CM 11 - LA MÉMOIRE

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

MÉMOIRE VIRTUELLE

▷ Lors de l'exécution du binaire, un espace mémoire particulier est associé à cette exécution : c'est le processus

▷ Les processus ont besoin de mémoire (ressource limitée sur un ordinateur)

▷ Il faut donc que les processus aillent chercher de la mémoire disponible pour pouvoir travailler (dans des systèmes d'exploitation multi-tâches)

▷ Que se passerait-il si deux processus voulaient accéder, au même instant, à la même zone mémoire ?

MÉMOIRE VIRTUELLE

▷ Chaque processus se voit allouer une plage de mémoire virtuelle comme s'il s'exécutait tout seul sur la machine

▷ L'espace d'adressage de chaque processus est divisé en pages de taille fixe

▷ Le système maintient une table d'association entre les pages du processus et leurs localisations effectives en mémoire

▷ Les valeurs d'adresses possibles sur une machine 32 bits vont de `0x00000000` à `0xFFFFFFFF`

MÉMOIRE VIRTUELLE

▷ Chaque processus se voit allouer une plage de mémoire virtuelle comme s'il s'exécutait tout seul sur la machine

▷ Le processus n'a alors plus à se soucier de l'implémentation de la mémoire

▷ Toutes les opérations bas niveau sont gérées par le noyau du système d'exploitation

▷ On obtient une couche d'abstraction qui simplifie le fonctionnement du processus

MÉMOIRE VIRTUELLE

▷ La taille des pages est fixe :

```
$> getconf PAGESIZE
4096
```

▷ Par conséquent, les adresses `0x00000000` à `0x00000FFF` ($2^{12} - 1$) appartiennent à la même page

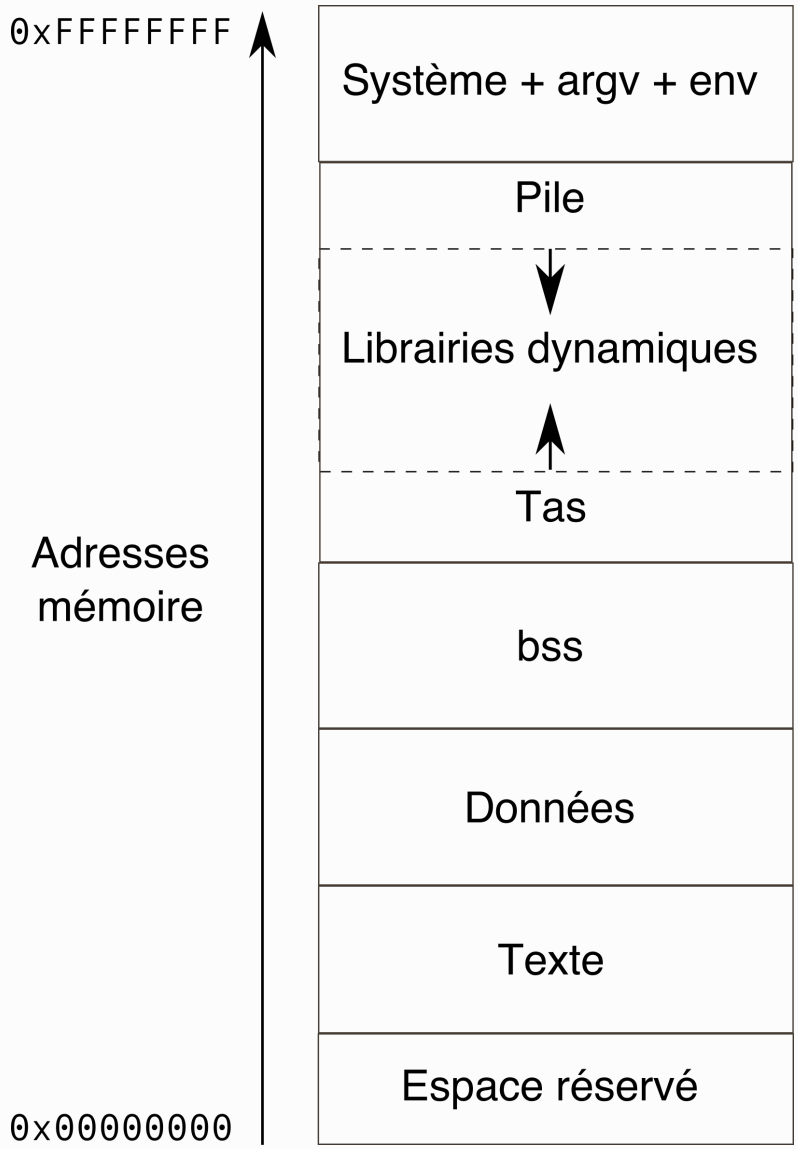
▷ Sur une machine 32 bits, l'espace d'adressage est donc divisé en $2^{(32-12)} = 1\,048\,576$ pages qui peuvent être

- ▷ non adressable (non associée)
- ▷ adressable : en lecture, écriture et/ou exécution

SEGMENTATION DE LA MÉMOIRE

LA MÉMOIRE

- La mémoire d'un processus est segmentée en 3 sections :
  - Texte (`.text`)
  - Données (`.data`)
  - bss (`.bss`)
- Et 2 zones mémoire
  - Tas (`heap`)
  - Pile (`stack`)



TEXTE (`.text`)

▷ La section `.text` contient le code du programme (sous la forme d'instructions en langage machine)

▷ C'est une section de taille fixée (et non variable) en lecture seule (une fois définie, elle est immuable) qui ne stocke que du code (pas des variables)

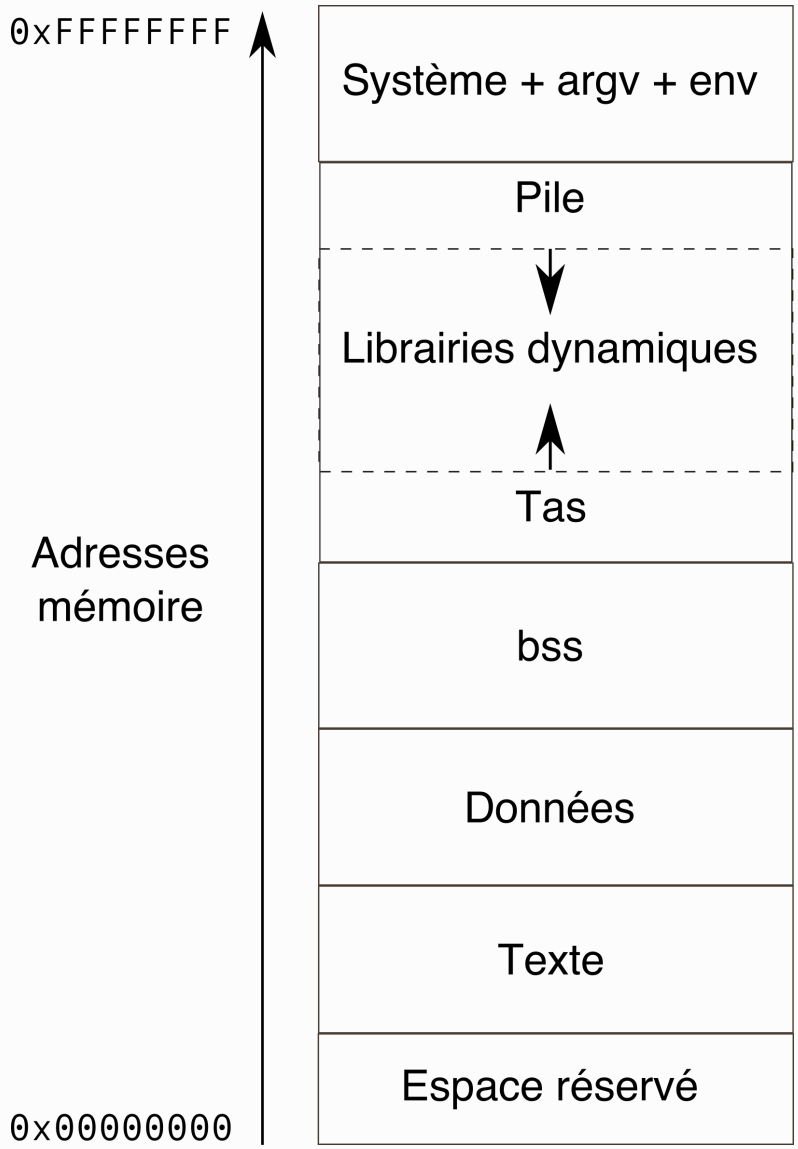
▷ Une erreur classique de "Segmentation Fault" peut survenir si l'utilisateur tente une écriture dans cette zone mémoire

▷ Le programme débutera son exécution par la première instruction de ce segment, puis exécutera les instructions suivantes une à une

▷ Comme nous l'avons vu, cette lecture n'est pas forcément linéaire du fait des structures de contrôles et des appels de fonctions

MODES D'ALLOCATIONS MÉMOIRES

- Il existe 3 modes d'allocations mémoires
  - Statique (dans `.data` et `.bss`)
  - Dynamique (dans le tas)
  - Automatique (dans la pile)
- Correspondant à 3 types de gestions différents



Données (`.data` et `.bss`)

▷ Les sections `.data` et `.bss` stockent les variables globales et statiques du programme

▷ Toute donnée initialisée est stockée dans la section `.data`, tandis que les autres le sont dans la section `.bss`

▷ Ce sont des sections de tailles fixes où l'écriture est partiellement possible (à l'exception des variables finales et statiques)

▷ Les données globales et statiques sont présentent dans le binaire et chargée en mémoire en même temps que le code

▷ Les adresses sont fixes et restent valides du lancement jusqu'à la fin du processus

RAPPEL SUR LES VARIABLES

- Une variable est un outil pour le programmeur, lui permettant de manipuler une zone mémoire et d'y associer une interprétation
- variable = (adresse, type, valeur)
- A l'exécution, le nom des variables et des fonctions n'existent plus
- Ainsi, lors de l'utilisation d'une variable v
  - v correspond à l'interprétation de la mémoire
  - &v correspond à l'adresse de cette mémoire

VARIABLES STATIQUES

▷ Pour une variable statique v

▷ v représente le contenu d'une zone mémoire fixe

▷ L'adresse mémoire &v sera toujours la même durant l'exécution du processus

▷ L'interprétation de cette zone dépend du type de v

▷ La valeur (i.e., le contenu de la mémoire) peut changer au cours de l'exécution

SEGMENTS PAR L'EXEMPLE

▷ Soit le programme basique en C suivant

```c
#include <stdlib.h>
int main(void) {
    return EXIT_SUCCESS;
}
```

▷ Examinons la taille de ses différentes sections (outil **`size`**)

```
$> gcc -m32 segments.c
$> size a.out
text    rodata    data    bss    total    filename
1349     8         8       4      1369     a.out
```

SEGMENTS PAR L'EXEMPLE

▷ Lors de l'ajout d'une variable globale non initialisée

```
#include <stdlib.h>
int gvar;
int main(void) {
    return EXIT_SUCCESS;
}
```

```
$> gcc -m32 segments.c
$> size a.out
text    rodata    data    bss    total    filename
1349    8         8       8      1373     a.out
```

▷ on remarque l'augmentation de 4 octets de la taille de la section `.bss` (pour stocker la variable globale non-initialisée)

SEGMENTS PAR L'EXEMPLE

▷ Lors de l'ajout d'une variable statique au sein de `main()`

```
#include <stdlib.h>
int gvar;
int main(void) {
    static int lvar;
    return EXIT_SUCCESS;
}
```

```
$> gcc -m32 segments.c
$> size a.out
text    rodata    data    bss    total    filename
1349    8         8       12     1377     a.out
```

▷ on remarque à nouveau l'augmentation de 4 octets de la taille de la section `.bss`

SEGMENTS PAR L'EXEMPLE

▷ Lors de l'initialisation de la variable statique

```
#include <stdlib.h>
int gvar;
int main(void) {
    static int lvar=1;
    return EXIT_SUCCESS;
}
```

```
$> gcc -m32 segments.c
$> size a.out
text    rodata    data    bss    total    filename
1349    8         12      8      1377     a.out
```

▷ on constate que la variable n'est plus stockée dans la section `.bss`, mais dans la section `.data`

SEGMENTS PAR L'EXEMPLE

▷ Enfin, si on initialise la variable globale

```
#include <stdlib.h>
int gvar=1;
int main(void) {
    static int lvar=1;
    return EXIT_SUCCESS;
}
```

```
$> gcc -m32 segments.c
$> size a.out
text    rodata    data    bss    total    filename
1349    8         16      4      1377     a.out
```

▷ alors les deux variables sont stockées dans la section `.data`, et non plus dans la section `.bss`

SEGMENTS PAR L'EXEMPLE

▷ Si la variable globale est de plus constante

```
#include <stdlib.h>
const int gvar=1;
int main(void) {
    static int lvar=1;
    return EXIT_SUCCESS;
}
```

```
$> gcc -m32 segments.c
$> size a.out
text    rodata    data    bss    total    filename
1349    12        12      4      1377     a.out
```

▷ alors cette dernière est stockée dans une section en lecture seule `.text` ou `.rodata` (read-only data)

VARIABLES STATIQUES

▷ Les chaînes de caractères sont des données statiques (correspondant à des tableaux de caractères) et mutualisées par unité de compilation (.c)

```c
//main.c
#include <stdlib.h>
#include <stdio.h>
int gvar=12, lvar;
void g();
void f(){
    char * s = "Hello, World!";
    printf("f: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

```c
//file1.c
#include <stdio.h>
void g(){
    char * s = "Hello, World!";
    printf("g: %p\n",s);
}
```

```
$> gcc main.c file1.c
$> ./a.out
main: 0x8048520
f: 0x8048520
g: 0x804853f
```

VARIABLES STATIQUES

▷ Les chaînes de caractères sont des données statiques (correspondant à des tableaux de caractères) et mutualisées par unité de compilation (.c)

```c
//main.c
#include <stdlib.h>
#include <stdio.h>
int gvar=12, lvar;
void g();
void f(){
    char * s = "Hello, World!";
    printf("f: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

```c
//file1.c
#include <stdio.h>
void g(){
    char * s = "Hello, World!";
    printf("g: %p\n",s);
}
```

```
$> gcc -m32 main.c file1.c
$> size a.out
text    rodata    data    bss    total
1622    60        12      8      1702
```

VARIABLES STATIQUES

▷ Les chaînes de caractères sont des données statiques (correspondant à des tableaux de caractères) et mutualisées par unité de compilation (.c)

```c
//main.c
#include <stdlib.h>
#include <stdio.h>
int gvar=12, lvar;
void g();
void f(){
    char * s = "Hello, World!";
    printf("f: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

```c
//file1.c
#include <stdio.h>
void g(){
    char s[] = "Hello, World!";
    printf("g: %p\n",s);
}
```

```
$> gcc -m32 main.c file1.c
$> size a.out
text    rodata    data    bss    total
1636    46        12      8      1702
```

VARIABLES STATIQUES

```c
//main.c
#include <stdlib.h>
#include <stdio.h>
int gvar=12, lvar;
void g();
void f(){
    char * s = "Hello, World!";
    printf("f: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

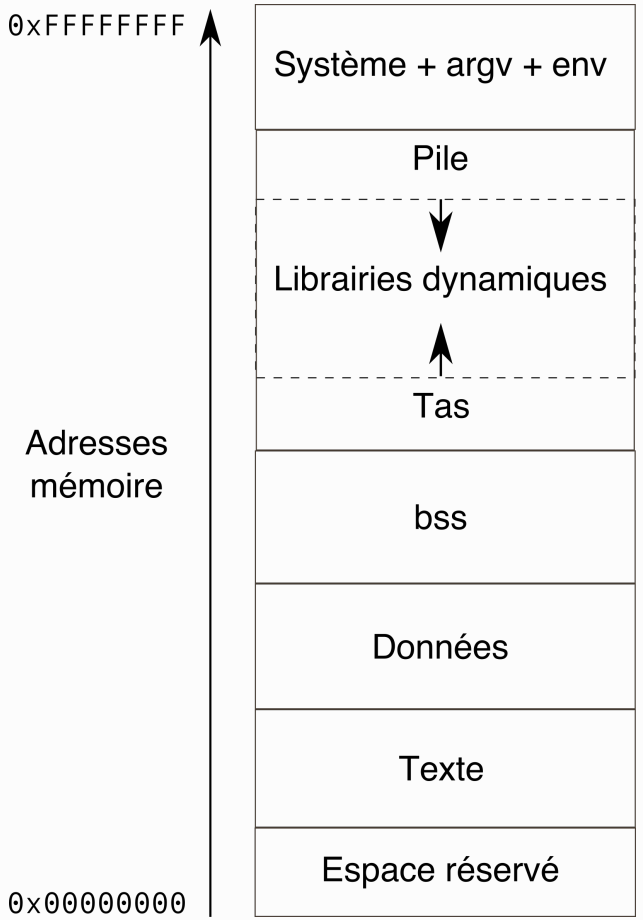
```c
//file1.c
#include <stdio.h>
void g(){
    char s[] = "Hello, World!";
    printf("g: %p\n",s);
}
```

```
$> gcc -m32 main.c file1.c
$> objdump -sj .text
a.out: format de fichier elf32-i386

Contenu de la section .text.:
 8048300 1.......PTRh....
 ...
 8048460 ......E.Hell.E.o
 8048470 , W.E.orldf.E.!.
 8048480 ....E.PhO....?..
 ...
 8048510 f3c3
```

LA ZONE MÉMOIRE TAS (heap)

▷ Le tas est un espace mémoire allouable dynamiquement par le programmeur (c.f. prochain cours)

▷ Cette zone mémoire n'a pas de taille fixe et augmente/diminue en fonction des demandes du programmeur, qui peut réserver (allocation) ou supprimer (libération) des blocs mémoires

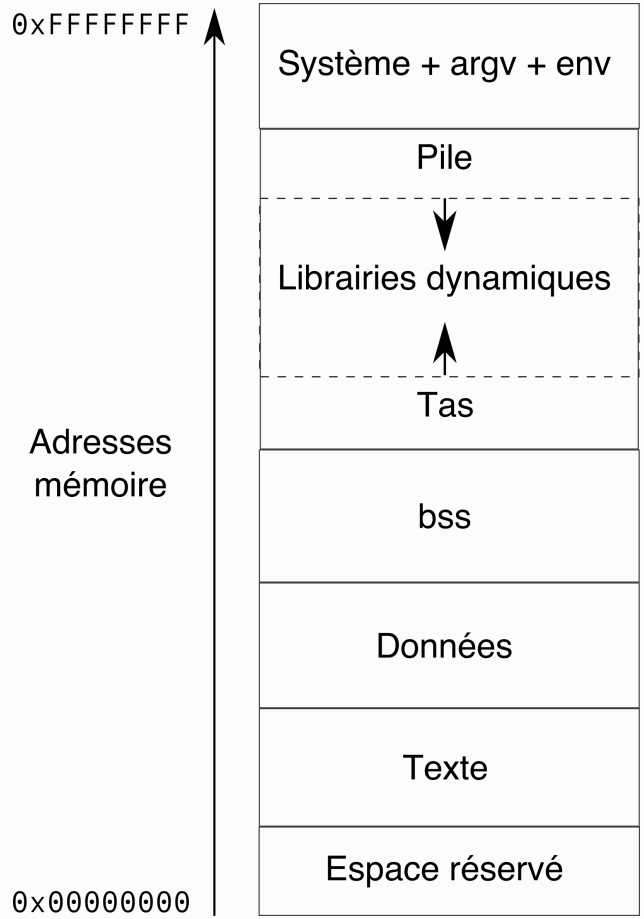


LA ZONE MÉMOIRE TAS (heap)

▷ Plus la taille du tas augmente, plus les adresses mémoires utilisées augmentent en s'approchant des adresses mémoires allouées aux librairies dynamiques

▷ La taille des blocs mémoires allouables sur le tas n'est pas limitée (sauf limite physique de la mémoire)

▷ Par ailleurs, les variables stockées dans le tas sont accessibles partout dans le programme, par l'intermédiaire des pointeurs

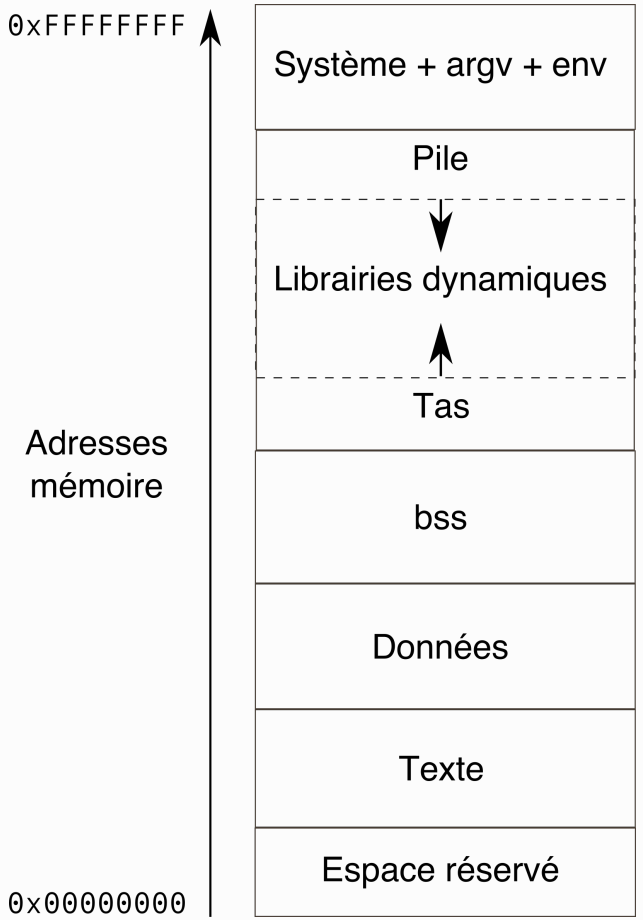


LA ZONE MÉMOIRE PILE (stack)

▷ La pile est un espace mémoire à taille variable

▷ Plus la taille de la pile augmente, plus les adresses mémoires utilisées diminuent en s'approchant des adresses mémoires allouées aux librairies dynamiques

▷ Elle correspond au mode d'allocation automatique lié aux fonctions (l'espace nécessaire est automatiquement réservé et libéré)



LA ZONE MÉMOIRE PILE (stack)

▷ Une pile a une structure LIFO (Last In, First Out).

▷ En d'autres termes, le dernier élément placé sur la pile sera le premier à être dépilé

▷ Dans le cas d'une pile d'assiettes, il faudra enlever la dernière assiette posée, puis l'avant-dernière etc. pour pouvoir récupérer la première assiette posée (c'est le même principe)

▷ La pile, ici, croît vers le bas ; ce que l'on nomme sommet de pile correspond à l'adresse la plus basse

▷ C'est cette structure qui permet l'utilisation de fonctions

RAPPELS SUR LES FONCTIONS

- La déclaration d'une fonction définit
  - Les types de ses paramètres
  - Le type de la valeur de retour
- La définition d'une fonction définit
  - Son code
  - Un ensemble abstrait de paramètres utilisables mais dont on ne connaît pas a priori la valeur
- L'appel d'une fonction définit
  - Des paramètres effectifs (via les arguments)
  - Un espace mémoire propre et temporaire

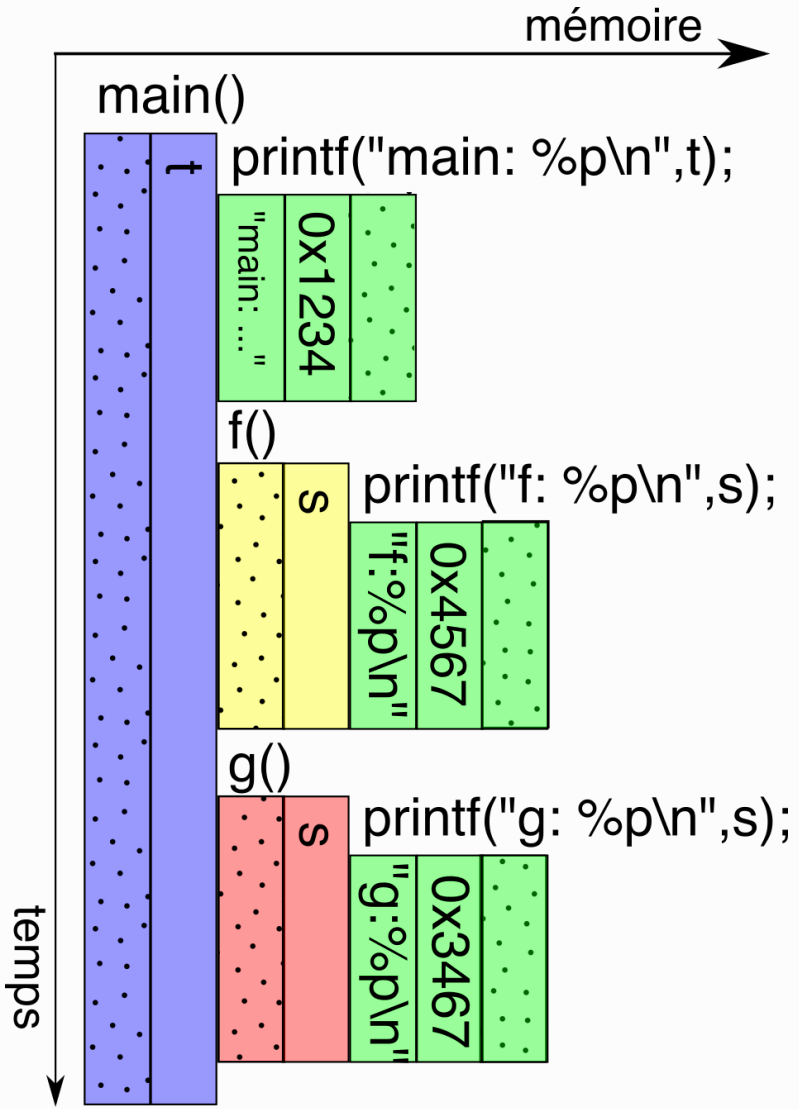
TRACE D'UN PROGRAMME

▷ La trace d'un programme utilisant des fonctions

▷ En ordonnée le temps est représenté

▷ En abscisse l'occupation mémoire

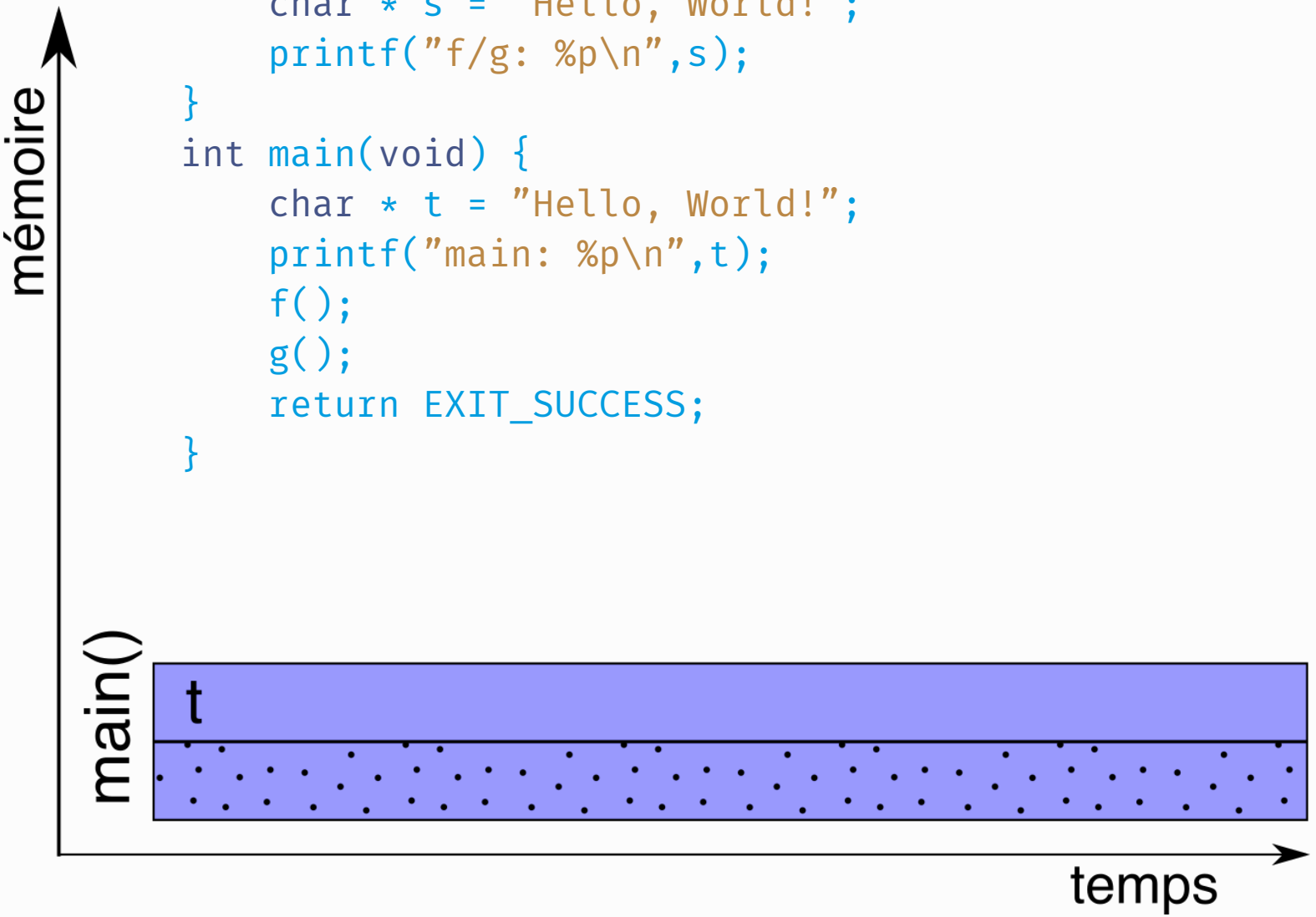
▷ L'appel d'une fonction occupe une portion de mémoire, puis la libère



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

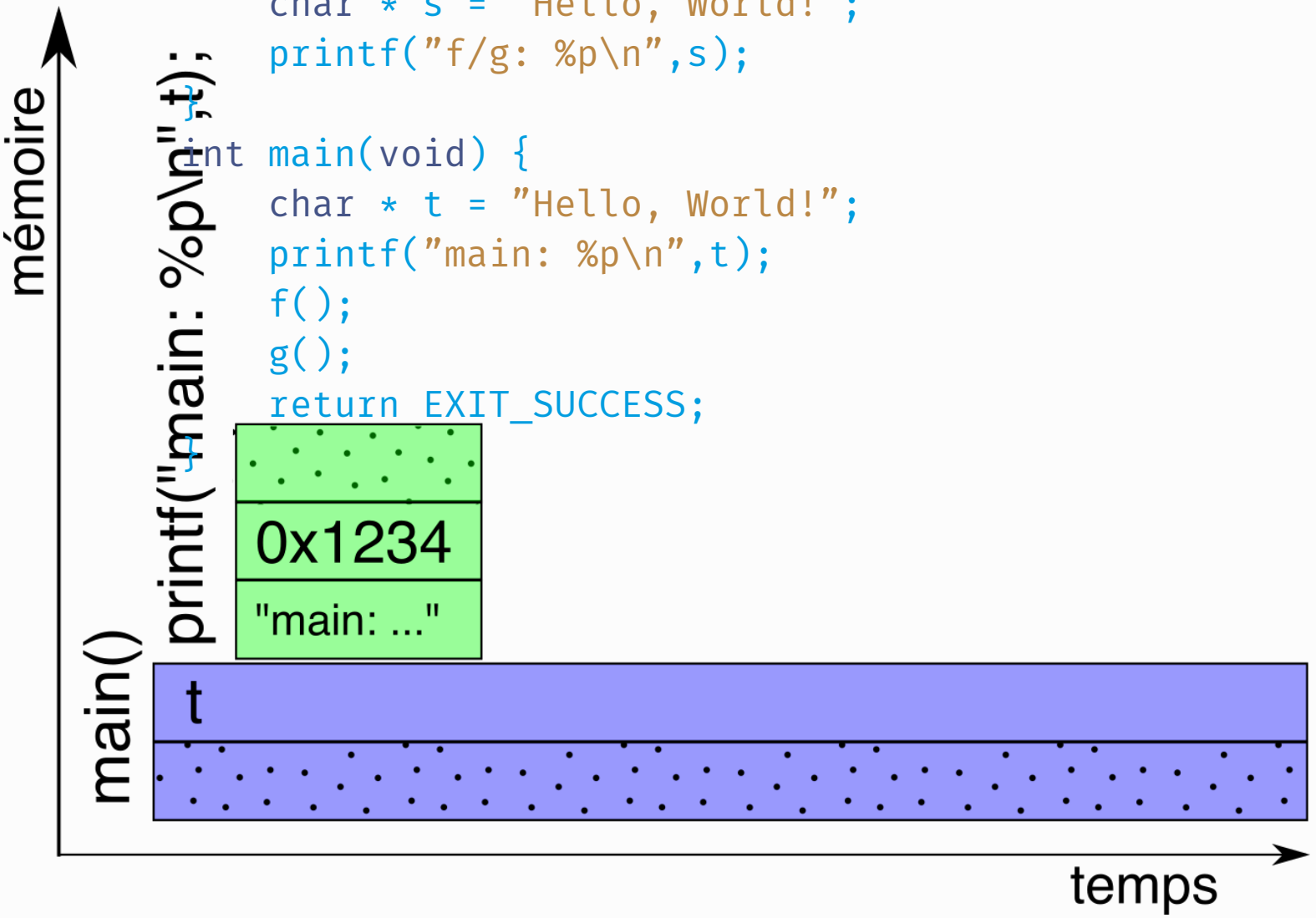
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

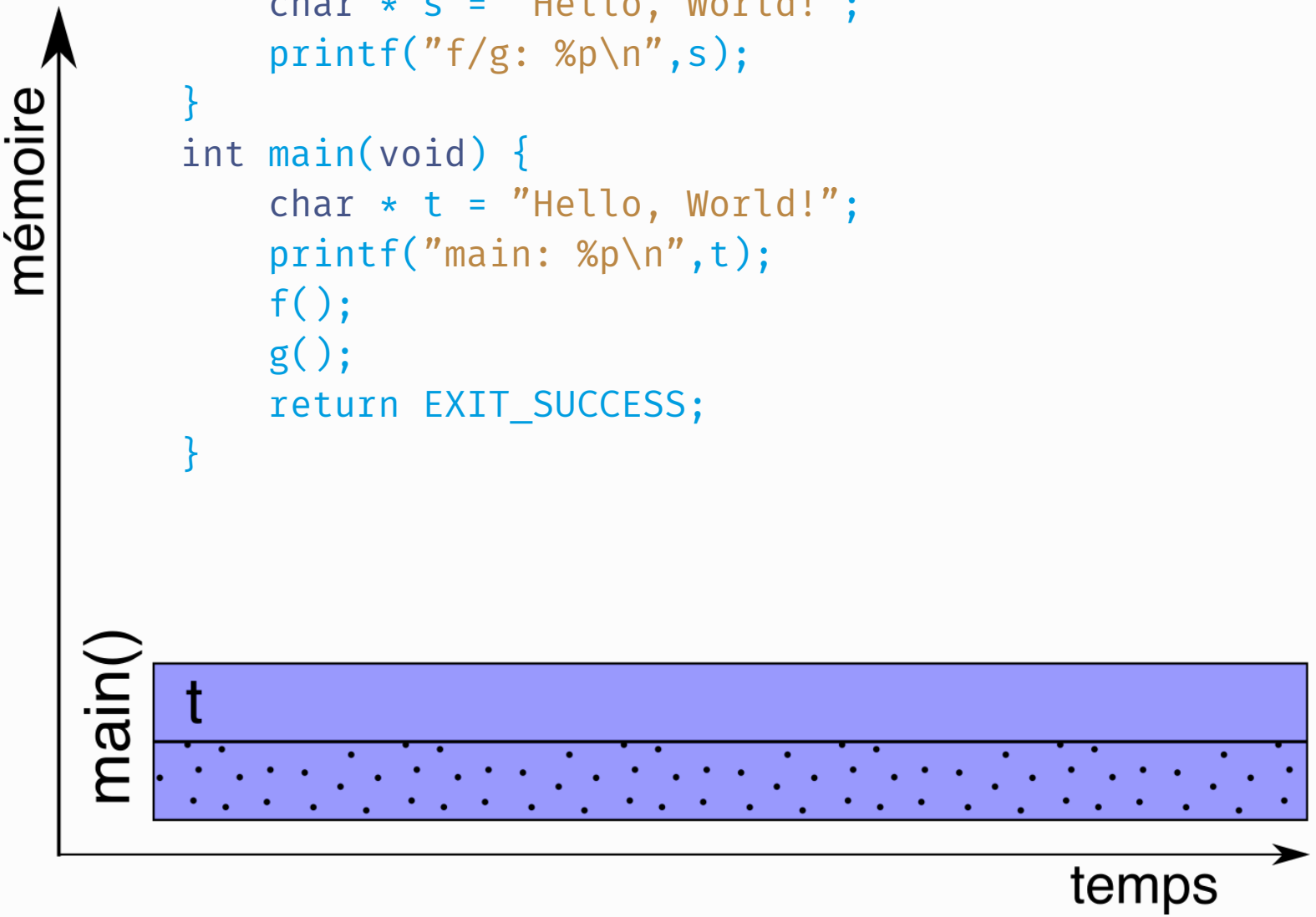
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);

int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

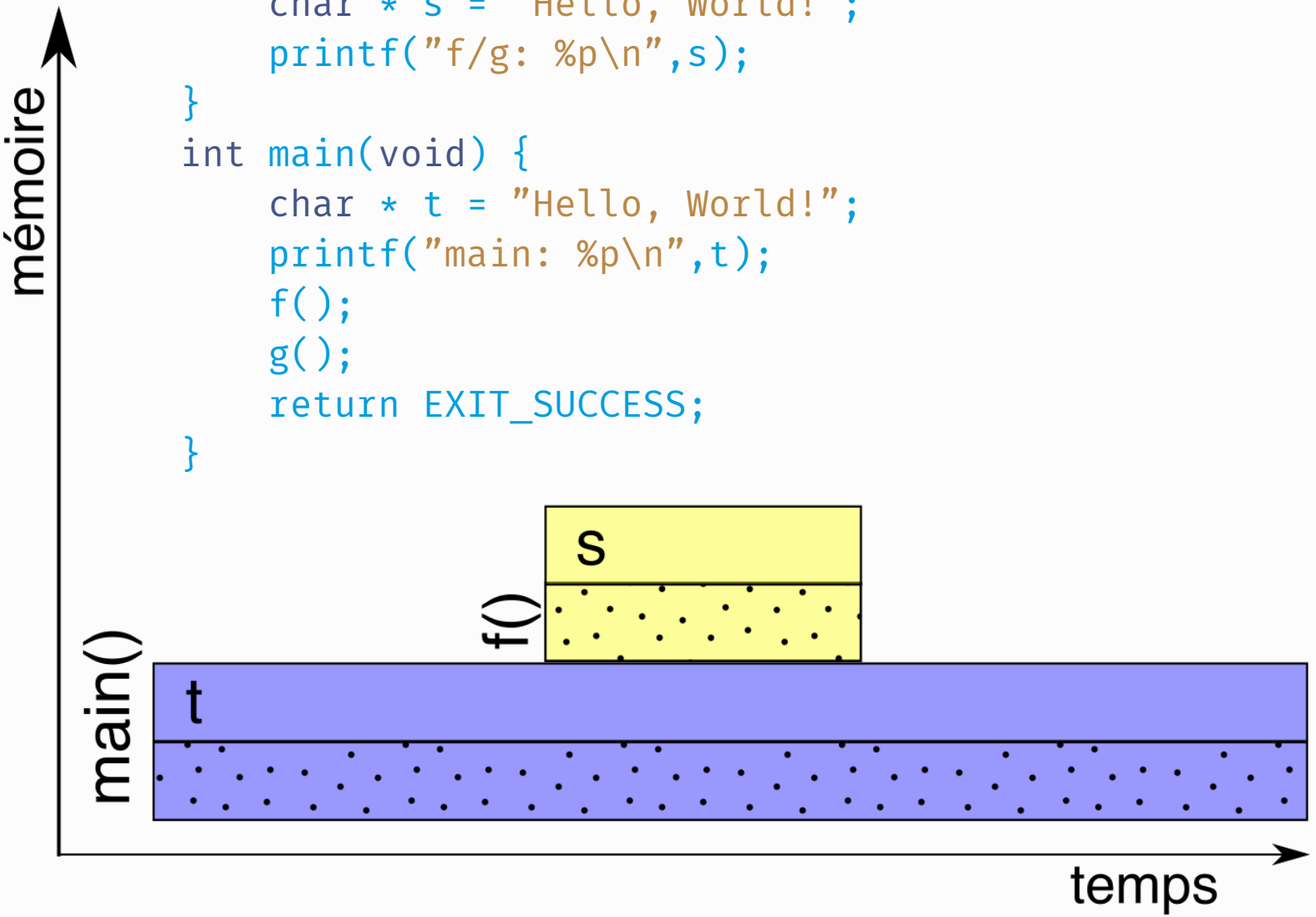
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

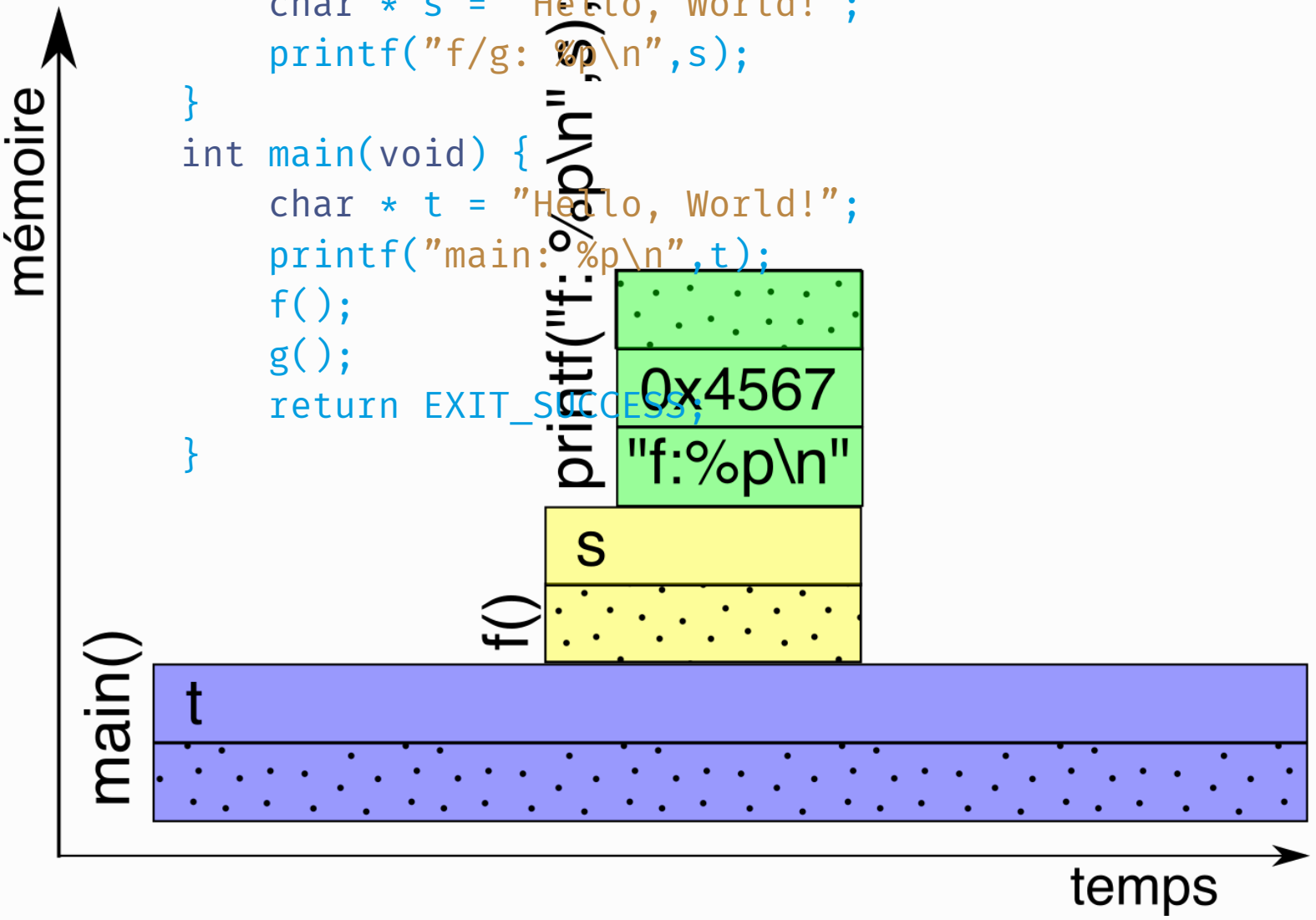
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

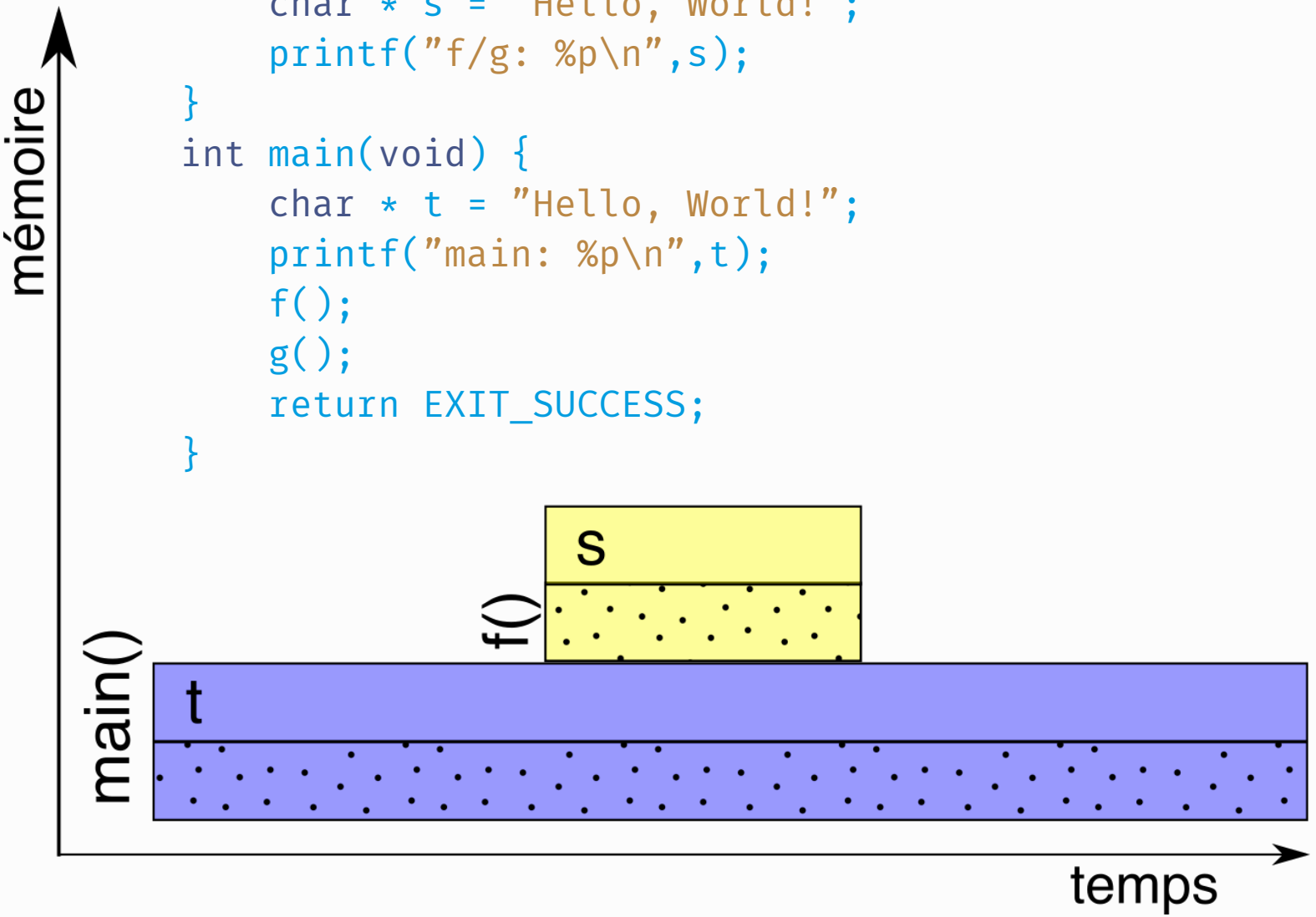
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

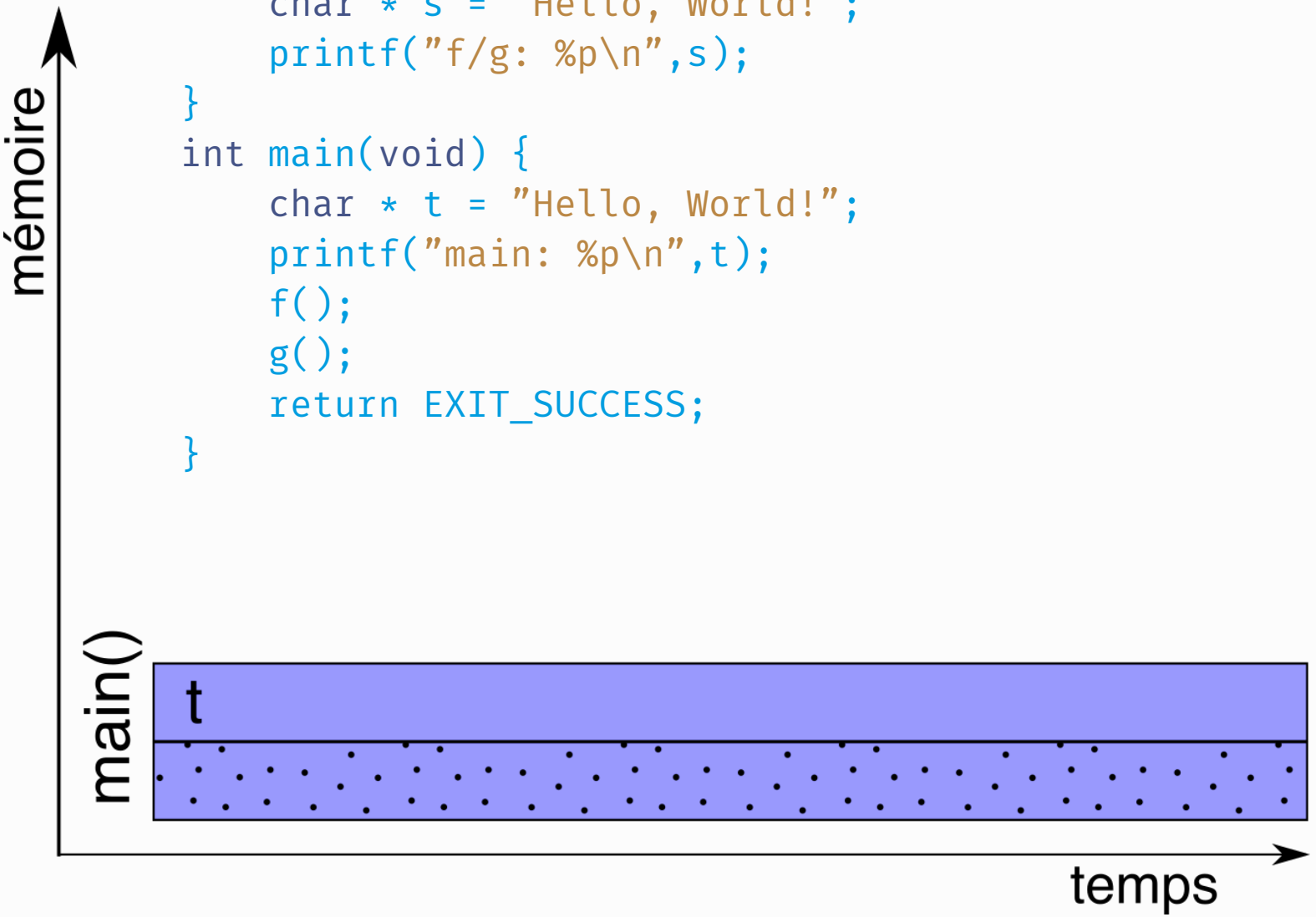
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

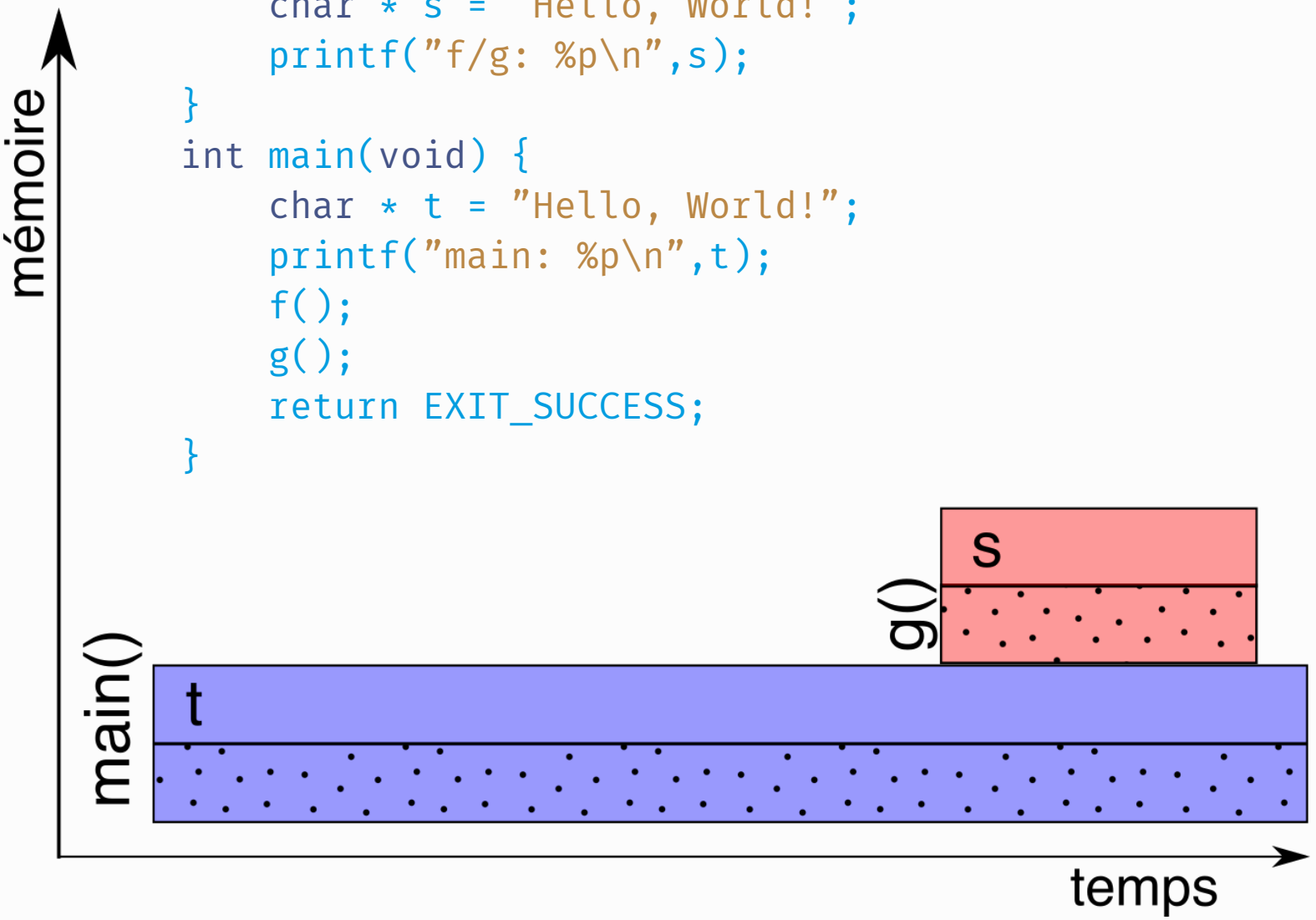
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

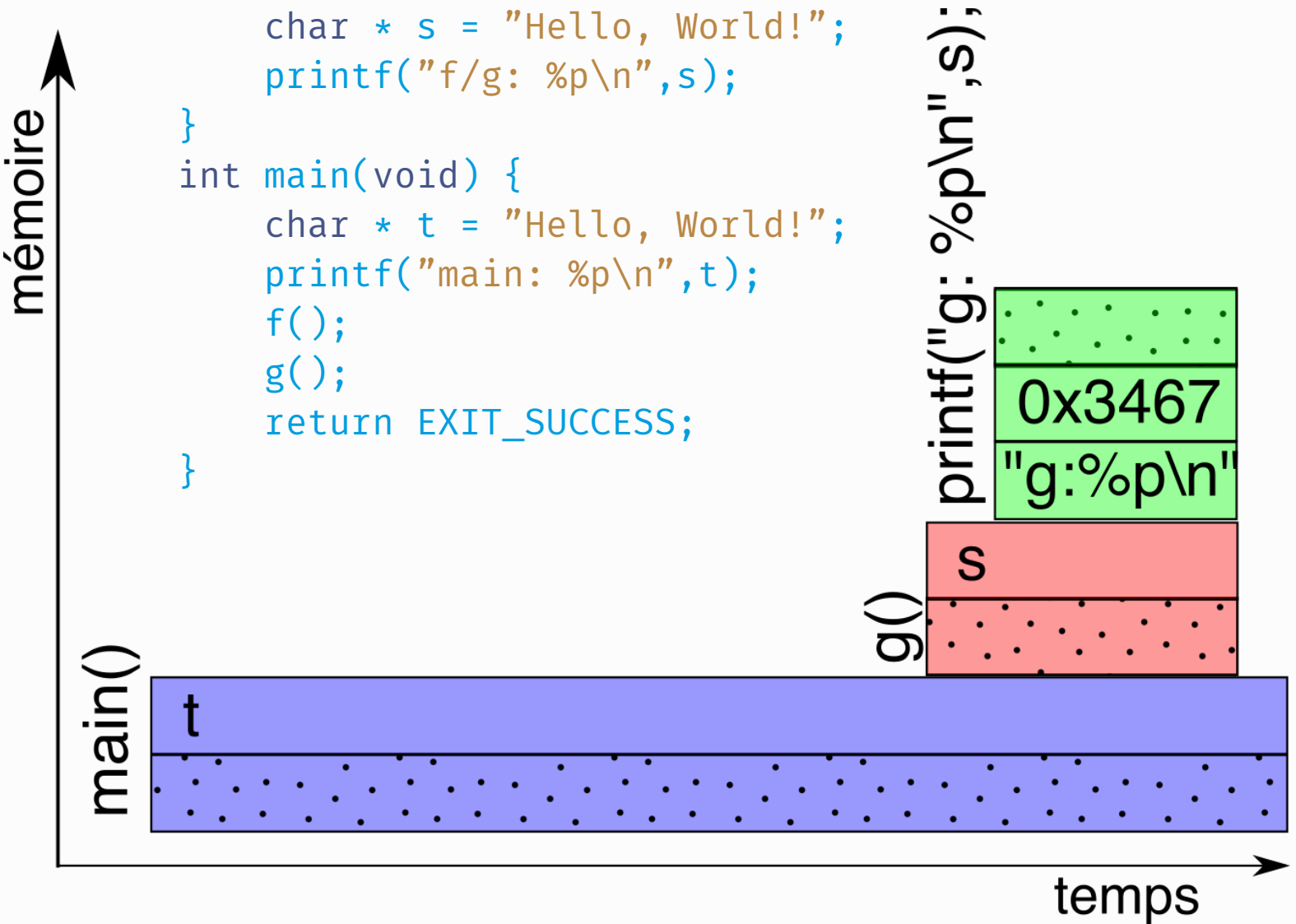
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

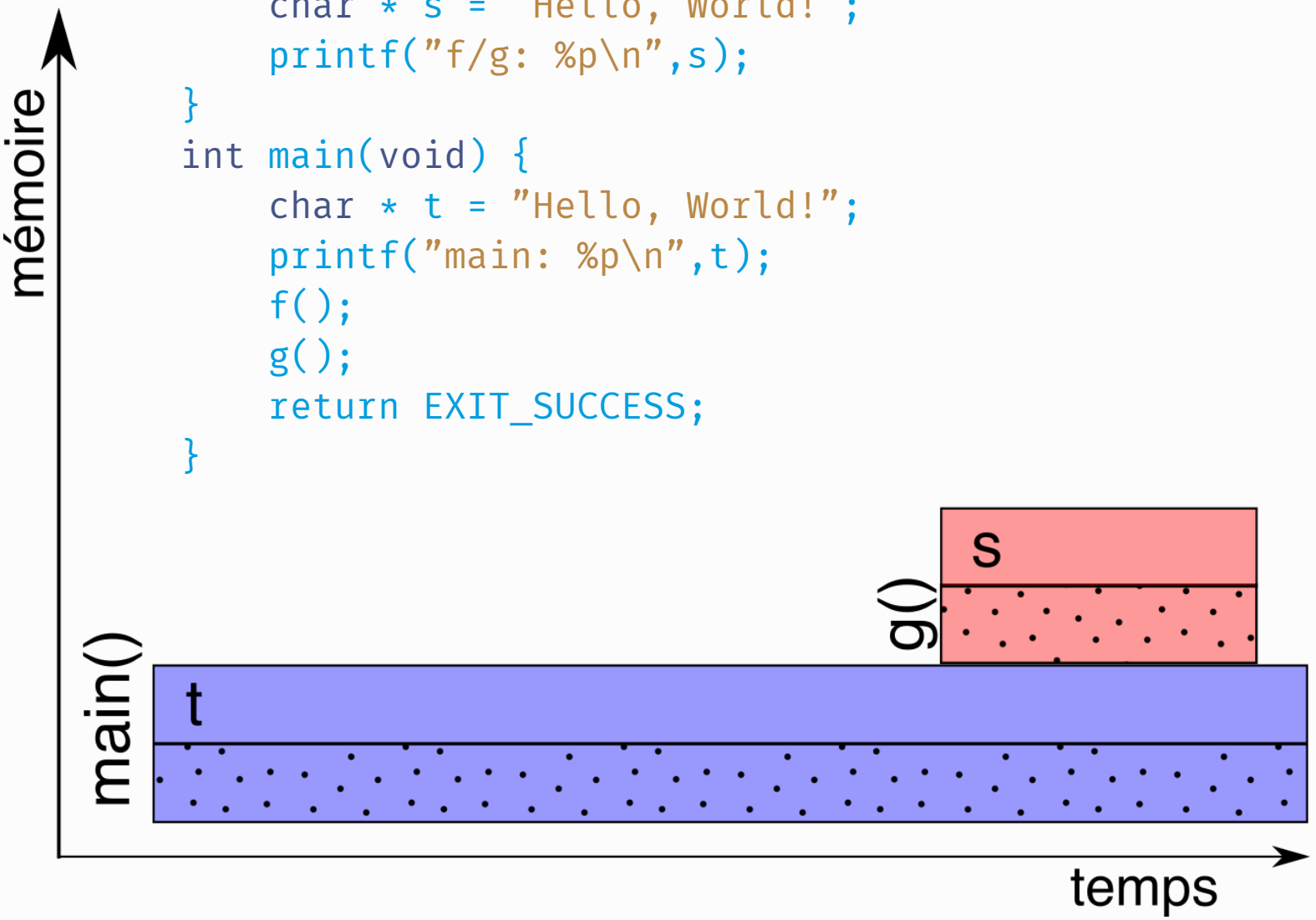
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

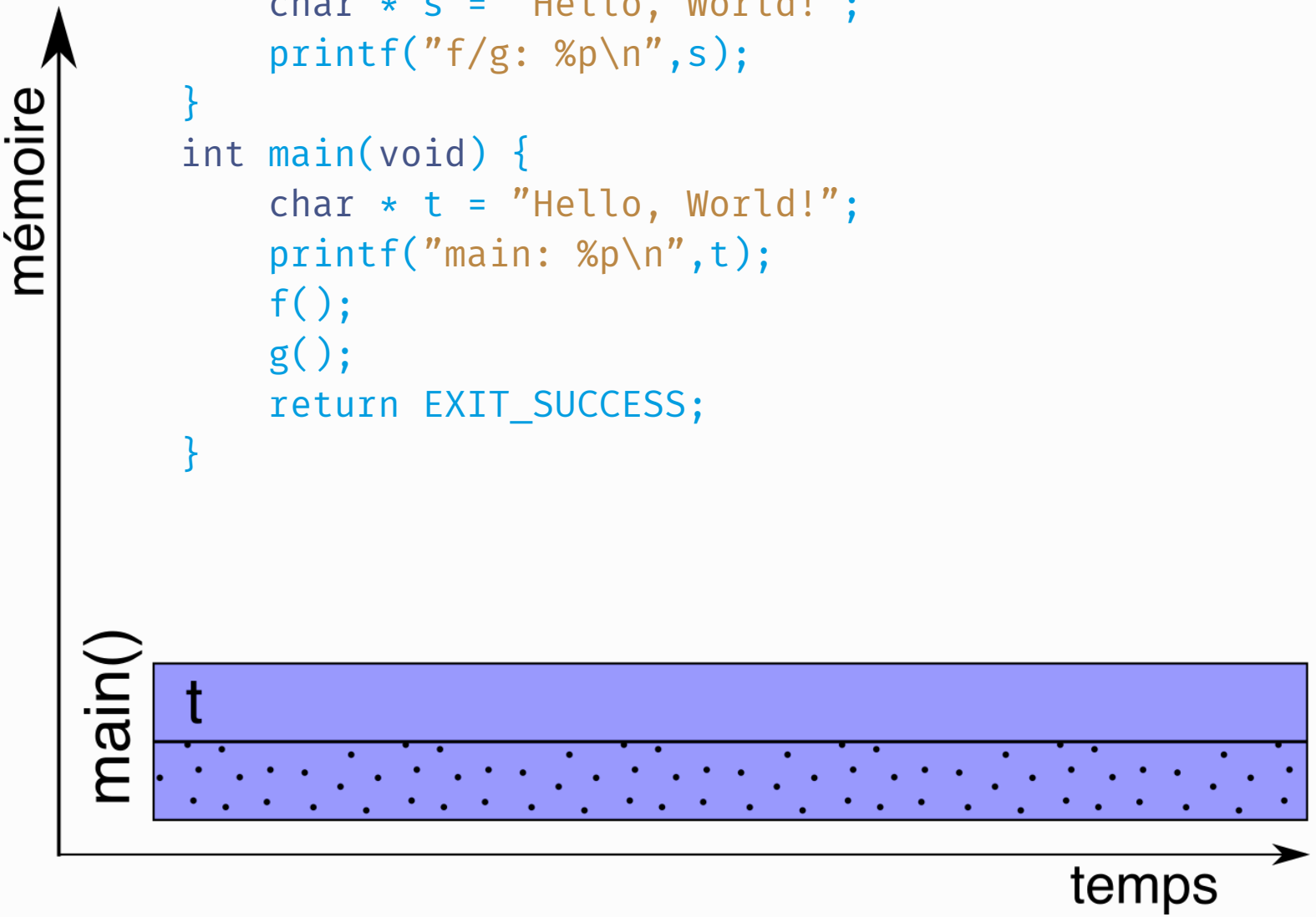
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

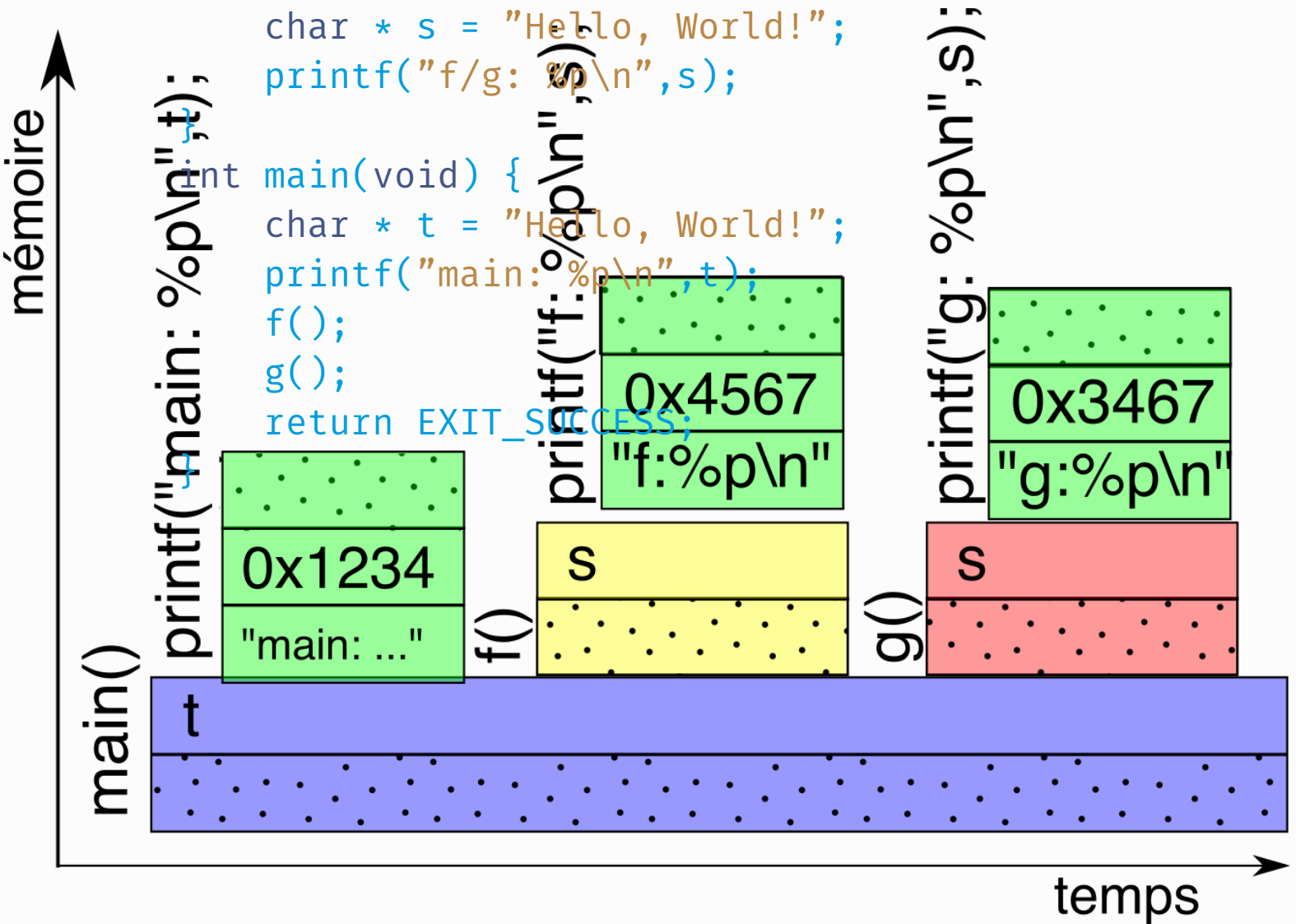
```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ On parle de pile d'appel car les fonctions s'empilent littéralement

```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}

int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```



PILE D'APPEL

▷ La pile d'appel est utilisée pour emmagasiner plusieurs valeurs

▷ Mais sa principale utilité est de garder la trace de l'endroit où chaque fonction active doit retourner à la fin de son exécution

▷ Pour ce faire, chaque appel de fonction place l'adresse de l'instruction suivant ce dernier (l'adresse de retour) sur la pile avant de transférer le contrôle de l'exécution à la fonction appelée

PILE D'APPEL

▷ C'est la fonction appelante qui pousse l'adresse de retour sur la pile, et la fonction appelée qui, quand elle se termine, récupère l'adresse de retour au sommet de la pile (et y transfère le contrôle)

▷ Si une fonction appelée appelle une autre fonction, elle poussera son adresse de retour sur la pile

▷ Les adresses de retour s'accumulent donc sur la pile et sont récupérées une à une lors de la fin de l'exécution des fonctions

▷ La pile permet de stocker également les variables locales de la fonction, une copie des arguments passés à la fonction, une valeur de retour

CONVENTIONS DE PASSAGE DE PARAMÈTRES

▷ Tous les éléments stockés dans la pile pour une fonction sont placés de manière stricte pour pouvoir y avoir accès facilement

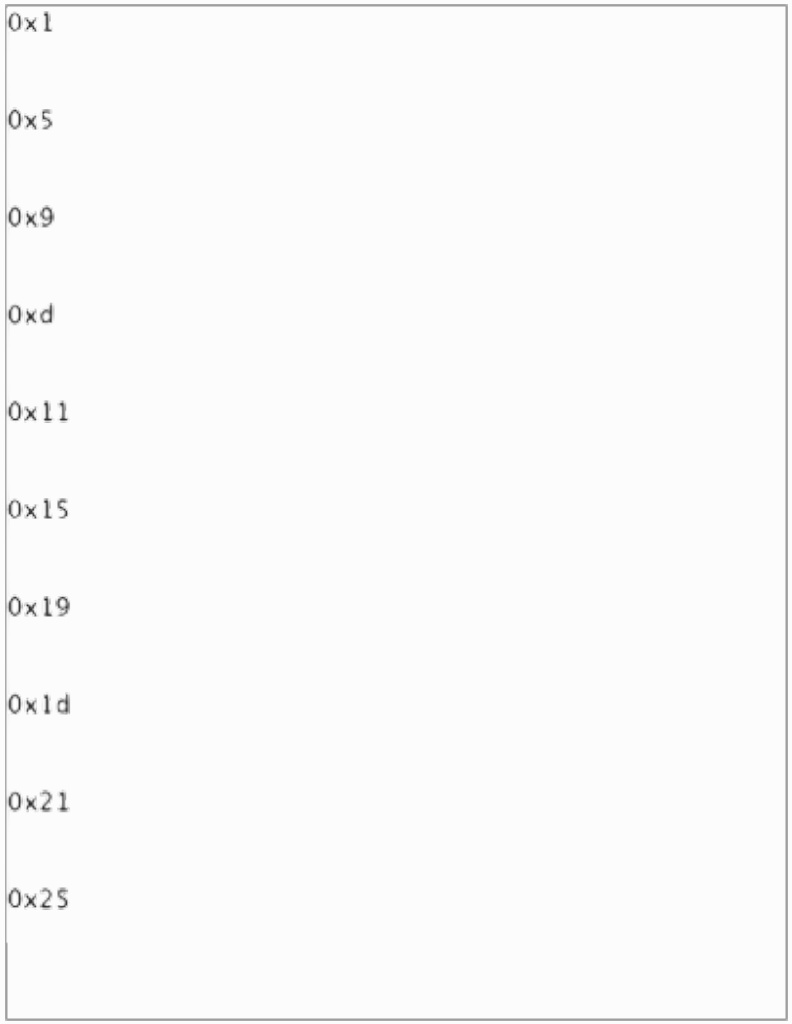
▷ Ce placement suit des conventions liées au processeur que les compilateurs doivent implémenter pour effectuer la traduction en langage machine

▷ Par exemple, en x86, tous les paramètres d'une fonction sont empilés en partant du dernier, suivi de l'adresse de retour et enfin des variables locales

▷ La valeur de retour d'une fonction est transmise via un registre particulier

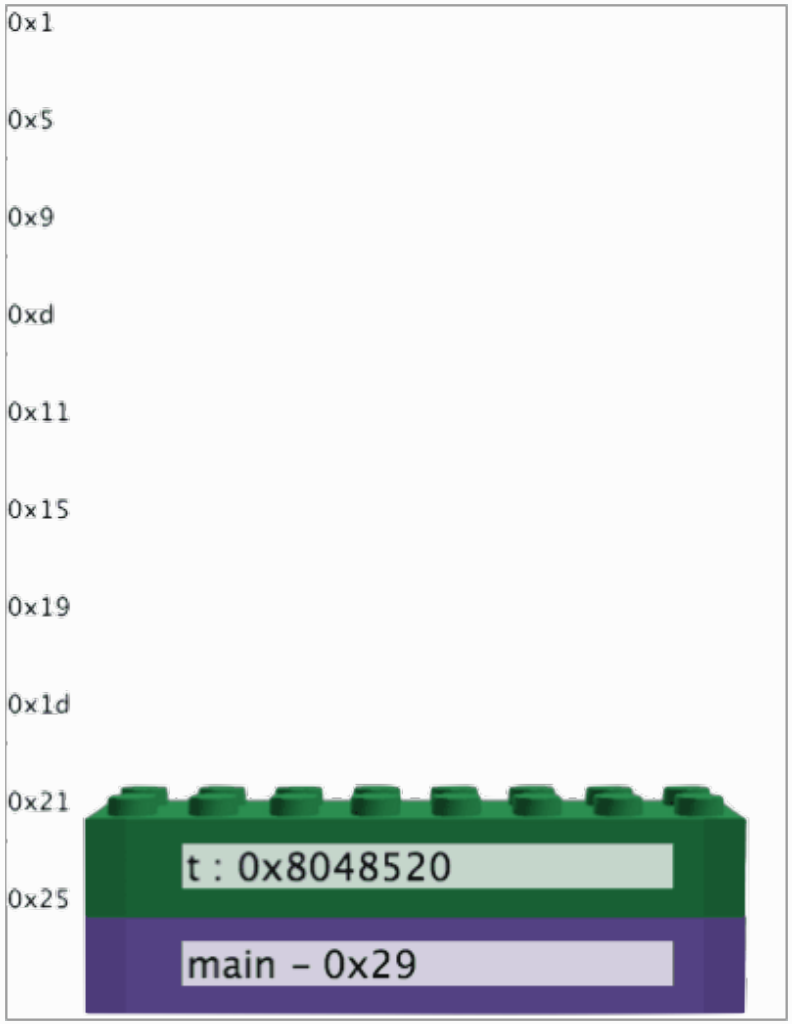
▷ En amd64, jusqu'à 6 paramètres sont directement transmis via des registres; le reste des éventuels paramètres étant empilés

CONVENTIONS DE PASSAGE DE PARAMÈTRES



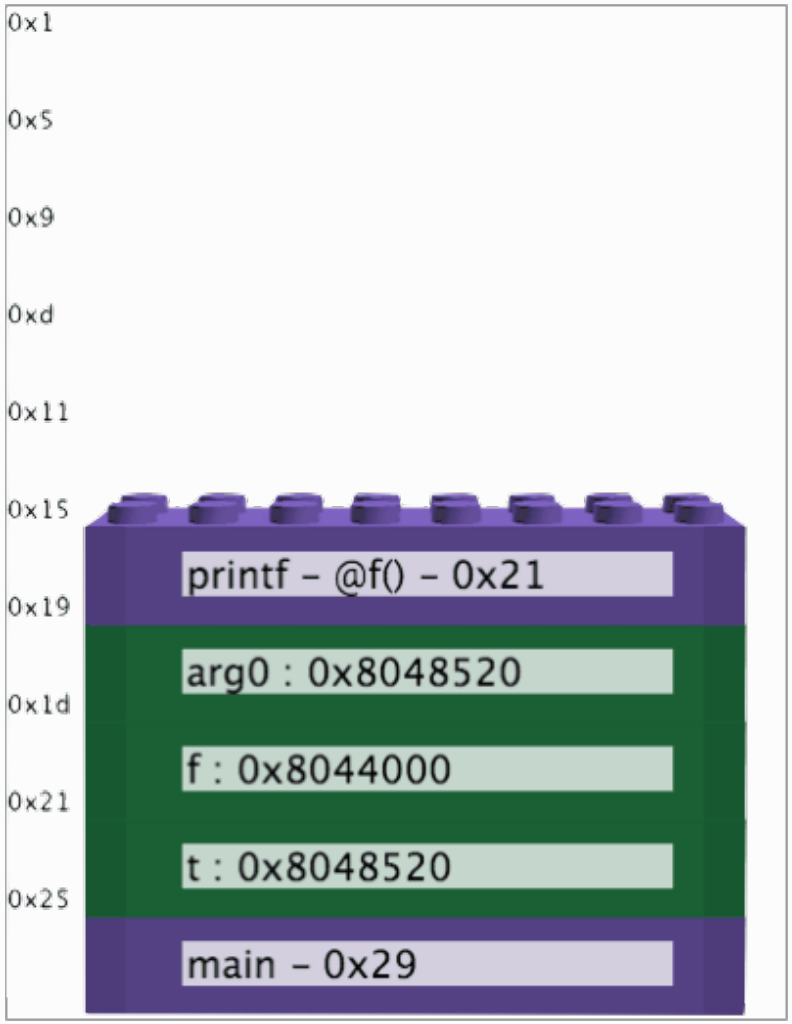

```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



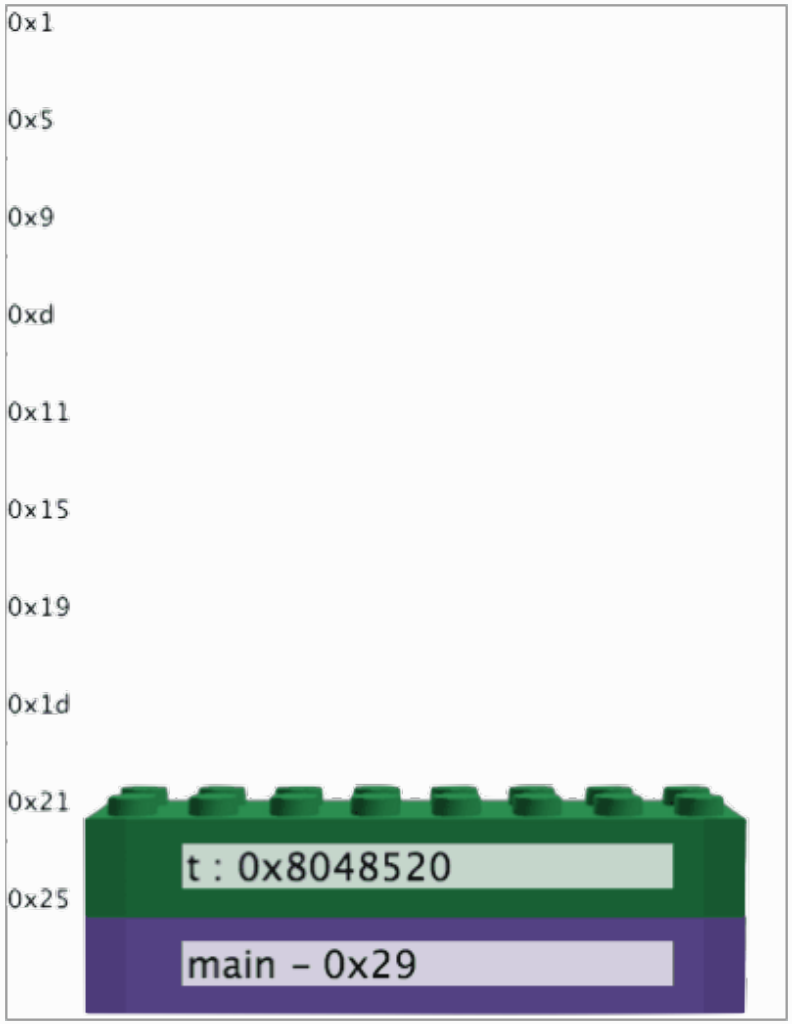

```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



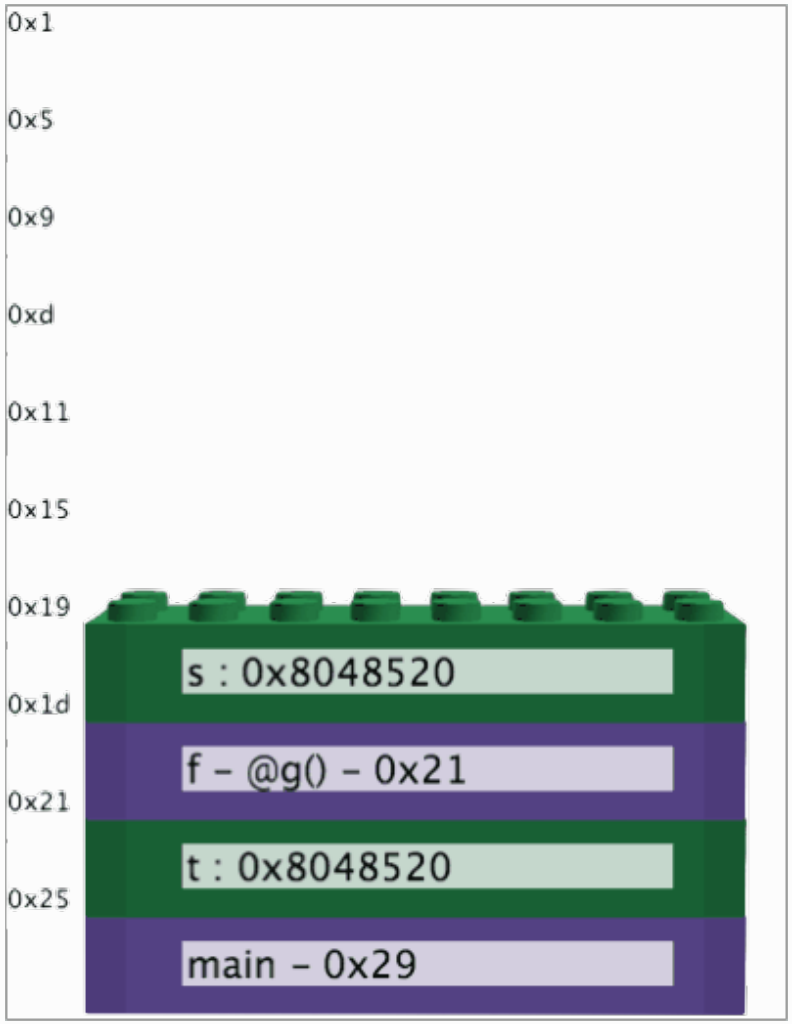

```c
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



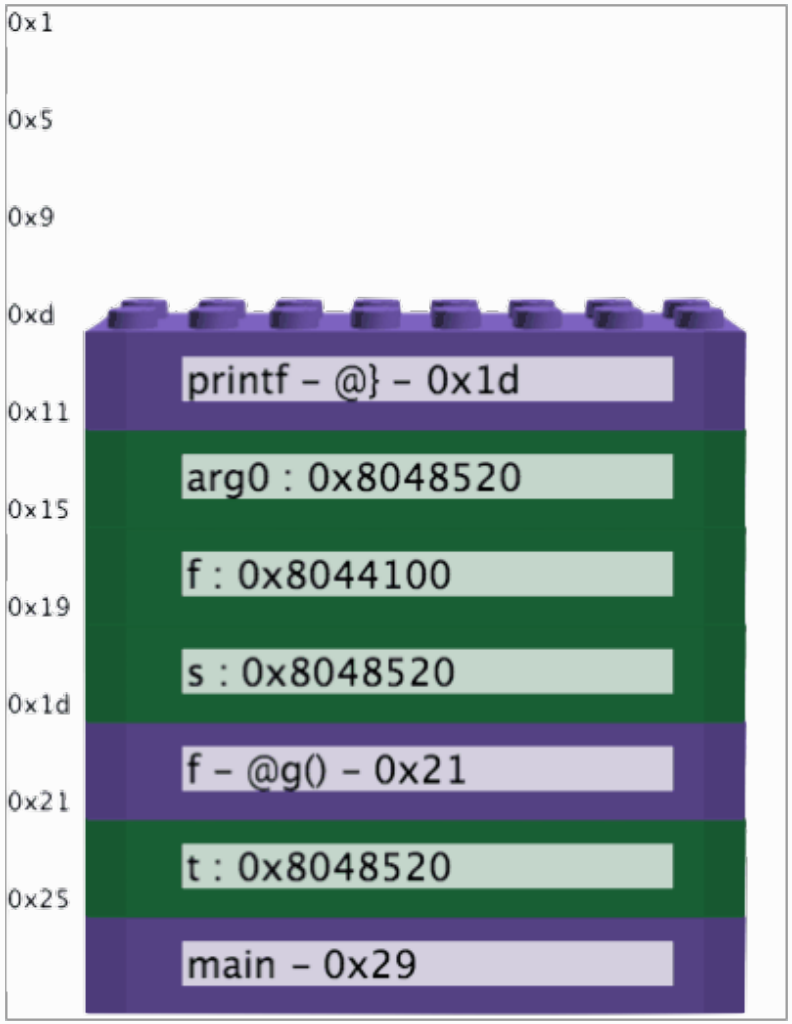

```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



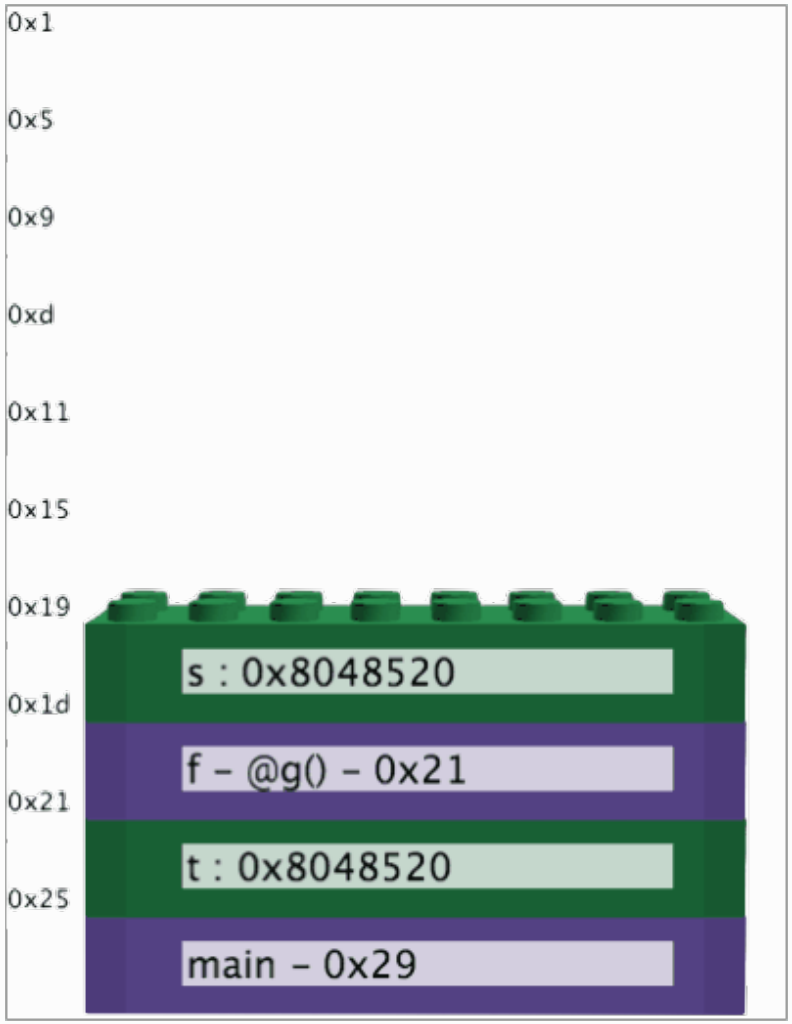

```c
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



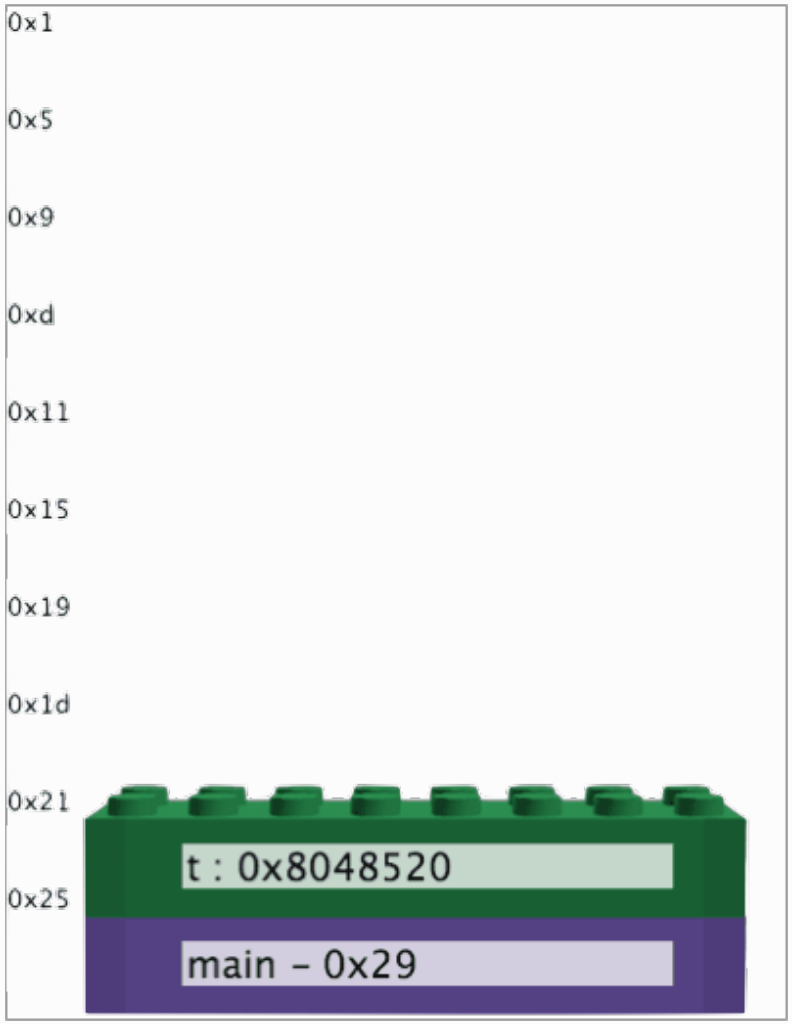

```c
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



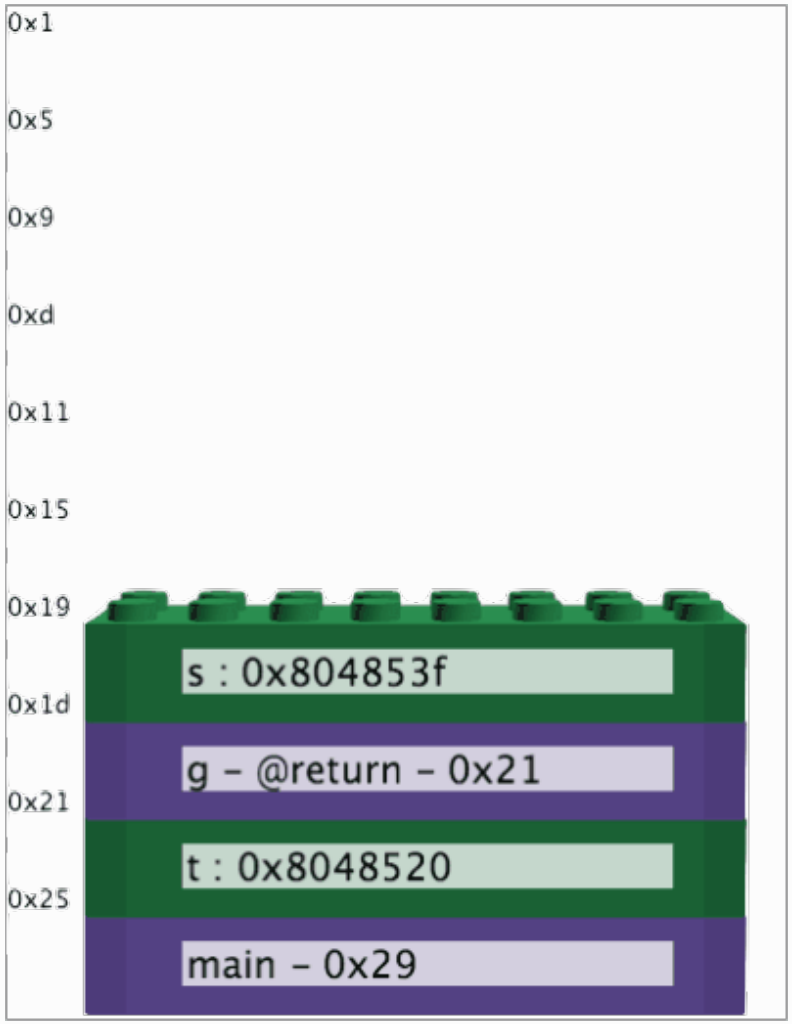

```c
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



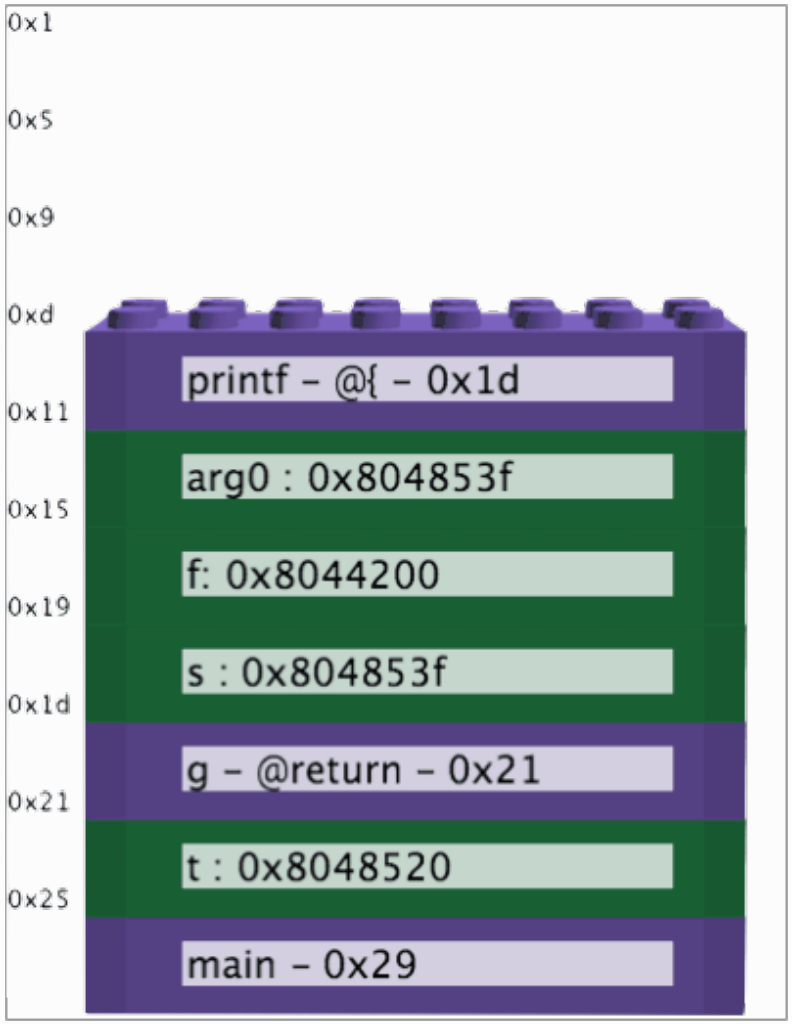

```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



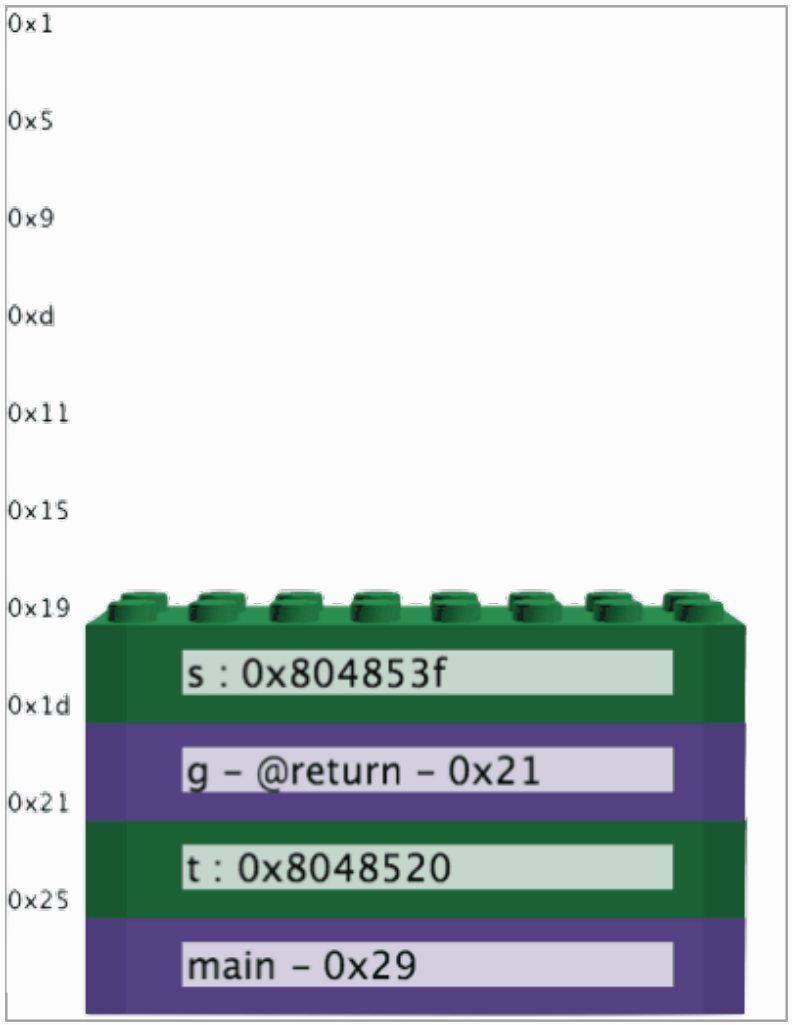

```c
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



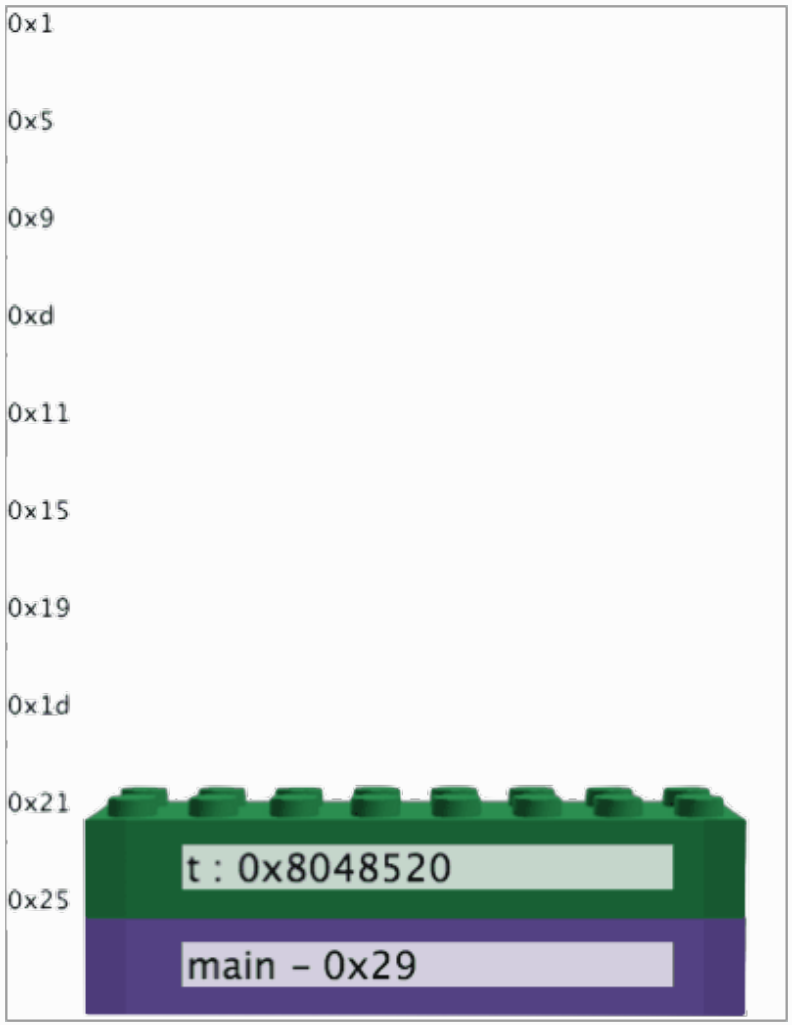

```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES



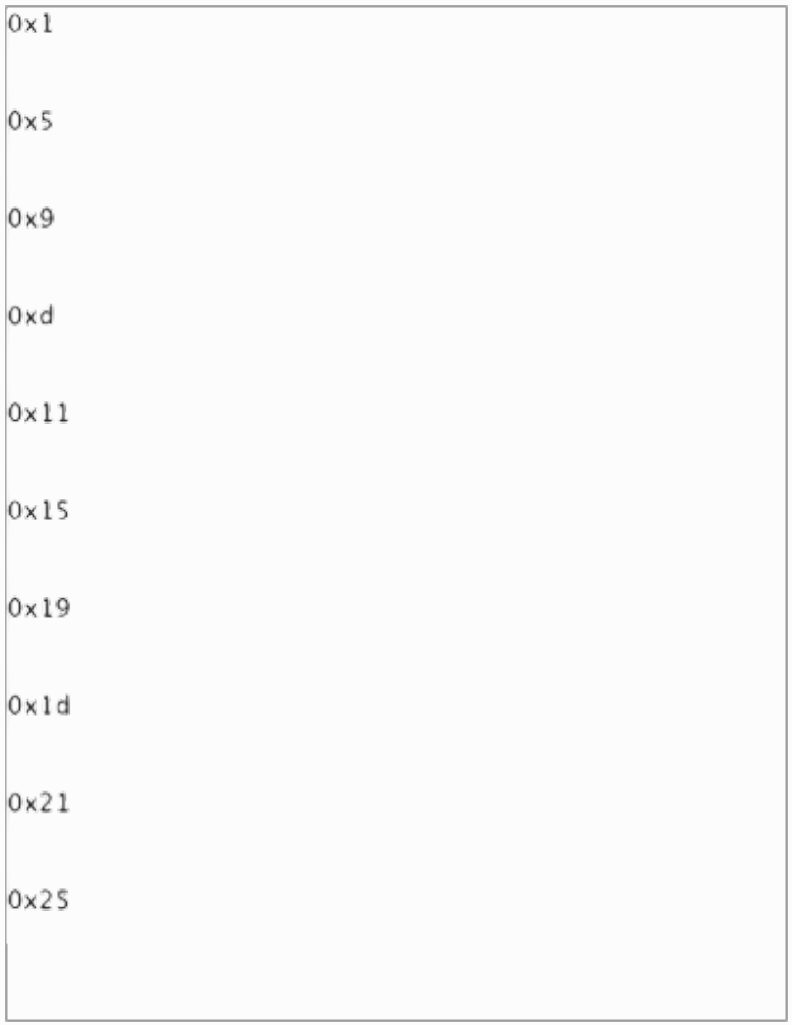

```c
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES




```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

CONVENTIONS DE PASSAGE DE PARAMÈTRES




```
void f/g(){
    char * s = "Hello, World!";
    printf("f/g: %p\n",s);
}
int main(void) {
    char * t = "Hello, World!";
    printf("main: %p\n",t);
    f();
    g();
    return EXIT_SUCCESS;
}
```

ALLOCATION AUTOMATIQUE

▷ La durée de vie d'une variable locale à une fonction est donc bien limitée à celle de la fonction

▷ Il ne faut donc bien jamais renvoyer directement ou indirectement l'adresse d'une variable locale

▷ Les arguments transmis lors de l'appel d'une fonction sont bien des copies (zone mémoire différente) et n'ont donc pas de liens directs avec les variables de la fonction appelante

ALLOCATION AUTOMATIQUE

▷ La taille de la pile d'un processus est limitée et fixée lors de l'exécution

▷ Sous linux, il est possible de la connaître et de la fixer à l'aide de la commande `ulimit`

DOGGY BAG

TO TAKE AWAY ...

▷ La mémoire est segmentée

▷ Tout ce qui est immuable est présent dans le binaire (le code et les constantes)

▷ La pile permet l'allocation automatique d'espace mémoire mais cette dernière est très gourmande (car stocké sur la pile pour chaque appel)

▷ Il peut s'avérer plus adapté de réserver et libérer de la mémoire à la demande (c.f. le prochain cours)

Questions?