack sur le dernier élément valide de la stack (pas sur le prochain emplacement libre)
— RBP (Base Pointer) : pointe vers la base du stack frame actuel
— La stack grandit vers le bas (adresses décroissantes)

3.2 Prologue et Epilogue de fonction

3.2.1 Prologue

Le **prologue** est la séquence d'instructions (assembleur) exécutée au **début** de chaque fonction. Son rôle est de :

1. Sauvegarder le RBP de la fonction appelante (pour pouvoir le restaurer plus tard)
2. Etablir une nouvelle frame de stack pour la nouvelle fonction
3. Réserver de l'espace pour les variables locales

```
Prologue standard :

push rbp                          ; Sauvegarde l'ancien RBP
mov rbp, rsp                    ; RBP = RSP (nouvelle frame)
sub rsp, N        ; Alloue N octets pour variables locales
```

Résultat : Après le prologue, le RBP pointe vers la base de la nouvelle frame, et le RSP pointe vers le sommet (après allocation des variables locales).

3.2.2 Epilogue

L'**épilogue** est la séquence d'instructions (assembleur) exécutée à la **fin** de chaque fonction, juste avant de retourner où elle avait été appelée. Son rôle est de :

1. Libérer l'espace alloué pour les variables locales
2. Restaurer le RBP de la fonction appelante
3. Retourner à l'adresse de retour

```
Epilogue standard :

mov rsp, rbp       ; Restaure le RSP (libère vars locales)
pop rbp                             ; Restaure l'ancien RBP
ret                               ; POP return address et JMP
```

Résultat : Après l'épilogue, la stack est revenue à l'état d'avant l'appel de la fonction, et le processeur saute à l'adresse de retour (dans la fonction appelante).

3.3 Exécution d'un programme

Imaginons un programme simple :

```
#include <stdio.h>

int min(int a, int b) {
    if (a < b) {
        return a;
    }
    return b;
}

int main() {
    int x = 2;
    int y = 3;
    int result = min(x, y);

    printf("min => %d\r\n", result);

    return 0;
}
```

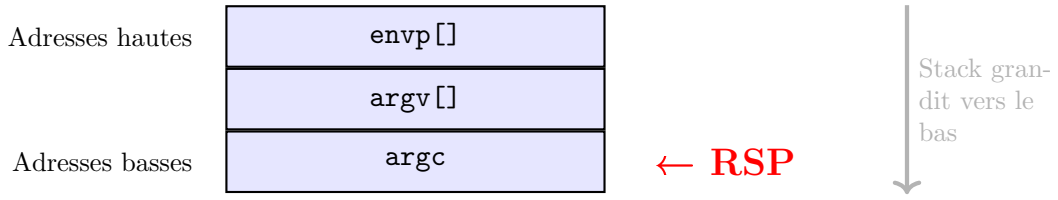
3.3.1 Avant l'appel du `main`

Quand le programme démarre, le système d'exploitation prépare l'environnement en plaçant sur la stack :

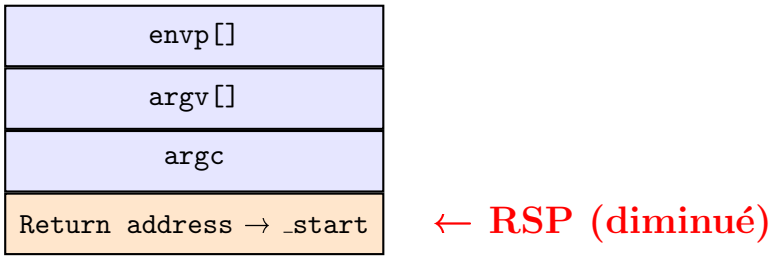
- `argc` (argument count) : nombre d'arguments passés au programme
- `argv[]` (argument vector) : tableau de pointeurs vers les chaînes d'arguments
- `envp[]` (environment pointer) : tableau de pointeurs vers les variables d'environnement

Exemple : `./program arg1 arg2`

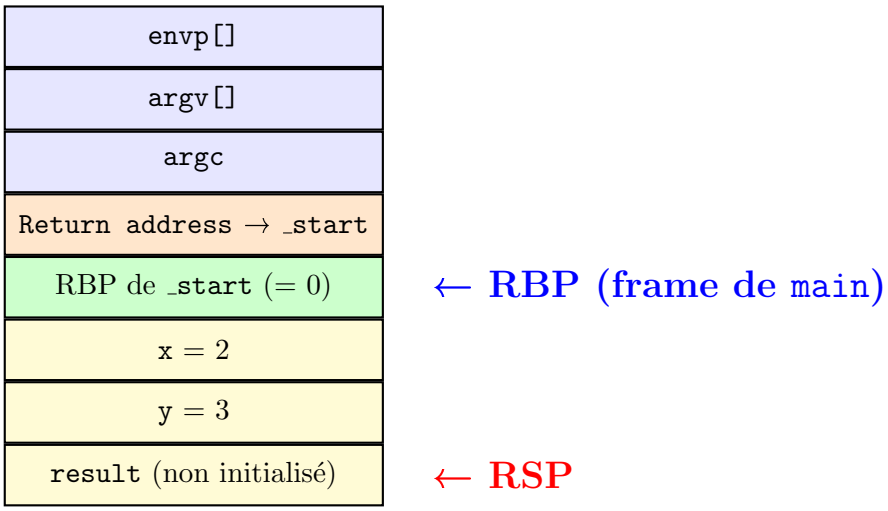
- `argc` = 3
- `argv[0]` = "./program"
- `argv[1]` = "arg1"
- `argv[2]` = "arg2"
- `argv[3]` = NULL
- `envp[]` contient des chaînes comme "PATH=/usr/bin", "HOME=/home/user", etc.



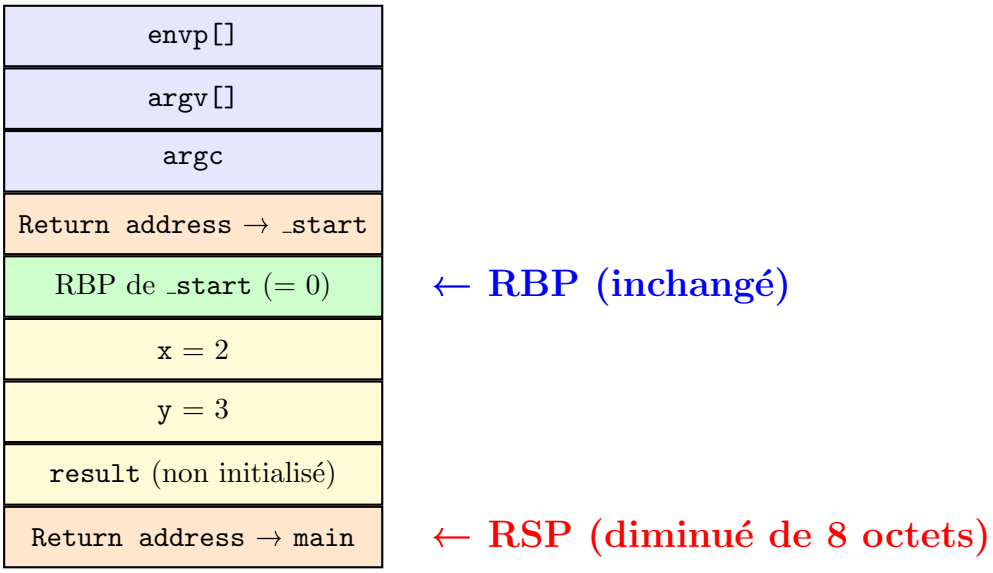
3.3.2 Instruction `call main`



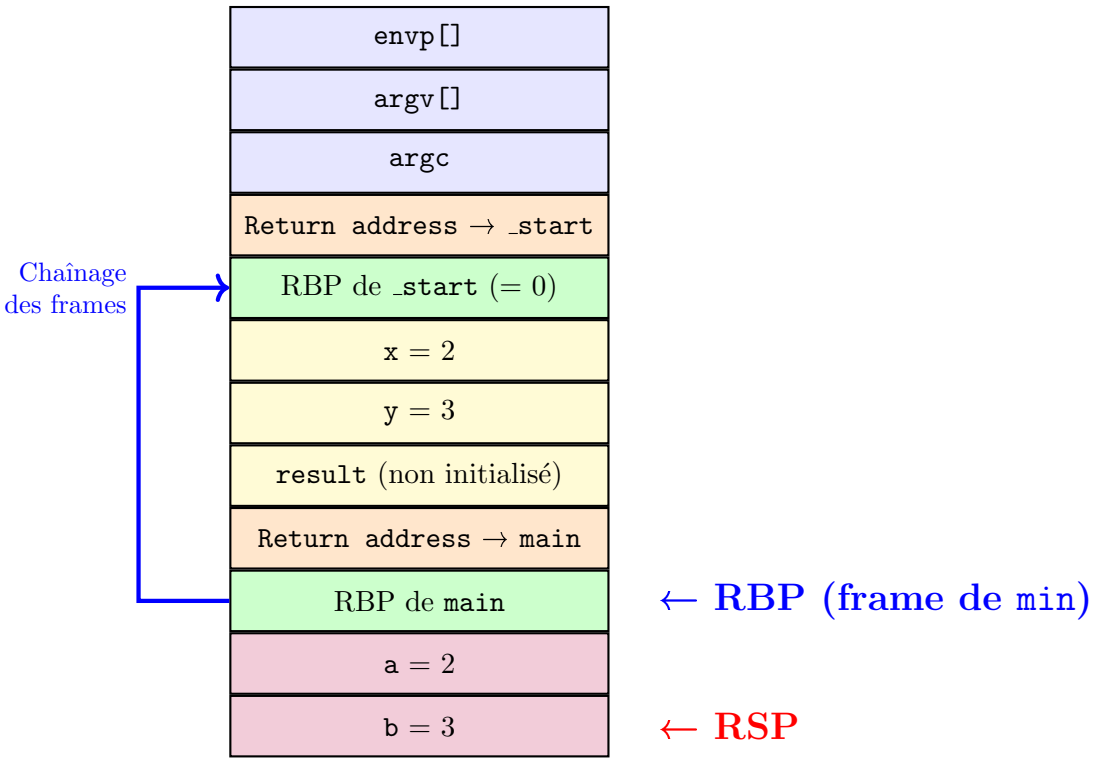
3.3.3 Prologue de `main`



3.3.4 Instruction `call min`



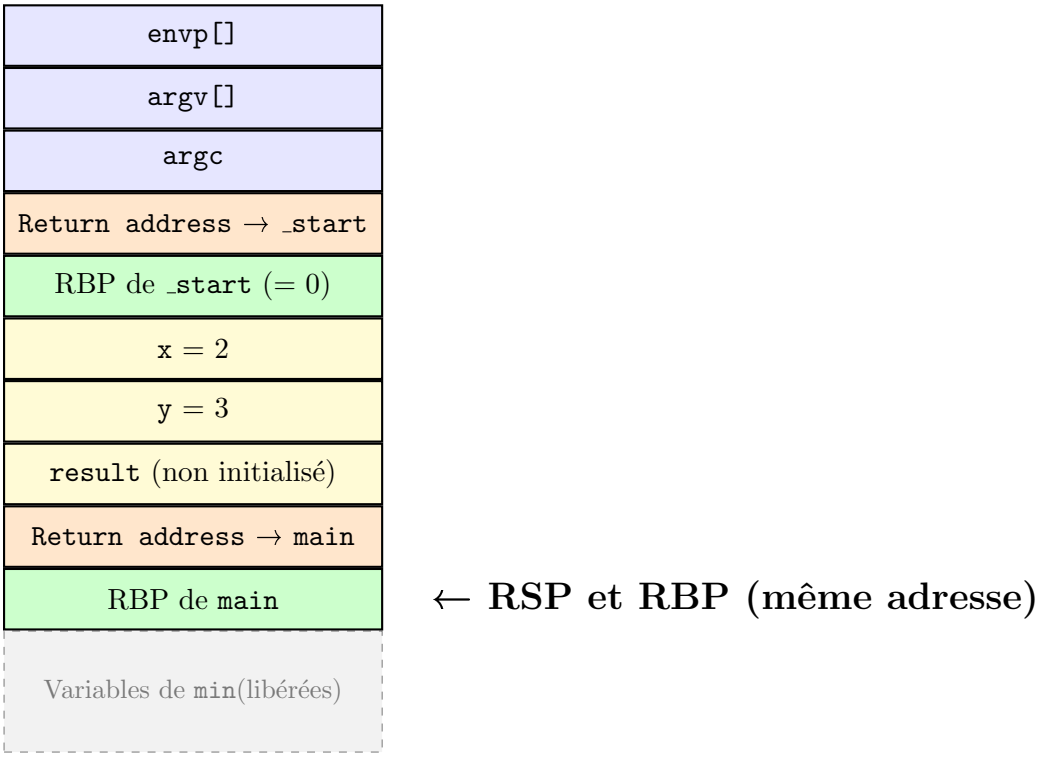
3.4 Prologue de `min`



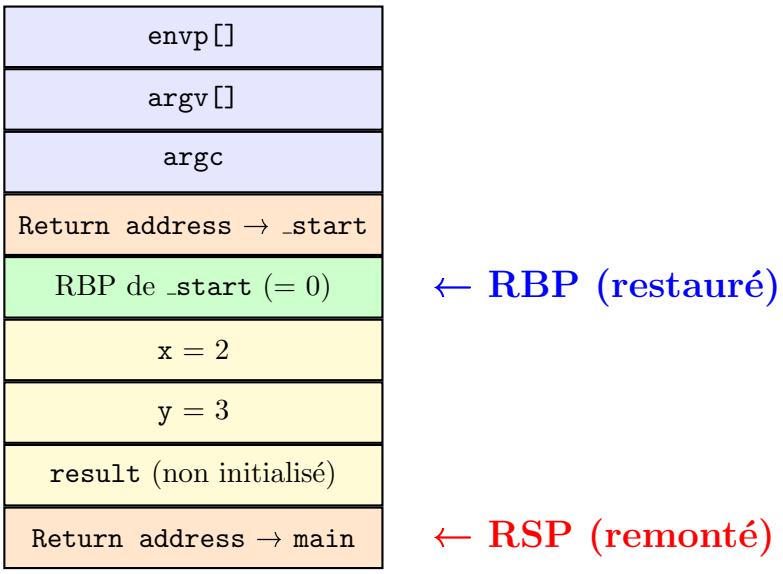
⚠représentation schématique : en absence de récursivité les registres sont utilisés en priorité pour les paramètres de fonctions

3.4.1 Epilogue de `min`

Le RSP remonte vers le RBP (libère les variables locales de `min`).



Le RSP remonte de 8 octets, le RBP est restauré avec la valeur sauvegardée.



`POP` de l'adresse de retour et saut à cette adresse (retour dans `main`).