



Programmation Impérative 2

C dynamique

Licence 2 Informatique

Antoine Spicher

`antoine.spicher@u-pec.fr`

Plan du cours : C dynamique



- Modèle de la mémoire
- Pointeurs
- Tableaux
- Chaînes de caractères
- Allocation dynamique

Modèle de la mémoire

■ Lieu où s'exécute un programme

■ Organisation en quatre parties

□ Le **code** (*text* en anglais)

Zone mémoire où se trouve la suite des *instructions* (i.e., assembleur) des fonctions d'un programme

□ Les **données statiques** (*data* en anglais)

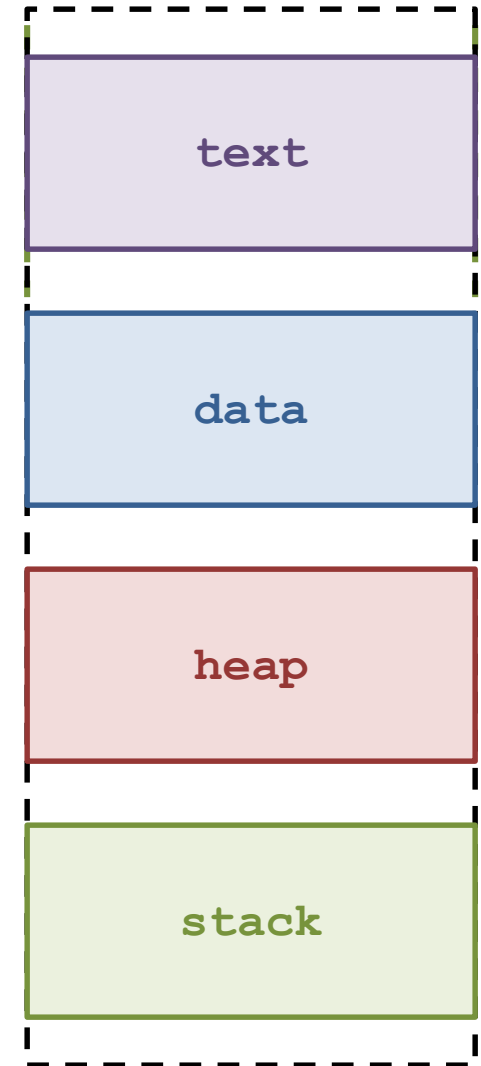
Zone mémoire où se trouve les *données statiques/globales* (i.e., permanentes et de taille fixe)

□ Le **tas** (*heap* en anglais)

Zone mémoire où se trouve les *données dynamiques* (i.e., allouées à l'exécution)

□ La **pile d'exécution** (*stack* en anglais)

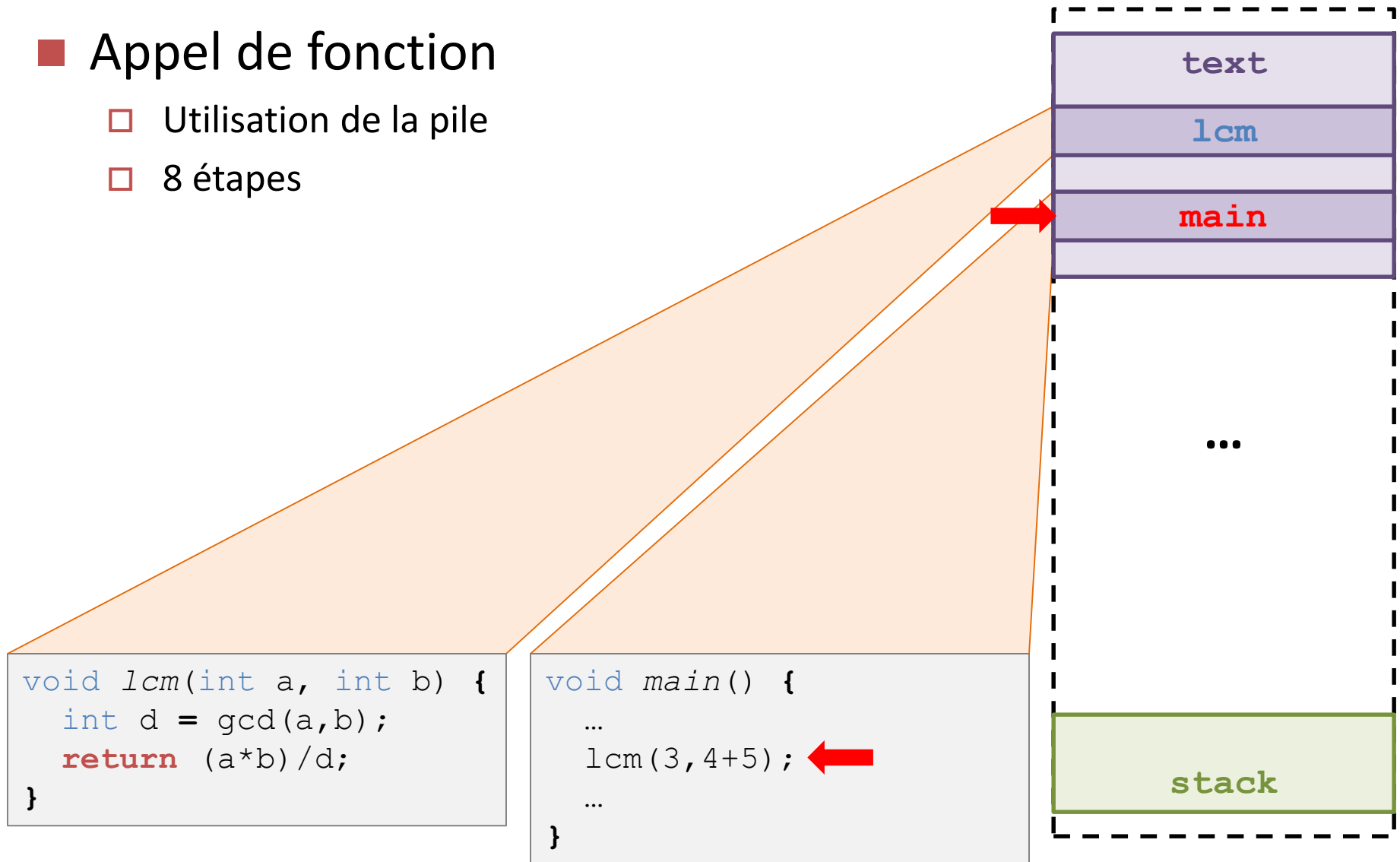
Zone mémoire utilisée pour stocker les *variables locales* (i.e., temporaires) et les *appels de fonctions* (i.e., arguments et adresse de retour)



Modèle de la mémoire

■ Appel de fonction

- Utilisation de la pile
- 8 étapes



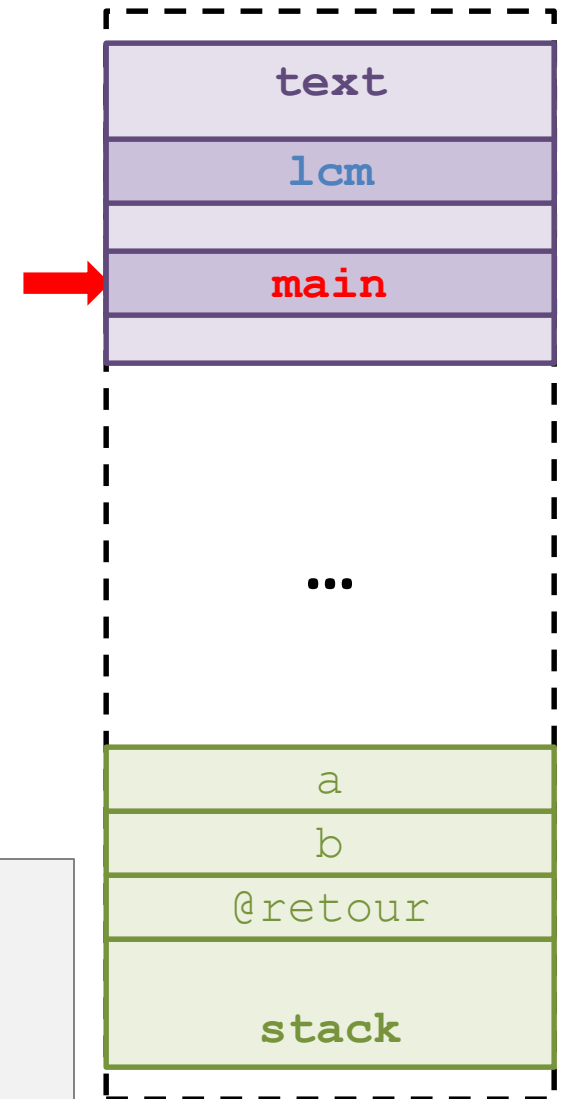
Modèle de la mémoire

■ Appel de fonction

- Utilisation de la pile
- 8 étapes
 - Allocation de la pile pour l'appel

```
void lcm(int a, int b) {  
    int d = gcd(a,b);  
    return (a*b)/d;  
}
```

```
void main() {  
    ...  
    lcm(3,4+5);  
    ...  
}
```



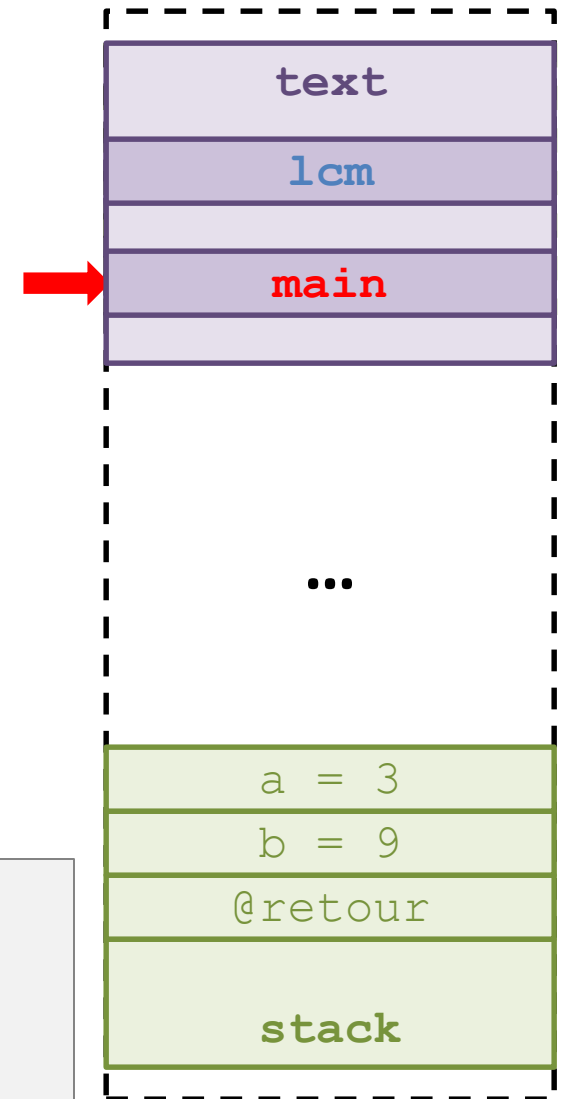
Modèle de la mémoire

■ Appel de fonction

- Utilisation de la pile
- 8 étapes
 - Allocation de la pile pour l'appel
 - Initialisation des arguments dans la pile

```
void lcm(int a, int b) {  
    int d = gcd(a,b);  
    return (a*b)/d;  
}
```

```
void main() {  
    ...  
    lcm(3, 4+5);  
    ...  
}
```




Modèle de la mémoire

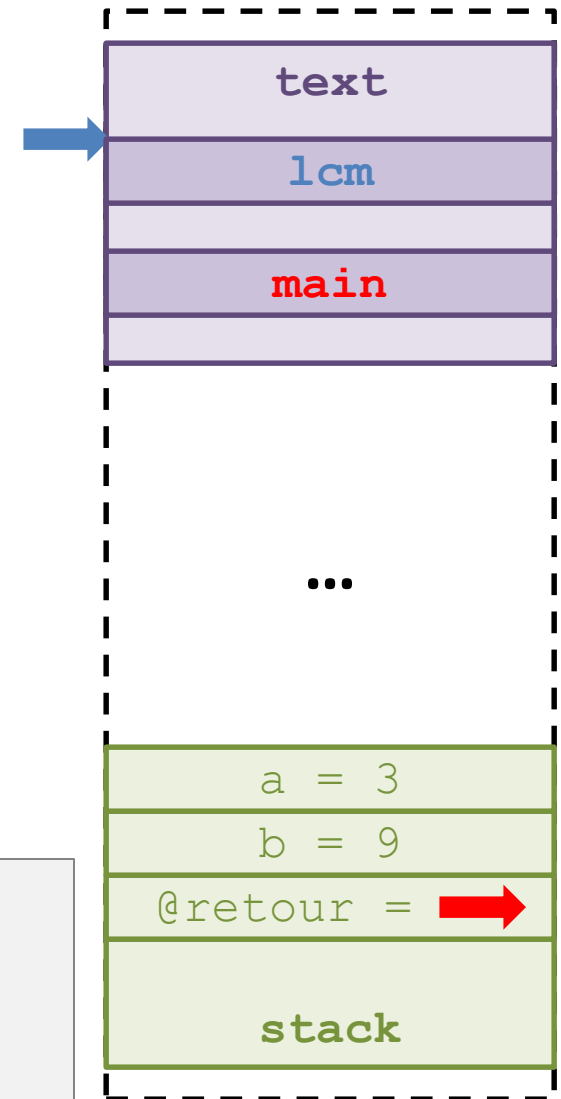
■ Appel de fonction

- Utilisation de la pile
- 8 étapes
 - Allocation de la pile pour l'appel
 - Initialisation des arguments dans la pile
 - Appel de la fonction (initialisation de l'@retour)

```
void lcm(int a, int b) {  
    int d = gcd(a,b);  
    return (a*b)/d;  
}
```



```
void main() {  
    ...  
    lcm(3, 4+5);  
    ...  
}
```




Modèle de la mémoire

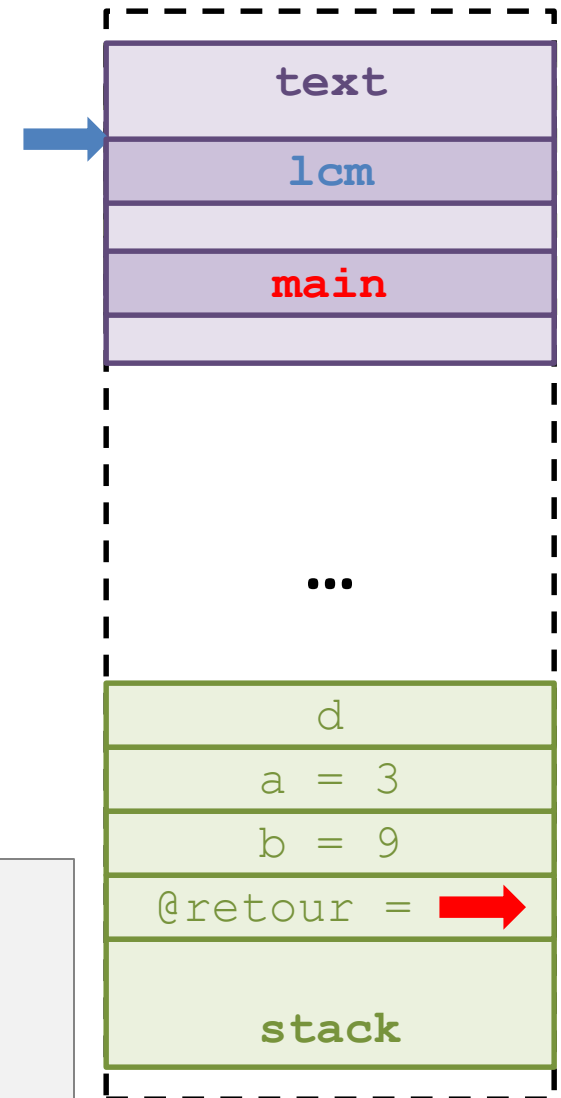
■ Appel de fonction

- Utilisation de la pile
- 8 étapes
 - Allocation de la pile pour l'appel
 - Initialisation des arguments dans la pile
 - Appel de la fonction (initialisation de l'@retour)
 - Allocation de la pile pour les variables locales

```
void lcm(int a, int b) {  
    int d = gcd(a,b);  
    return (a*b)/d;  
}
```



```
void main() {  
    ...  
    lcm(3, 4+5);  
    ...  
}
```




Modèle de la mémoire

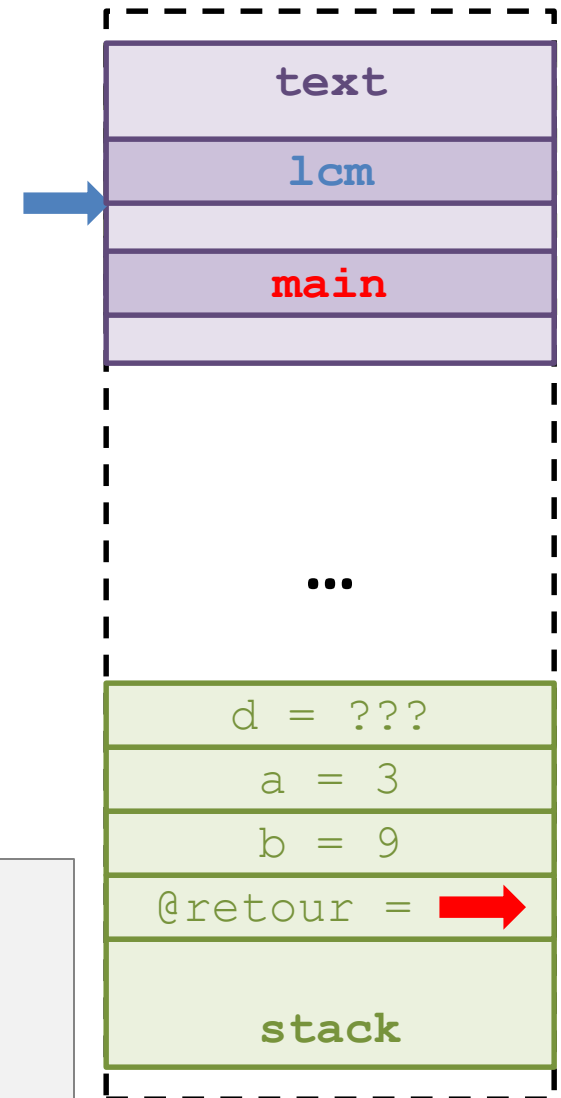
■ Appel de fonction

- Utilisation de la pile
- 8 étapes
 - Allocation de la pile pour l'appel
 - Initialisation des arguments dans la pile
 - Appel de la fonction (initialisation de l'@retour)
 - Allocation de la pile pour les variables locales
 - Exécution du code de la fonction

```
int lcm(int a, int b) {  
    int d = gcd(a,b);  
    return (a*b)/d;  
}
```



```
void main() {  
    ...  
    lcm(3, 4+5);  
    ...  
}
```




Modèle de la mémoire

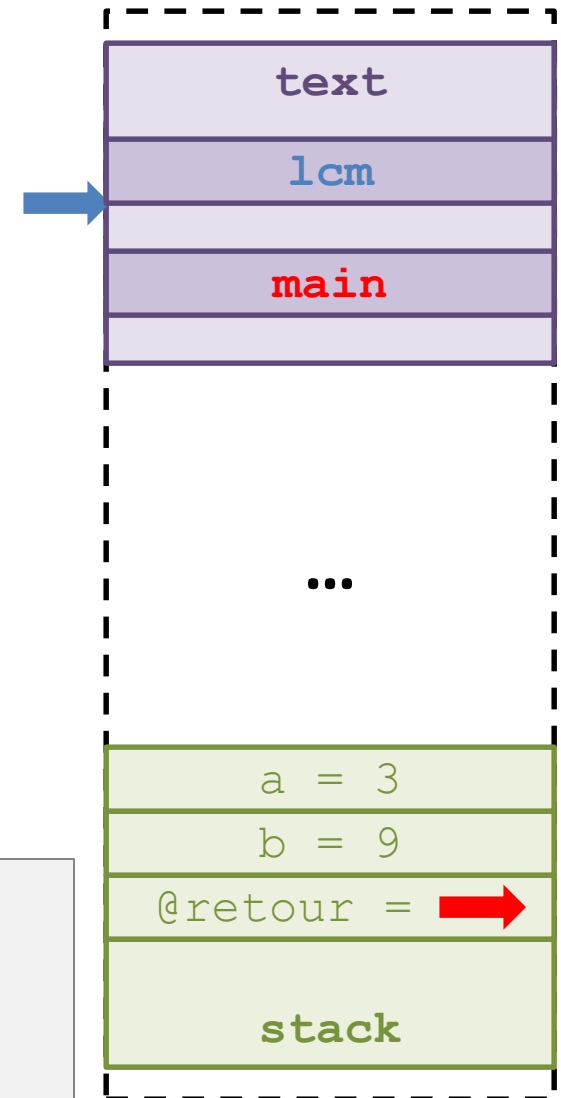
■ Appel de fonction

- Utilisation de la pile
- 8 étapes
 - Allocation de la pile pour l'appel
 - Initialisation des arguments dans la pile
 - Appel de la fonction (initialisation de l'@retour)
 - Allocation de la pile pour les variables locales
 - Exécution du code de la fonction
 - Désallocation des variables locales

```
void lcm(int a, int b) {  
    int d = gcd(a,b);  
    return (a*b)/d;  
}
```



```
void main() {  
    ...  
    lcm(3, 4+5);  
    ...  
}
```



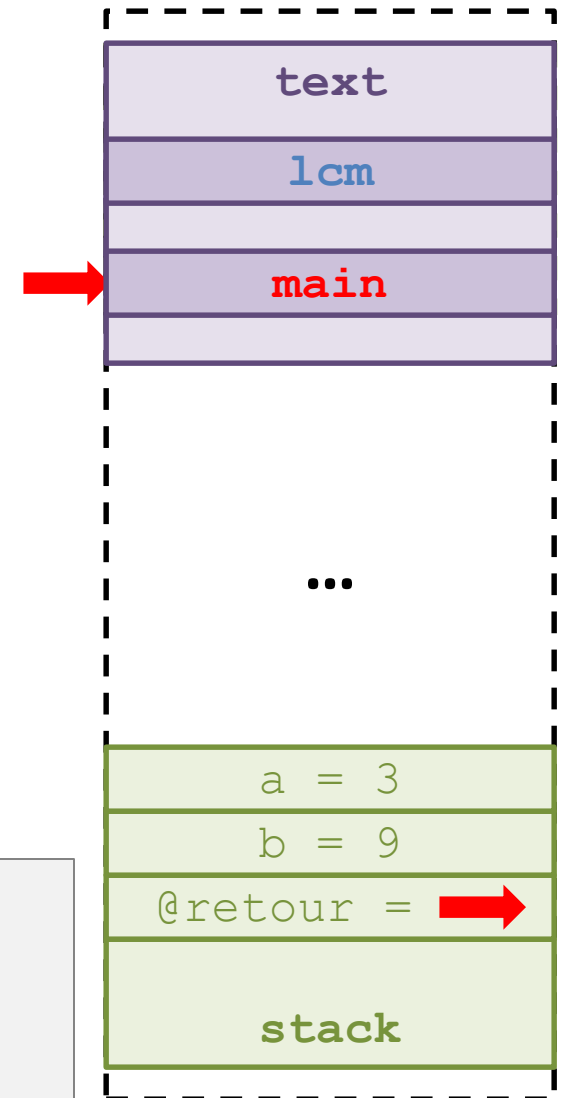
Modèle de la mémoire

■ Appel de fonction

- Utilisation de la pile
- 8 étapes
 - Allocation de la pile pour l'appel
 - Initialisation des arguments dans la pile
 - Appel de la fonction (initialisation de l'@retour)
 - Allocation de la pile pour les variables locales
 - Exécution du code de la fonction
 - Désallocation des variables locales
 - Retour à l'appelant (utilisation de l'@retour)

```
void lcm(int a, int b) {  
    int d = gcd(a,b);  
    return (a*b)/d;  
}
```

```
void main() {  
    ...  
    lcm(3, 4+5);  
    ...  
}
```



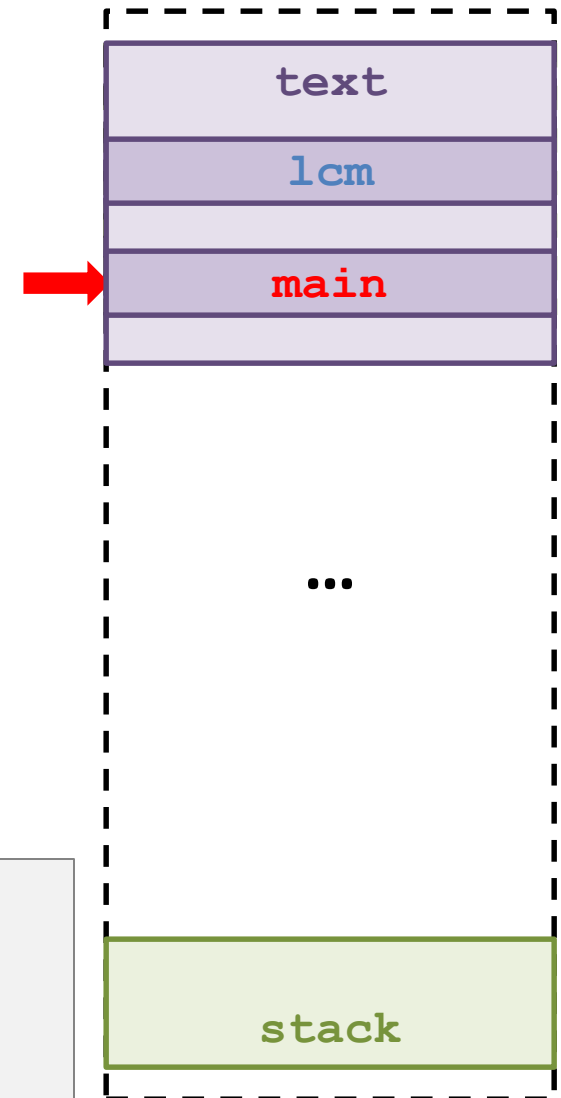
Modèle de la mémoire

■ Appel de fonction

- Utilisation de la pile
- 8 étapes
 - Allocation de la pile pour l'appel
 - Initialisation des arguments dans la pile
 - Appel de la fonction (initialisation de l'@retour)
 - Allocation de la pile pour les variables locales
 - Exécution du code de la fonction
 - Désallocation des variables locales
 - Retour à l'appelant (utilisation de l'@retour)
 - Désallocation des arguments et de l'@retour

```
void lcm(int a, int b) {  
    int d = gcd(a,b);  
    return (a*b)/d;  
}
```

```
void main() {  
    ...  
    lcm(3, 4+5);  
    ...  
}
```



Plan du cours : C dynamique



- Modèle de la mémoire
- Pointeurs
- Tableaux
- Chaînes de caractères
- Allocation dynamique

Pointeurs

■ Notion de variable

- Une *variable* **n'est pas** simplement un *nom pour une valeur*
- Une *variable* est un *emplacement mémoire où se trouve une valeur*
- Une *variable* a une taille (**sizeof**)
- Une *variable* possède une *adresse* ➡

```
type name = value;
```

sizeof(type) = nbr d'octets

```
int n = 3;
```

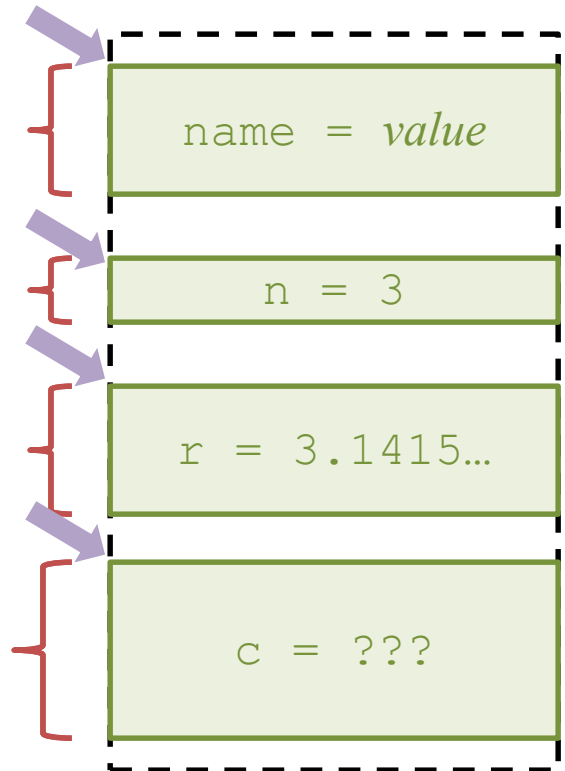
sizeof(int) = 4

```
double r = M_PI;
```

sizeof(double) = 8

```
tarot_card c;
```

sizeof(tarot_card) = ?



Pointeurs

■ Notion de pointeur

- Un *pointeur* représente une *adresse en mémoire*
- Un *pointeur* est une *valeur*
au même titre que 3, 4.5, 'a', "helloworld", etc.
- Un *pointeur* possède un *type*
le type d'un pointeur dépend du type la valeur pointée

■ Notation du type des pointeurs

Extension de la grammaire

```
<type> ::= 'int' | 'float' | 'double' | etc.  
        | ['enum' | 'struct' | 'union'] <name>  
        | <type> '*' | 'void' '*'
```



Pointeurs

■ Pointeurs et valeurs

- Les *pointeurs* sont des *entiers*

Une adresse prise au hasard a peu de chance de pointer vers une zone « autorisée » de la mémoire (e.g., l'accès à la valeur lève un *segmentation fault*)

```
float* r = 0x0493A94C;  
char* txt = "helloworld";  
printf("r=%p, txt=%p\n", r, txt); /* r=0x0493A94C, txt=0x080484F0 */
```

- La taille d'un *pointeur* est *fixe* (ne dépend pas de la valeur pointée)

```
printf("%d, %d, %d\n", sizeof(char*), sizeof(int*), sizeof(float*));  
/* 4, 4, 4 */
```

- Le pointeur `NULL`

- Ne pointe vers aucune valeur
- Valeur référence

```
printf("NULL = %p\n", NULL);  
/* r=(nil) */
```

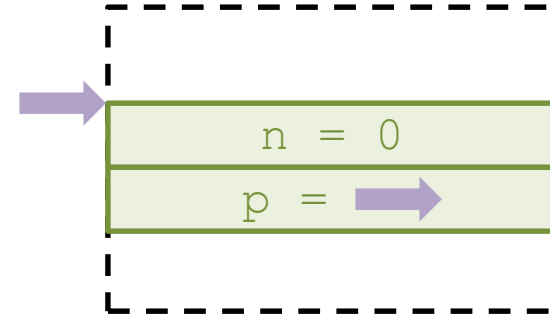
- *Toujours initialiser un pointeur avec `NULL`*
- *Quand une variable de type pointeur est `NULL`, elle n'est pas initialisée*

Pointeurs

■ Pointeurs et variables

- Accéder à l'adresse d'une variable : **&**

```
int  n = 0;
int* p = &n;
```



Attention **&** (**&n**) est *illégal*

&n est un *pointeur*, donc une *valeur*, donc *n'est pas une variable*

- Accéder à la *variable pointée* : *****

```
int  n = 0;
int* p = &n;
printf("%d, %d\n", n, *p); /* 0, 0 */
n = 1;
printf("%d, %d\n", n, *p); /* 1, 1 */
*p = 2;
printf("%d, %d\n", n, *p); /* 2, 2 */
```

Attention ***** (***n**) est *légal* si **n** est une variable de type *pointeur de pointeur*

Pointeurs

- L'opérateur `*` des types et `*` des variables coïncident

```
char** x;
```

≡

```
char* *x;
```

≡

```
char **x;
```

`x` est une variable de type
pointeur de pointeur vers un char

`*x` est une variable de type
pointeur vers un char

`**x` est une variable de type
char

Exercice : quel est le type de `p` ?

```
int x = * (* (* (* (& (* (&p) ) ) ) ) ) ;
```

Pointeurs

■ Pointeurs de fonction

- Une *fonction* est *suite d'instructions* dans la zone mémoire **text**
- Une *fonction* possède une *adresse en mémoire*
celle de la première instruction à exécuter
- Les pointeurs permettent la *manipulation de variables de fonctions*

```
int succ(int i) {  
    return i+1;  
}
```

```
int odd(int i) {  
    return i%2;  
}
```

```
int main() {  
    int (*f)(int) = NULL;  
    f = succ;  
    printf("%d\n", f(3)); /* 4 */  
    f = odd;  
    printf("%d\n", f(3)); /* 1 */  
    return 0;  
}
```

Plan du cours : C dynamique



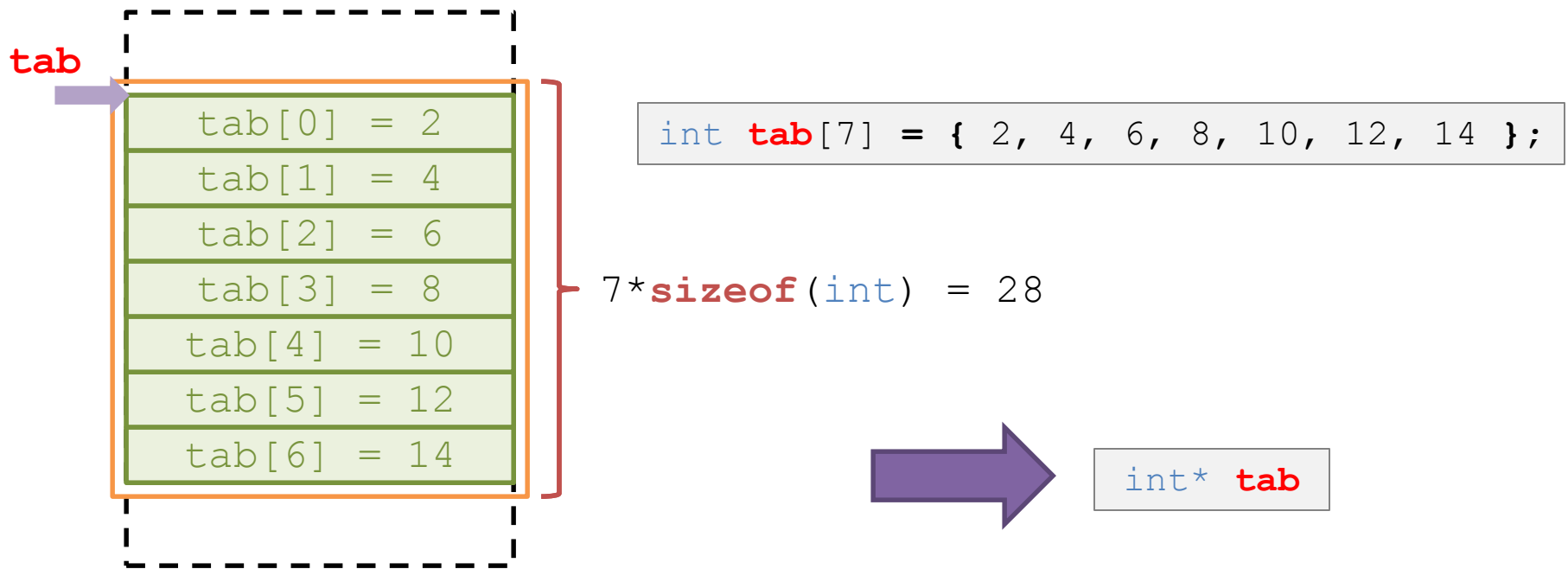
- Modèle de la mémoire
- Pointeurs
- Tableaux
- Chaînes de caractères
- Allocation dynamique

Tableaux

■ Représentation mémoire

- Zone contigüe de mémoire : séquence des éléments dans l'ordre
- Taille de la zone : nombre d'éléments = **sizeof**(type des éléments)
- *Un tableau est un pointeur* vers la zone mémoire

Il s'agit donc du pointeur vers le 1^{er} élément



Tableaux

■ Interprétations d'un type

```
char** x;
```

≡

```
char* *x;
```

≡

```
char **x;
```

x est une variable de type
pointeur de pointeur vers un char

*x est une variable de type
pointeur vers un char

**x est une variable de type
char

|||

x est une variable de type
tableau de pointeurs vers un char

*x est une variable de type
tableau de char

|||

|||

x est une variable de type
pointeur vers un tableau de char

|||

x est une variable de type
tableau de tableaux de char

7 interprétations possibles

Tableaux

■ Arithmétique des pointeurs

- Équivalence entre l'utilisation des tableaux et des pointeurs

```
tab[i];
```

≡

```
(tab+i)[0];
```

≡

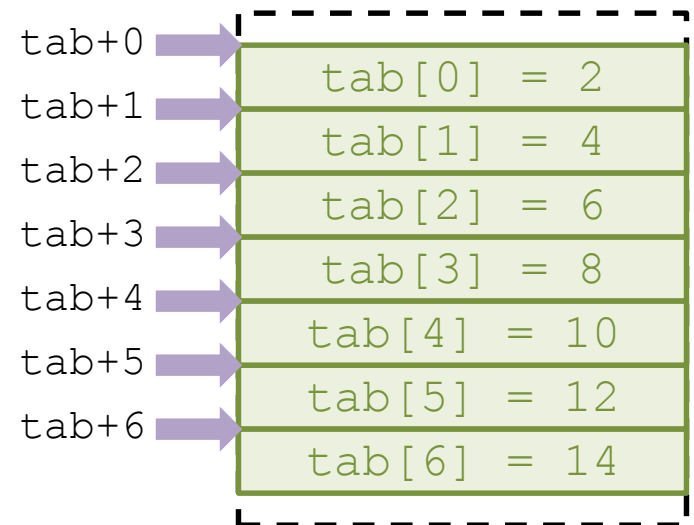
```
*(tab+i);
```

- *Comment C connaît le nombre d'octets à ajouter à l'adresse ?*

Utilisation du typage : les éléments sont des entiers

$\text{tab} + i : @ \text{ du tableau} + i \times \text{sizeof}(\text{int})$

```
int tab[7] = { ... };
```



Tableaux

■ Typage et arithmétique des pointeurs

- Utilisation de la *conversion de type explicite* (ou *cast*)

```
struct st { int a; int b; };
```

```
printf("%d = 2*%d\n", sizeof(struct st), sizeof(int));  
/* 8 = 2*4 */
```

```
struct st t[3]; int i;  
  
for(i=0; i<3; i++) {  
    t[i].a=i; t[i].b=100+i;  
    printf("t[i].a=%d, t[i].b=%d\n", t[i].a, t[i].b);  
}
```


Tableaux

■ Typage et arithmétique des pointeurs

- Utilisation de la *conversion de type explicite* (ou *cast*)

```
struct st { int a; int b; };
```

```
printf("%d = 2*%d\n", sizeof(struct st), sizeof(int));  
/* 8 = 2*4 */
```

```
t[i].a=0, t[i].b=100  
t[i].a=1, t[i].b=101  
t[i].a=2, t[i].b=102
```

output du
1^{er} printf


Tableaux

■ Typage et arithmétique des pointeurs

- Utilisation de la *conversion de type explicite* (ou *cast*)

```
struct st { int a; int b; };
```

```
printf("%d = 2*%d\n", sizeof(struct st), sizeof(int));  
/* 8 = 2*4 */
```

```
struct st t[3]; int i;  
  
for(i=0; i<3; i++) {  
    t[i].a=i; t[i].b=100+i;  
    printf("t[i].a=%d, t[i].b=%d\n", t[i].a, t[i].b);  
}  
  
int* s = (int*)t;   
for(i=0; i<6; i++)  
    printf("s[i]=%d\n", s[i]);
```

Tableaux

■ Typage et arithmétique des pointeurs

- Utilisation de la *conversion de type explicite* (ou *cast*)

```
struct st { int a; int b; };
```

```
printf("%d = 2*%d\n", sizeof(struct st), sizeof(int));  
/* 8 = 2*4 */
```

```
t[i].a=0, t[i].b=100  
t[i].a=1, t[i].b=101  
t[i].a=2, t[i].b=102
```

output du
1^{er} printf

```
s[i]=0  
s[i]=100  
s[i]=1  
s[i]=101  
s[i]=2  
s[i]=102
```

output du
2^{ème} printf

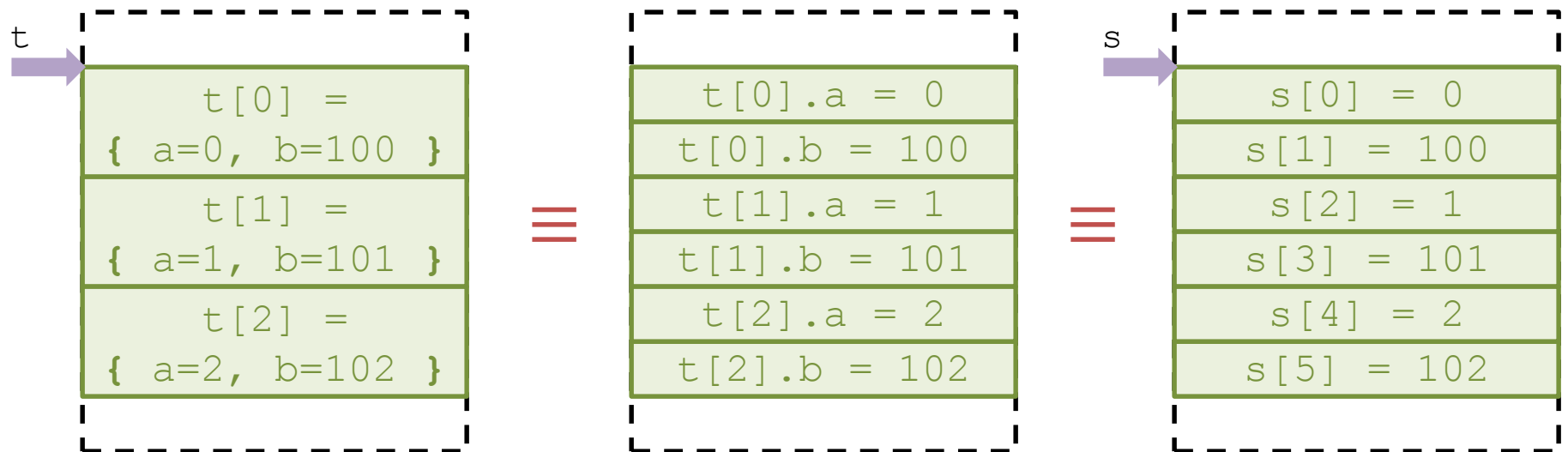
Tableaux

■ Typage et arithmétique des pointeurs

- Utilisation de la *conversion de type explicite* (ou *cast*)

```
struct st { int a; int b; };
```

```
printf("%d = 2*%d\n", sizeof(struct st), sizeof(int));  
/* 8 = 2*4 */
```



Plan du cours : C dynamique



- Modèle de la mémoire
- Pointeurs
- Tableaux
- Chaînes de caractères
- Allocation dynamique

Chaînes de caractères

■ Définition

- Tableau de caractères se terminant par le caractère spécial `'\0'`

```
char string[11] = "Helloworld";
```

- Pointeur vers le premier caractère

```
char* string = "Helloworld";
```



■ Manipulation des chaînes

- Opérations classiques : `#include <string.h>`

`strlen, str(n)cpy, str(n)cat, str(n)cmp, strchr, strrchr, strstr`

- Formater les chaînes : `#include <stdio.h>`

`printf, fprintf, sprintf, asprintf (GNU uniquement),
scanf, fscanf, sscanf`

Chaînes de caractères

■ Opérations classiques

□ `size_t strlen(const char* s)`

Calcule la longueur de la chaîne s

□ `char* strcpy(char* dst, const char* src)`

Copie la chaîne src à l'emplacement pointé par dst (une place suffisante aura due être allouée)

□ `size_t strncpy(char* dst, const char* src, size_t n)`

Similaire à `strcpy` mais copie au plus n caractères

□ `char* strcat(char* dst, const char* src)`

Concatène (copie) la chaîne src à la suite de la chaîne dst (une place suffisante aura due être allouée)

□ `size_t strncat(char* dst, const char* src, size_t n)`

Similaire à `strcat` mais concatène au plus n caractères

Chaînes de caractères

■ Opérations classiques

□ `int strcmp(const char* s1, const char* s2)`

Compare les chaînes de caractères en utilisant l'ordre des entiers (qui coïncide avec l'ordre lexicographique dans le code ASCII) ; retourne un entier négatif, nul ou positif si s1 est respectivement inférieur, égal ou supérieur à s2

□ `int strncmp(const char* s1, const char* s2, size_t n)`

Similaire à `strcmp` mais compare au plus n caractères

□ `char* strchr(const char* s, int c)`

Recherche la première occurrence du caractère c dans la chaîne s ; renvoie le pointeur vers ce caractère dans s s'il existe, `NULL` sinon

□ `char* strrchr(const char* s, int c)`

Similaire à `strchr` mais retourne un pointeur vers la dernière occurrence de c

□ `char* strstr(const char* s1, const char* s2)`

Recherche la première occurrence de la sous-chaîne s2 dans la chaîne s1 ; renvoie le pointeur vers cette occurrence si elle existe, `NULL` sinon

Plan du cours : C dynamique



- Modèle de la mémoire
- Pointeurs
- Tableaux
- Chaînes de caractères
- Allocation dynamique

Allocation dynamique



■ Motivation

- Conserver des données complexes (e.g., en jouant sur les adresses)
- Utiliser la mémoire uniquement si nécessaire

■ Utilisation de la mémoire

□ Stack

- Données : variables locales et arguments
- Durée de vie : **courte** (le temps d'exécution du bloc)

□ Data

- Données : variables globales et statiques
- Durée de vie : **permanente** (le temps d'exécution du programme)

□ Heap

- Données : variables allouées dynamiquement
- Durée de vie : **intermédiaire** (entre l'allocation et la désallocation)

Allocation dynamique

■ malloc

□ Prototype

```
#include <stdlib.h>

void* malloc(size_t size);
```

□ Action (> **man malloc**)

« Alloue size octets dans le tas (heap), et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone n'est pas initialisé. Si la taille size est nulle ou s'il n'y a pas assez de mémoire disponible, renvoie le pointeur NULL. »

```
int* tab = (int*) malloc(30*sizeof(int));

if (tab == NULL) {
    /* gestion de l'erreur */
}
```

Allocation dynamique

■ calloc

□ Prototype

```
#include <stdlib.h>

void* calloc(size_t nmemb, size_t size);
```

□ Action (> **man calloc**)

« Alloue dans le tas (heap) la mémoire nécessaire pour un tableau de nmemb éléments de taille size octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone est initialisé à 0. Si la taille size ou le nombre d'élément nmemb est nulle ou s'il n'y a pas assez de mémoire disponible, renvoie le pointeur NULL. »

```
int* tab =
    (int*) calloc(30, sizeof(int));

if (tab == NULL) /* erreur */
```

```
int* tab =
    (int*) malloc(30*sizeof(int));

if (tab == NULL) /* erreur */

for (i=0; i<30; i++) tab[i]=0;
```

Allocation dynamique

■ `realloc`

□ Prototype

```
#include <stdlib.h>

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

□ Action (> `man realloc`)

« Modifie la taille du bloc mémoire pointé par ptr pour l'amener à une taille de size octets, et renvoie le pointeur vers la zone mémoire. Si le bloc pointé par ptr ne pas être étendu à size octets, un nouveau bloc est alloué, l'ancien est copié dedans puis désalloué (`free`). Si l'allocation échoue, `realloc` n'a aucun effet sur le bloc pointé par ptr et retourne `NULL`. Si la taille size est nulle, l'appel est équivalent à `free`. Si le pointeur ptr est `NULL`, l'appel est équivalent à `malloc`. »

Allocation dynamique

■ free

□ Prototype

```
#include <stdlib.h>

void free(void* ptr);
```

□ Action (> **man free**)

« Libère (désalloue) l'espace mémoire pointé par ptr qui doit avoir été obtenu par un appel à `malloc`, `calloc` ou `realloc`. Dans le cas contraire, le comportement de `free` n'est pas défini. Si ptr est `NULL`, `free` est sans effet. »

*A chaque **malloc** doit correspondre un **free***

LE credo en C

Allocation dynamique : *listes* chaînées

■ Problématique

□ Contraintes d'utilisation des tableaux

- Données contiguës en mémoire
- Nombre (max.) d'éléments (*i.e.*, taille du tableau) connu dès l'allocation
- Opérations à éviter : insertion et suppression d'un élément

```
int* tab = (int*)malloc(4 * sizeof(int));
```

```
tab[0] = 27;
```

```
tab[1] = 4;
```

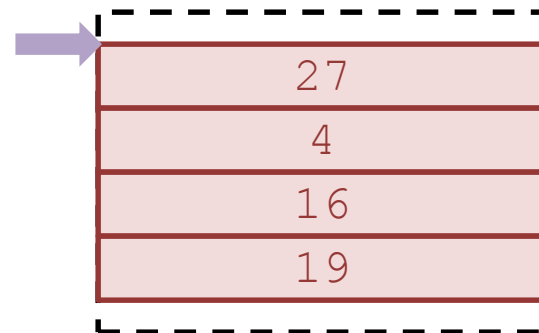
```
tab[2] = 16;
```

```
tab[3] = 19;
```

```
/* Insérer 1 entre 27 et 4 */
```

```
/* Supprimer 16 */
```

```
free(tab);
```

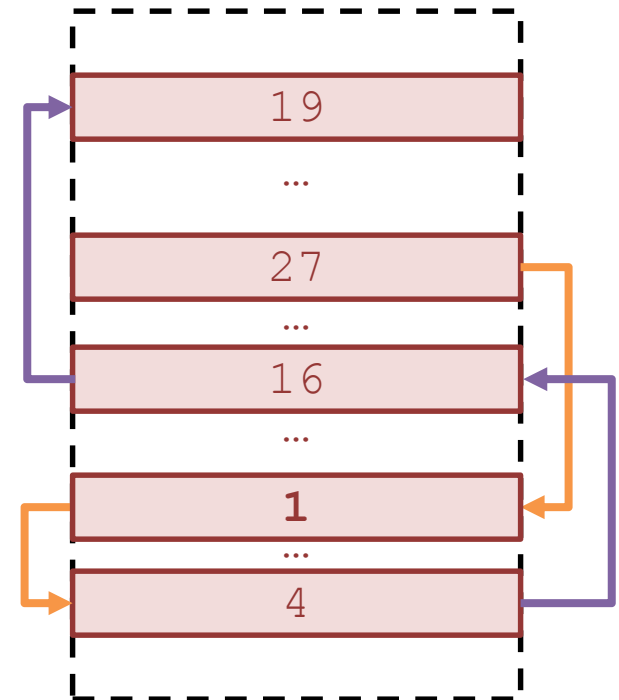
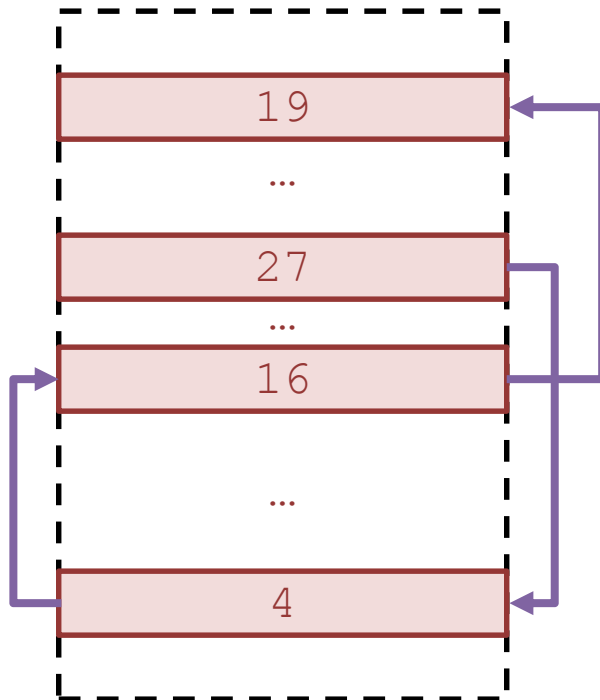


Allocation dynamique : *listes chaînées*

■ Problématique

□ Utilisation d'une liste chaînée

- Placer les données arbitrairement en mémoire
- Allouer l'espace mémoire nécessaire à la demande
- Modifier la mémoire localement lors de l'insertion et la suppression

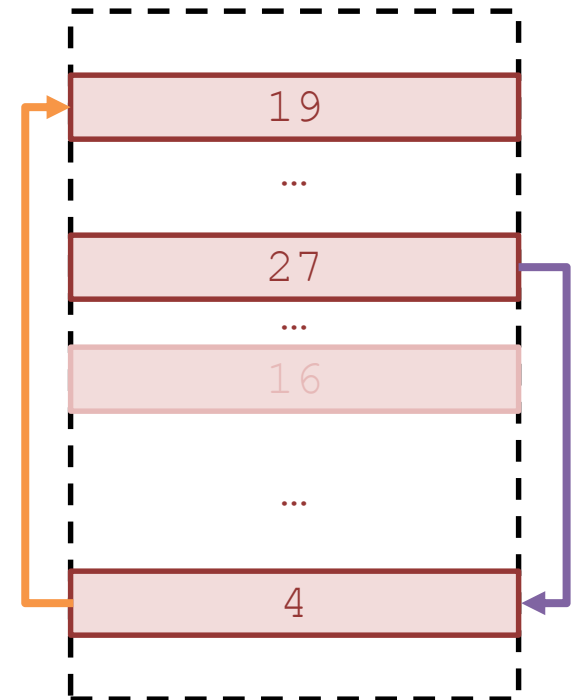
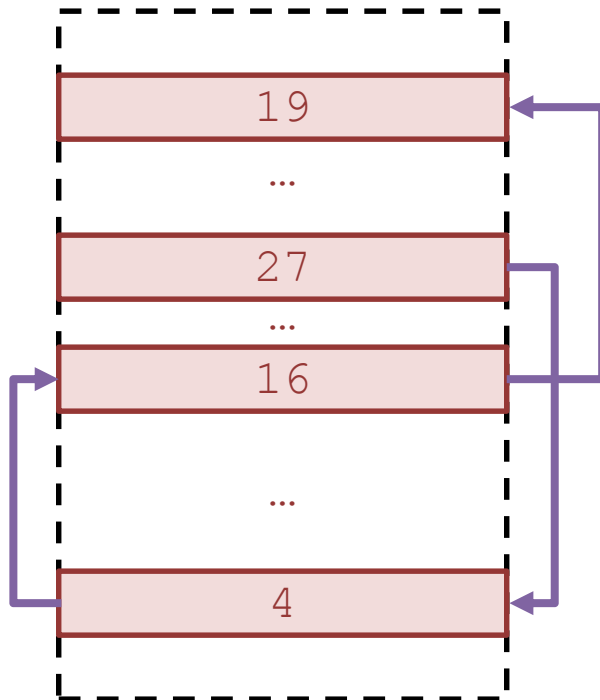


Allocation dynamique : *listes* chaînées

■ Problématique

□ Utilisation d'une liste chaînée

- Placer les données arbitrairement en mémoire
- Allouer l'espace mémoire nécessaire à la demande
- Modifier la mémoire localement lors de l'insertion et la suppression



Allocation dynamique : *listes* chaînées


■ Définition (possible) du type en C

□ Pour chaque élément, stocker

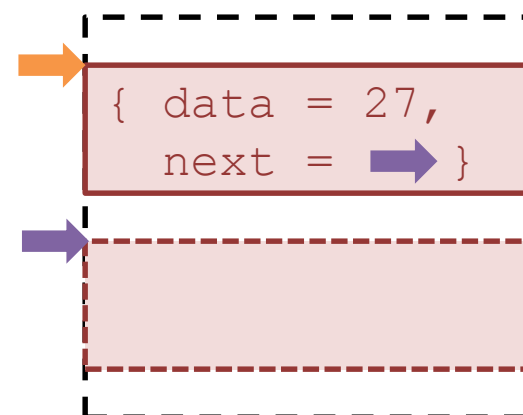
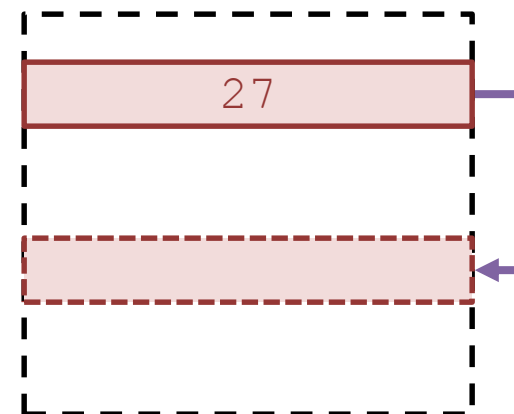
- La donnée : **27**
- L'adresse de l'élément suivant : 

```
struct block_s {  
    int          value;  
    struct block_s* next;  
};  
  
typedef struct block_s block;
```

□ Liste d'éléments

- Tête de la liste : 
- Pointeur vers le premier élément

```
typedef block* list;
```



Allocation dynamique : *listes* chaînées

■ Liste chaînée et pointeur **NULL**

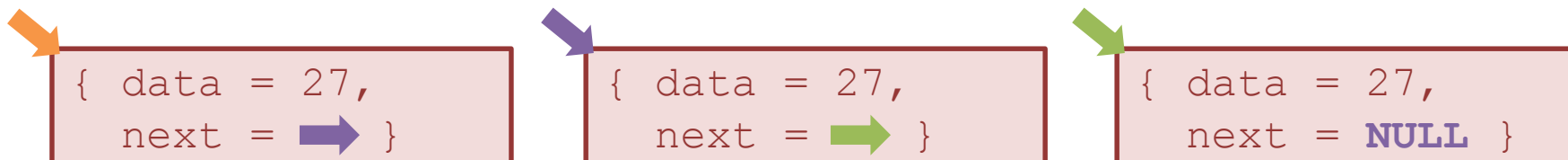
□ Liste vide

Absence de premier élément

```
#define EMPTY ((list) NULL)
```

□ Bloque-élément sans élément suivant

Absence de dernier élément



□ Il s'agit du même cas

- **Queue** de la liste
- **Pointeur** vers l'élément suivant = liste

```
struct block_s* next;
```

≡


```
block* next;
```

≡

```
list next;
```

Allocation dynamique : *listes* chaînées

■ Construction de liste



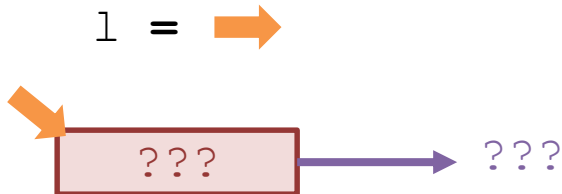
```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```

```
l = ???
```

Allocation dynamique : *listes* chaînées

■ Construction de liste

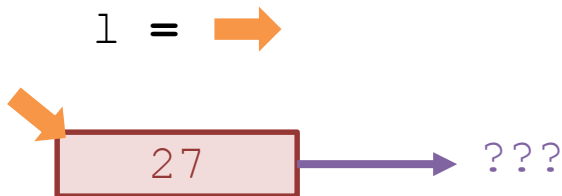
```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

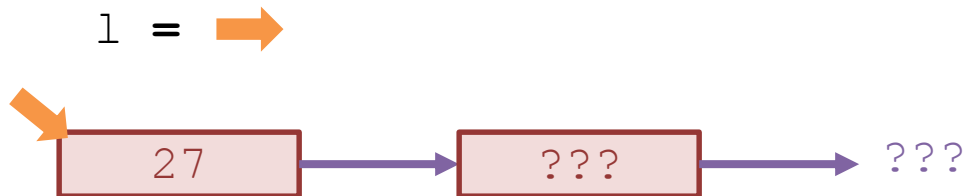
```
list l;  
l = (block*) malloc(sizeof(block));  
→ l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

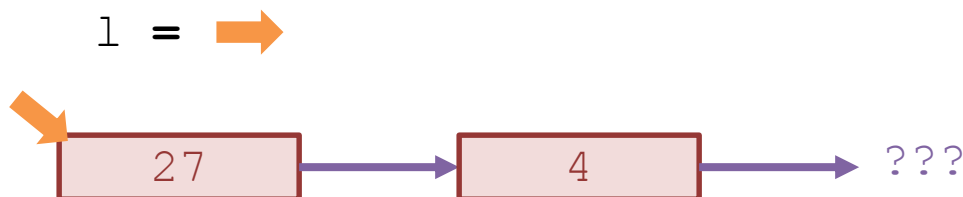
```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
→ l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

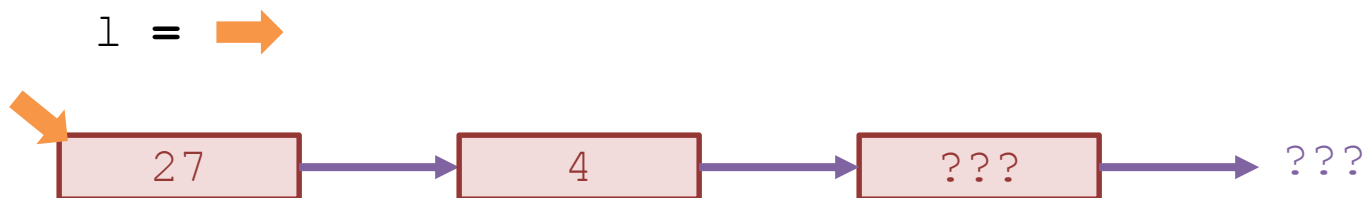
```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
→ l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

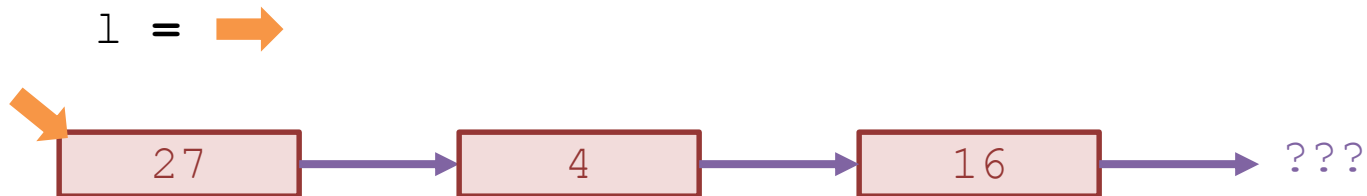
```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
→ l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

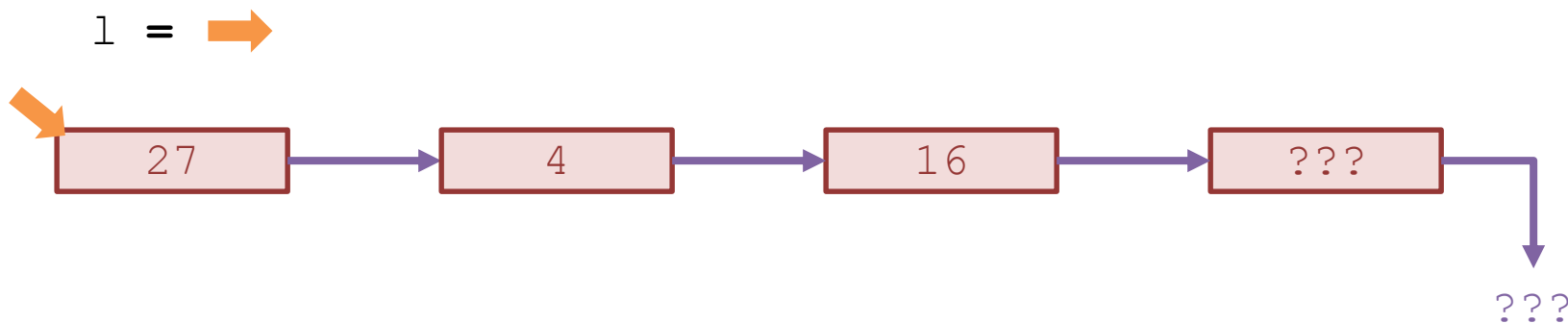
```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

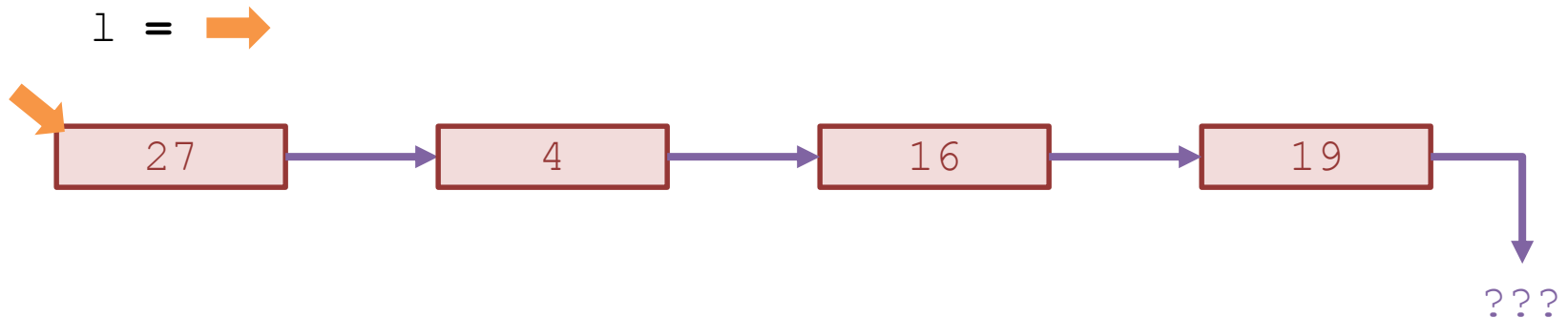
```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
→ l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

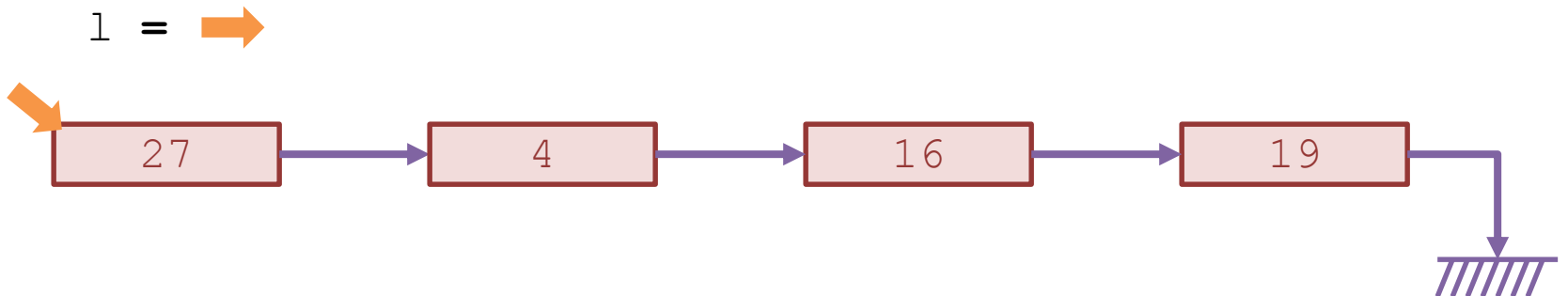
```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```

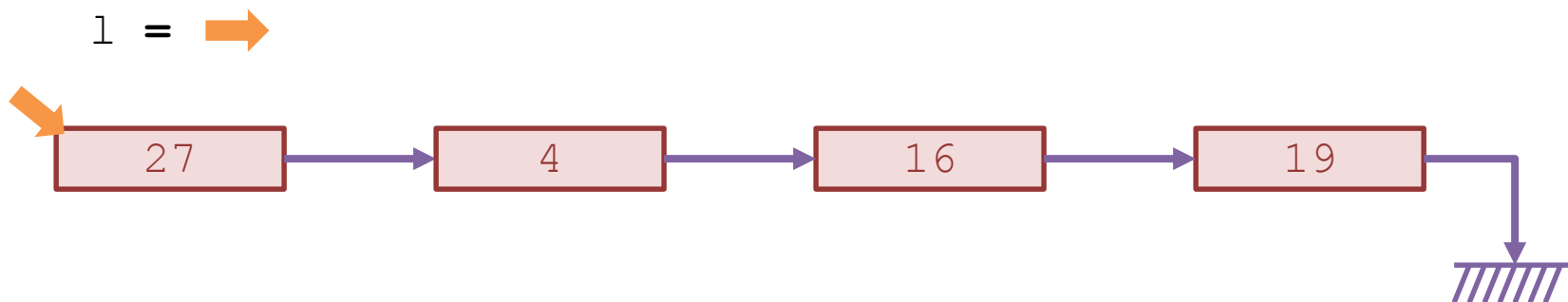


Allocation dynamique : *listes* chaînées

■ Construction de liste

Création d'un bloc

```
list l;  
l = (block*) malloc(sizeof(block));  
l->data = 27;  
l->next = (block*) malloc(sizeof(block));  
l->next->data = 4;  
l->next->next = (block*) malloc(sizeof(block));  
l->next->next->data = 16;  
l->next->next->next = (block*) malloc(sizeof(block));  
l->next->next->next->data = 19;  
l->next->next->next->next = NULL;
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

Création d'un bloc

```
block* new_block(int v, block* n) {  
    block* b = (block*)malloc(sizeof(block));  
    if (b == NULL) {  
        fprintf(stderr,  
            "List: new_block: memory allocation failure\n");  
        exit(EXIT_FAILURE);  
    }  
    b->data = v;  
    b->next = n;  
    return b;  
}
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

Création d'un bloc : MAUVAIS CODES

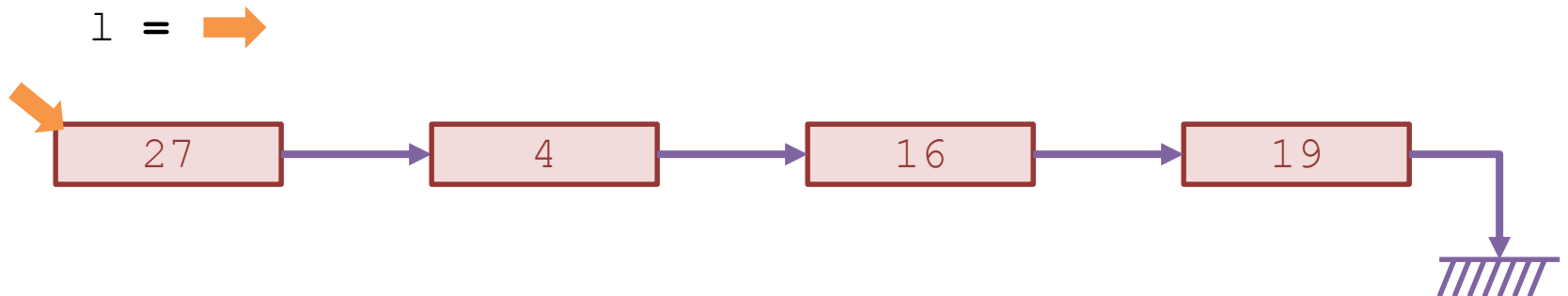
```
block new_block(int v, block n) {  
    block b;  
    b.data = v;  
    b.next = &n;  
    return b;  
}
```

```
block* new_block(int v, block* n) {  
    block b;  
    b.data = v;  
    b.next = &n;  
    return b;  
}
```


Allocation dynamique : *listes* chaînées

■ Construction de liste

```
list l;  
l = new_block(27, NULL);  
l->next = new_block(4, NULL);  
l->next->next = new_block(16, NULL);  
l->next->next->next = new_block(19, NULL);
```

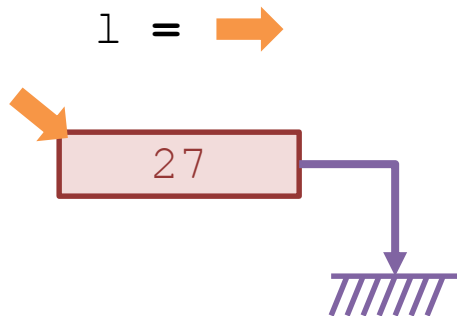


Allocation dynamique : *listes* chaînées

■ Construction de liste

Éviter les appels répétés à « ->next »

```
list l;  
l = new_block(27, NULL);  
l->next = new_block(4, NULL);  
l = l->next;  
l->next = new_block(16, NULL);  
l = l->next;  
l->next = new_block(19, NULL);
```

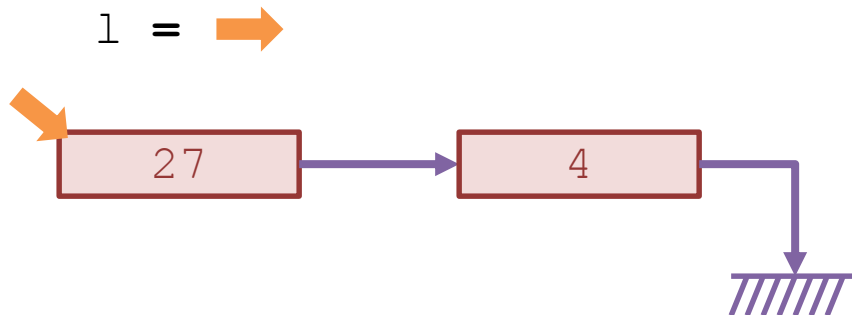


Allocation dynamique : *listes* chaînées

■ Construction de liste

Éviter les appels répétés à « ->next »

```
list l;  
l = new_block(27, NULL);  
l->next = new_block(4, NULL);  
l = l->next;  
l->next = new_block(16, NULL);  
l = l->next;  
l->next = new_block(19, NULL);
```

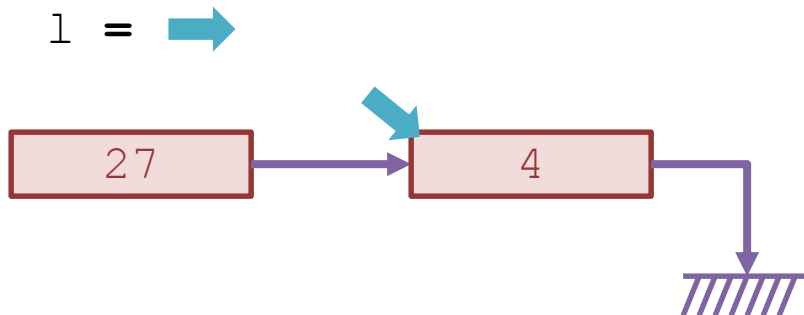


Allocation dynamique : *listes* chaînées

■ Construction de liste

Éviter les appels répétés à « ->next »

```
list l;  
l = new_block(27, NULL);  
l->next = new_block(4, NULL);  
→ l = l->next;  
l->next = new_block(16, NULL);  
l = l->next;  
l->next = new_block(19, NULL);
```

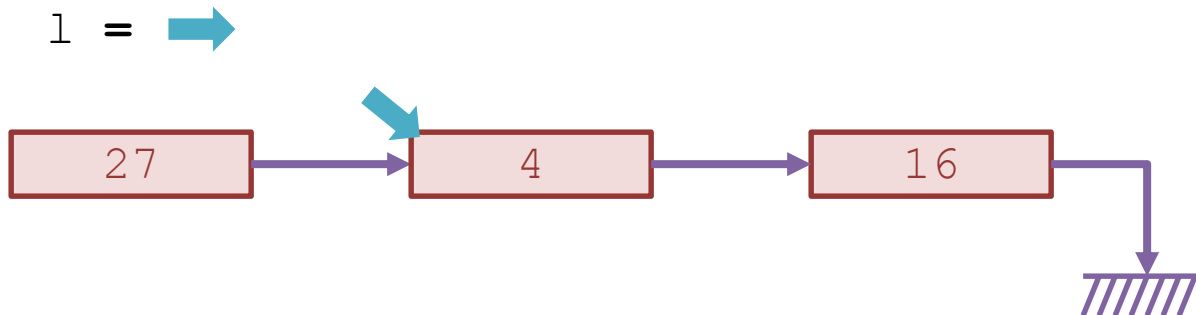


Allocation dynamique : *listes* chaînées

■ Construction de liste

Éviter les appels répétés à « ->next »

```
list l;  
l = new_block(27, NULL);  
l->next = new_block(4, NULL);  
l = l->next;  
→ l->next = new_block(16, NULL);  
l = l->next;  
l->next = new_block(19, NULL);
```

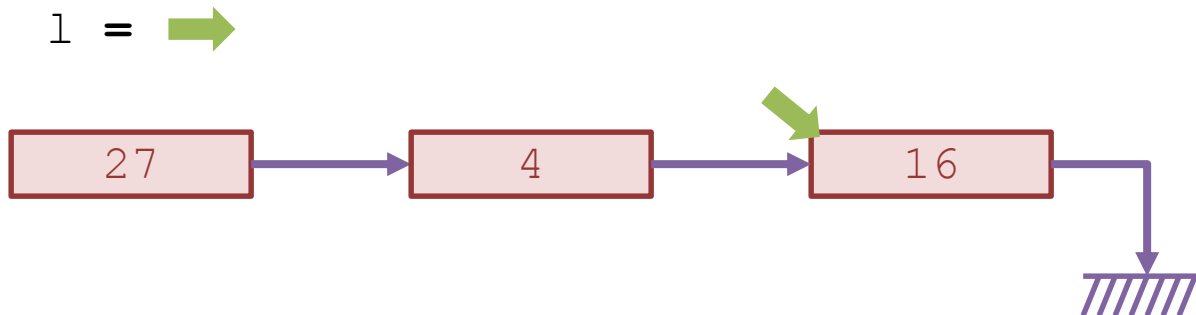


Allocation dynamique : *listes* chaînées

■ Construction de liste

Éviter les appels répétés à « ->next »

```
list l;  
l = new_block(27, NULL);  
l->next = new_block(4, NULL);  
l = l->next;  
l->next = new_block(16, NULL);  
l = l->next;  
l->next = new_block(19, NULL);
```



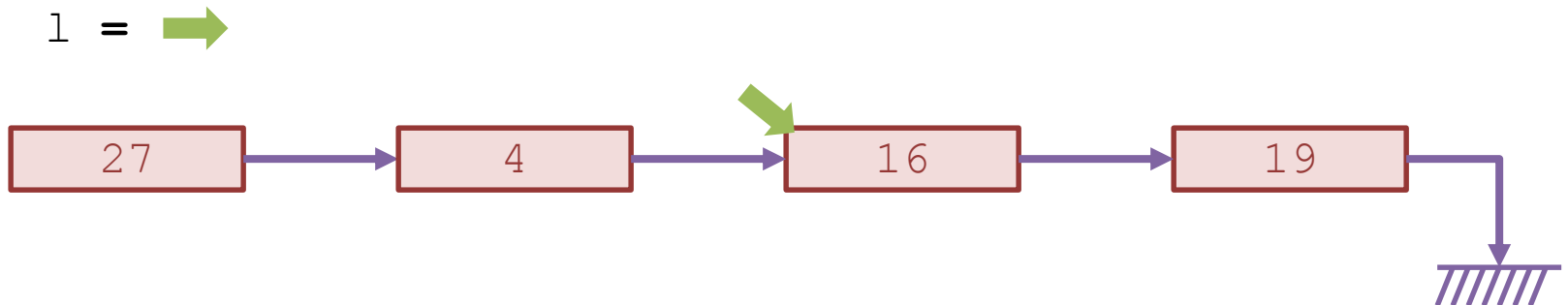
Allocation dynamique : *listes* chaînées

■ Construction de liste

Éviter les appels répétés à « ->next »

```
list l;  
l = new_block(27, NULL);  
l->next = new_block(4, NULL);  
l = l->next;  
l->next = new_block(16, NULL);  
l = l->next;  
l->next = new_block(19, NULL);
```

Attention à ne pas perdre la tête



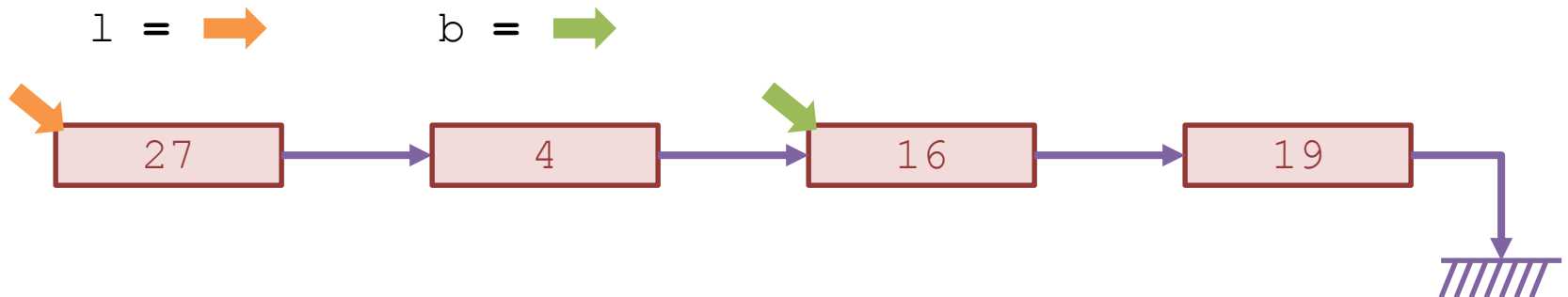
Allocation dynamique : *listes* chaînées

■ Construction de liste

Éviter les appels répétés à « ->next »

Un algorithme apparaît

```
list l;  
block* b = l = new_block(27, NULL);  
b->next = new_block(4, NULL);  
b = b->next;  
b->next = new_block(16, NULL);  
b = b->next;  
b->next = new_block(19, NULL);
```

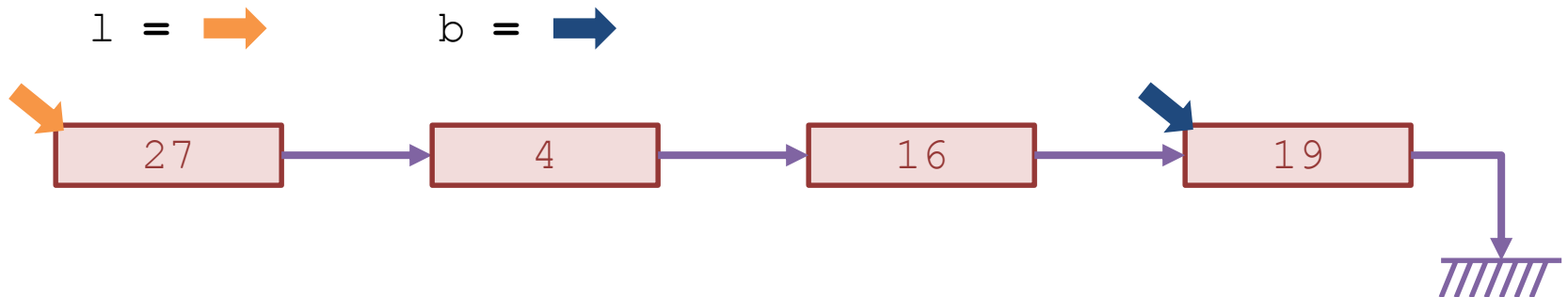


Allocation dynamique : *listes* chaînées

■ Construction de liste

Algorithme de construction

```
int i, data[4] = { 27, 4, 16, 19 };  
list l;  
block* b = l = new_block(data[0], NULL);  
for(i = 1; i < 4; i++) {  
    b->next = new_block(data[i], NULL);  
    b = b->next;  
}
```



Allocation dynamique : *listes* chaînées

■ Construction de liste

Algorithme de construction : conversion d'un tableau en liste


- Prérequis : `data` est correctement alloué et de taille `size`
- Cas particulier : le tableau de taille 0

```
list list_of_array(int* data, unsigned int size) {  
    if (size == 0) return EMPTY;  
    unsigned int i;  
    list l;  
    block* b = l = new_block(data[0], NULL);  
    for(i = 1; i < size; i++) {  
        b->next = new_block(data[i], NULL);  
        b = b->next;  
    }  
    return l;  
}
```

Allocation dynamique : *listes* chaînées

■ Construction de liste

Alternative : itérer dans l'ordre inverse




```
list l;  
l = EMPTY;  
l = new_block(19, l);  
l = new_block(16, l);  
l = new_block(4, l);  
l = new_block(27, l);
```

```
l = ???
```


Allocation dynamique : *listes* chaînées

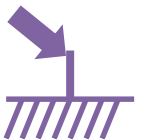
■ Construction de liste

Alternative : itérer dans l'ordre inverse



```
list l;  
l = EMPTY;  
l = new_block(19, l);  
l = new_block(16, l);  
l = new_block(4, l);  
l = new_block(27, l);
```

l = 



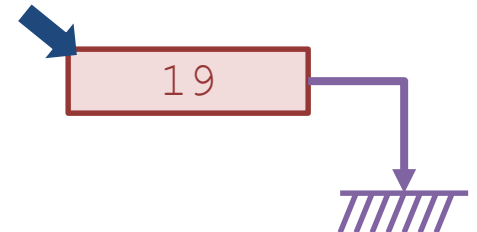
Allocation dynamique : *listes* chaînées

■ Construction de liste

Alternative : itérer dans l'ordre inverse

```
list l;  
l = EMPTY;  
→ l = new_block(19, l);  
l = new_block(16, l);  
l = new_block(4, l);  
l = new_block(27, l);
```

l = ➡




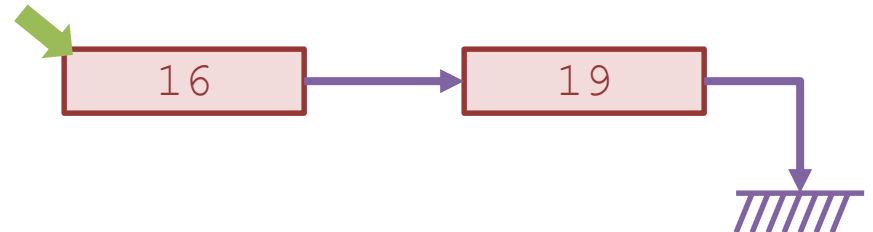
Allocation dynamique : *listes* chaînées

■ Construction de liste

Alternative : itérer dans l'ordre inverse

```
list l;  
l = EMPTY;  
l = new_block(19, l);  
l = new_block(16, l);  
l = new_block(4, l);  
l = new_block(27, l);
```

l = 



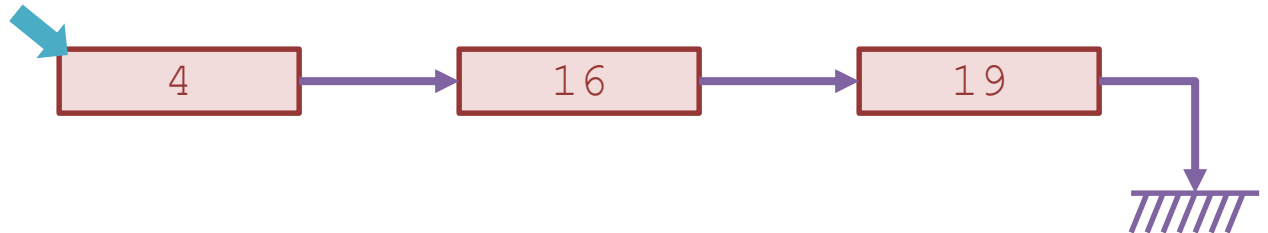
Allocation dynamique : *listes* chaînées

■ Construction de liste

Alternative : itérer dans l'ordre inverse

```
list l;  
l = EMPTY;  
l = new_block(19, l);  
l = new_block(16, l);  
→ l = new_block(4, l);  
l = new_block(27, l);
```

l = →

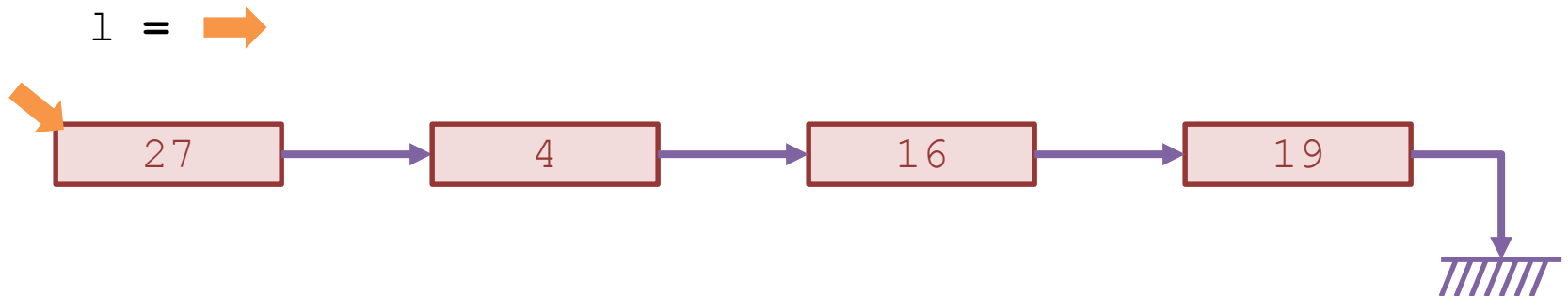


Allocation dynamique : *listes* chaînées

■ Construction de liste

Alternative : itérer dans l'ordre inverse

```
list l;  
l = EMPTY;  
l = new_block(19, l);  
l = new_block(16, l);  
l = new_block(4, l);  
→ l = new_block(27, l);
```



Allocation dynamique : *listes* chaînées

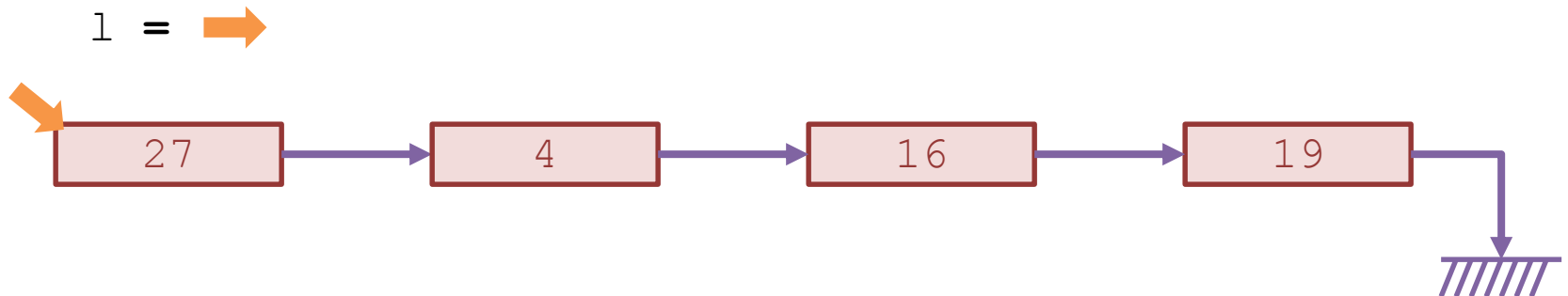
■ Construction de liste

Alternative : itérer dans l'ordre inverse

Un algorithme apparaît

```
list l;  
l = EMPTY;  
l = new_block(19, l);  
l = new_block(16, l);  
l = new_block(4, l);  
l = new_block(27, l);
```

Plus naturel, à la façon d'une pile



Allocation dynamique : *listes* chaînées

■ Construction de liste

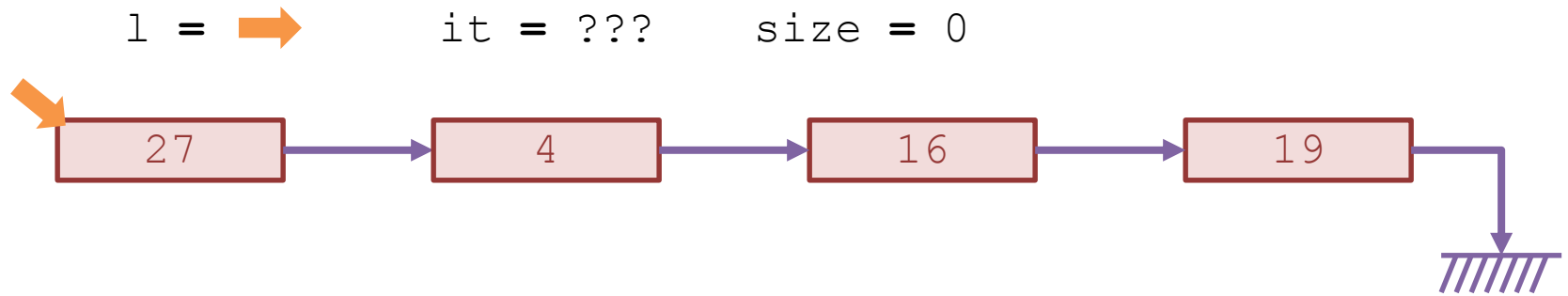
Algorithme de construction (bis) : conversion d'un tableau en liste

- Prérequis : `data` est correctement alloué et de taille `size`
- Aucun cas particulier

```
list list_of_array(int* data, unsigned int size) {  
    list l = EMPTY;  
    while (size > 0)  
        l = new_block(data[--size], l);  
    return l;  
}
```

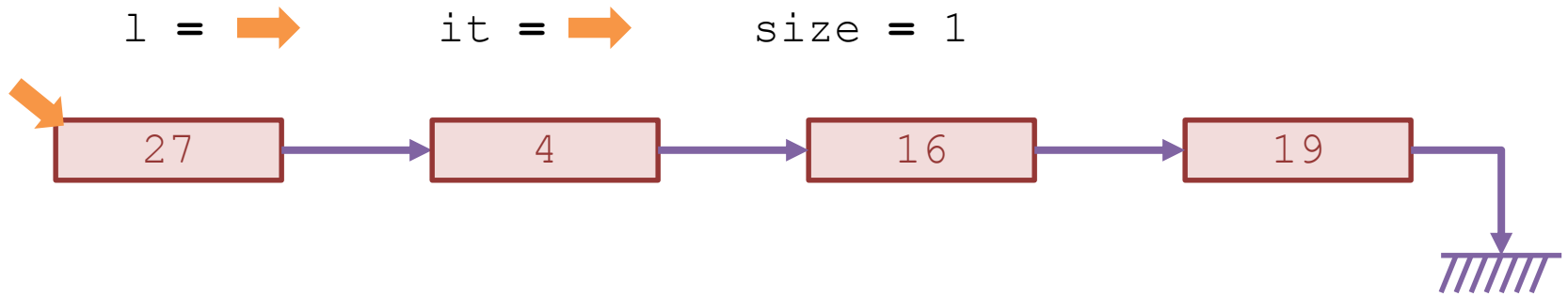
Allocation dynamique : *listes* chaînées

■ Evaluer la longueur d'une liste



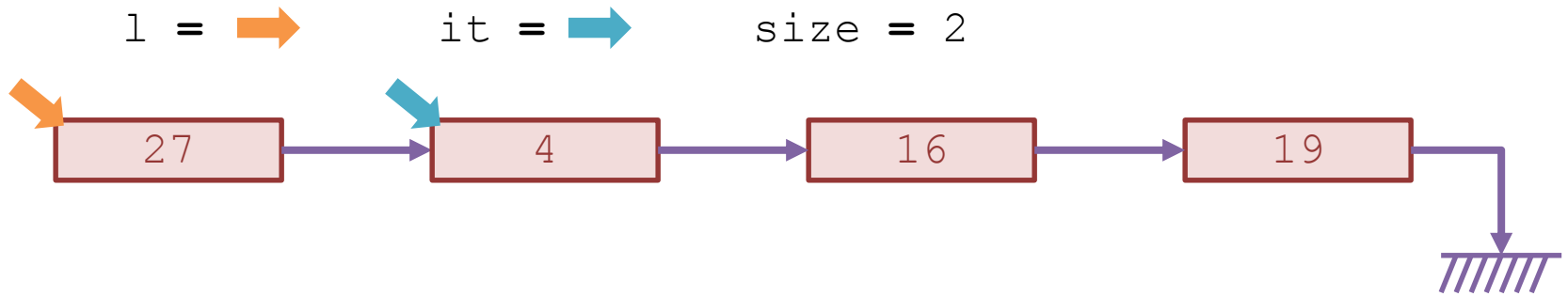
Allocation dynamique : *listes* chaînées

■ Evaluer la longueur d'une liste



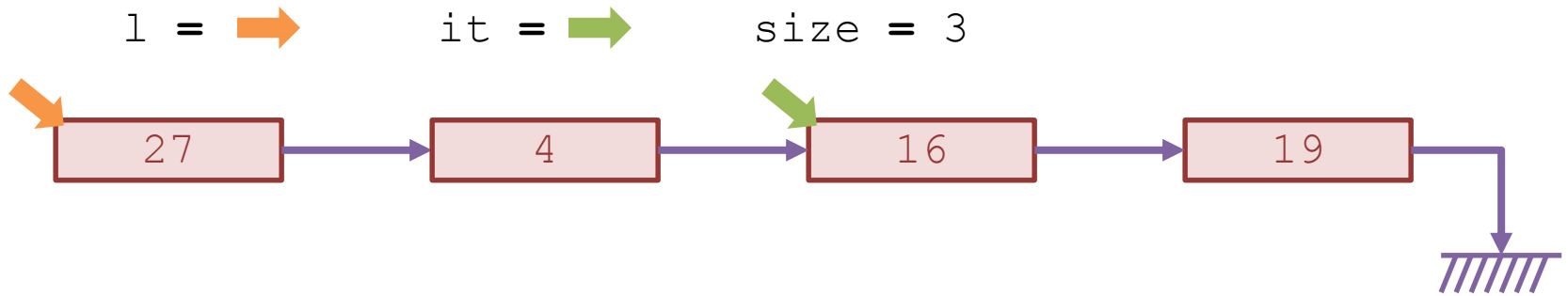
Allocation dynamique : *listes* chaînées

■ Evaluer la longueur d'une liste



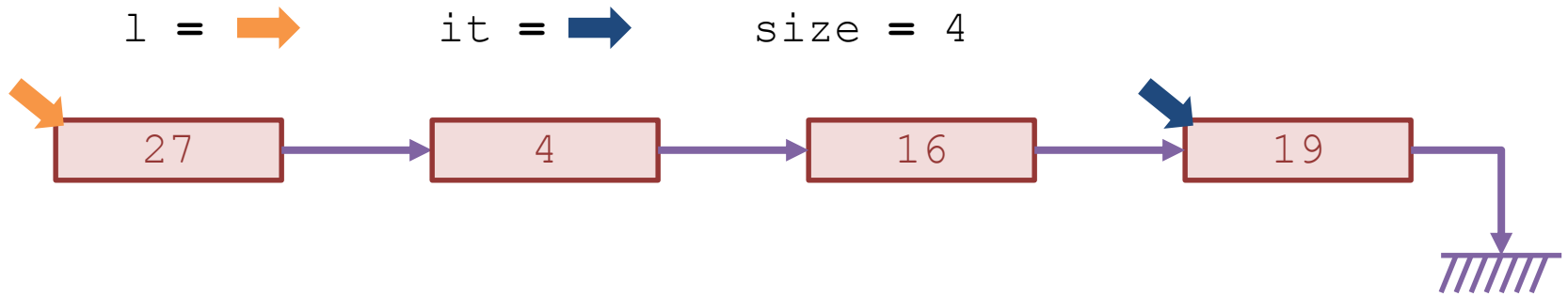
Allocation dynamique : *listes* chaînées

■ Evaluer la longueur d'une liste



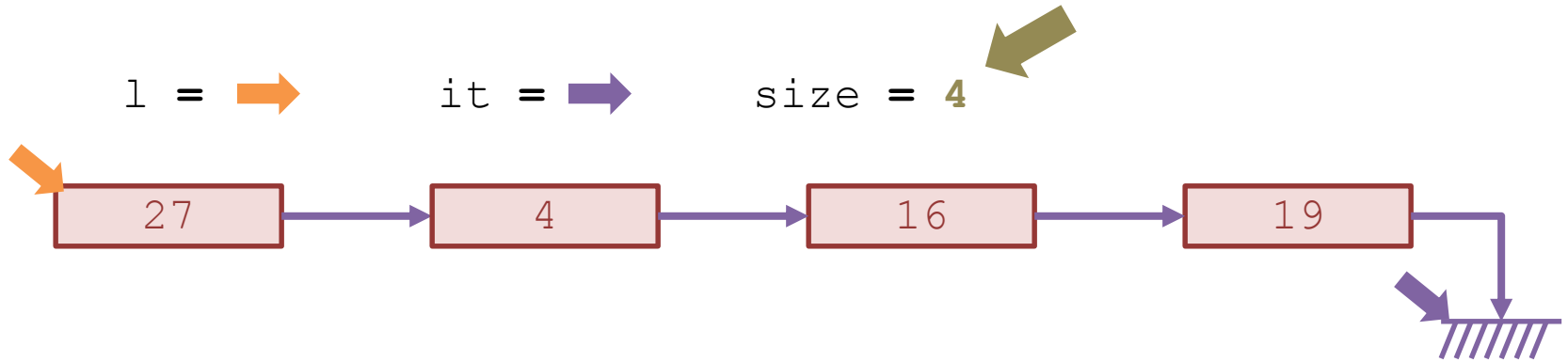
Allocation dynamique : *listes* chaînées

■ Evaluer la longueur d'une liste



Allocation dynamique : *listes* chaînées

■ Evaluer la longueur d'une liste

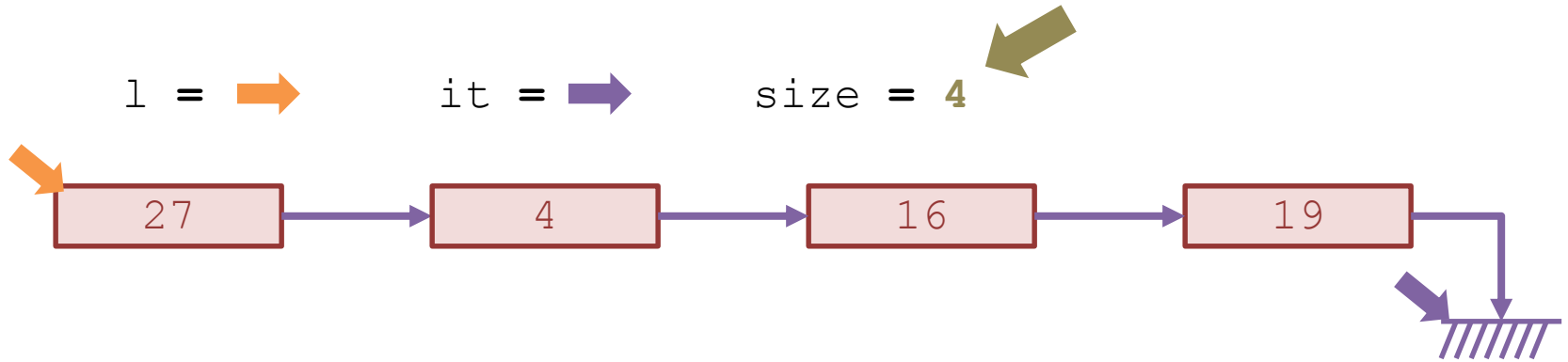


```
unsigned int length (list l) {  
    unsigned int size = 0;  
    block* it = l;  
    while (it != EMPTY) {  
        size++;  
        it = it->next;  
    }  
    return size;  
}
```

```
unsigned int length (list it) {  
    unsigned int size = 0;  
    while (it != EMPTY) {  
        size++;  
        it = it->next;  
    }  
    return size;  
}
```


Allocation dynamique : *listes* chaînées

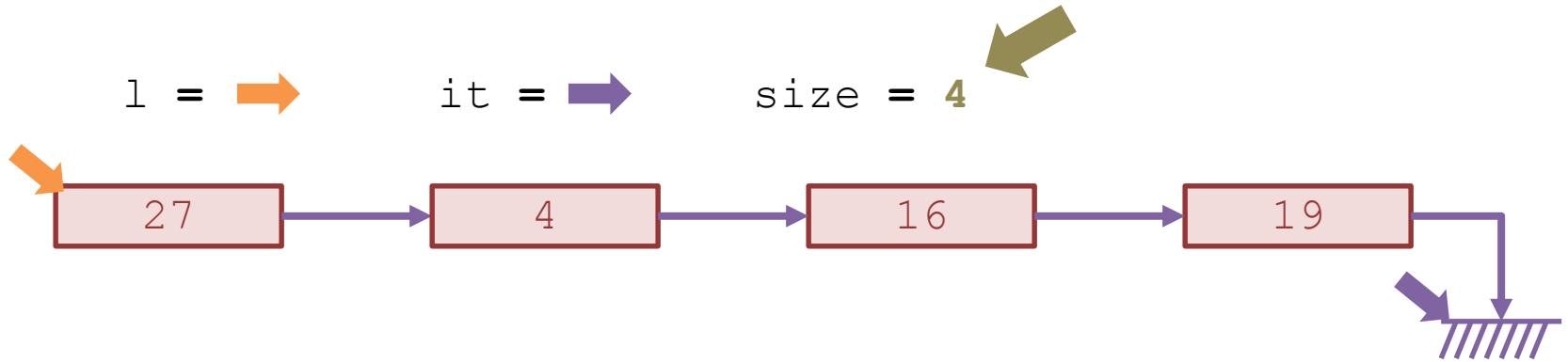
■ Evaluer la longueur d'une liste



```
unsigned int length (list l) {  
    unsigned int size = 0;  
    block* it;  
    for(it = l; it != EMPTY; it = it->next)  
        size++;  
    return size;  
}
```

Allocation dynamique : *listes* chaînées

■ Evaluer la longueur d'une liste





```
unsigned int length (list it) {  
    unsigned int size = 0;  
    for(; it != EMPTY; it = it->next)  
        size++;  
    return size;  
}
```

Allocation dynamique : *listes* chaînées

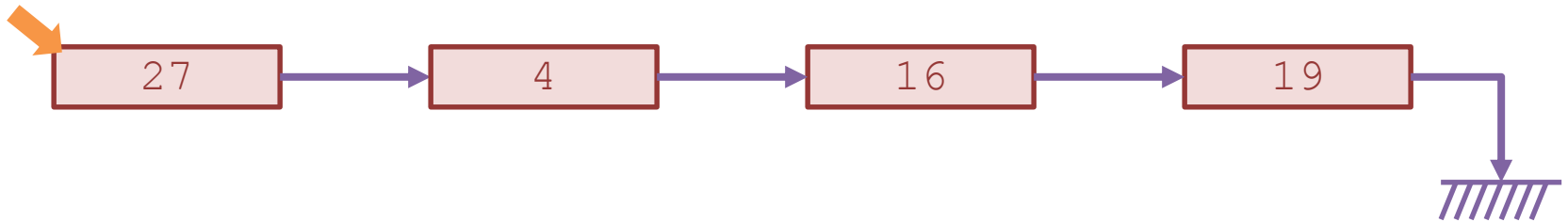
■ Insérer à la $n^{\text{ième}}$ position

$n = 2$

$l =$ 

$curr =$ 

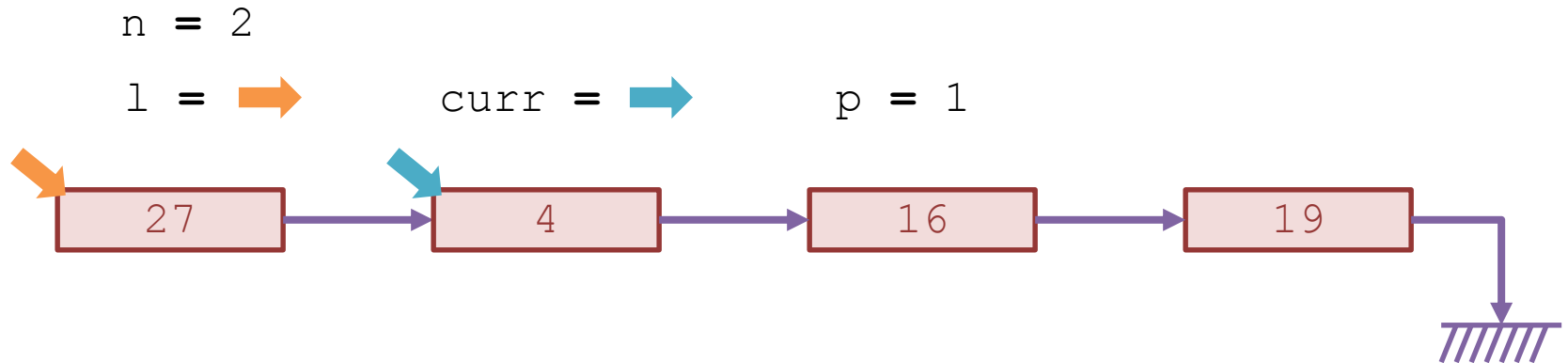
$p = 0$



```
void insert(list l, int v, unsigned int n) {  
    unsigned int p = 0;  
    block *curr = l;  
    while( ??? p++ ??? )  
        curr = curr->next;  
    ...  
}
```

Allocation dynamique : *listes* chaînées

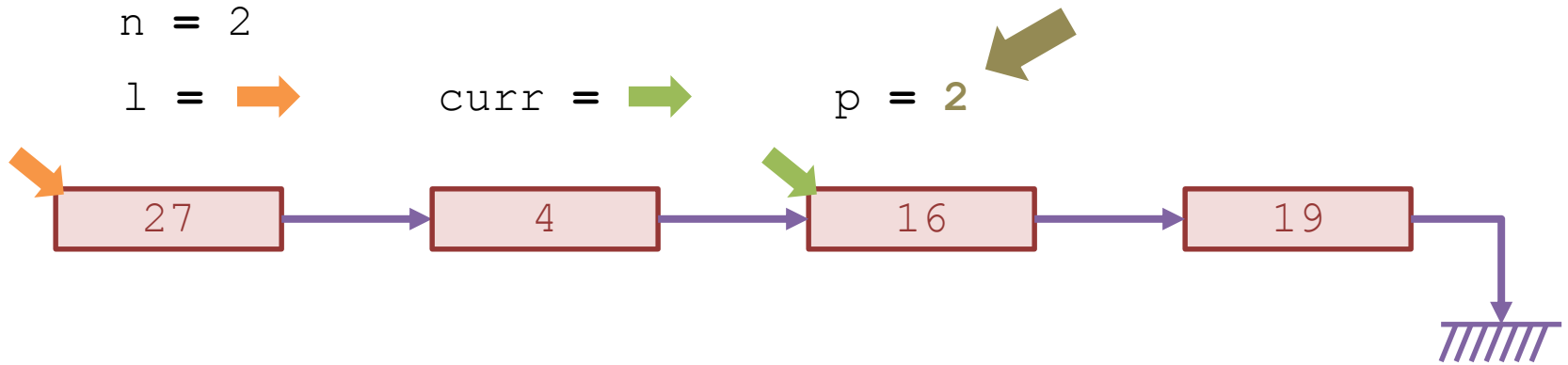
■ Insérer à la $n^{\text{ième}}$ position



```
void insert(list l, int v, unsigned int n) {  
    unsigned int p = 0;  
    block *curr = l;  
    while( ??? p++ ??? )  
        curr = curr->next;  
    ...  
}
```

Allocation dynamique : *listes* chaînées

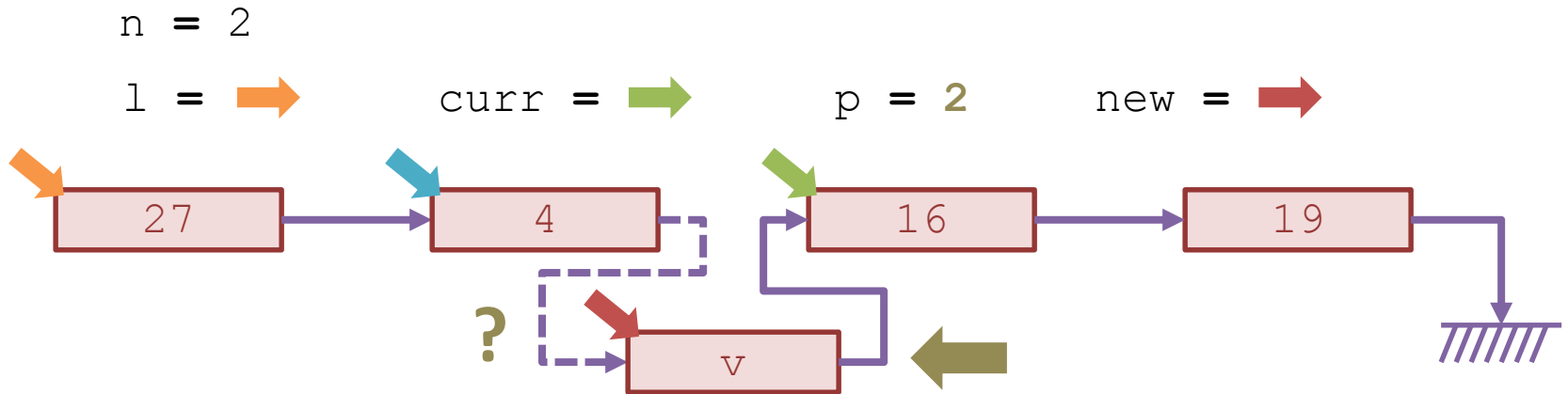
■ Insérer à la $n^{\text{ième}}$ position



```
void insert(list l, int v, unsigned int n) {  
    unsigned int p = 0;  
    block *curr = l;  
    while (p++ < n)  
        curr = curr->next;  
    ...  
}
```

Allocation dynamique : *listes* chaînées

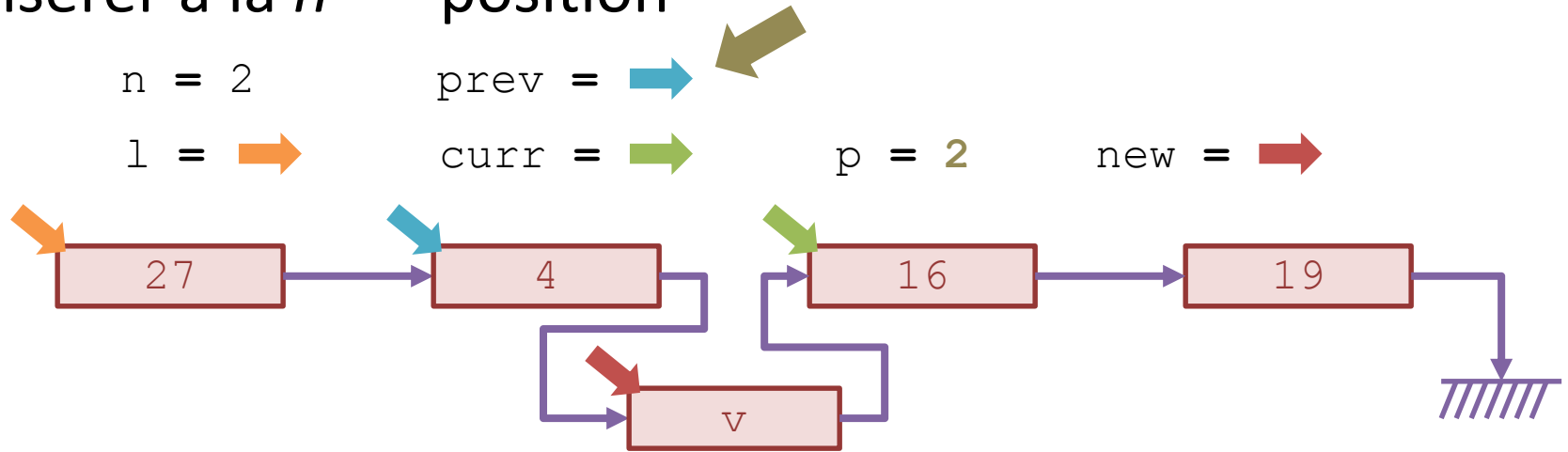
■ Insérer à la $n^{\text{ième}}$ position






```
void insert(list l, int v, unsigned int n) {  
    unsigned int p = 0;  
    block *curr = l;  
    while (p++ < n)  
        curr = curr->next;  
    block* new = new_block(v, curr);  
    ...  
}
```

Allocation dynamique : *listes* chaînées

■ Insérer à la $n^{\text{ième}}$ position



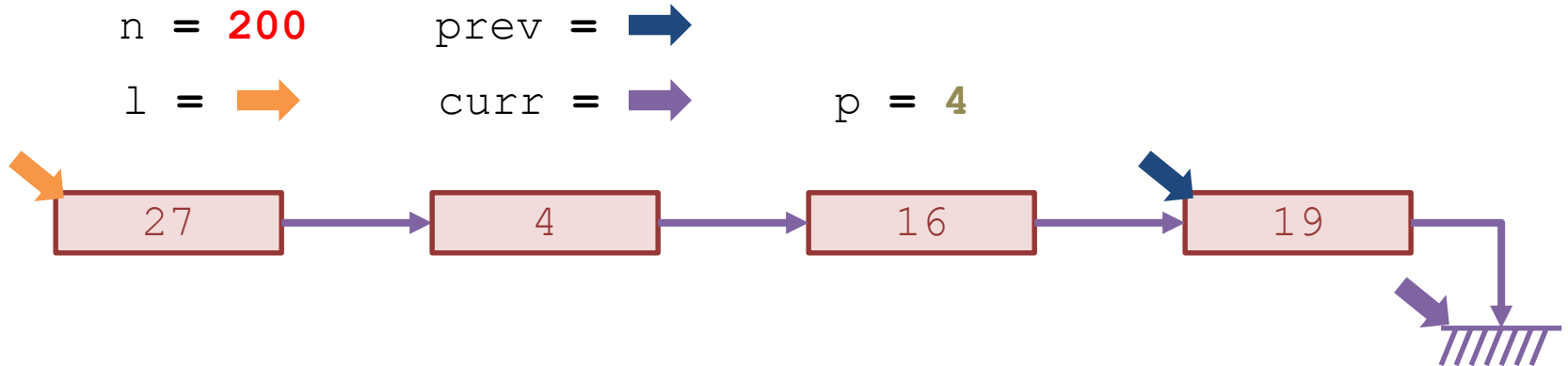
```
void insert(list l, int v, unsigned int n) {
    unsigned int p = 0;
    block *curr = l, *prev;
    while (p++ < n) {
        prev = curr;
        curr = curr->next;
    }
    block* new = new_block(v, curr);
    prev->next = new;
}
```




Allocation dynamique : *listes* chaînées

■ Insérer à la $n^{\text{ième}}$ position

Attention aux cas limite

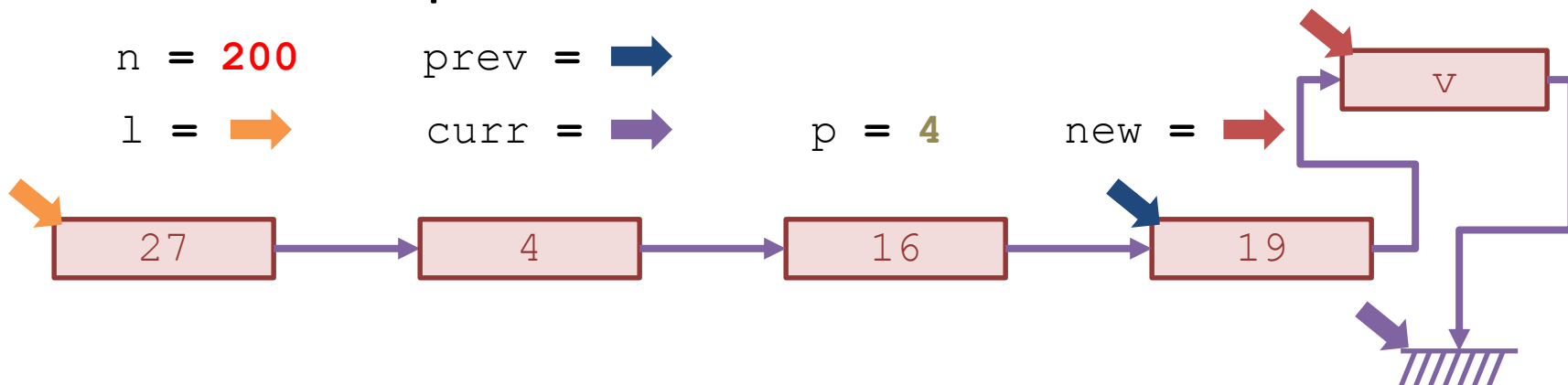


```
void insert(list l, int v, unsigned int n) {  
    unsigned int p = 0;  
    block *curr = l, *prev;  
    while (p++ < n) {  
        prev = curr;  
        curr = curr->next;  segmentation fault  
    }  
    block* new = new_block(v, curr);  
    prev->next = new;  
}
```


Allocation dynamique : *listes* chaînées

■ Insérer à la $n^{\text{ième}}$ position

Attention aux cas limite



```
void insert(list l, int v, unsigned int n) {  
    unsigned int p = 0;  
    block *curr = l, *prev;  
    while ((p++ < n) && (curr != NULL)) {  
        prev = curr;  
        curr = curr->next;  
    }  
    block* new = new_block(v, curr);  
    prev->next = new;  
}
```

Allocation dynamique : *listes* chaînées

■ Insérer à la $n^{\text{ième}}$ position

Attention aux cas limite

$n = 0$

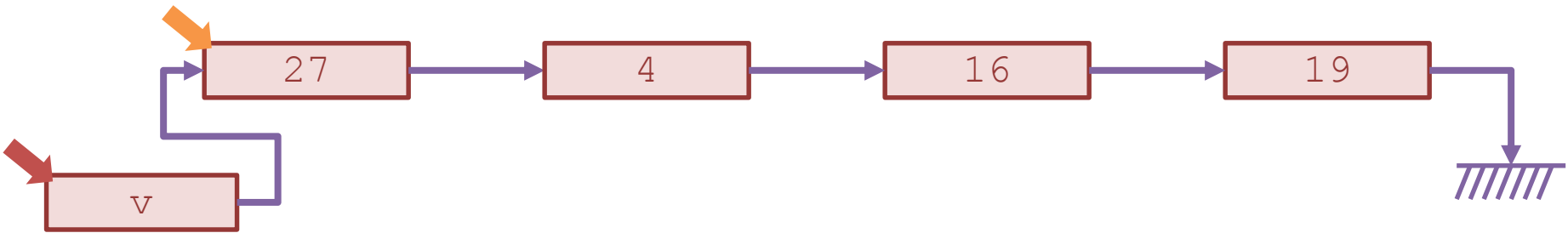
$prev = ???$

$l = \rightarrow$

$curr = \rightarrow$

$p = 0$

$new = \rightarrow$



```
void insert(list l, int v, unsigned int n) {  
    unsigned int p = 0;  
    block *curr = l, *prev;  
    while ((p++ < n) && (curr != NULL)) {  
        prev = curr;  
        curr = curr->next;  
    }  
    block* new = new_block(v, curr);  
    prev->next = new; segmentation fault  
}
```

Allocation dynamique : *listes* chaînées

■ Insérer à la $n^{\text{ième}}$ position

Attention aux cas limite

$n = 0$

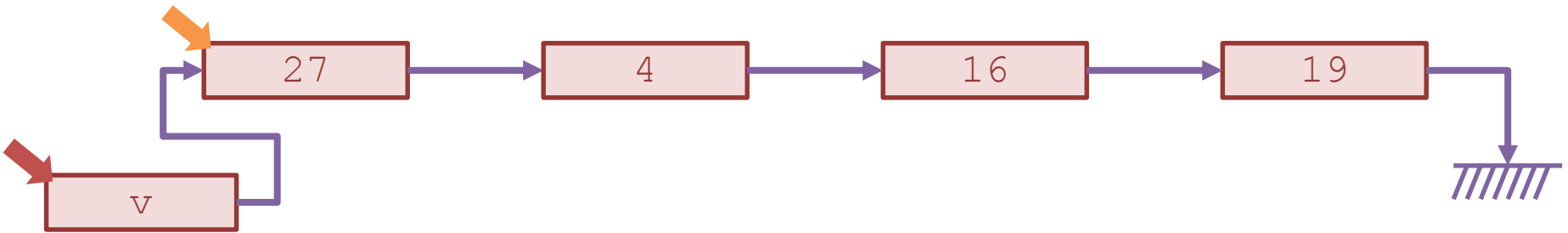
$prev = \text{NULL}$

$l = \rightarrow$

$curr = \rightarrow$

$p = 0$

$new = \rightarrow$



```
void insert(list l, int v, unsigned int n) {  
    unsigned int p = 0;  
    block *curr = l, *prev = NULL;  
    while ((p++ < n) && (curr != NULL)) {  
        prev = curr;  
        curr = curr->next;  
    }  
    block* new = new_block(v, curr);  
    if (prev != NULL) prev->next = new;  
}
```

Allocation dynamique : *listes* chaînées

■ Insérer à la $n^{\text{ième}}$ position

Attention aux cas limite

$n = 0$

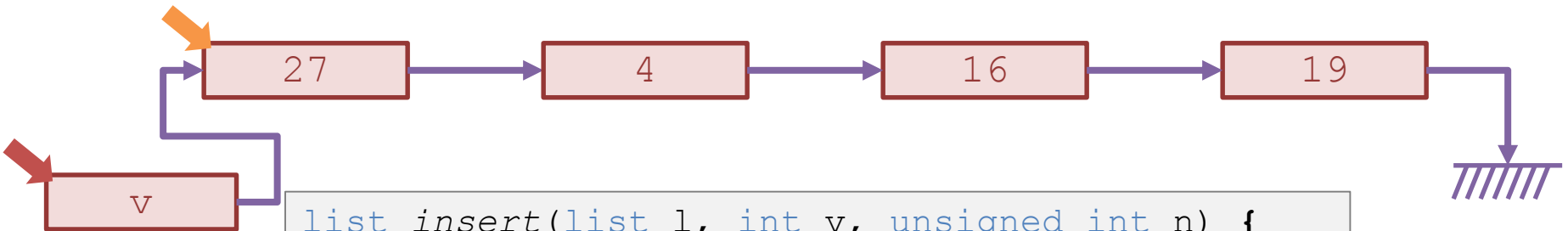
$prev = \text{NULL}$

$l = \rightarrow$

$curr = \rightarrow$

$p = 0$

$new = \rightarrow$



```
list insert(list l, int v, unsigned int n) {
    unsigned int p = 0;
    block *curr = l, *prev = NULL;
    while ((p++ < n) && (curr != NULL)) {
        prev = curr;
        curr = curr->next;
    }
    block* new = new_block(v, curr);
    if (prev == NULL) return new;
    prev->next = new;
    return l;
}
```

Allocation dynamique : *listes* chaînées

■ Insérer à la $n^{\text{ième}}$ position

Version avec boucle **for**

```
list insert(list l, int v, unsigned int n) {
    block *curr, *prev = NULL;
    for (curr = l; (n-- > 0) && (curr != NULL);
        curr = curr->next)
        prev = curr;
    block* new = new_block(v, curr);
    if (prev == NULL) return new;
    prev->next = new;
    return l;
}
```

Allocation dynamique : *listes* chaînées

■ Autres fonctions sur les listes chaînées (cf. TD/TP)

- ☐ Afficher une liste
- ☐ Retourner le $n^{\text{ième}}$ élément
- ☐ Supprimer le $n^{\text{ième}}$ élément
- ☐ Supprimer toute la liste
- ☐ Convertir une liste en tableau
- ☐ Concaténer deux listes
- ☐ Copier une liste
- ☐ Appliquer une fonction f sur chaque élément
- ☐ Conserver uniquement les éléments vérifiant un prédicat P
- ☐ ...

Allocation dynamique : *listes* chaînées

■ Quelques autres structures chaînées

- Listes *simplement* chaînées
 - De flottant, de double, de long, etc.
 - De n'importe quoi (type `void*`)
- Listes doublement chaînées
- Listes cycliques
- Arbres
 - n*-aires, binaires, rouge-noir, bien balancé, etc.
- Graphes...