

Get up to speed with this  
fun and friendly road map to the technology

# **Introduction au C**

FOR  
**DUMMIES<sup>®</sup>**

CD-ROM loaded  
with goodies

**A Reference  
for the  
Rest of Us!**

**B. Quoitin**  
**Ecole Polytechnique de Louvain**  
**Université catholique de Louvain**



# Table des matières

- Rappels (?)
- Mémoire
- Exécution d'un programme
- Le segment de pile
- Les pointeurs
- Allocation dynamique de mémoire

# Rappel

- Bases

- 2 (binaire) :  $[0, 1]$
- 10 (décimale) :  $[0, 9]$
- 16 (hexadécimale) :  $[0, 9] \cup [A, F]$

# Rappel

- Correspondance entre bases

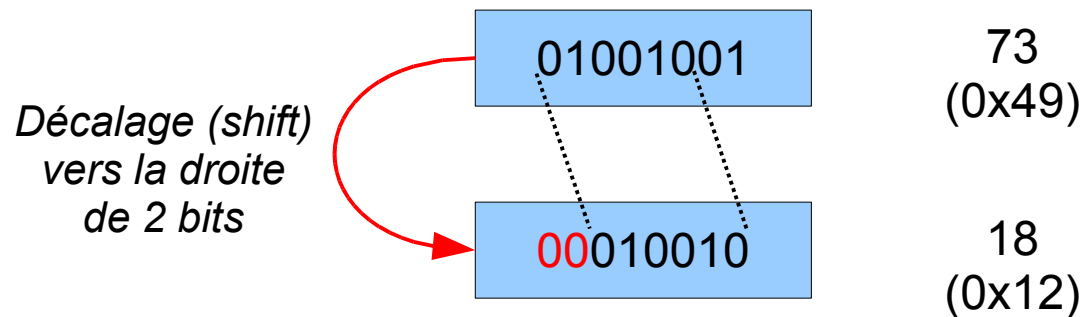
$$\begin{aligned} 1219 &= 1024 + 128 + 64 + 2 + 1 \\ &= 2^{10} + 2^7 + 2^6 + 2^1 + 2^0 \\ &= \text{b}10011000011 \\ &\quad \underbrace{\quad\quad\quad}_4 \quad \underbrace{\quad\quad}_C \quad \underbrace{\quad}_3 \\ &= 4 * 16^2 + C * 16^1 + 3 * 16^0 \\ &= 0x4C3 \end{aligned}$$

Note: il peut être utile de connaître la plupart des exposants de 2

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, ...

# Rappel

- Opérations utiles

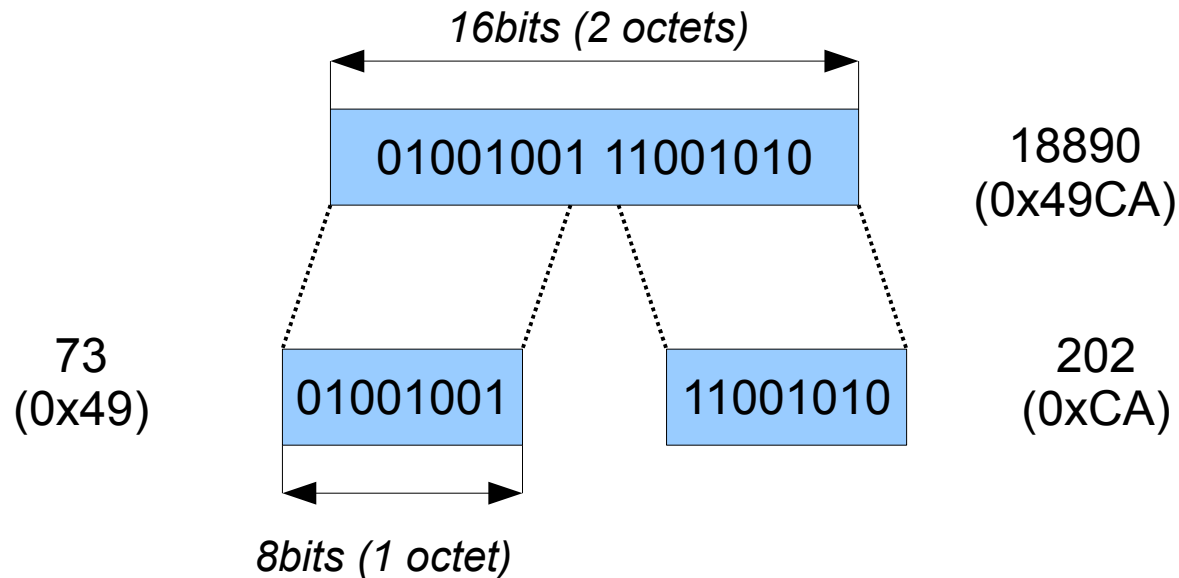


Note:

- Le **décalage vers la droite de  $N$  bits** revient à effectuer une division entière par  $2^N$ . Par exemple, un décalage de 2 bits vers la droite effectue une division par 4.
- Symétriquement, le **décalage de  $N$  bits vers la gauche** revient à effectuer une multiplication par  $2^N$ . Attention toutefois au dépassement de la capacité (les bits de gauche sont perdus).

# Rappel

- Taille des données



Note: un entier non signé  $E_{16}$  de taille 16 bits est égal à la concaténation des bits de 2 entiers non signés  $E_{8G}$  et  $E_{8D}$  de taille 8 bits. Numériquement,

$$E_{16} = (E_{8G} * 2^8) + E_{8D}$$

$E_{8G}$  est l'octet de poids fort (gauche) et  $E_{8D}$  est l'octet de poids faible (droite).

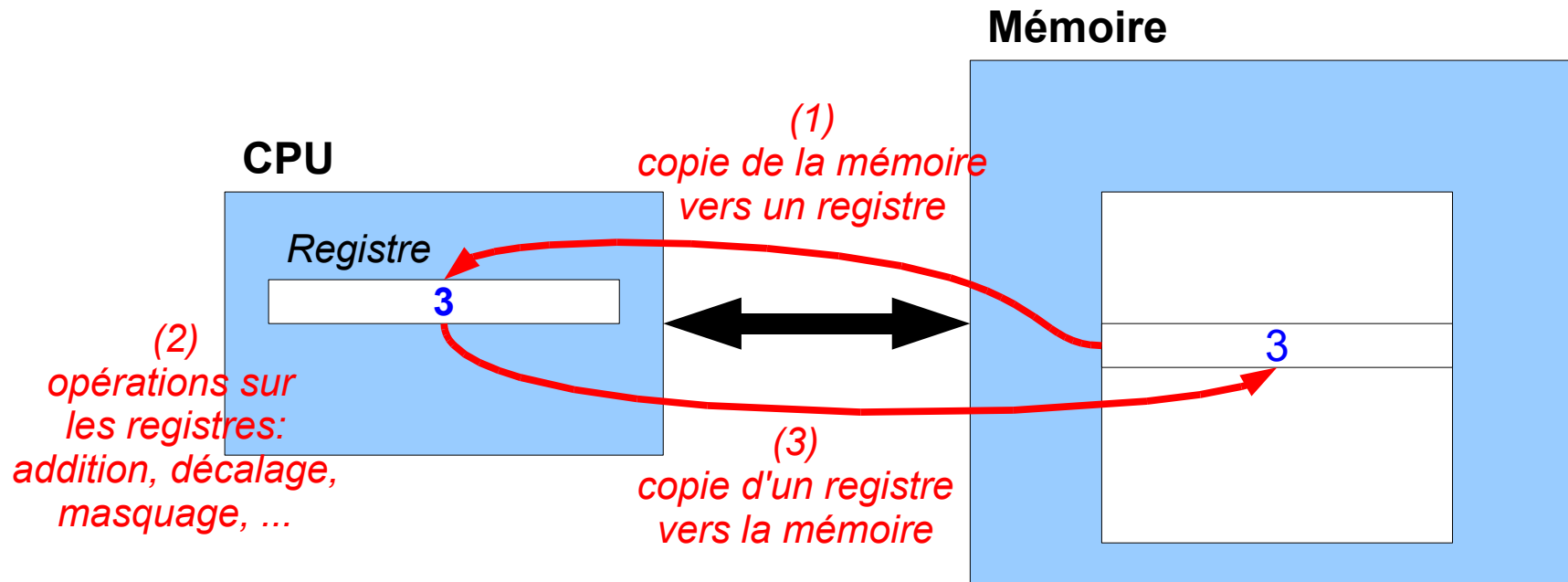
# Mémoire

# Mémoire

- La mémoire vue par un programme C
  - Concept d'adresse
  - Représentation des données
  - Plus bas niveau qu'en Java
  - Comprendre la réalité physique des pointeurs afin d'éviter les erreurs de manipulation classiques



# Mémoire

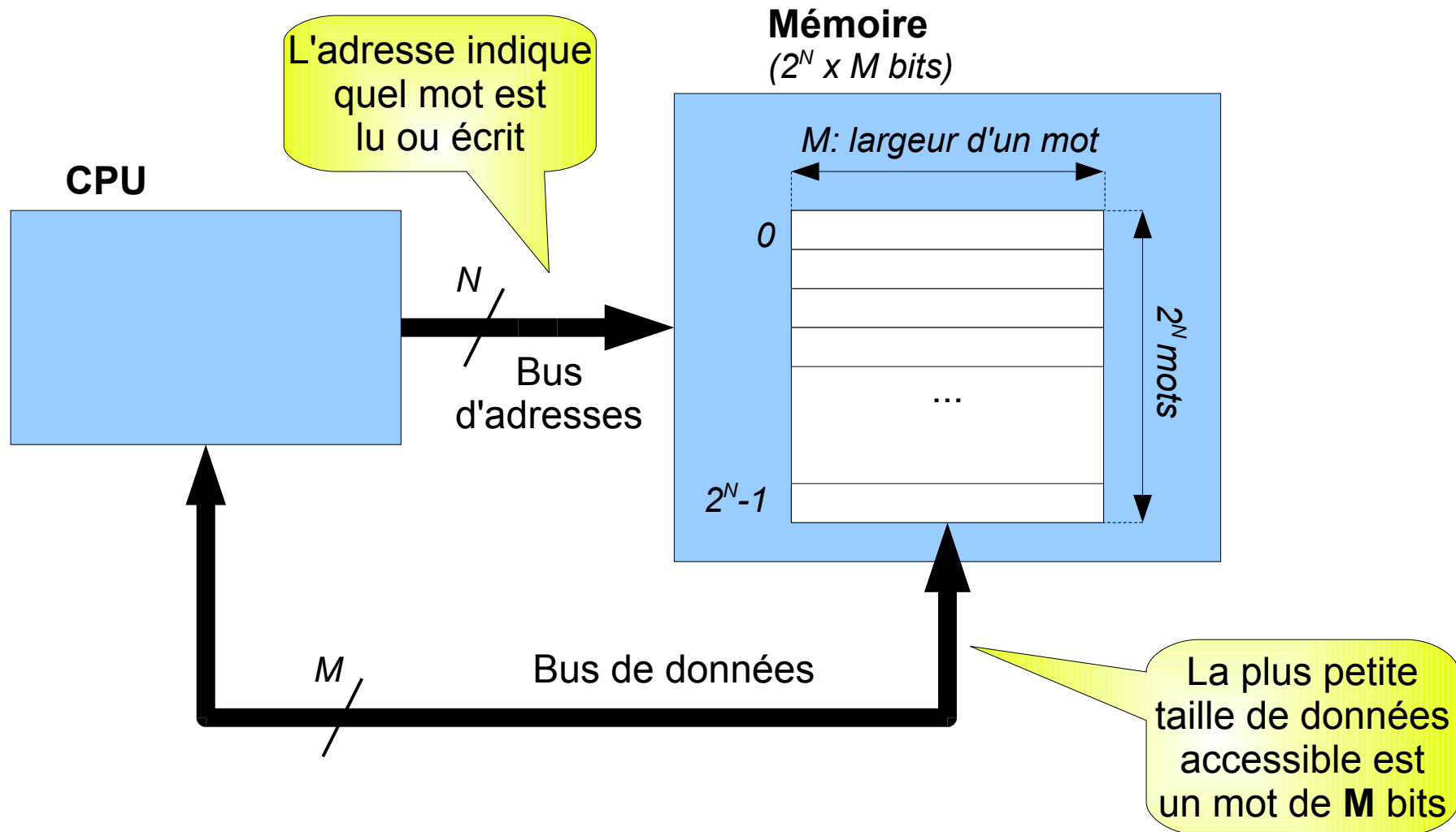


## Notes importantes:

En général, les CPU ne travaillent **pas directement** sur les données en mémoire. Celles-ci sont rapatriées dans des registres du CPU. Elles y sont modifiées, puis éventuellement recopiées en mémoire.

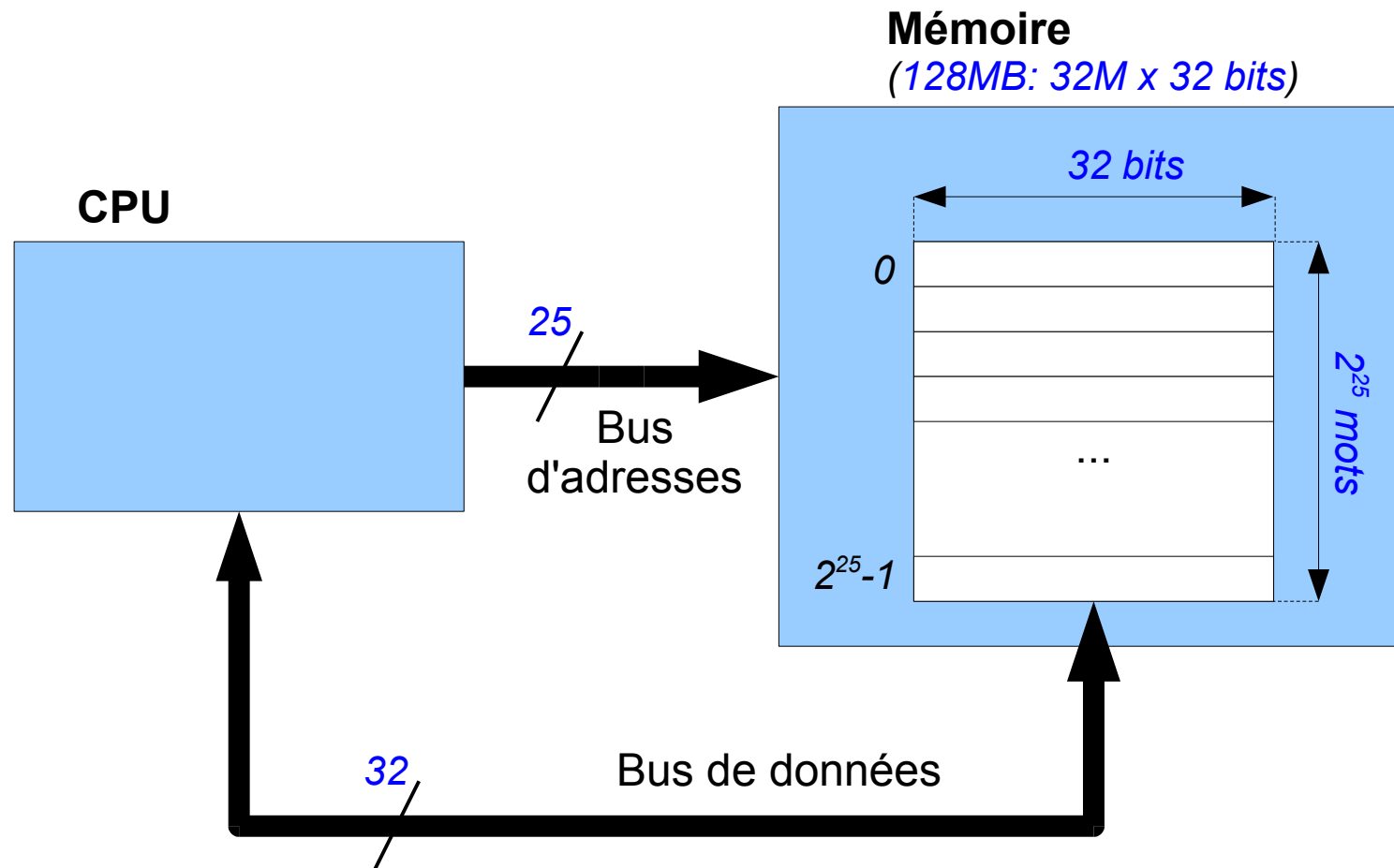
Un CPU a un **nombre limité de registres** (de l'ordre d'une  $10^{\text{aine}}$ ), ce qui implique qu'il peut traiter un nombre limité de données à la fois.

# Mémoire

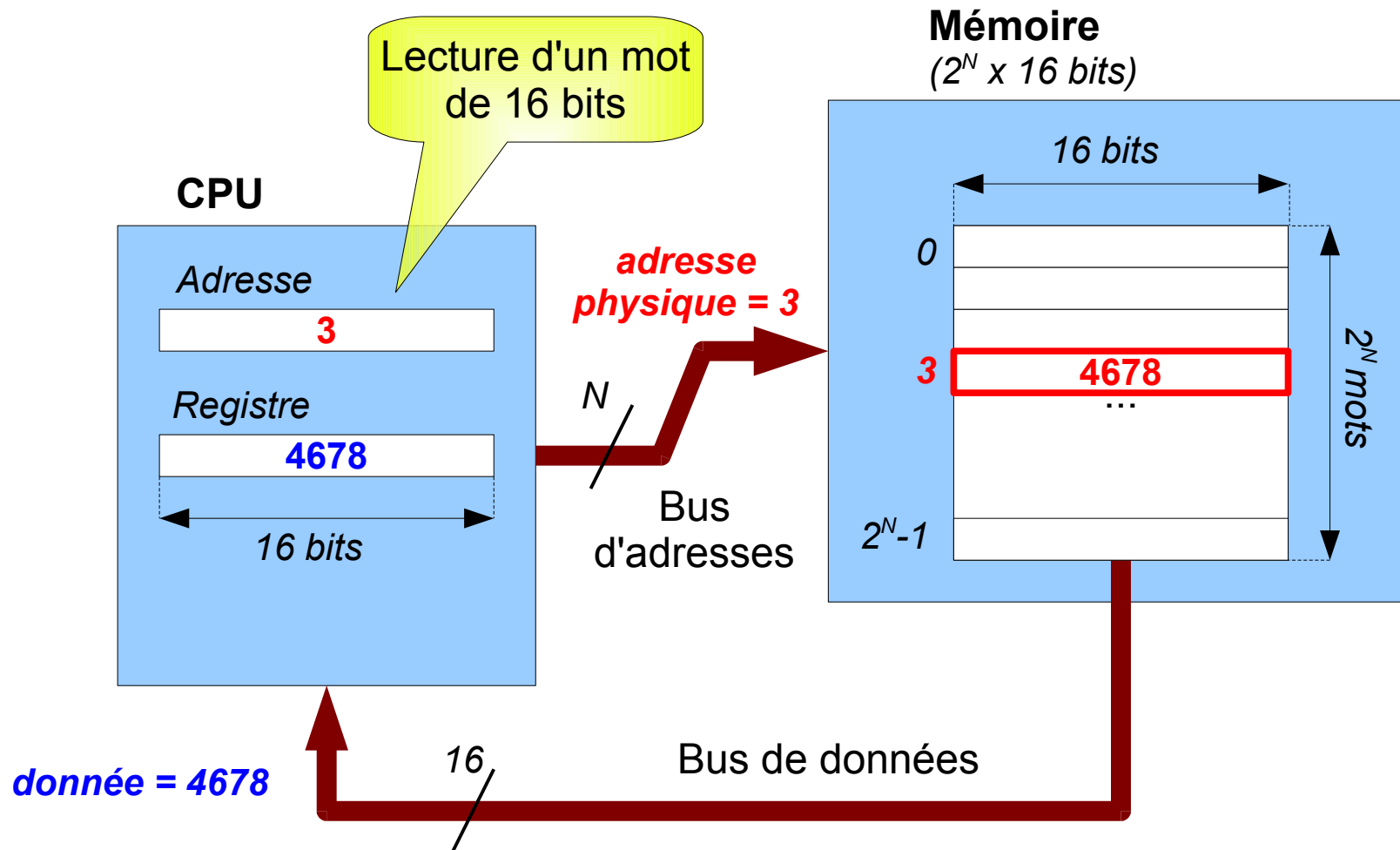


Notes: parfois les bus d'adresses et de données sont partagés.

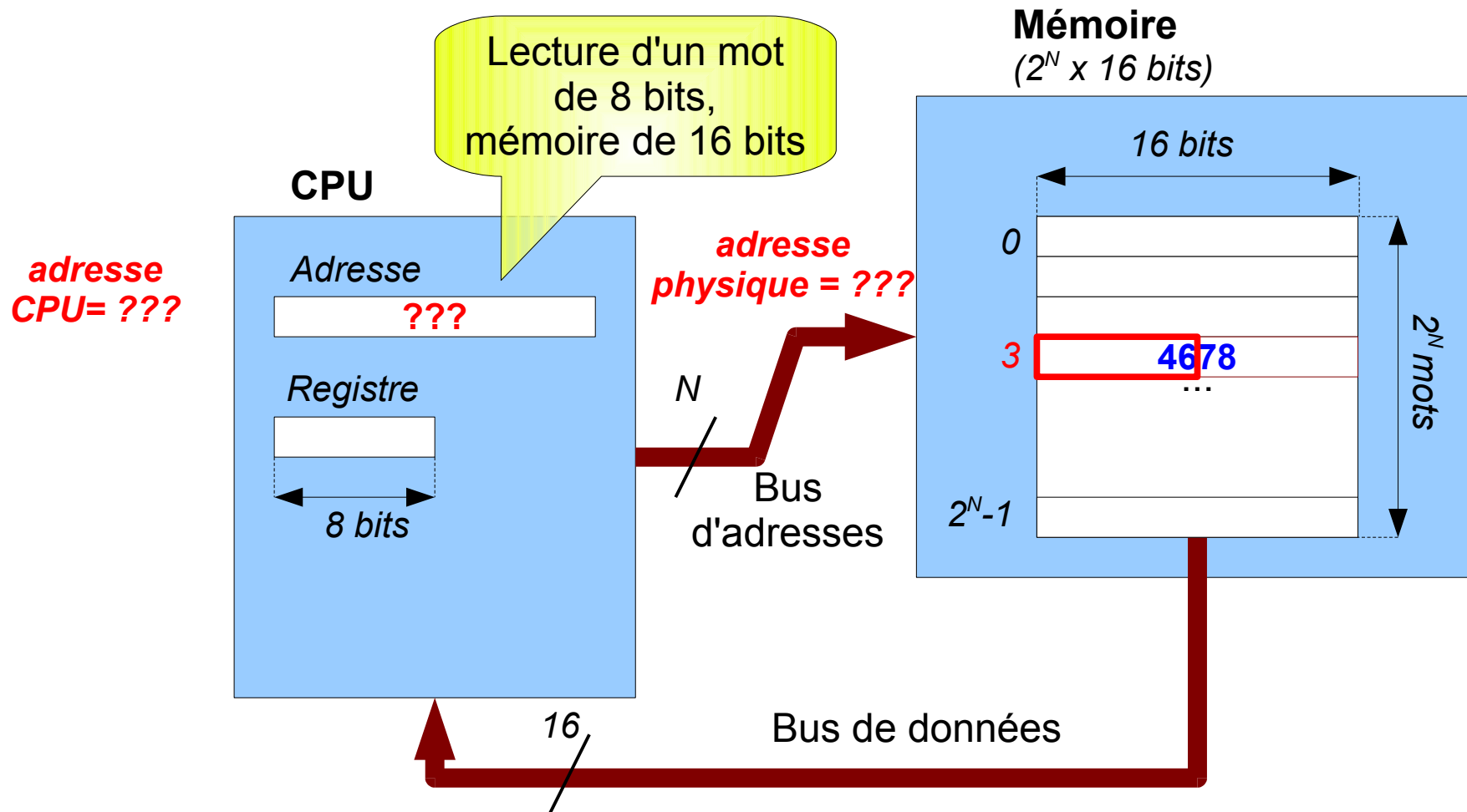
# Mémoire (exemple)



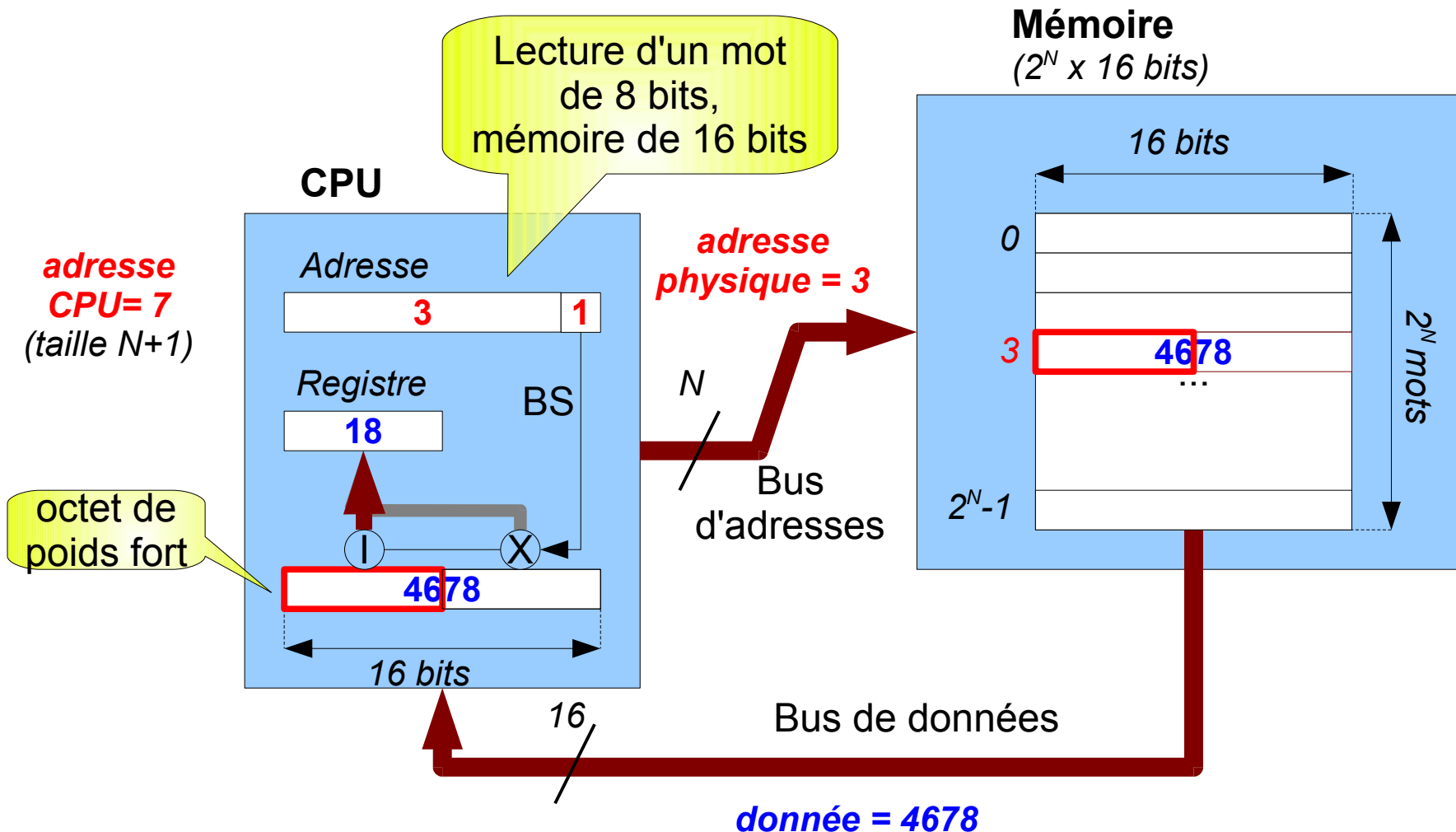
# Mémoire



# Mémoire



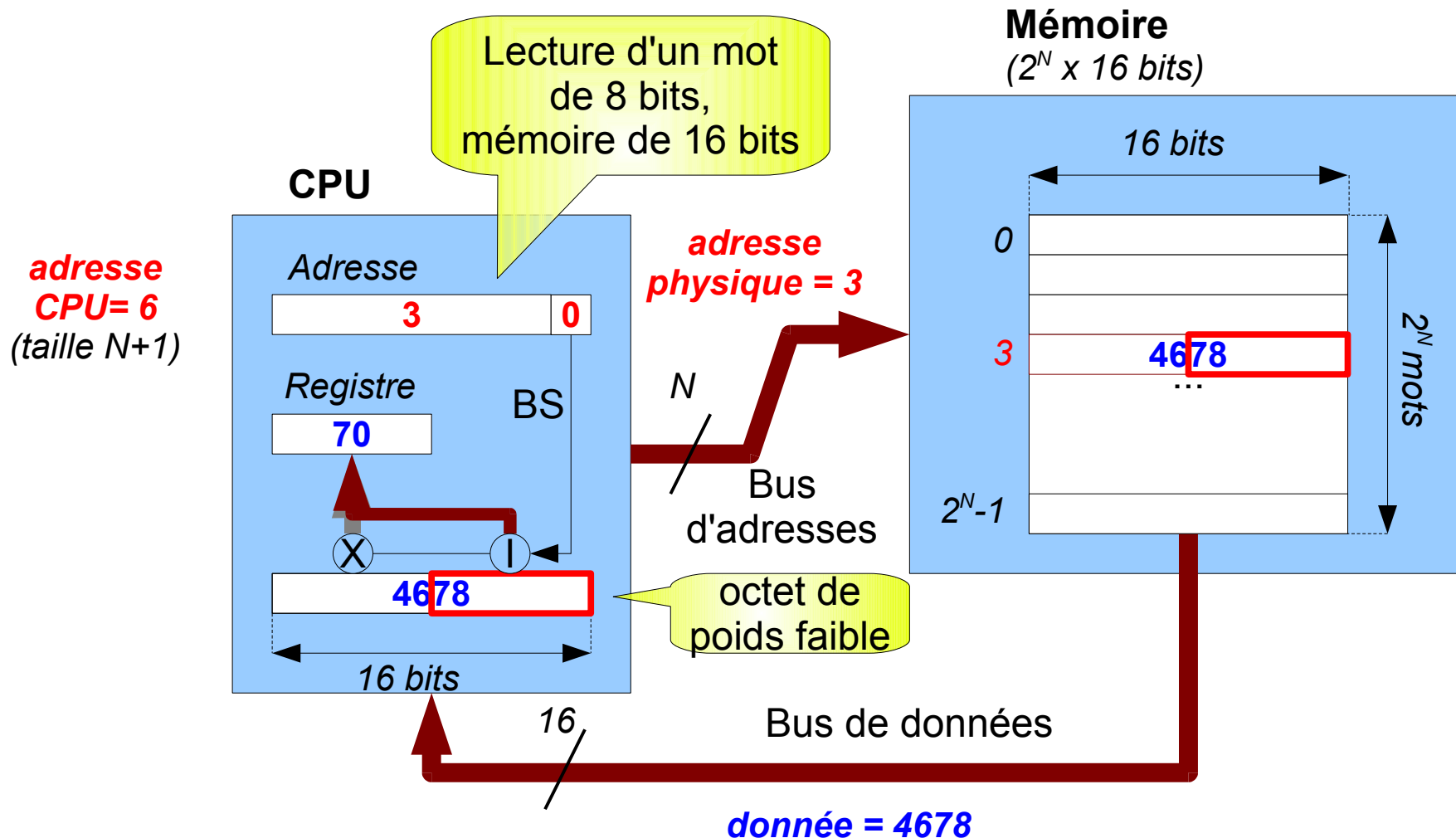
# Mémoire



## Détails:

- adresse CPU = (adresse physique # BS) =  $(3 \times 2) + 1 = 7$
- donnée:  $4678 = 18 \times 2^8 + 70$

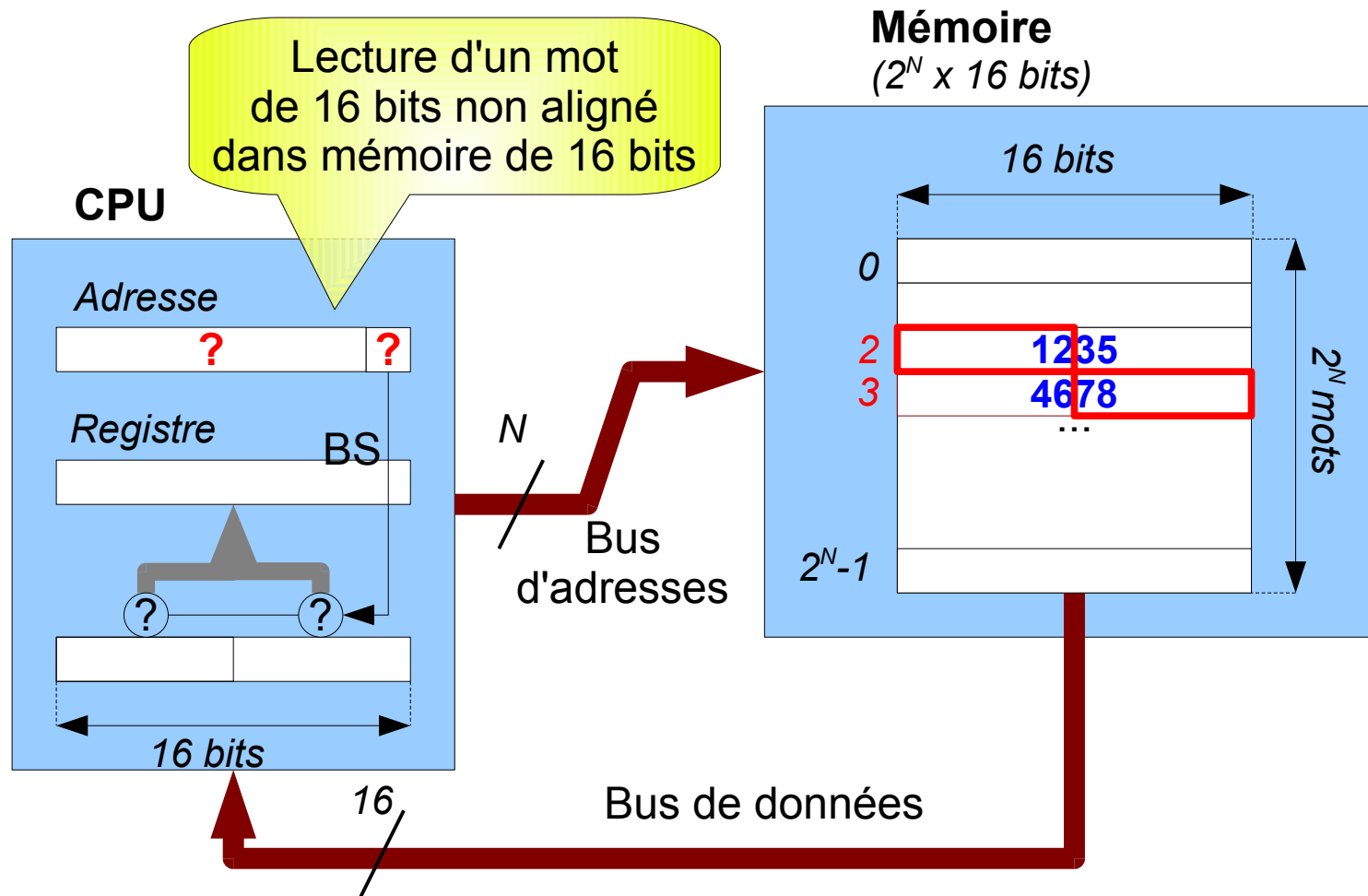
# Mémoire



## Détails:

- adresse CPU = (adresse physique # BS) =  $(3 \times 2) + 0 = 6$
- donnée:  $4678 = 18 \times 2^8 + 70$

# Mémoire



Note: pour lire 16 bits à l'adresse CPU 5, 2 lectures sont nécessaires. Il faut d'abord lire l'octet de poids fort à l'adresse physique 2 puis l'octet de poids faible à l'adresse physique 3.



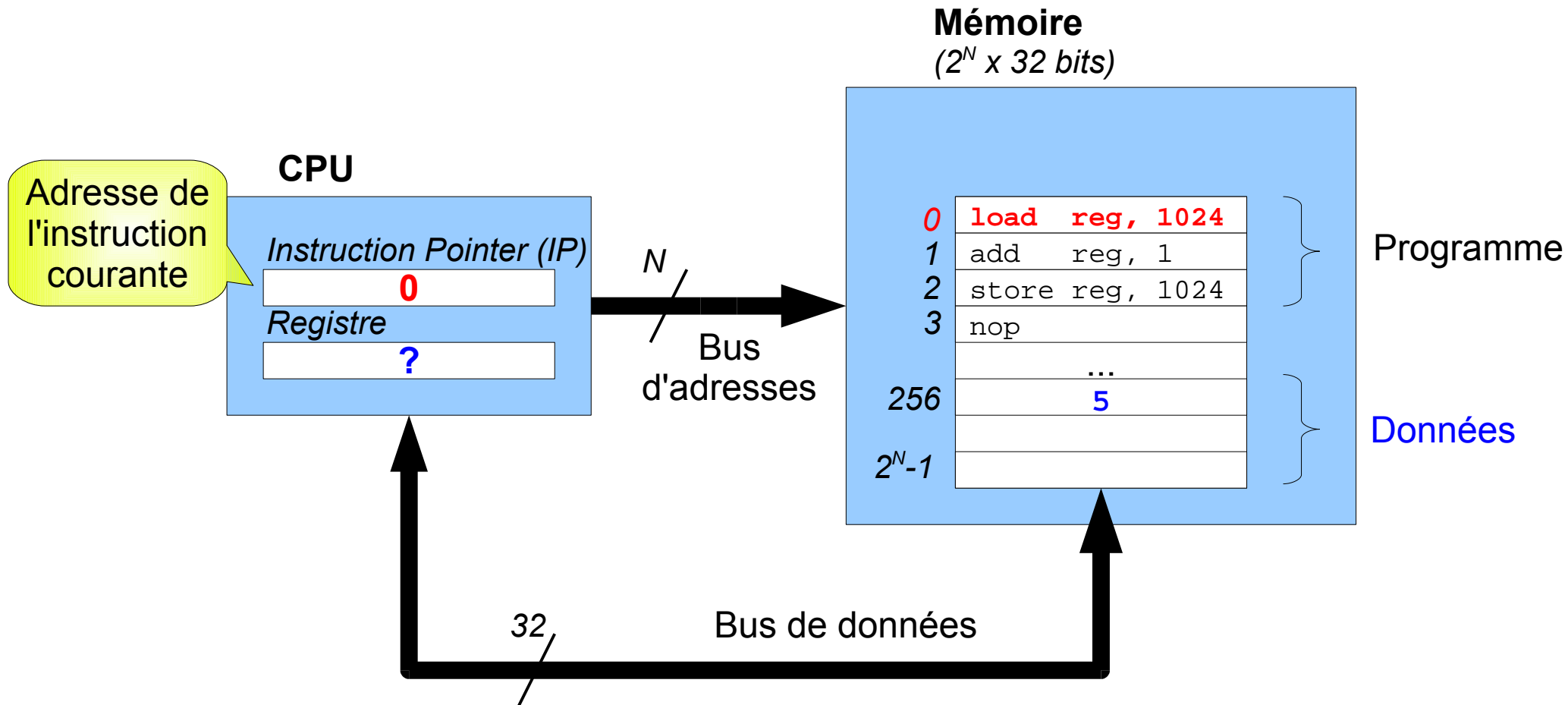
# Mémoire

- Résumé

- Le traitement des données nécessite leur rapatriement dans des registres du CPU.
- Le CPU permet d'adresser un octet (taille 8bits)
- La mémoire permet d'adresser un mot de M bits (en général  $M \geq 8$ )
- Aligner les données en mémoire
  - plus efficace
  - le compilateur s'en charge en général

# Exécution d'un programme

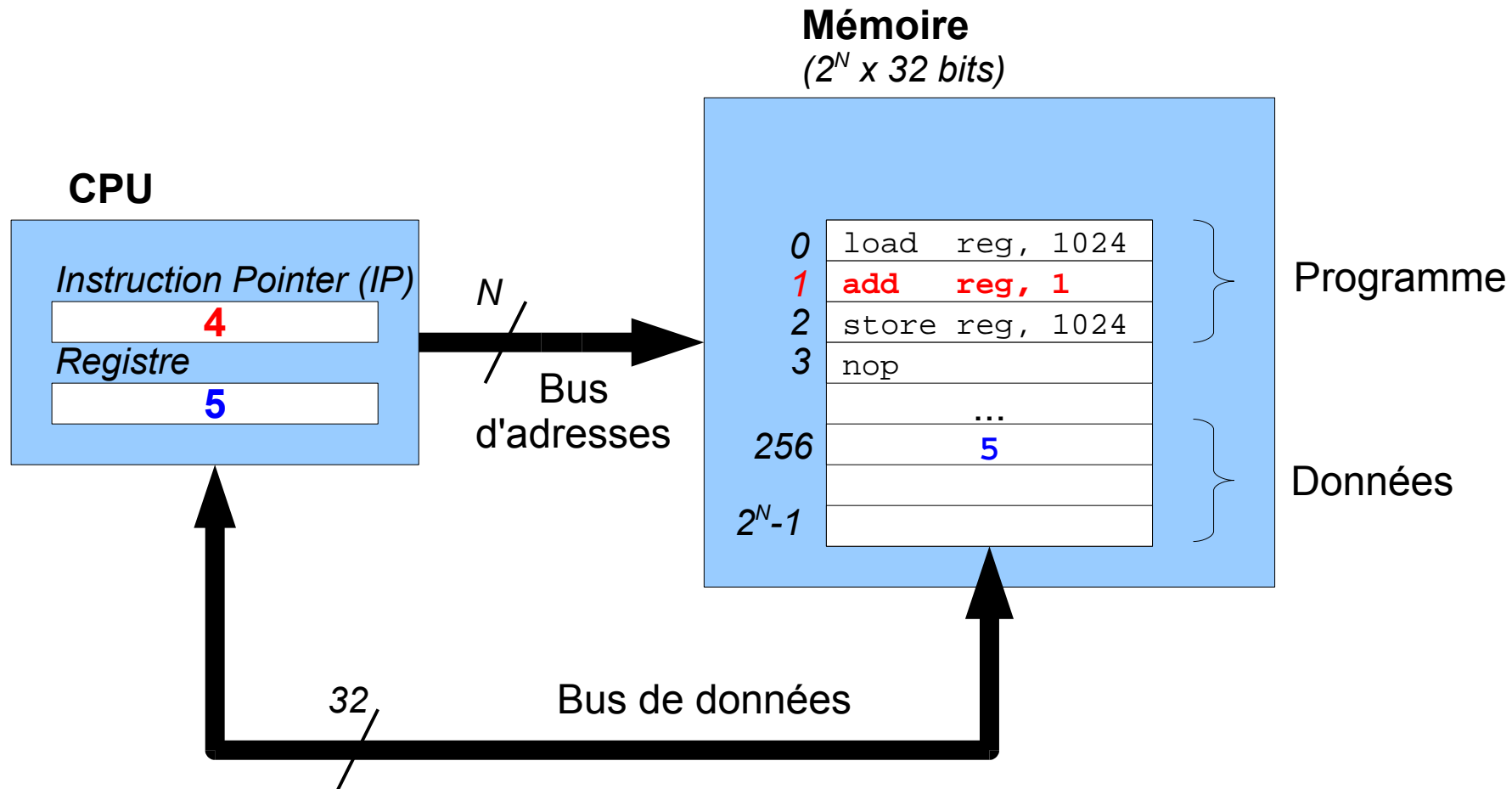
# Exécution d'un programme



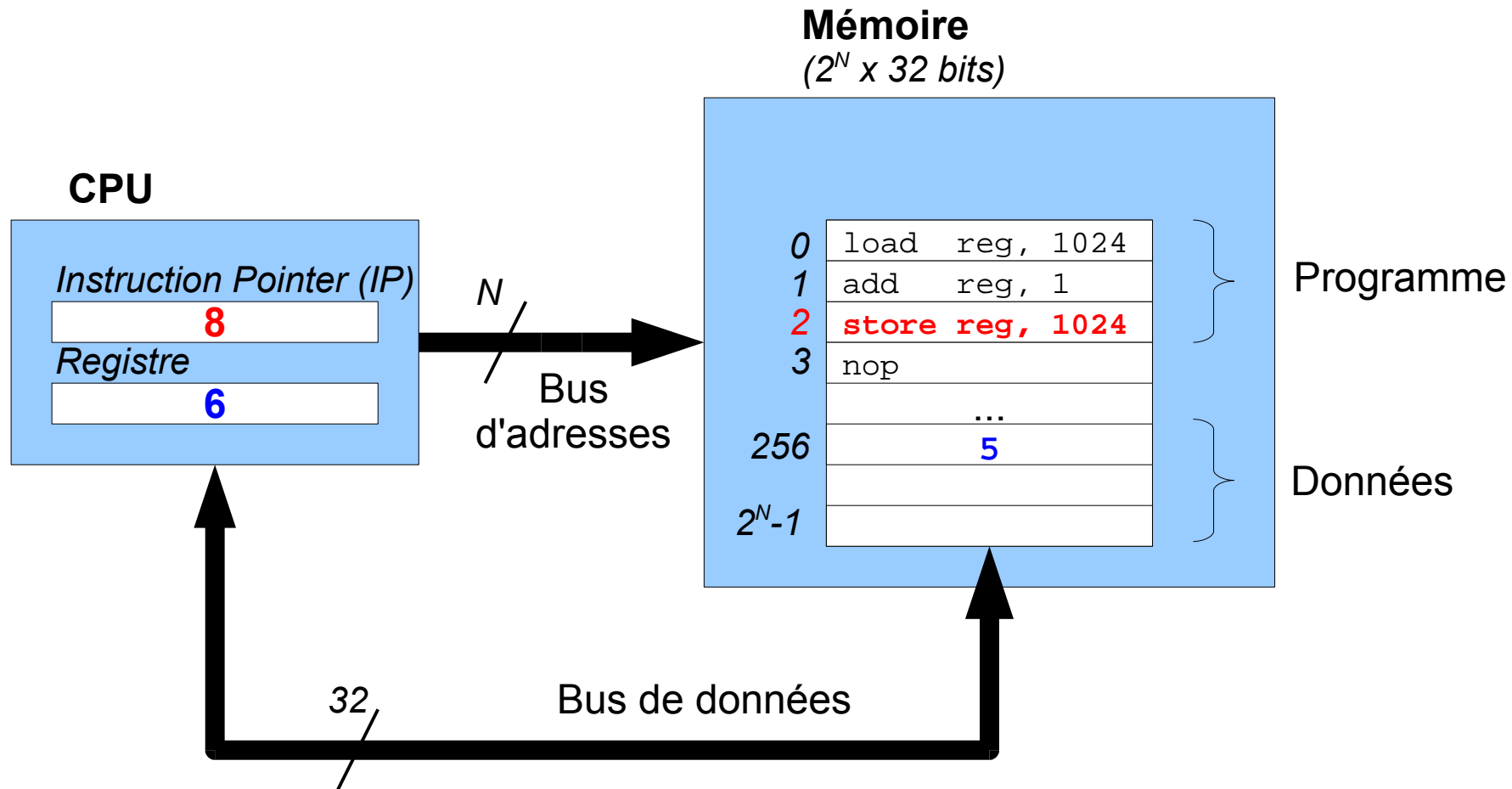
Hypothèse: longueur d'une instruction = 1 mot = 4 octets = 32 bits

Note: les instructions sont aussi rapatriées de la mémoire vers le CPU (non montré dans l'illustration par simplicité).

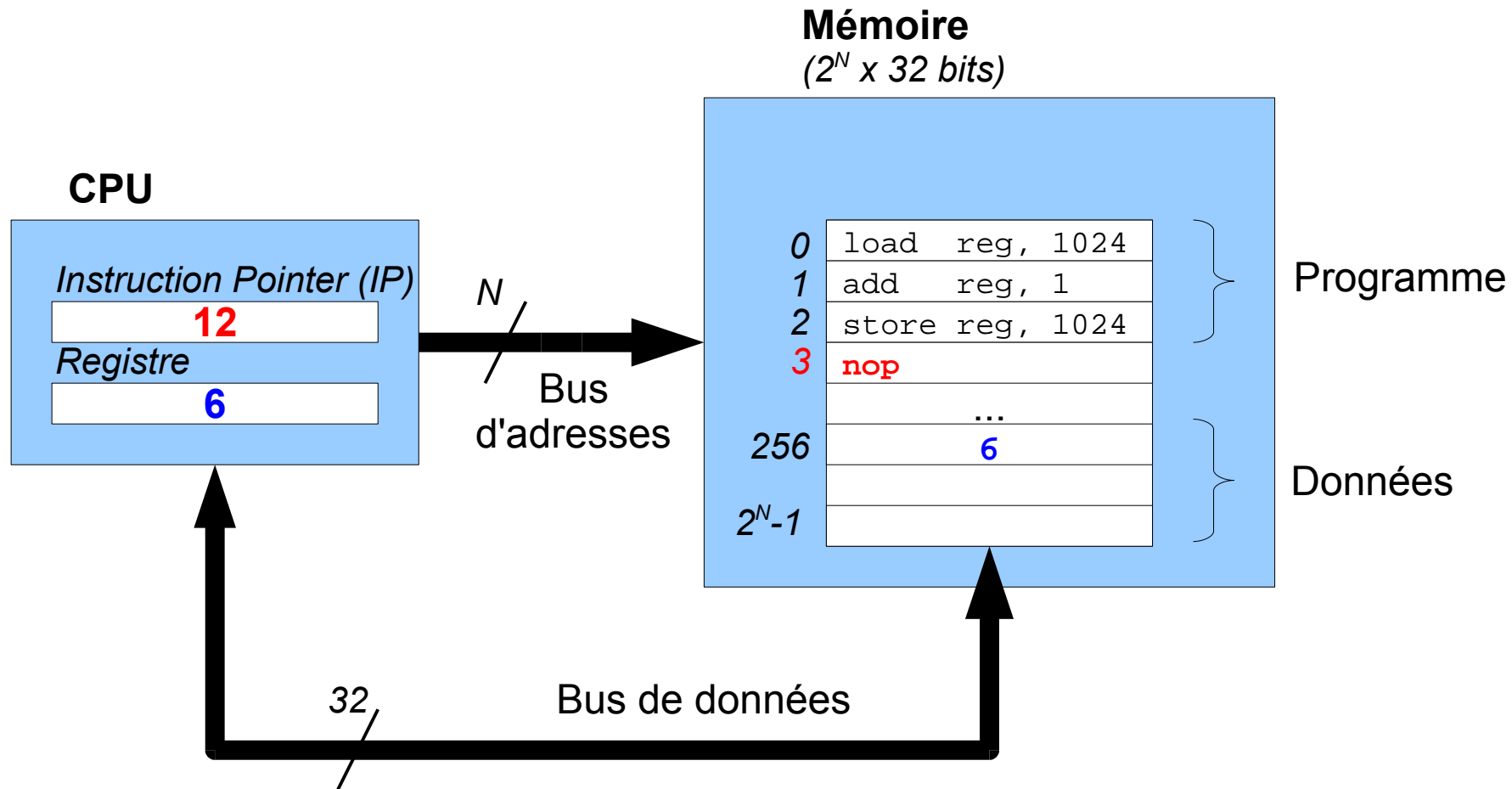
# Exécution d'un programme



# Exécution d'un programme

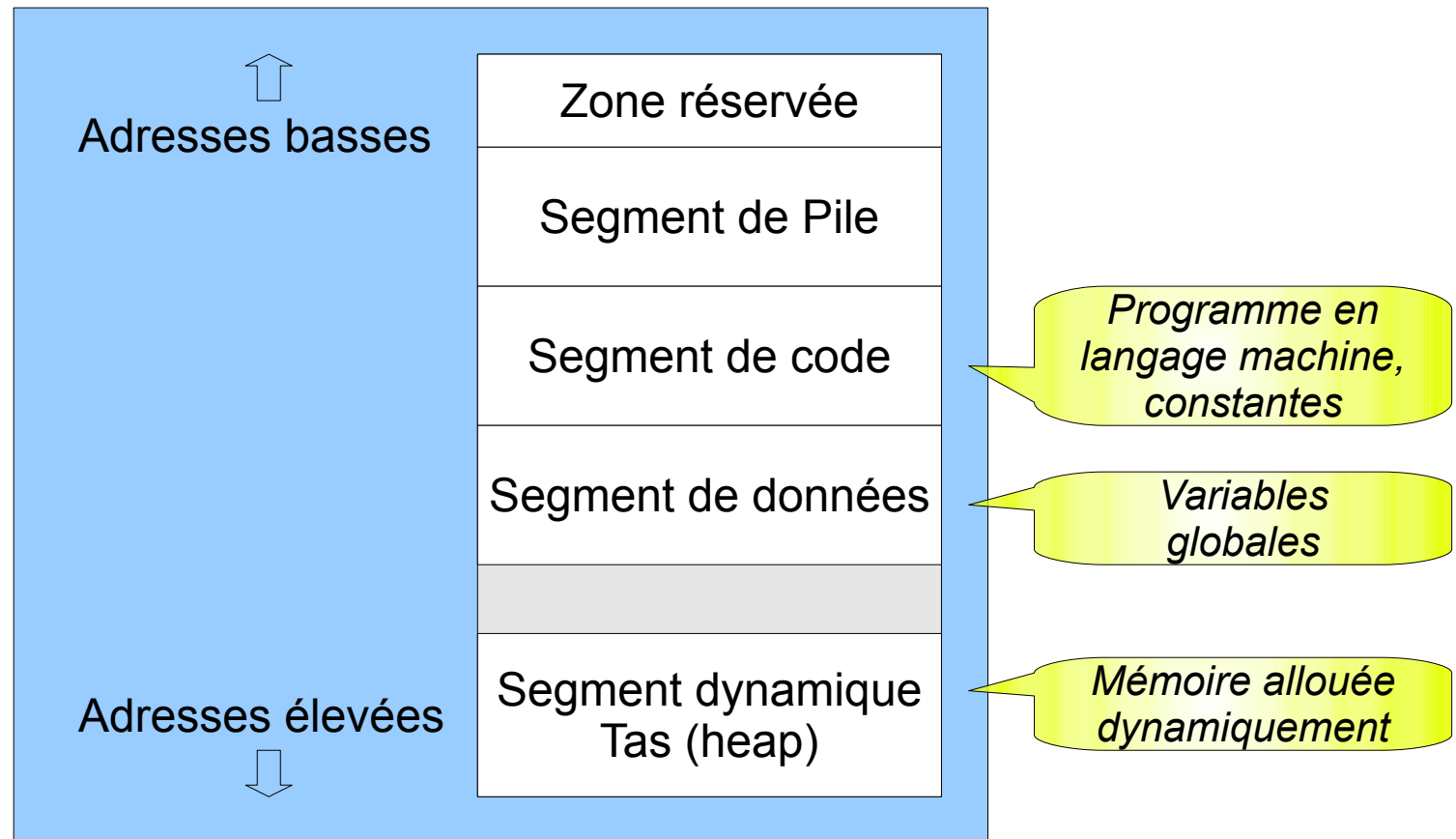


# Exécution d'un programme



# Programme en mémoire

## Mémoire



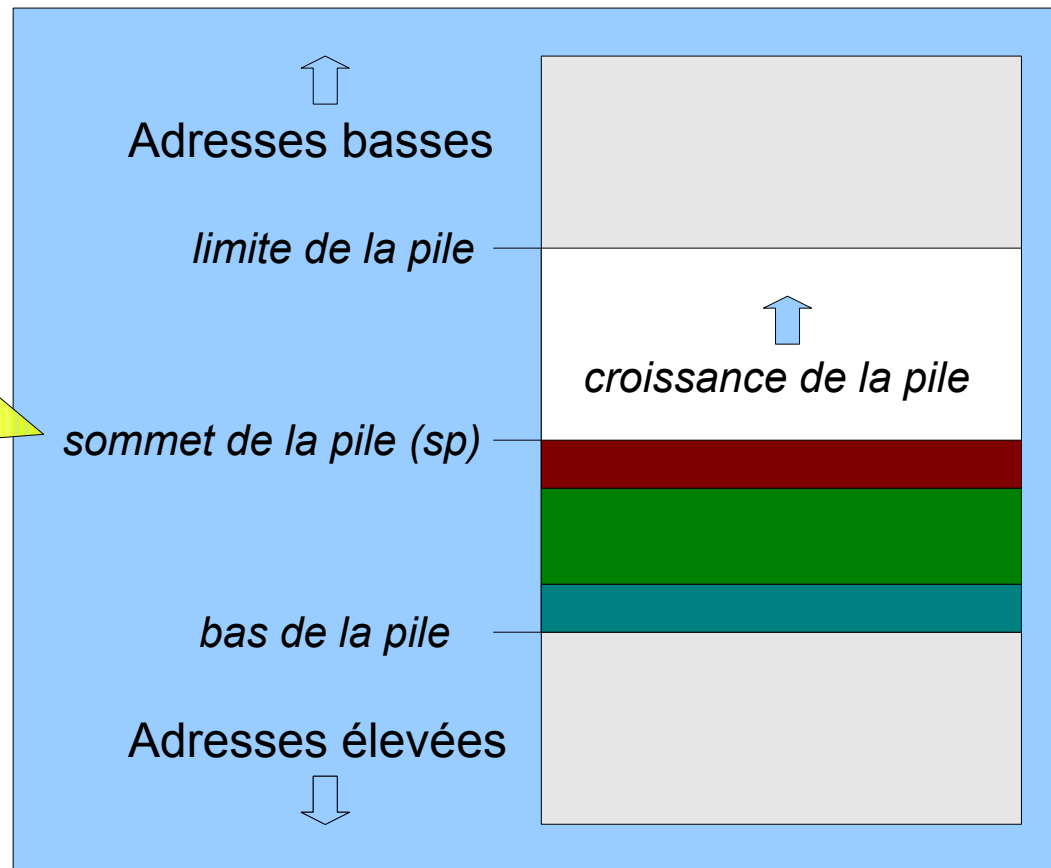
Note: l'agencement des différents segments diffère d'un système à l'autre.

# Le Segment de Pile



# Segment de pile

## Mémoire



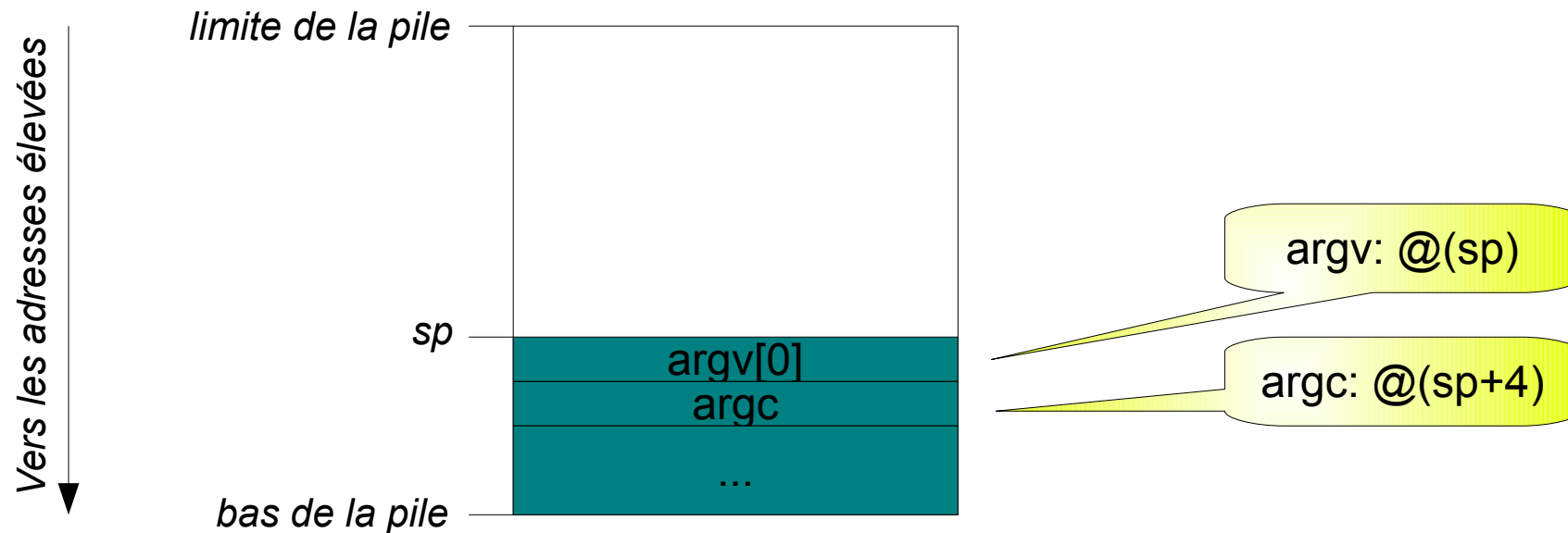
# Segment de pile

- Utilisation
  - Passage des arguments de fonctions
    - permet les appels récursifs
  - Variables locales aux fonctions
  - Sauvegarde du contenu de registres
  - Sauvegarde de l'adresse de retour d'une fonction

# Segment de pile

- Passage d'arguments

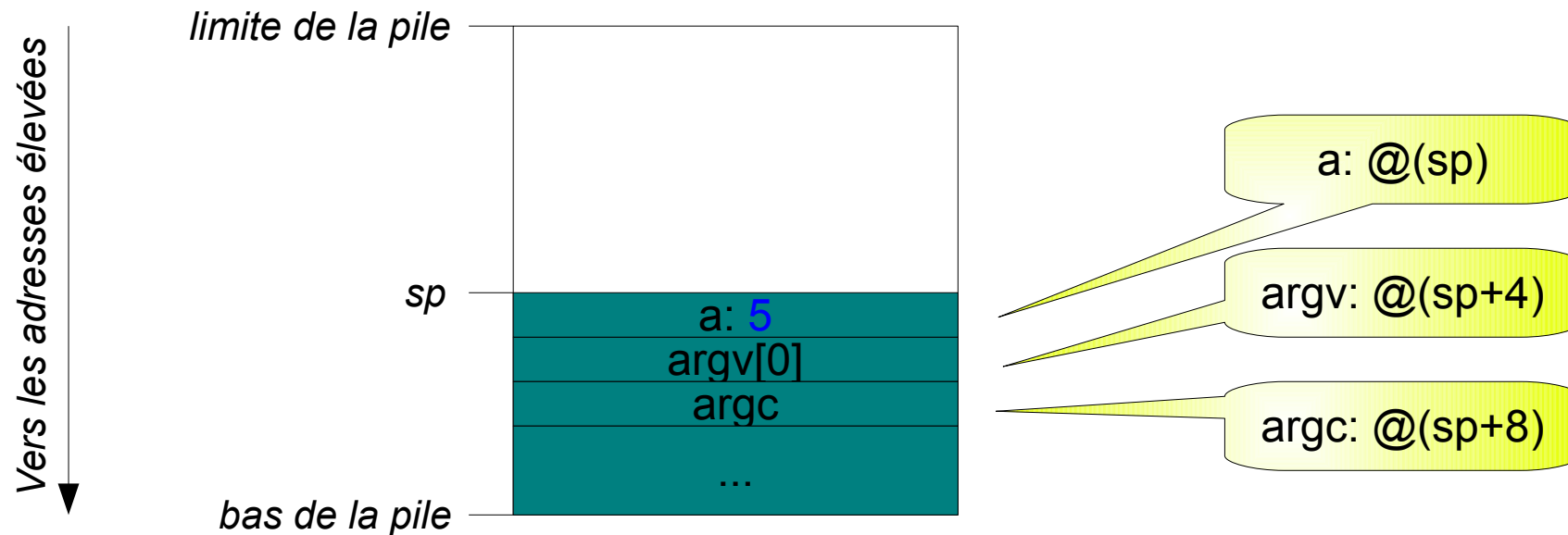
```
int main(int argc, char * argv[]) {  
  
}
```



# Segment de pile

- Variables locales

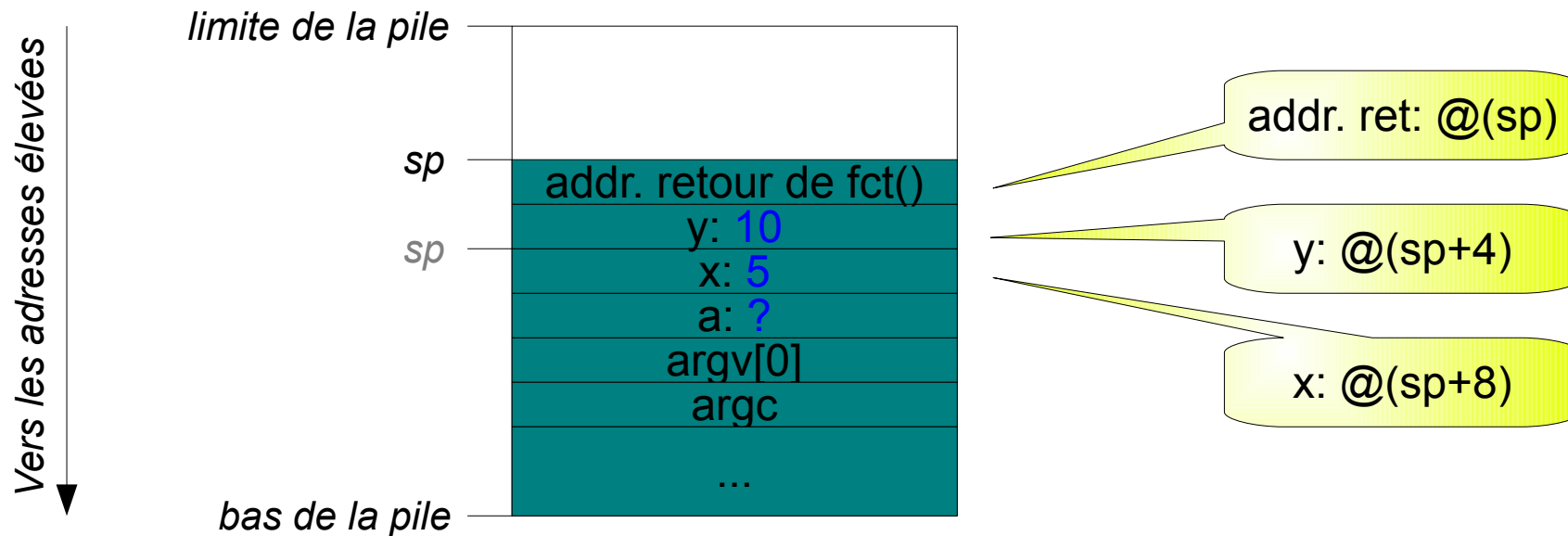
```
int main(int argc, char * argv[]) {  
    int a= 5;  
}
```



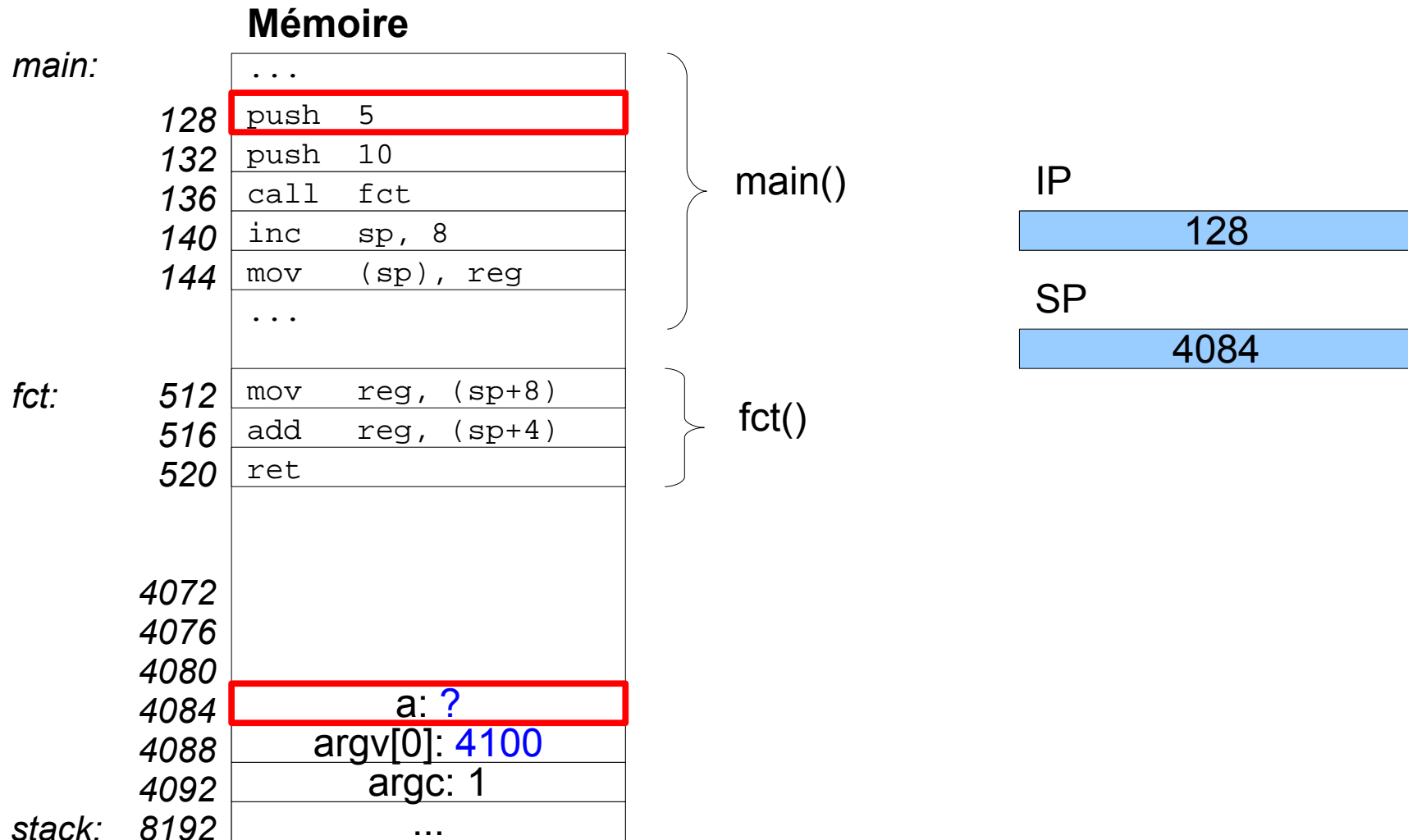
# Segment de pile

- Adresse de retour

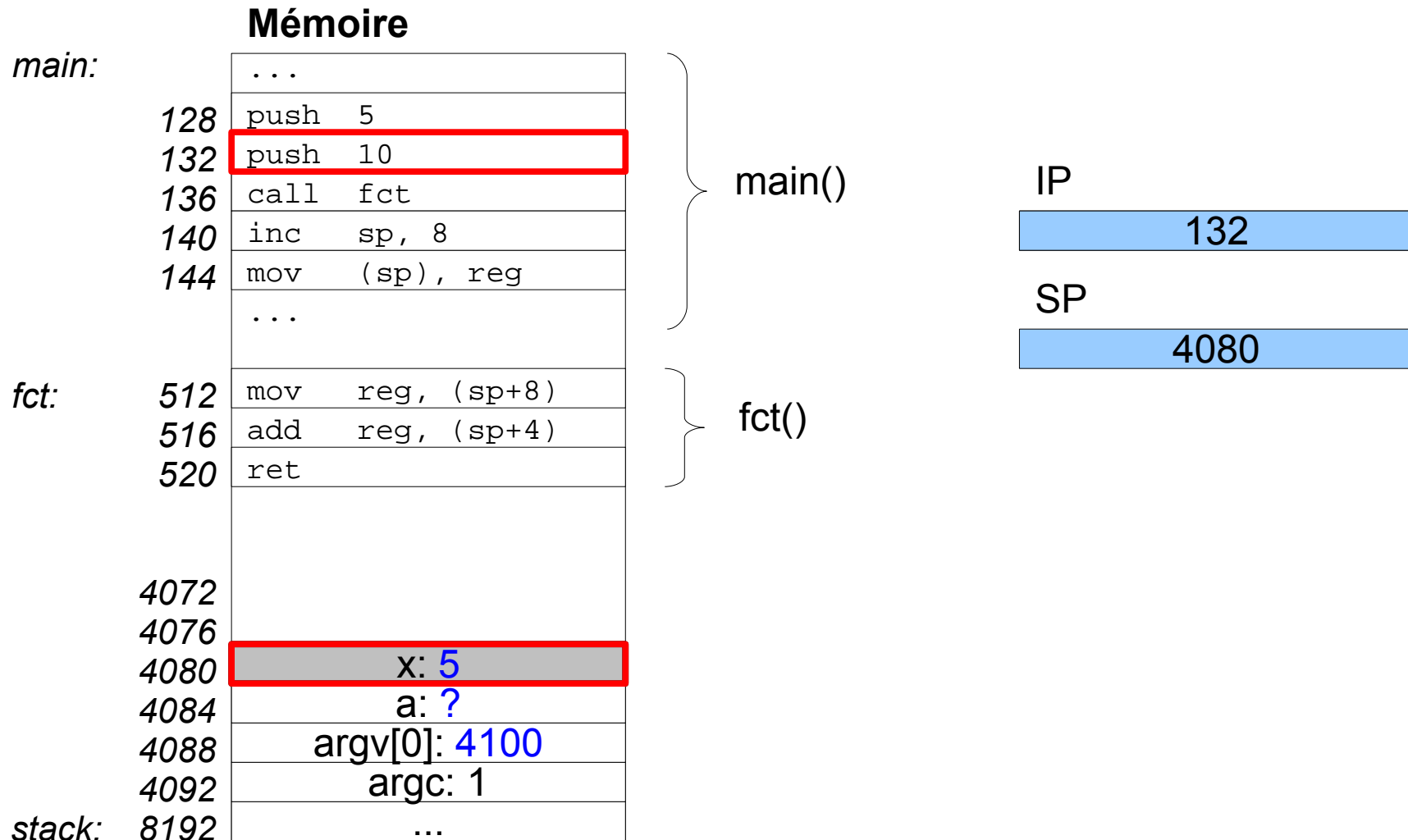
```
int fct(int x, int y) {  
    return x+y;  
}  
int main(int argc, char * argv[]) {  
    int a= fct(5, 10);  
}
```



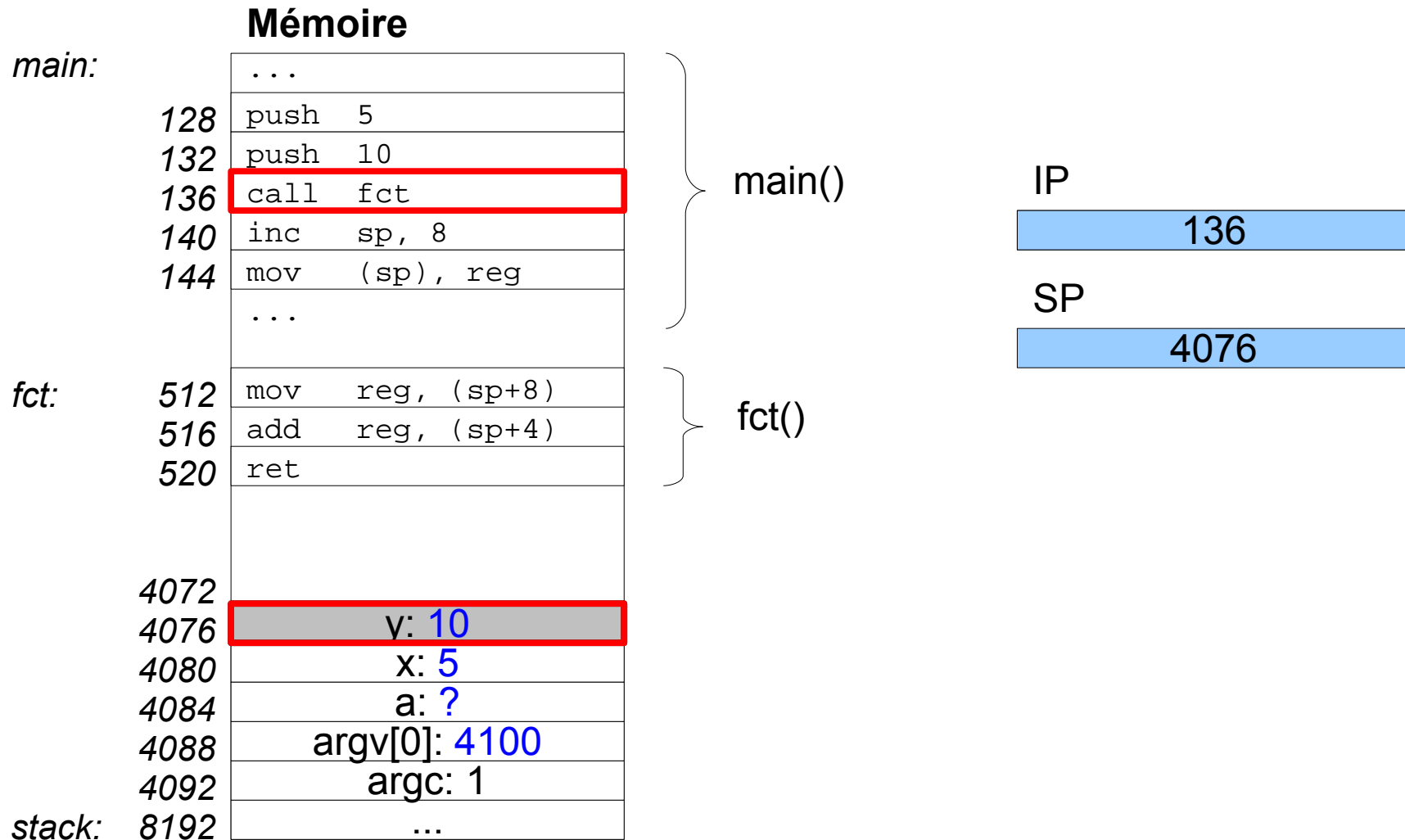
# Exécution d'un programme



# Exécution d'un programme

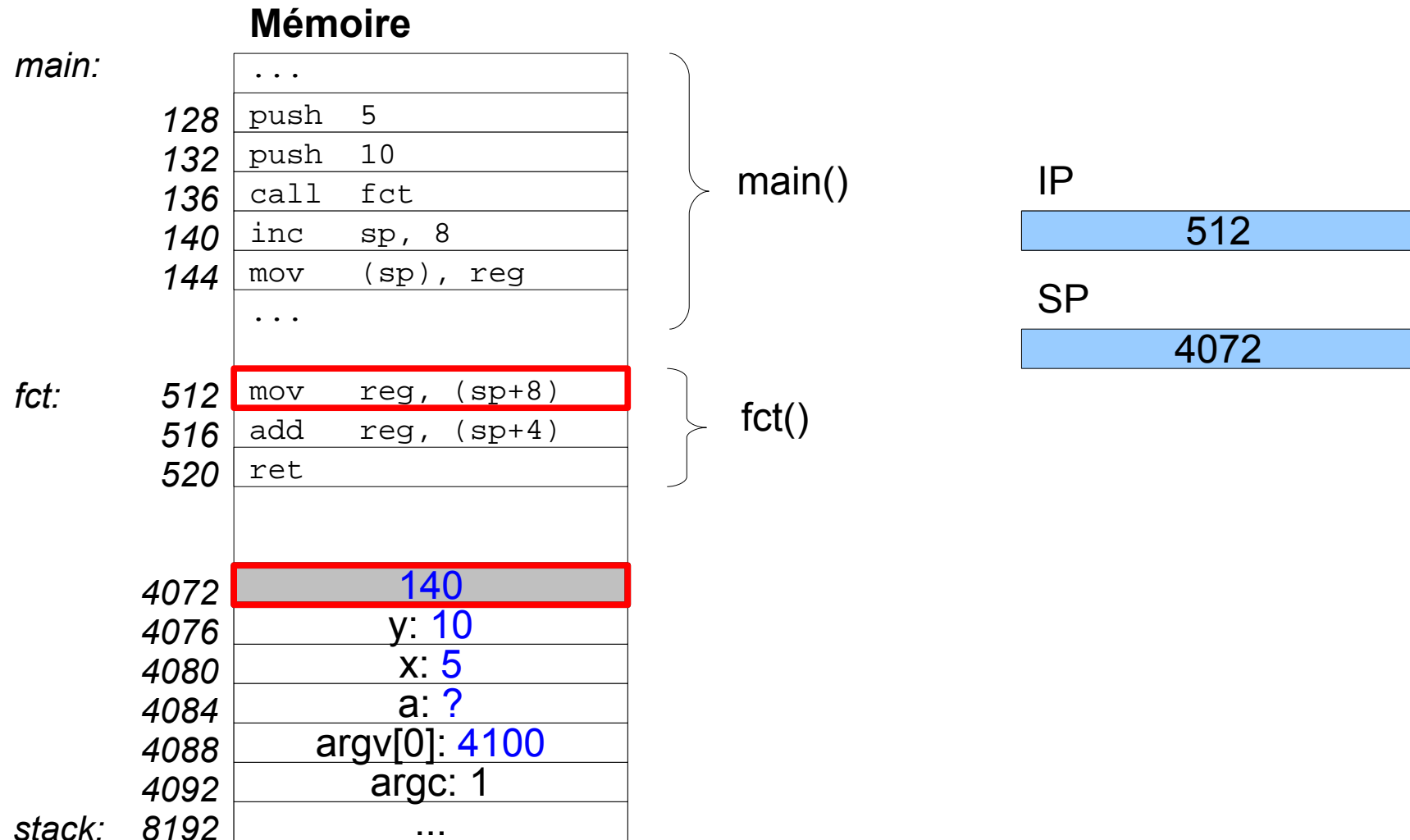


# Exécution d'un programme

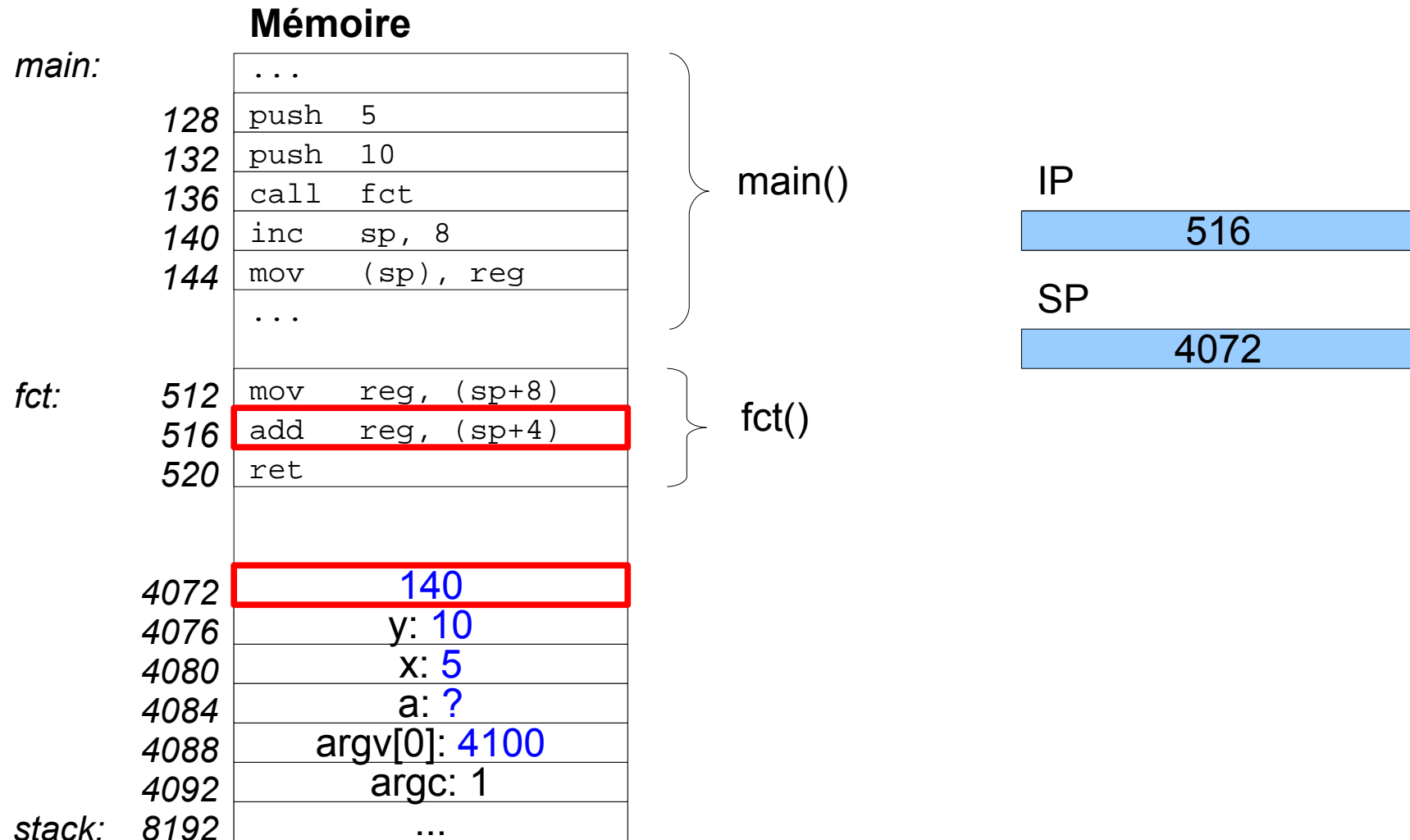




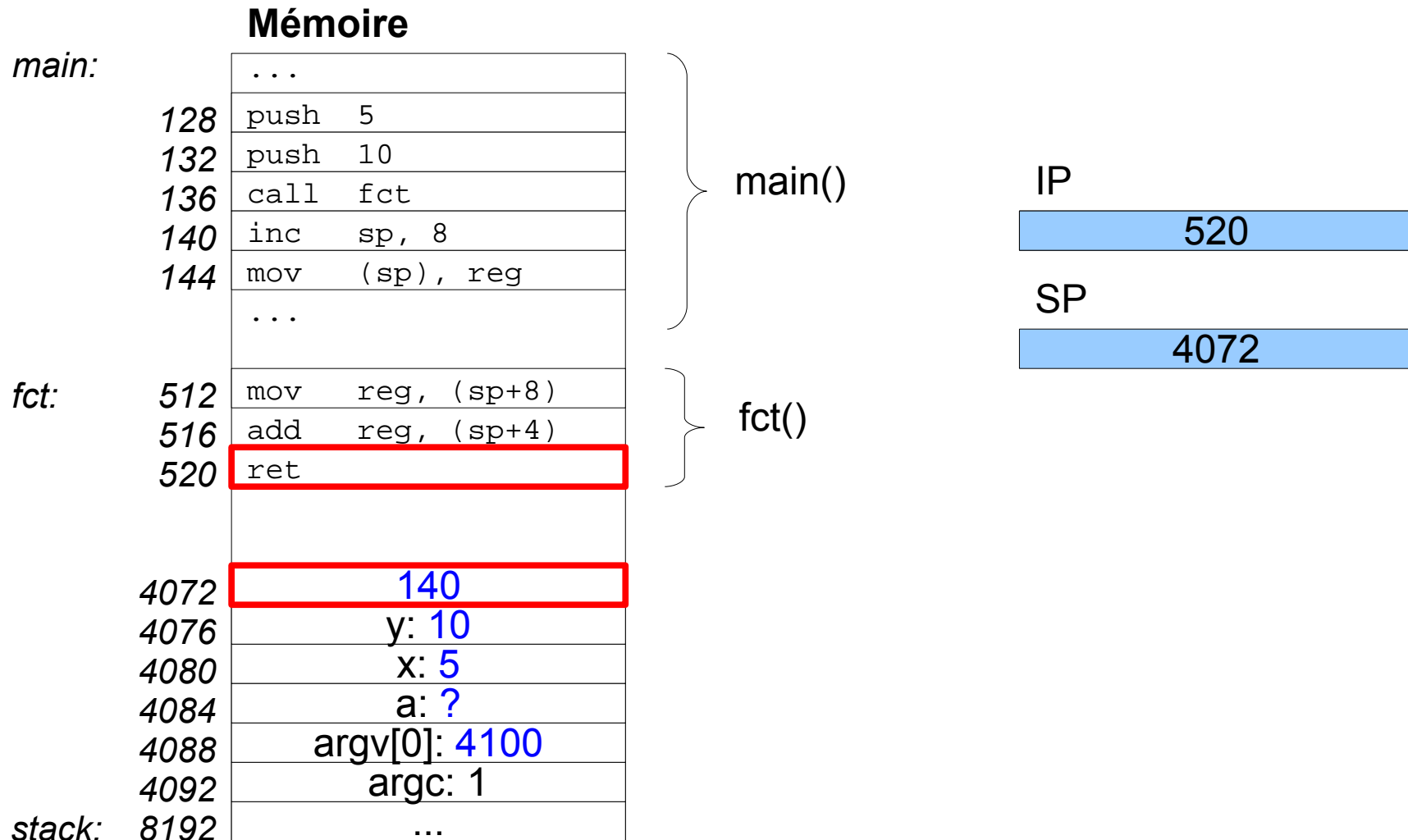
# Exécution d'un programme



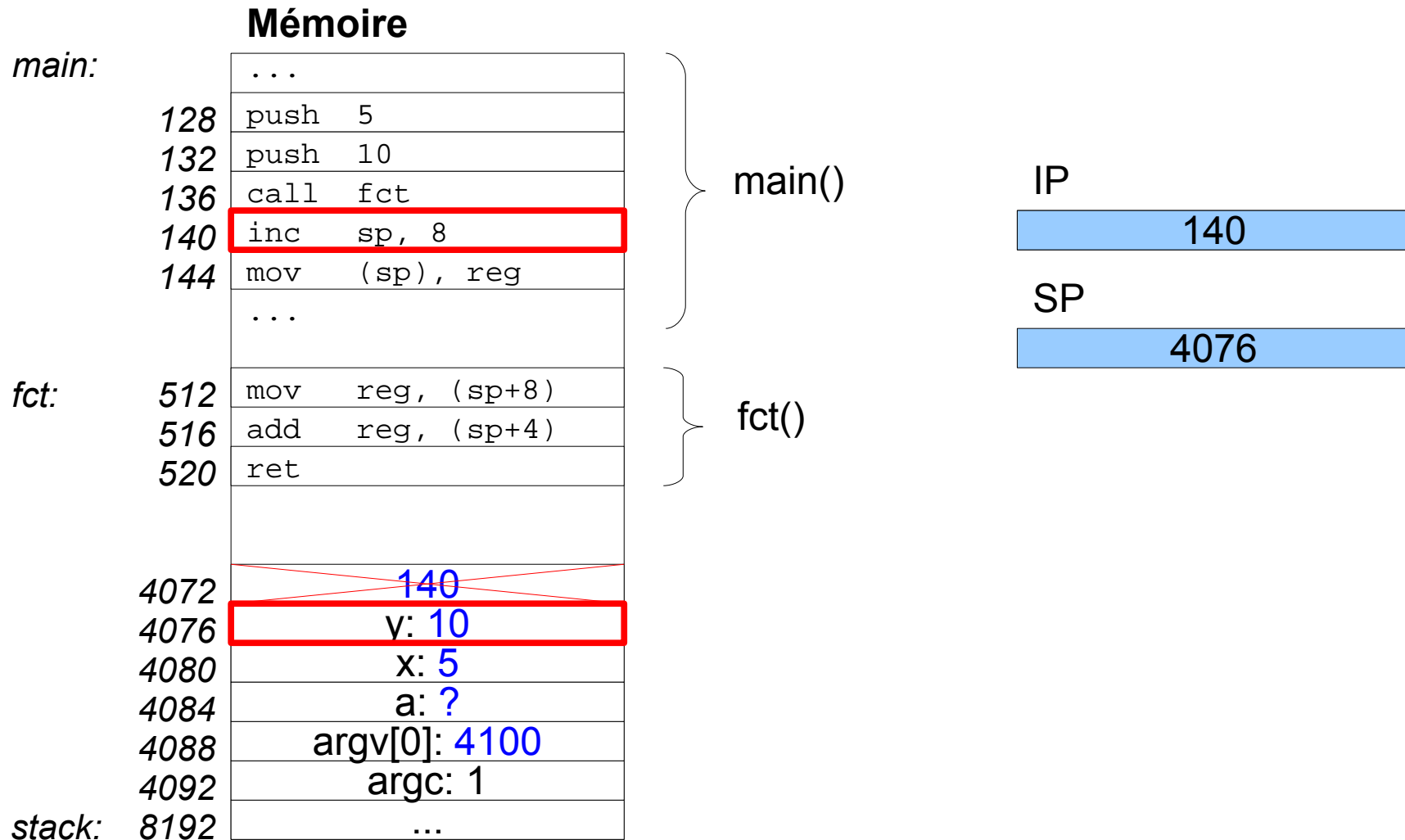
# Exécution d'un programme



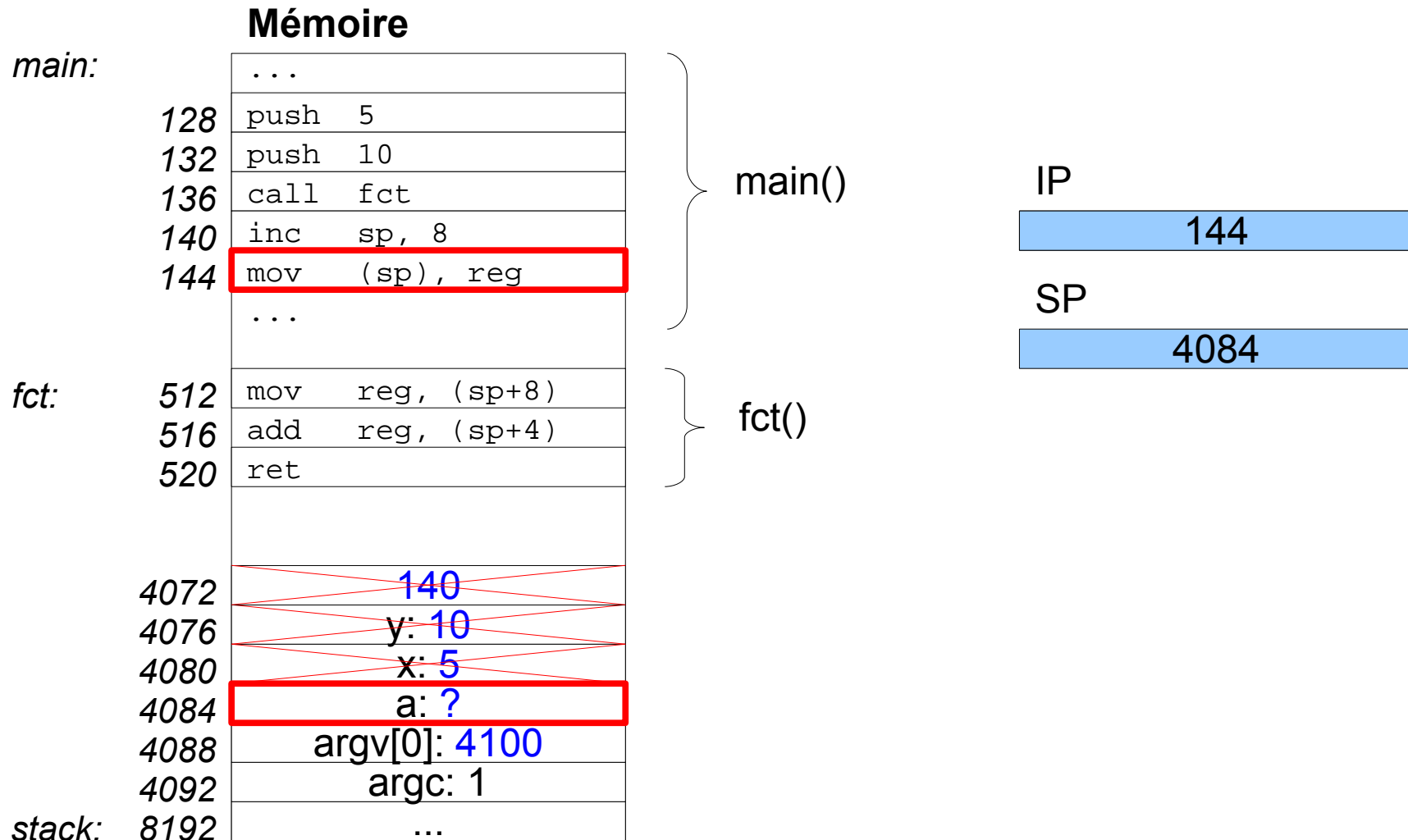
# Exécution d'un programme



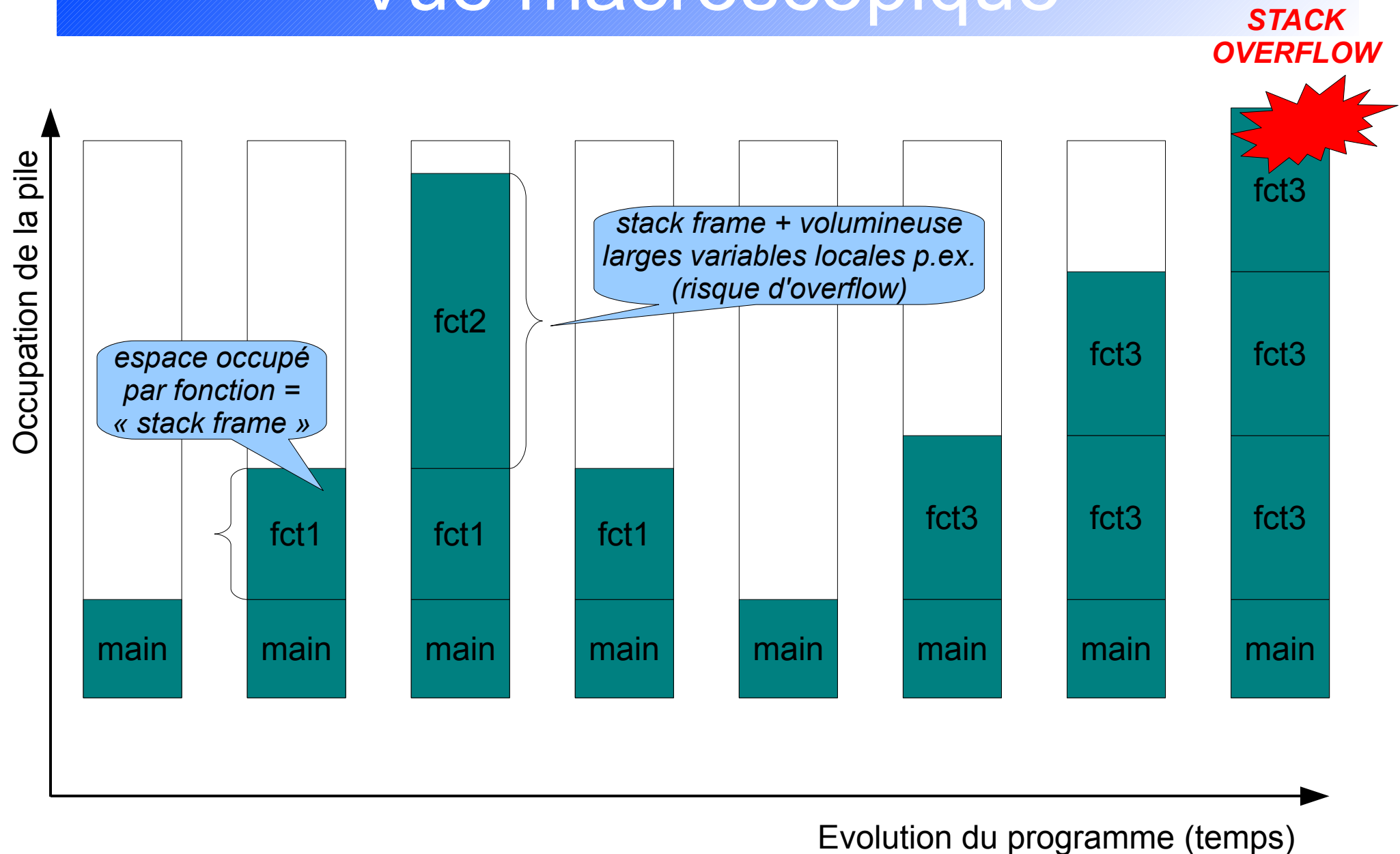
# Exécution d'un programme



# Exécution d'un programme



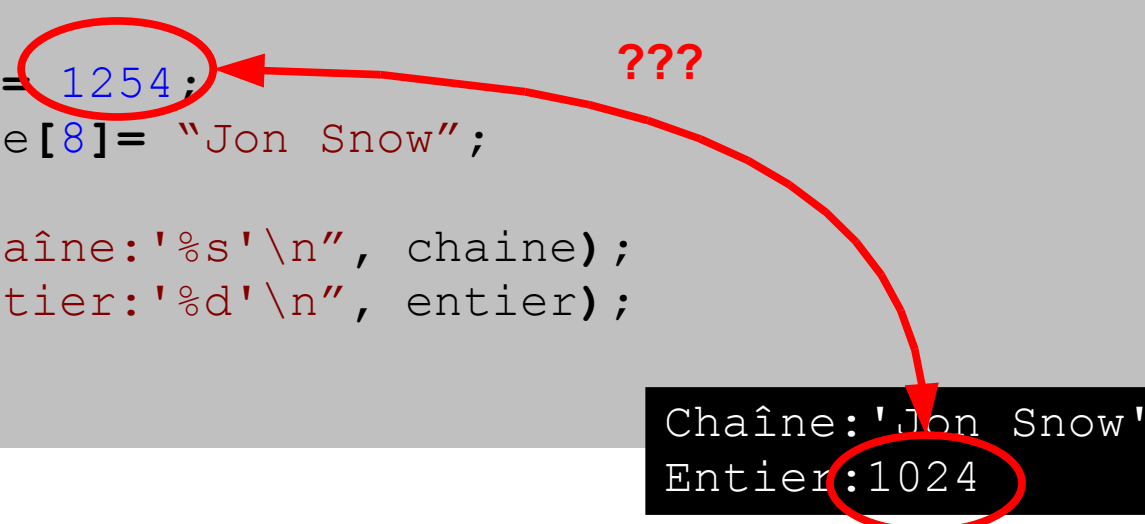
# Vue macroscopique



# Buffer overflow

```
int main()
{
    int entier= 1254;
    char chaine[8]= "Jon Snow";

    printf("Chaîne: '%s'\n", chaine);
    printf("Entier: '%d'\n", entier);
    return 0;
}
```



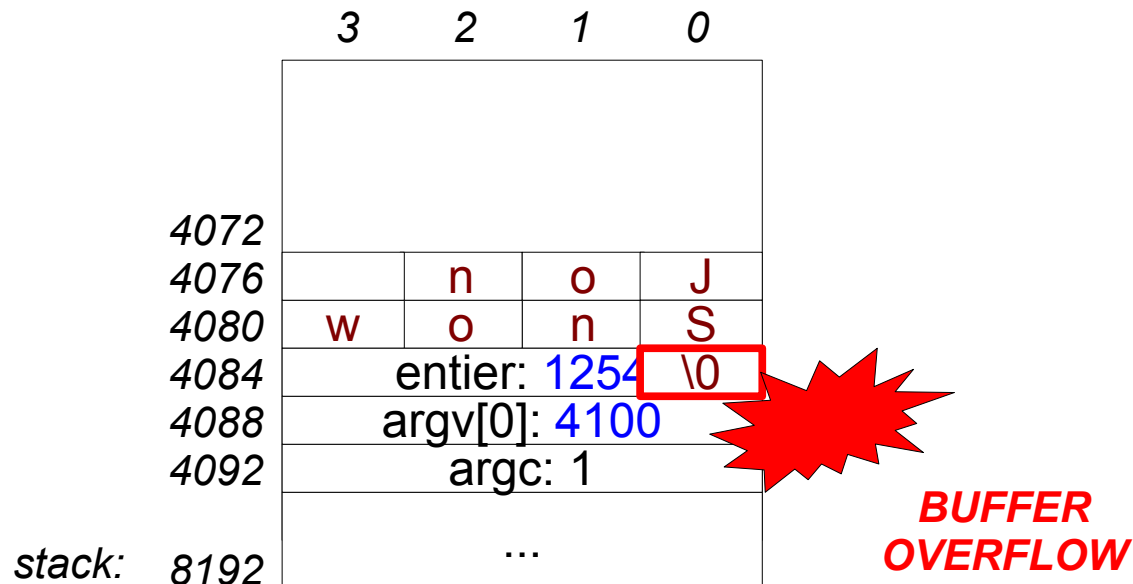
Chaîne: 'Jon Snow'  
Entier: 1024

# Buffer overflow

```
int main()
{
    int entier= 1254;
    char chaine[8]= "Jon Snow";

    printf("Chaîne: '%s'\n", chaine);
    printf("Entier: '%d'\n", entier);
    return 0;
}
```

Chaîne: 'Jon Snow'  
Entier: 1024





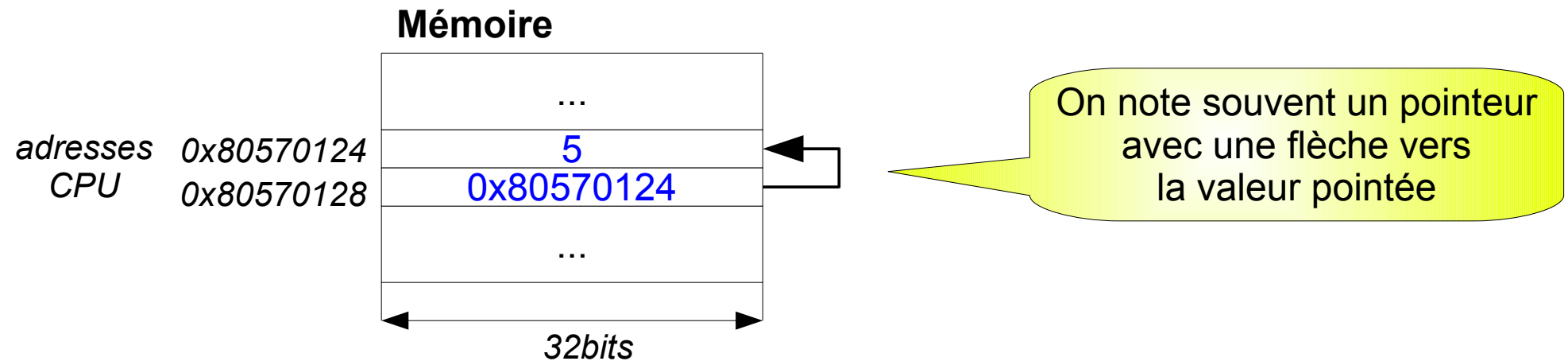
# Segment de pile

- Résumé
  - Utilité de la pile
    - passer des arguments
    - stocker le contenu de variables locales
    - sauvegarder l'adresse de retour des fonctions
  - Données « dépilées » peuvent être écrasées dans la suite du programme (ne plus les utiliser)
  - Risque de stack overflow
    - trop d'appels de fonctions en chaîne
    - variables locales trop grosses

# Les Pointeurs

# Pointeurs

- Définition
  - Variable dont le contenu est une adresse



# Pointeurs

- Déclaration

```
type * identifiant;
```

L'astérisque (\*) est utilisée pour la déclaration d'un pointeur et pour le déréférencement.

- Exemple

```
int var= 5;  
  
int * ptr_var= &var;  
  
printf("Adresse de 'var':%p\n", ptr_var);  
printf("Contenu de 'var':%d\n", var);  
printf("Contenu de 'var':%d\n", *ptr_var);
```

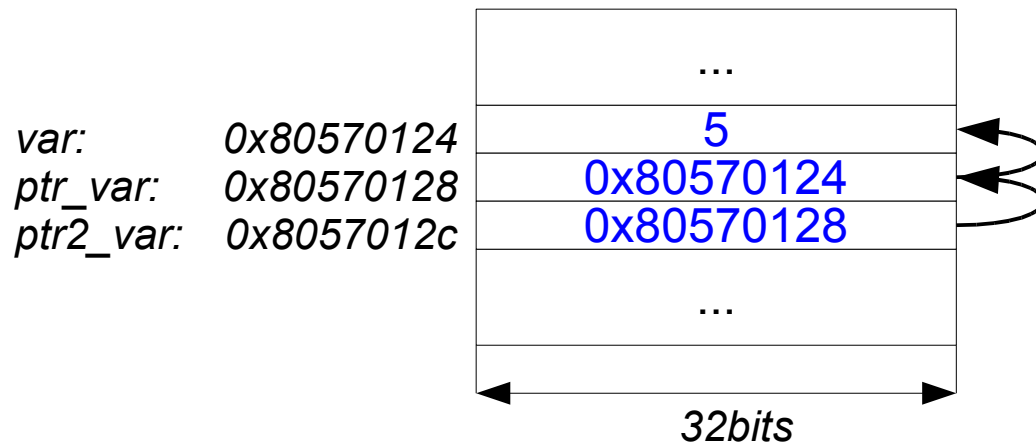
```
Adresse de 'var':0x80570124  
Contenu de 'var':5  
Contenu de 'var':5
```

# Pointeurs

- Pointeur de pointeur de pointeur de ...

```
int var= 5;  
int * ptr_var= &var;  
int ** ptr2_var= &ptr_var;
```

## Mémoire



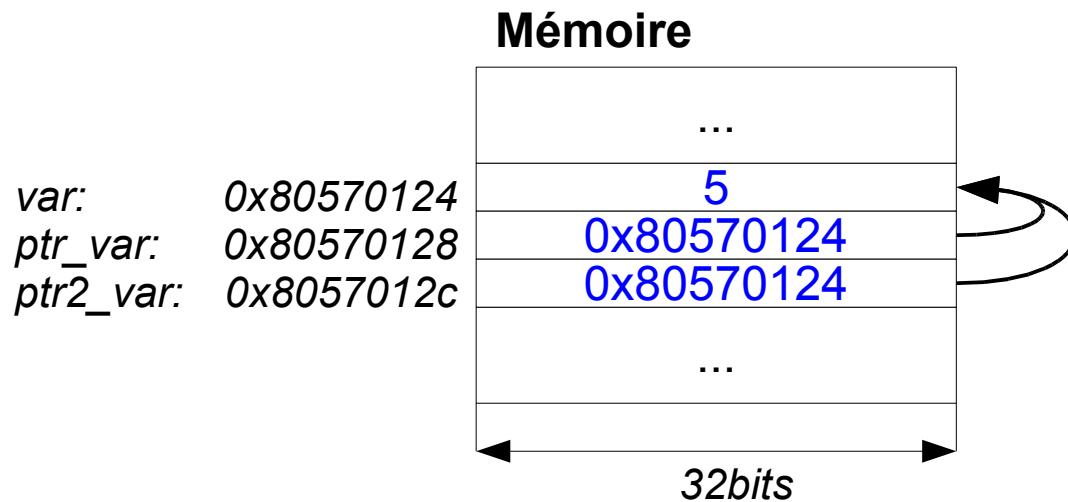
```
var= 10;  
*ptr_var= 10;  
**ptr2_var= 10;
```

Ces 3 expressions  
assignent la valeur 10  
à la variable 'var'.

# Pointeurs

- Aliasing

```
int var= 5;  
int * ptr_var= &var;  
int * ptr2_var= &var;
```

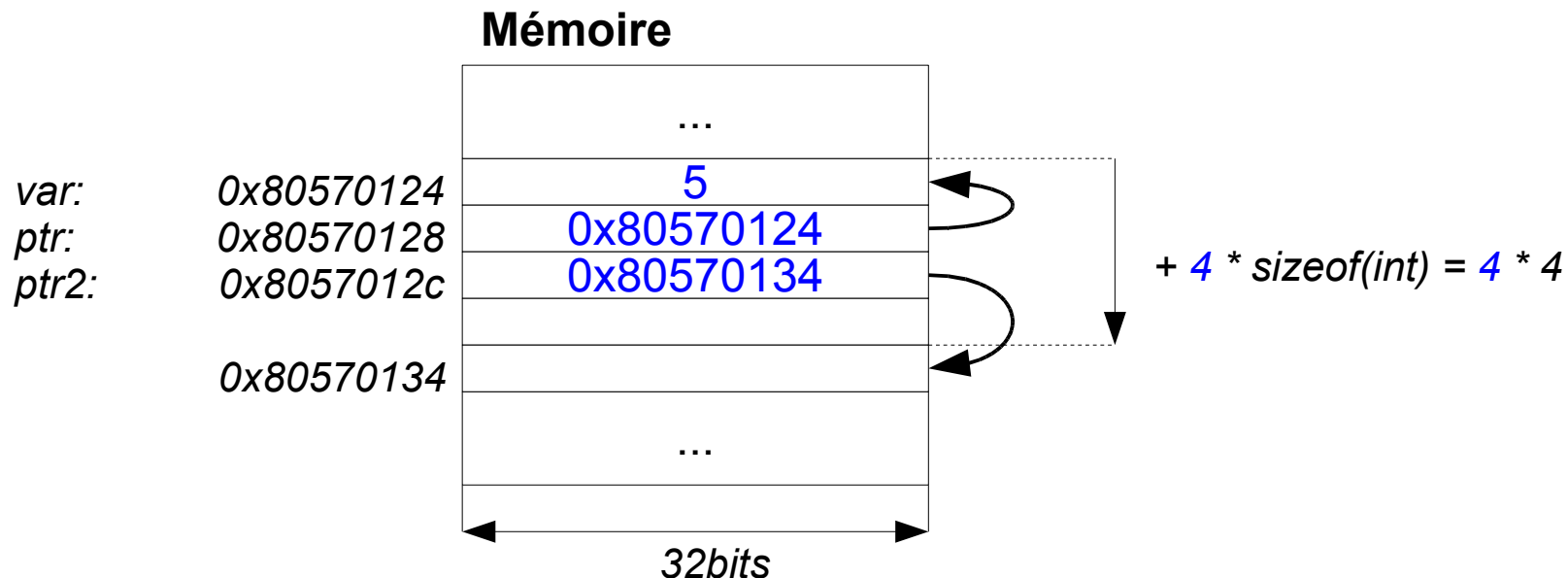


Si on modifie `*ptr_var`,  
alors `*ptr2_var` est  
aussi modifié !

# Pointeurs

- Arithmétique des pointeurs

```
int var= 5;  
int * ptr= ...;  
int * ptr2= ptr + 4;
```



```
Pointeur' = Pointeur + (offset * sizeof(type))
```

# Pointeurs

- Valeur spéciale **NULL**
  - marquer des pointeurs invalides
  - parfois utilisée pour retourner des erreurs

```
#include <stdlib.h>

void * ptr= NULL;
```



# **Allocation dynamique de mémoire**

# Allocation dynamique

- Allocation de mémoire sur le tas (heap)
  - gérée par le programmeur (la pile est gérée par le compilateur)
  - grands blocs de mémoire (taille tas > taille pile)
  - gestion des erreurs
- Fonctions de la librairie C

```
#include <stdlib.h>

void * malloc(size_t size);

void * realloc(void * ptr, size_t size);

void free(void * ptr);
```

# Allocation dynamique

- Exemple

```
#include <stdlib.h>
#define TAB_SIZE 10000

int main()
{
    long * dyn_tab= (long *) malloc(TAB_SIZE*sizeof(long));
    unsigned int index;

    srand(2009);

    for (index= 0; index < TAB_SIZE; index++)
        dyn_tab= random();

    free(dyn_tab);
}
```

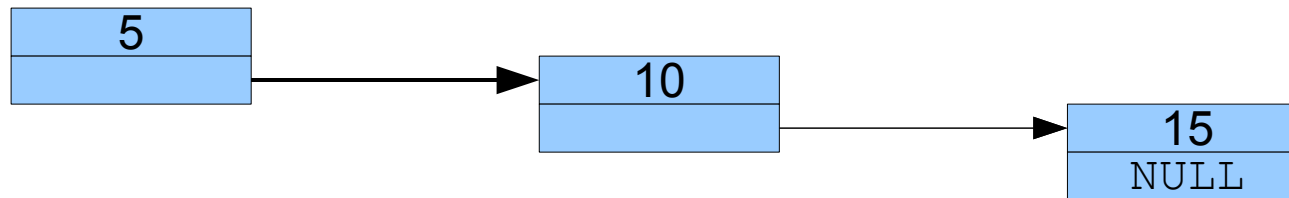
# Allocation dynamique

- Gestion des erreurs

```
void * ptr= malloc(...*sizeof(...));  
if (ptr == NULL) {  
    fprintf(stderr, "ERREUR: allocation de mémoire impossible\n");  
    exit(EXIT_FAILURE);  
}
```

# Pointeurs

- Exemple d'utilisation (liste chaînée)



- Définition d'un nœud

```
typedef struct type_noeud
{
    int          valeur;
    struct type_noeud * suivant;
} type_noeud;
```

# Pointeurs

- Exemple d'utilisation (liste chaînée)

```
void insertion_noeud(type_noeud * base, int valeur)
{
    type_noeud * nouveau=
        (type_noeud *) malloc(sizeof(type_noeud));
    nouveau->valeur= valeur;

    while ((base->suivant != NULL) &&
           (base->suivant->valeur < valeur))
        base= base->suivant;

    nouveau->suivant= base->suivant;
    base->suivant= nouveau;
}
```

# Gestion de la mémoire

## **Quand utiliser la pile ? Quand utiliser le tas ?**

- Pile
  - variables locales de petite taille
  - création/destruction plus rapide que sur le tas
- Tas
  - blocs de grande taille
  - blocs dont la taille n'est pas connue à l'avance
  - structures de données qui évoluent avec le temps

# FIN

## Questions ?



# Références

- **The C Programming Language (2<sup>nd</sup> edition)**, B. Kernighan and D. Ritchie, Addison-Wesley, 1988.
- **The System V Interface Definition (4<sup>th</sup> edition)**, SCO, <http://www.sco.com/developers/devspecs/>
- **Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching (3rd Edition)**, Robert Sedgewick, Addison-wesley, 1997.

# Remerciements

Merci à Van Nam Nguyen et Sébastien Barré pour leurs commentaires sur cette version du cours.