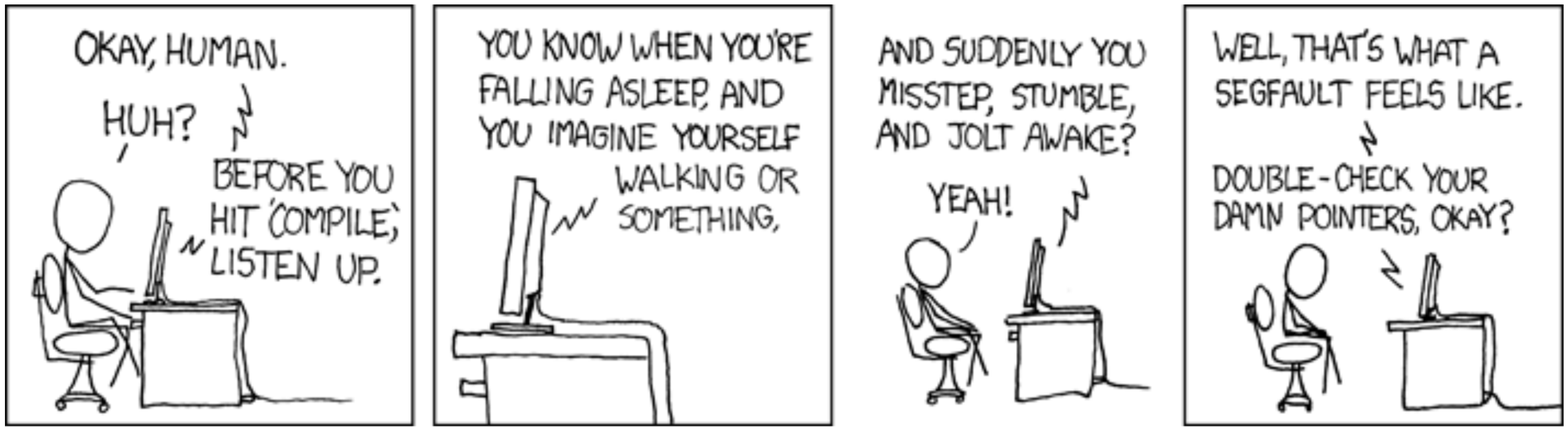
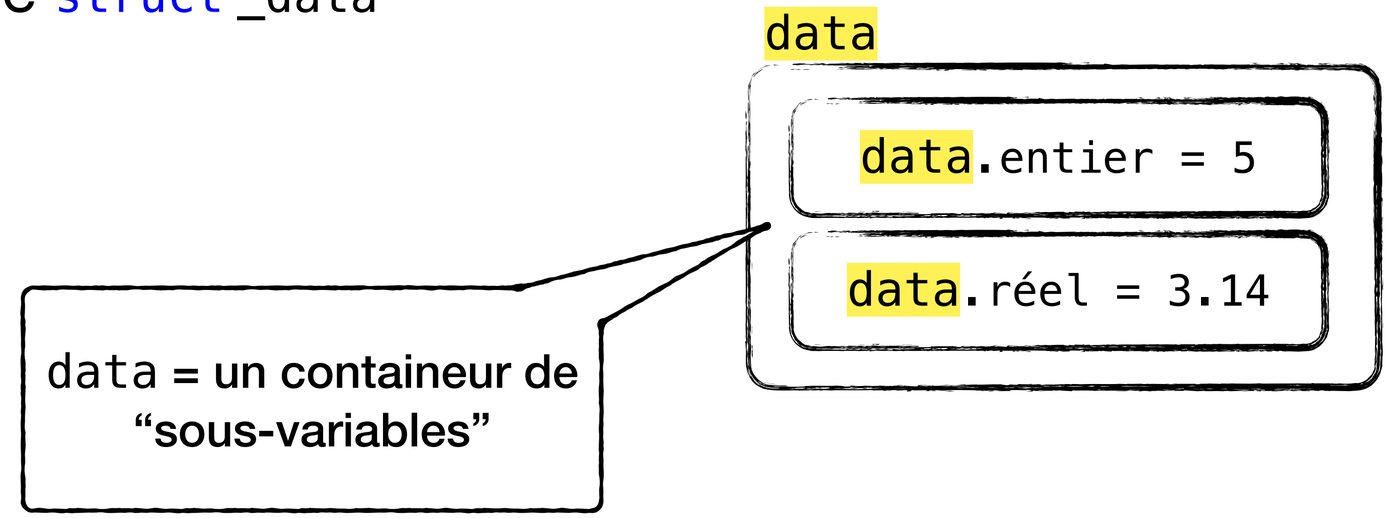
Listes


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE', LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING.
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Rappel struct

- `struct _data` est un type composé
- `data_t` est un synonyme de `struct _data` donc aussi un type
- `data` est une variable de type `struct _data`

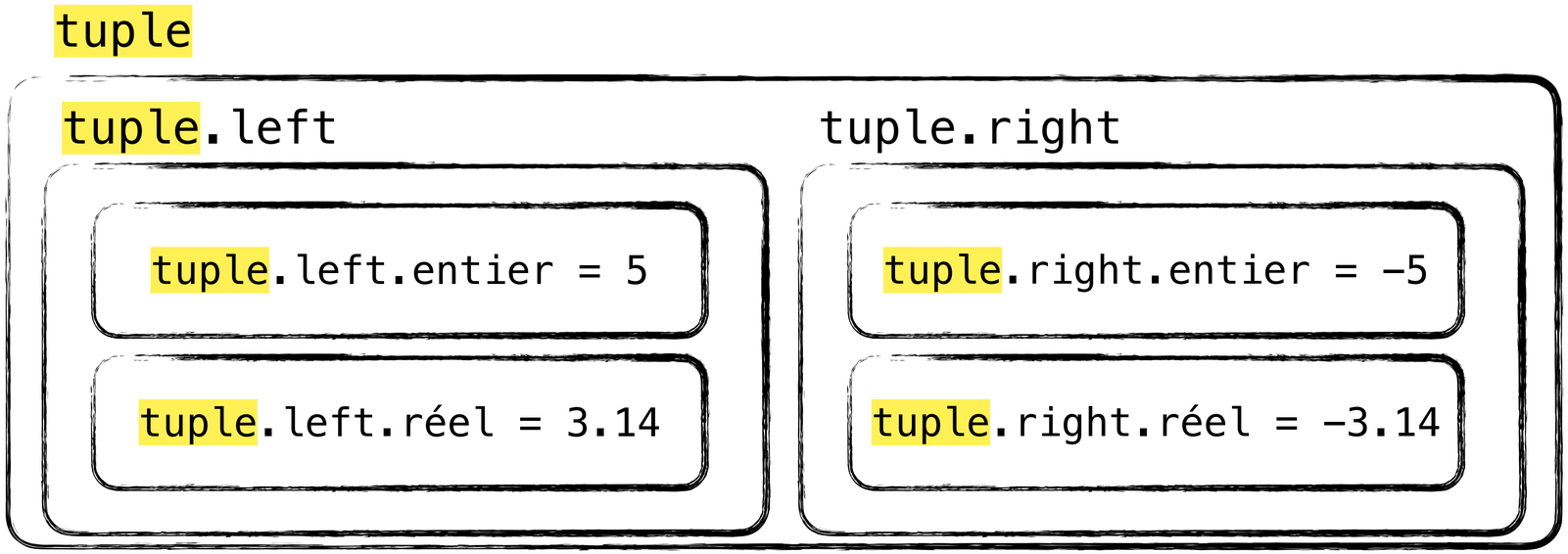
```c
typedef struct _data
{
  int entier;
  float réel;
} data_t;

data_t data = {5, 3.14};
```



Rappel struct

- `struct _tuple` est aussi un type composé
- Ses deux membres ont des types composés
- `tuple` est une variable de type `struct _tuple`

```c
typedef struct _tuple
{
  data_t left;
  data_t right;
} tuple_t;

tuple_t tuple = {
  {5, 3.14}, {-5, -3.14}
};
```



PTT

```c
void temps_total(
  const data_t *pdata,
  int *total_minutes)
{
  for (int i = 0; i < 10; i++)
    total_minutes[i] = 0;

  call_record_t* current =
    pdata->records;
  while (
    current < pdata->records + pdata->M)
  {
    total_minutes[current->no_client]
      += current->minutes;
    current++;
  }
}
```

```c
typedef struct call_record
{
    int no_client;    // numéro client
    int no_tel_appel; // numéro de tel appelé
    int date;         // la date
    int minutes;      // la durée de l'appel
} call_record_t;

typedef struct tarif
{
    int tel;                // numéro de tel
    float chf_par_minute; // tarif
} tarif_t;

typedef struct data
{
    int M, N;
    call_record_t *records;
    tarif_t *tarifs;
} data_t;
```

Erratum

- Sa taille n'est pas vraiment 9 (=3 + 2 + 4)

```
printf("chem_element_t: %ld\n",
       sizeof(chem_element_t));
// Affiche 12
```

- Les compilateurs alignent (souvent) la mémoire à des multiples de 4
- Padding…
- Il faut **toujours** compter sur `sizeof`

```
typedef struct chem_element
{
  char symbol[3];
  short at_no;
  float at_mass;
} chem_element_t;
```

Tableaux statiques

- Pour un type T nous pouvons définir un tableau de N éléments de type T

```
T tableau_statique[10]; // variable de type T[10]
```

- N est une constante
- Parfois une variable est permise par certains compilateurs

\+ Facile à déclarer et à utiliser

\- Durée de vie limitée au bloc/fonction

\- Taille fixe - on ne peut pas la changer

Tableaux dynamiques

- Pour un type T nous pouvons allouer un tableau de taille N de type T*

```
T *tableau_dynamique = malloc(N * sizeof(T)); // variable de type T*
```

- N peut être une variable

+ Durée de vie illimitée (jusqu'à la fin du programme)

- Gestion de la mémoire: allouer le bon nombre d'octets, libérer la mémoire…

- Taille fixe - on ne peut pas la changer sans tout recommencer

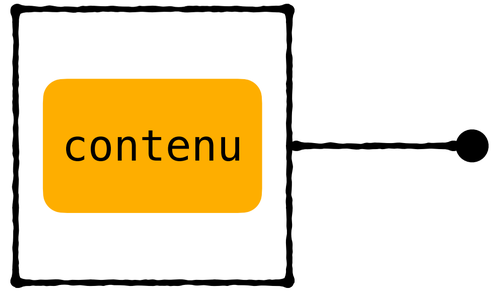
Stockage de taille flexible

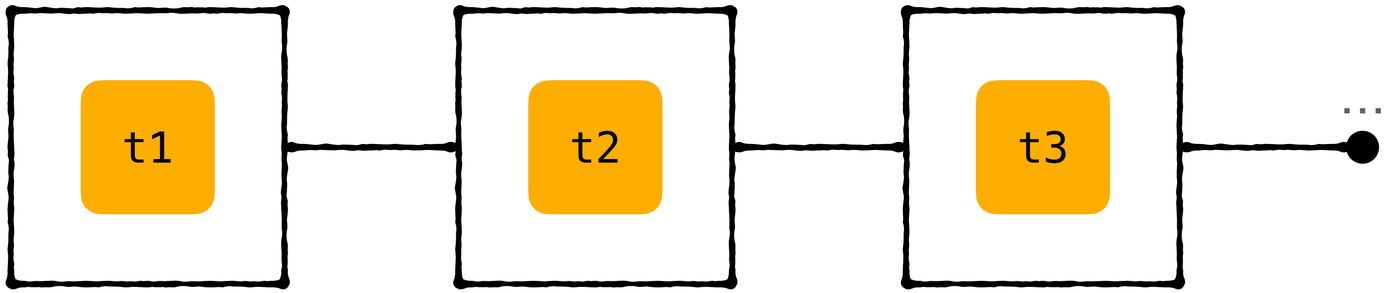
Spécifications

- Peut-on définir une structure de données de taille variable?
- On a des éléments d'un certain type T qu'on veut stocker
- On ne sait pas combien il y en a
- On les lit un par un dans un ordre quelconque
- On veut pouvoir rajouter/enlever des éléments sans tout recréer

L’idée

- Définir une “cellule” qui
  - peut stocker un objet de type T
  - peut être “reliée” à une autre cellule
- On veut aussi pouvoir enlever une cellule





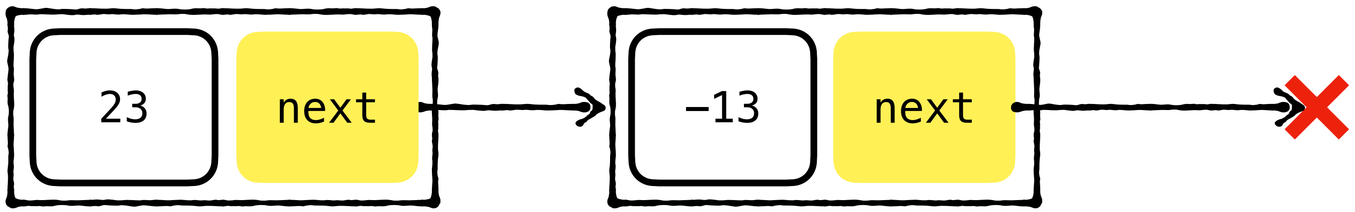
Une cellule d'entiers

Implémentation

```
typedef struct _cell
{
    int contenu;
    ??? next;
} cell_t;
```

À la place de ??? on devrait mettre…

Option 1	Option 2	Option 3
`struct _cell`	`struct _cell*`	`cell_t*`



Une cellule d’entiers

Option 1

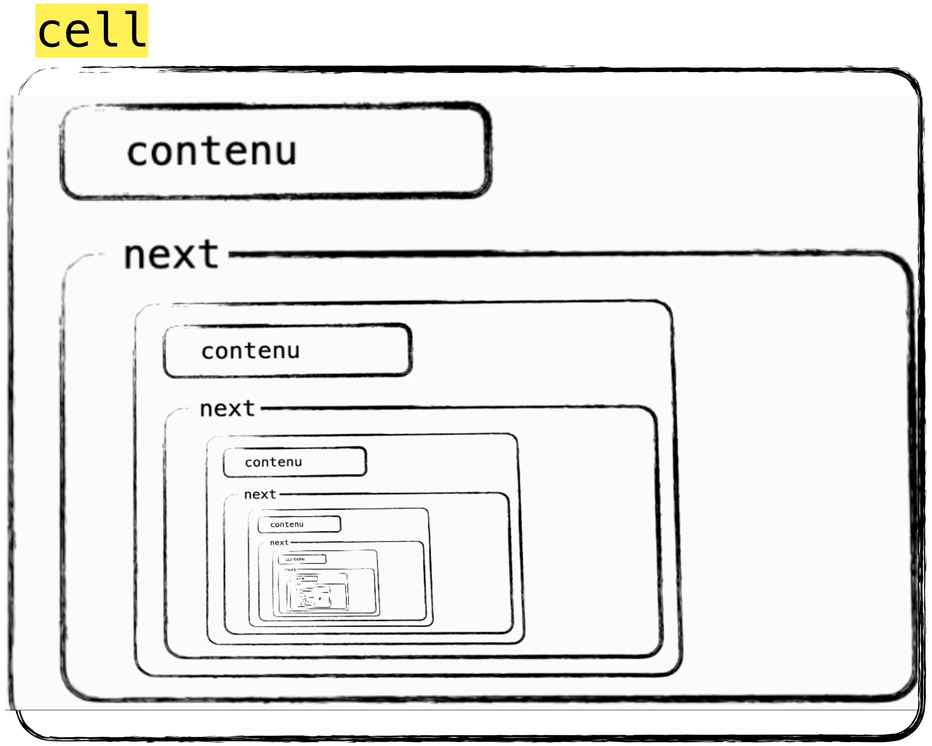
Cela créerait un type de **taille infinie** !

```
typedef struct _cell
{
    int contenu;
    struct _cell next;
} cell_t;
```

Option 1
`struct _cell`

```
test.c:45:18: error: field has incomplete type 'struct _cell'
    struct _cell next;
```

= on veut utiliser un type qui n’est pas encore complètement défini



https://javier.xyz/droste-creator

Une cellule d’entiers

Option 3

```
typedef struct _cell
{
    int contenu;
    cell_t* next;
} cell_t;
```


```
shallow.c:45:5: error: unknown type name 'cell_t'
    cell_t *next;
    ^
```

Le type `cell_t` est inconnu — c’est un synonyme qui est en train d’être défini...

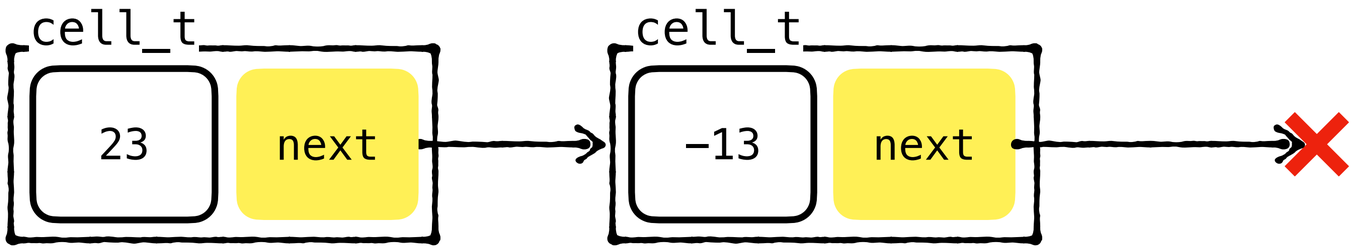
Une cellule d'entiers

Option 2

À la place de ??? on devrait mettre…

```
typedef struct _cell
{
    int contenu;
    struct _cell* next;
} cell_t;
```

Option 2
`struct _cell*`



Que représente ❌ ?

NULL

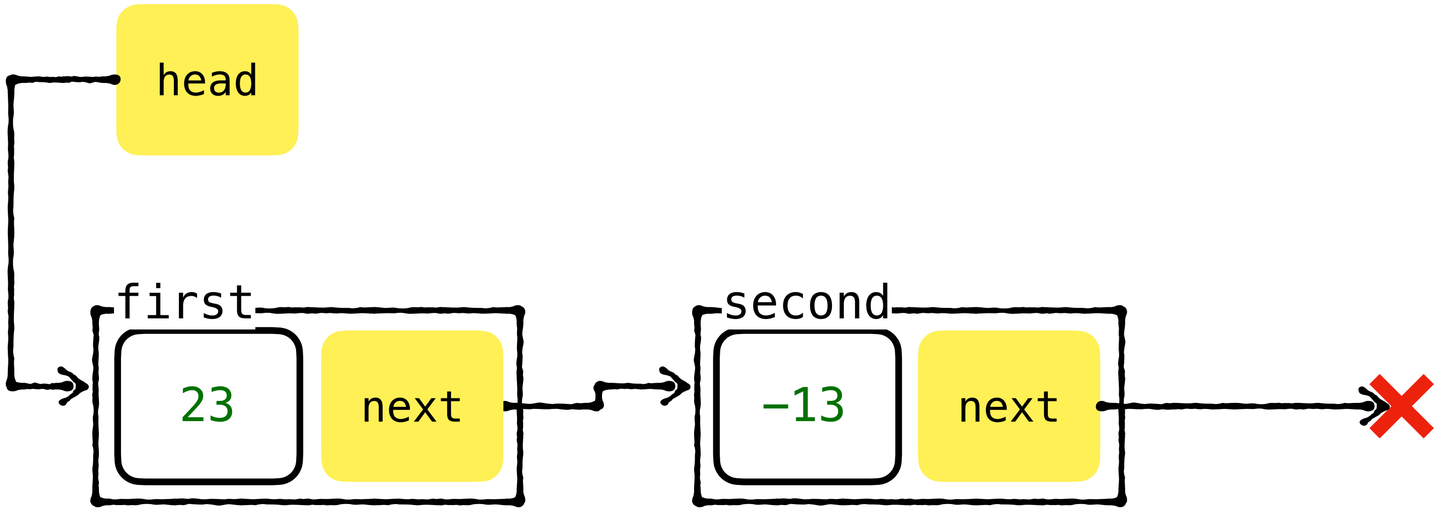
- Souvent quand un pointeur n’est pas utilisé, on lui affecte la valeur NULL
- Comme “zéro” mais pour les pointeurs
- On peut tester si un pointeur est égal à NULL

```
typedef struct _cell
{
    int contenu;
    struct _cell* next;
} cell_t;
```

Exemple

Statique

```
typedef struct _cell
{
    int contenu;
    struct _cell* next;
} cell_t;

cell_t first, second;

first.contenu = 23;
first.next = &second;

second.contenu = -13;
second.next = NULL;

cell_t *head = &first;
```



Exemple

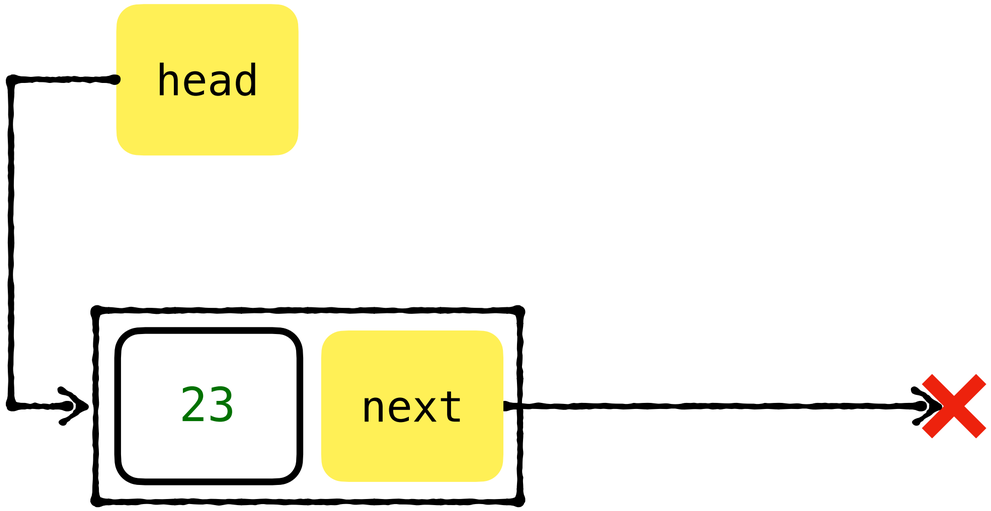
Dynamique


```c
typedef struct _cell
{
    int contenu;
    struct _cell* next;
} cell_t;

cell_t *head = NULL;
```

Exemple

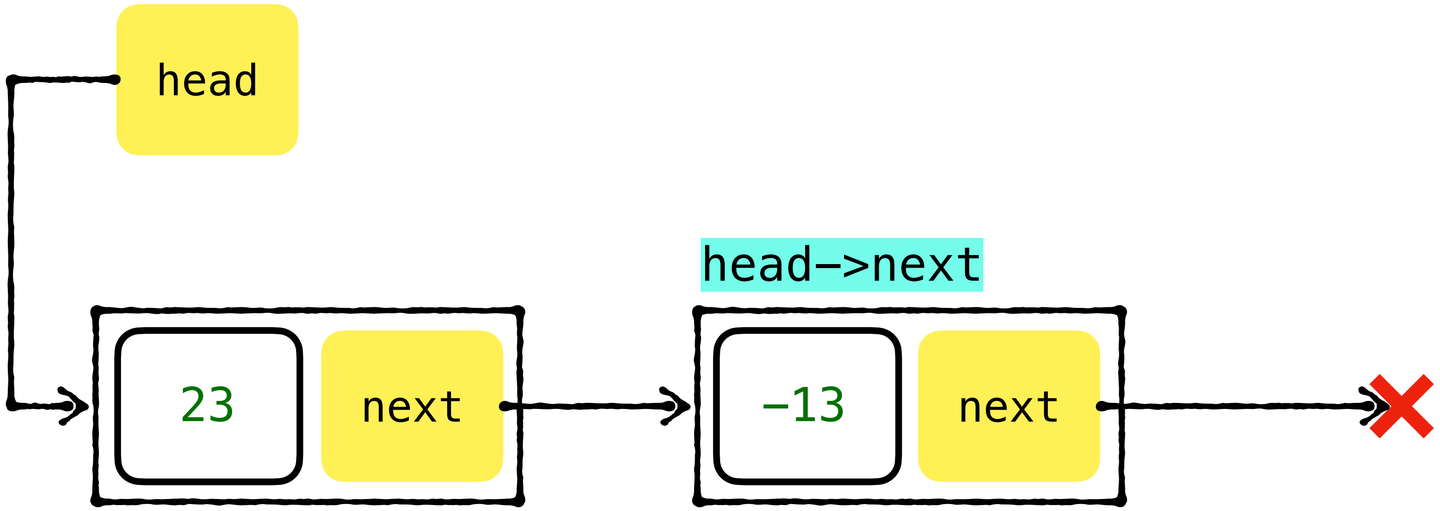
Dynamique

```c
typedef struct _cell
{
    int contenu;
    struct _cell* next;
} cell_t;

cell_t *head = NULL;

head = malloc(sizeof(cell_t));
head->contenu = 23;
head->next = NULL;
```



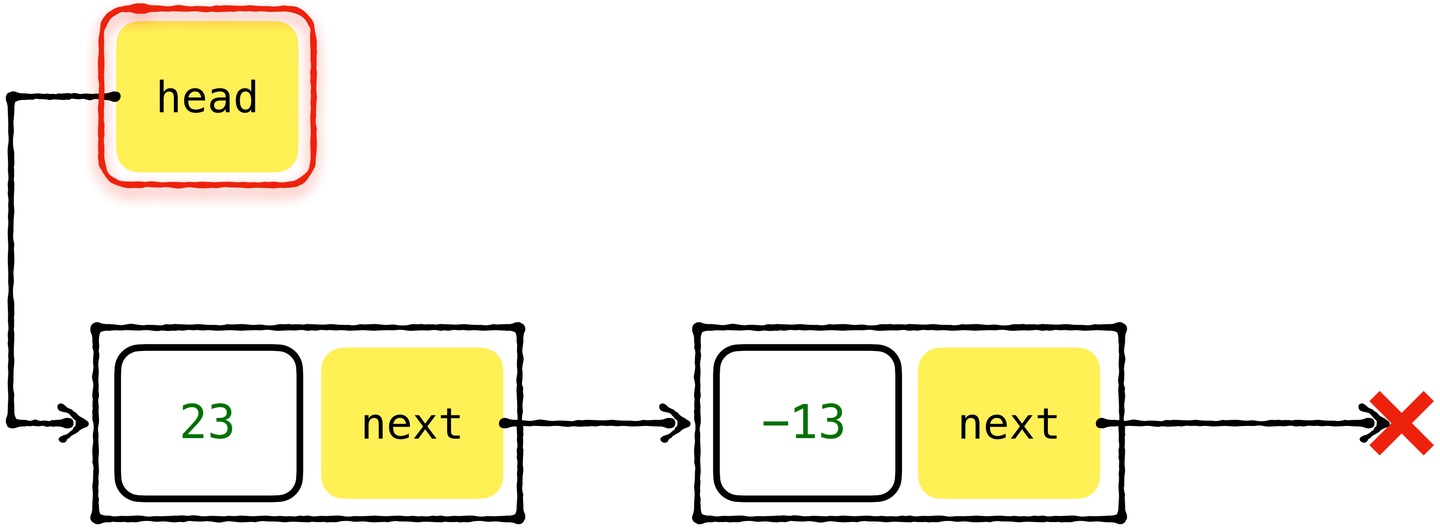
Exemple

Dynamique



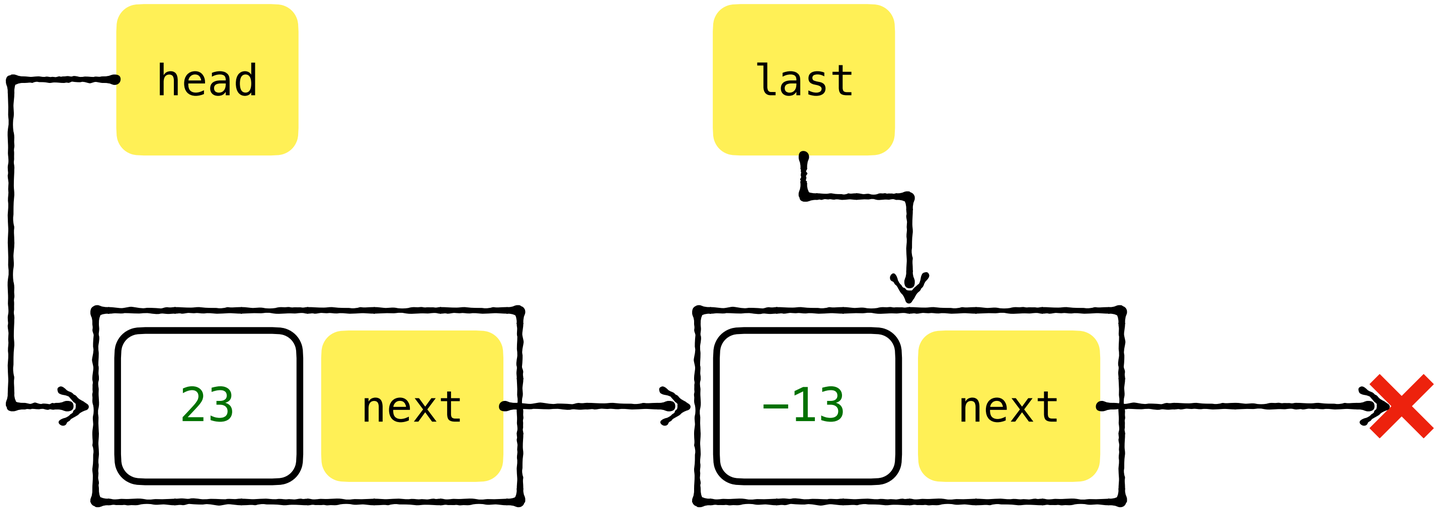

```c
typedef struct _cell
{
    int contenu;
    struct _cell* next;
} cell_t;

cell_t *head = NULL;

head = malloc(sizeof(cell_t));
head->contenu = 23;
head->next = NULL;

head->next = malloc(sizeof(cell_t));
head->next->contenu = -13;
head->next->next = NULL;
```

Tête de liste



- La variable `head`
- Pointe vers le début
- Il ne faut pas perdre la tête…
- Si on la “perd” (on lui affecte une autre valeur), alors **on ne peut plus récupérer la liste**

Fin de liste



- On veut agrandir la liste
- Une façon de faire:
  - Chercher à partir de `head` jusqu'à ce que `next` soit `NULL`
- Enfiler un nouveau élément coûte un temps de parcours ~ taille de la liste
- Alternative: garder (et maintenir!) un pointeur vers le dernier élément

Mettre tout ensemble

```
typedef struct _list
{
    cell_t *head;
    cell_t *last;
} list_t;
```

- On peut définir la liste vide:

```
const list_t liste_vide = {NULL, NULL};
```

- Chaque fois qu'on veut initialiser une liste, il faudrait faire

```
list_t nouvelle_liste = liste_vide; // Valide, car on peut affecter des struct
```

Une nouvelle structure de données!

- Liste chaînée = *linked list*
- Caractéristiques:

  \+ Peut grandir et rétrécir facilement

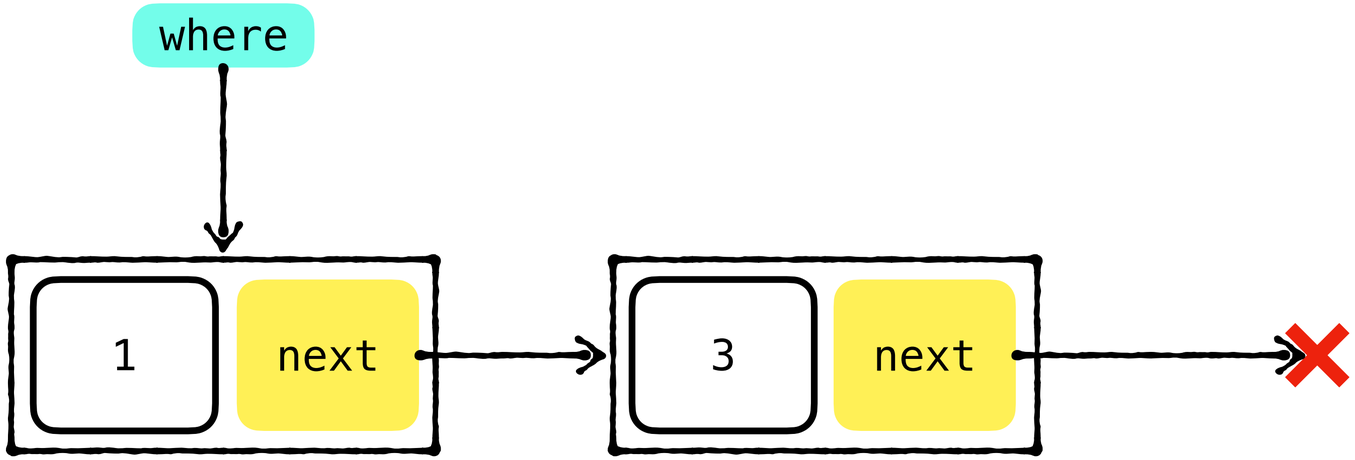
  \+ Insertion d'un élément en temps constant (même au milieu!)

  \- Temps linéaire en la taille pour accéder à un élément arbitraire

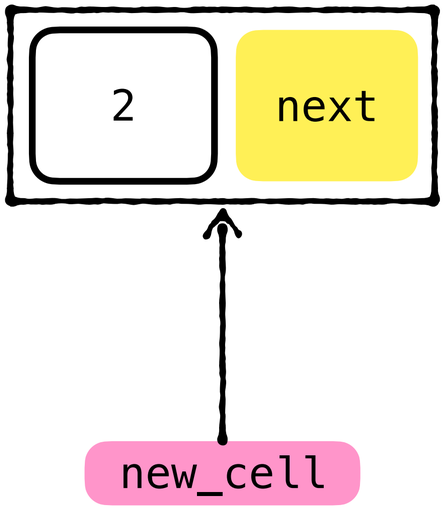
Opérations

```
void insert_head(list_t *plist, int valeur);

int delete_head(list_t *plist);

int insert_after(list_t *plist, cell_t *where, int valeur);

int delete_after(list_t *plist, cell_t *where);

cell_t* find_first(const list_t *plist, int valeur);

void delete_list(list_t *plist);
```

Insérer

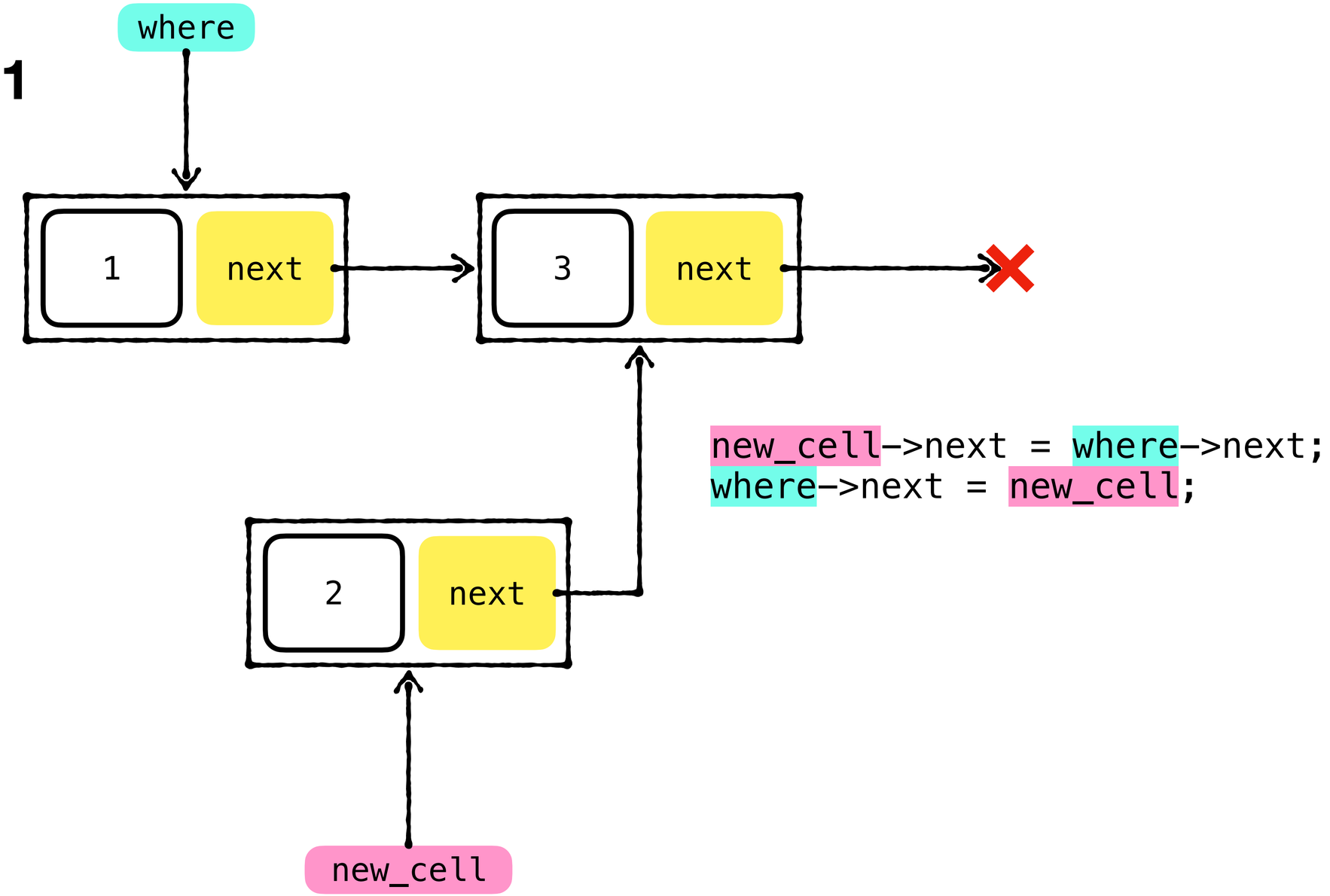



```
new_cell = malloc(sizeof(cell_t));

new_cell->contenu = 2
```



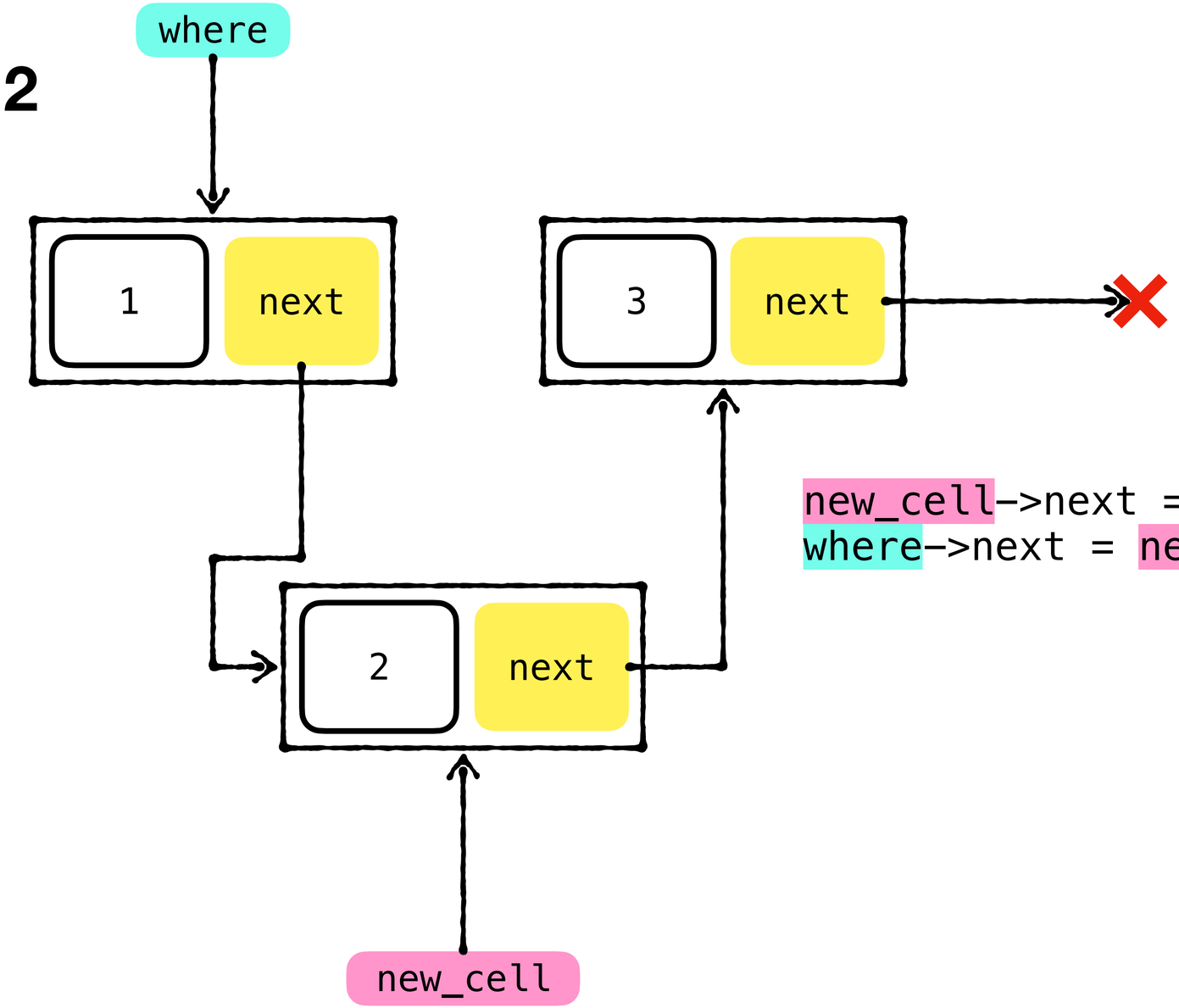
Insérer

Recâblage 1



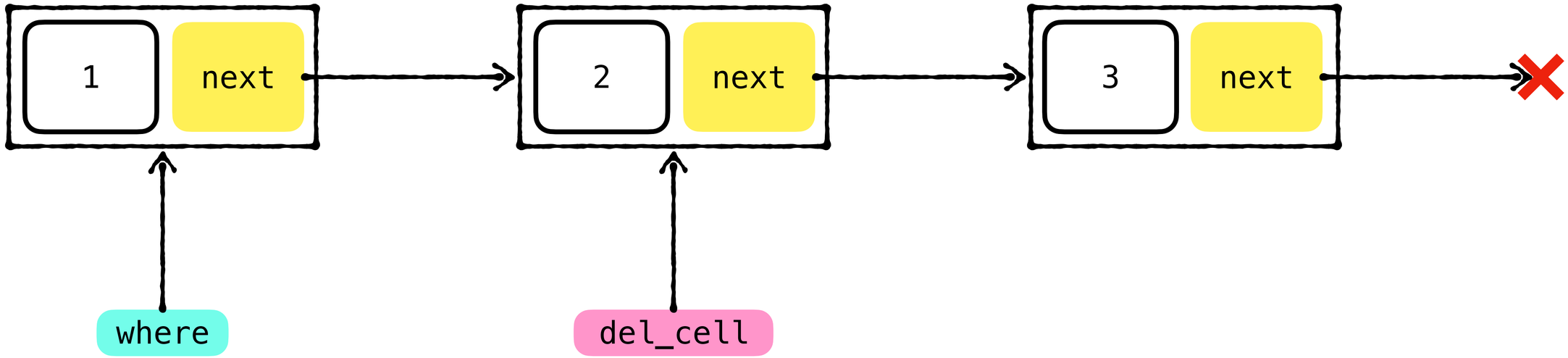
Insérer

Recâblage 2




```
new_cell->next = where->next;
where->next = new_cell;
```

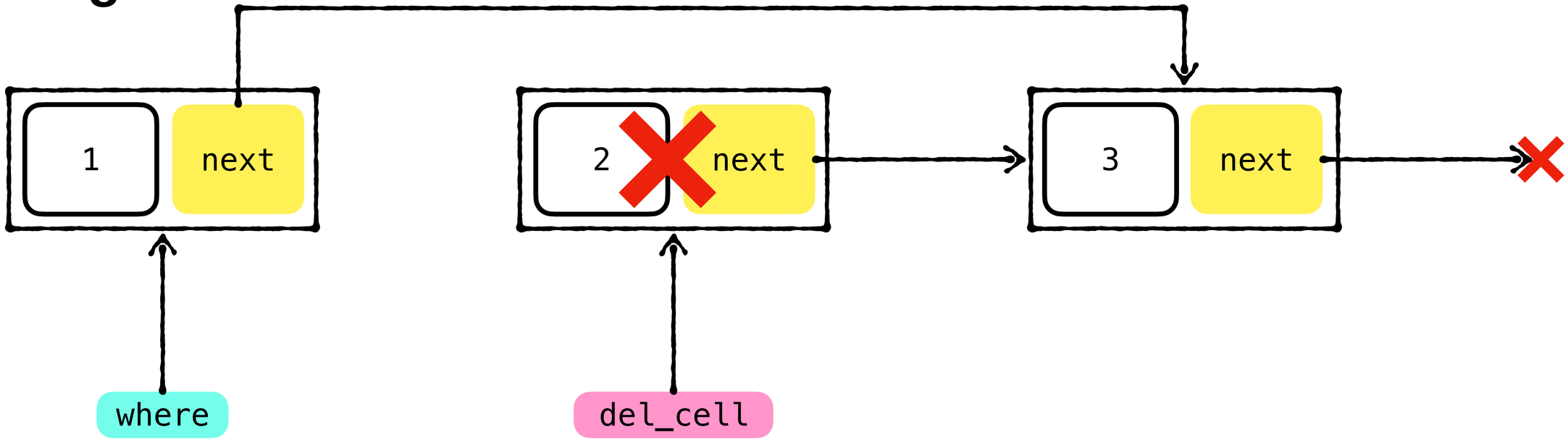
Supprimer




```
where->next = del_cell->next;
```

Supprimer

Recâblage




```
where->next = del_cell->next;
free(del_cell);
```

Programmation Orientée Objet

Object-Oriented Programming (OOP)

- Langages: C++, Java, Scala, Python
- On regroupe ensemble données (=“membres”) et fonctions (=“méthodes”) dans le même type/objet
- (Presque) **tout est un objet**
- Mécanismes:
  - Polymorphisme
  - Héritage
  - …