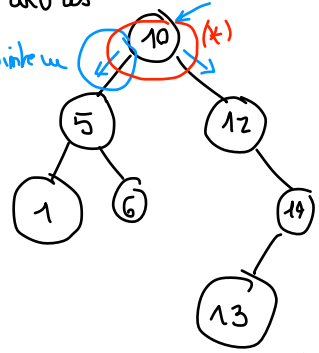
§5. Structures, types, listes

Structures de données C
- → tableaux ~> n données du même type
- → pointeurs ~> adresses

↓
stocker exemples des données de types ≠

base → Algo ↓ structures de données avancées
~> arbres



(*) stocker 1 donnée (10) + 2 pointeurs.
types ≠

I. Structures C

Pour cela en C on peut déclarer des types "struct"
↓
regroupement de données de types ≠
↓
l'accès se fait en assignant un nom / label à chaque champ.

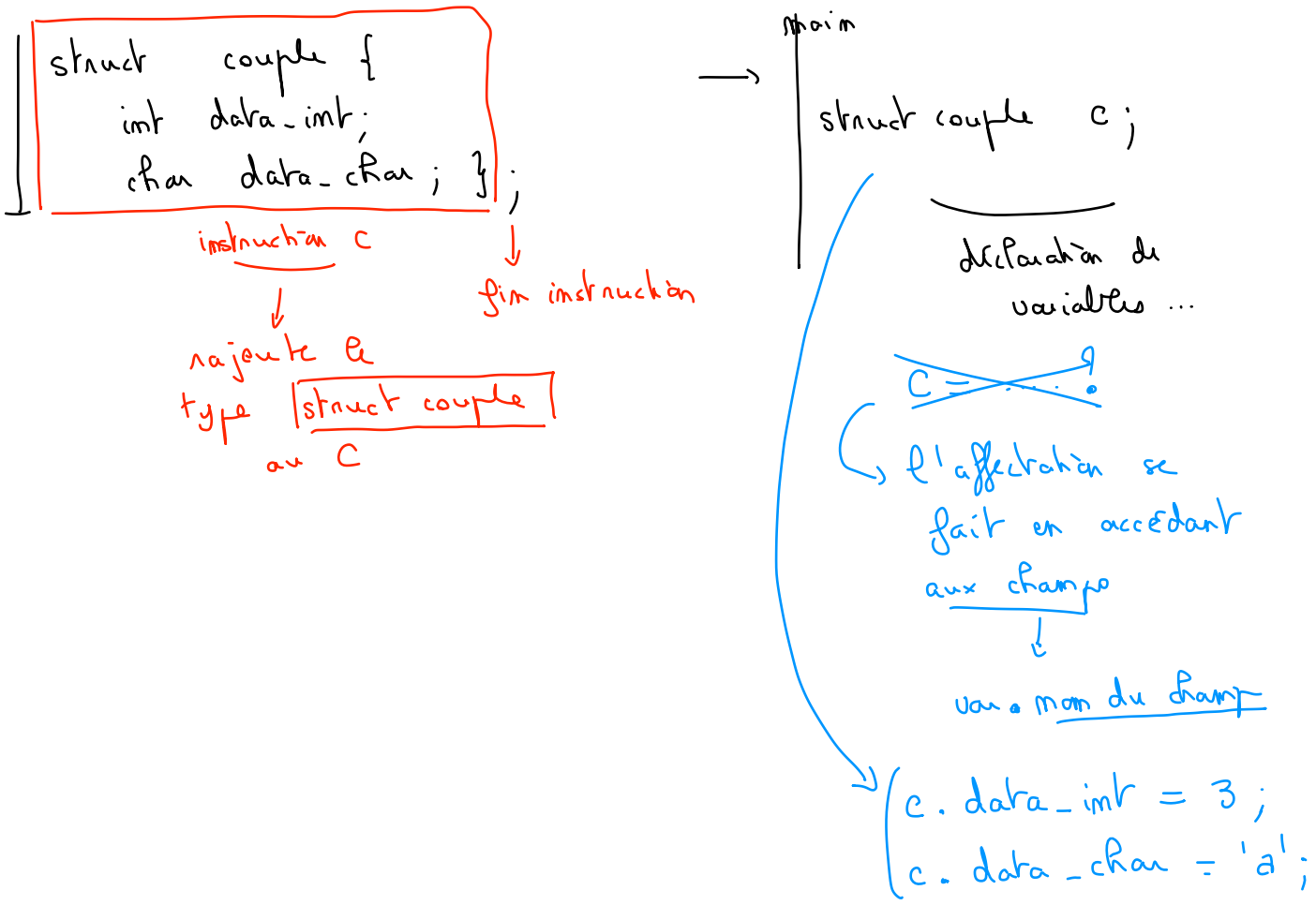
Syntaxe:

```
struct nom_de_la_structure {
    type 1  nom 1 ;  ← champ 1
    type 2  nom 2 ;  ← champ 2
        ⋮              ⋮
};
```

→ définir le type `struct nom_de_la_structure`

→ accès aux champs : var.nom_du champ

ex: couple : entier, caractère ~> 2 champs
↓ data_int
↘ data_char
noms des champs



Pb / inconvénient : "struct ..." très long ....

Redéfinitions de types

Par contre, le C permet de redéfinir des noms pour des types :

Syntaxe :

synonyme

typedef ancien_nom nouveau_nom ;

ex : typedef struct couple mon_couple ;

ex très fréquent :

typedef int bool ; booléens → V/F

En C int ma_fct (...)

↳ ! V/F ou int

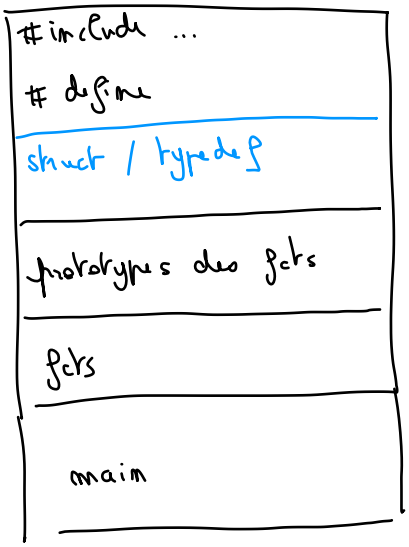
Portable ~> typedef int bool;

⋮

bool ma_fct(....)
{
⋮
}

config pour l'utilisateur
↓
directement V/F comme return

"Logistique"



II. Structure de données listes chaînées

Algo
↳ structure de données avancées
~> listes
~> files d'attente
~> piles
~> autres

+ tas
+ tables hachage / map / tableaux associatifs

① Intuition

Liste: → données à la suite les une des autres (séquentielles 1 → 2 → 3 → 4 ...)

≠ tableaux
→ ajout / suppression d'un élément
⊕ extension / diminution
simples.

→ ajout / suppression en tête / au début seulement.
conséquence

début → fin

ex: liste [1 ; 2 ; 3 ; 4]
ajout suppression — tête

[1;2;3;4] ⇝ ajoute 5 [5;1;2;3;4]

tête

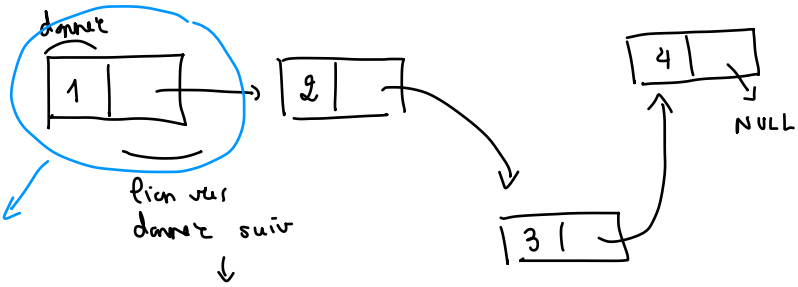
[1;2;3;4] ⇝ récupère la val. de tête 1

⇝ supprime tête [2;3;4]

② Implémentation / type abstrait de données

Liste ⇝ utilise des pointeurs pour relier les données.

[1;2;3;4] ⇝



struct ⇐ (contenir/regrouper
- 1 donnée
- 1 ptrs

Liste ⇝ combinaison struct + pointeurs

↘ fournir à l'utilisateur des fcts pour manipuler les listes → ...

- ajoute
- val. de la tête
- supprime en tête

Bas niveau, code

- ↓ structure de données
- ↓ (*) fct manipulation → type abstrait de données

Utilisateur a besoin de listes pour stocker des données

ex: dictionnaire listes de chaines de caract

["a" ; "abc" ; "def"]

Utilisateur ↓ utilise les listes → sans utiliser les ptrs seulement les fct (*)

pour faire autre chose

utilisent les listes

Dictionnaire → ajouter mots
→ voir si un mot est ds le dico ....

fonctions s/ dico

Type abstrait de données Listes (TAD)

→ ensemble de fct de base suffisant pour manipuler les listes

→ ce sont les seules à manipuler les pointeurs / struct ....

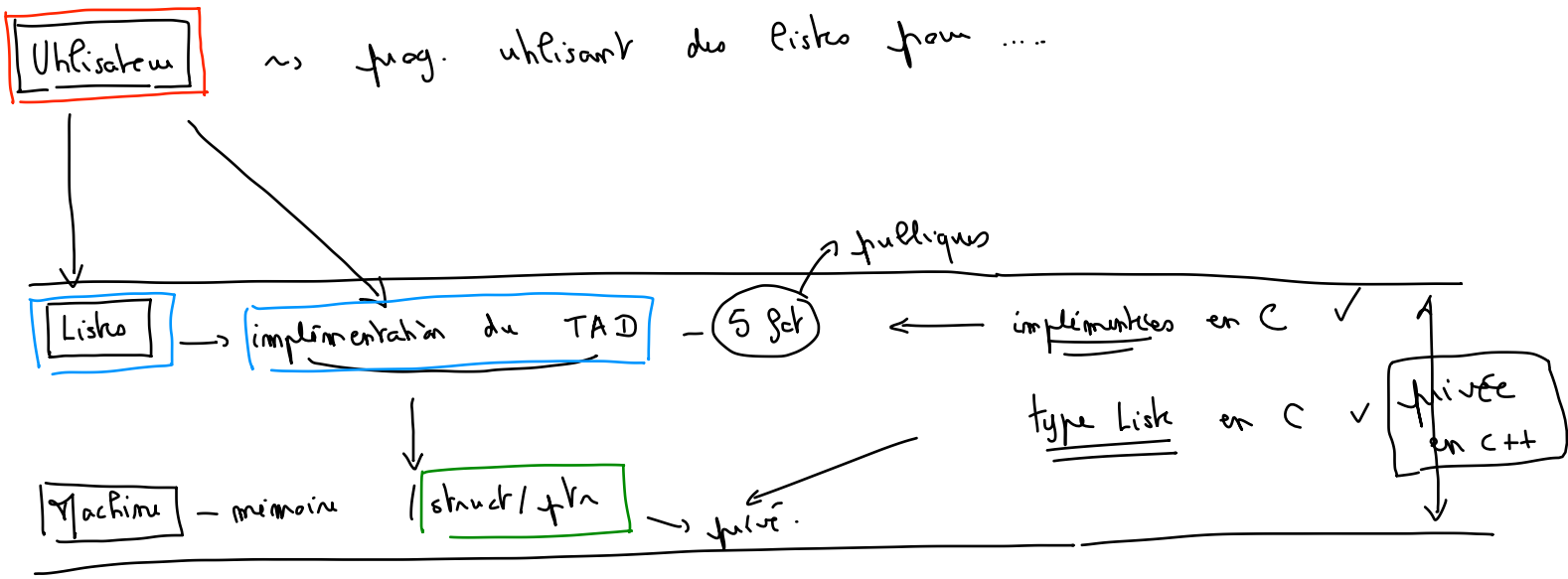
[1 ; 2 ; 3 ; 4]
tête — queue de la liste

TAD Listes

→ créer_liste_vide
→ est_liste_vide
→ ajouter : ajoute 1 élt en tête
→ tête : récupère la valeur de la tête
→ queue : ~~supprime la tête~~ renvoie la suite de la liste sans la tête

---

Utilisateur ~> prog. utilisant des listes pour ....



ex: fct calculant la longueur d'une liste

l = [1 ; 2 ; 3 ; 4] → supprimer tête → liste vide

```
int longueur (Liste l)
{
    int cpt = 0;
    while (! est_liste_vide (l))
    {
        cpt = cpt + 1;
        l = queue (l);
    }
    return cpt;
}
```

NON est_liste_vide

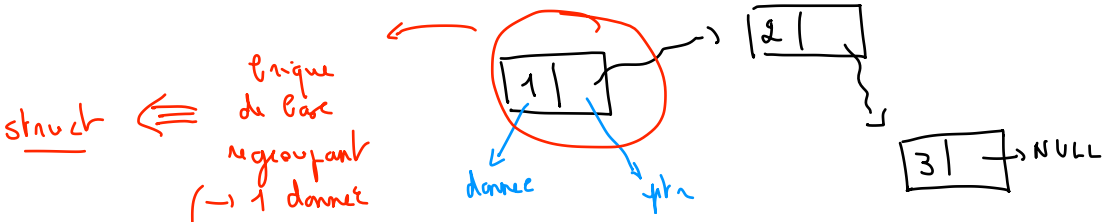
→ copie l

→ retourne l

OK avec TAD.

③ Implémentation du type / fcts du TAD (camemouis...)

Idée : [1;2;3] ⟶ relie les données ⟶ ptrs



maillon ⟶ struct regroupant → 1 donnée
→ 1 ptr sur le maillon suivant

Liste ⟶ manipulée par sa tête

type Liste ≡ maillon (de tête)

maillon contient → 1 donnée
→ 1 ptr

ex: [1;2;3]

↳ codé

liste.

après 3 ⟶ liste vide ↳ NULL

ptr ≠ maillon

si liste = maillon

[] liste vide ↳

on ne peut pas coder []

liste → maillon * : @ du maillon de tête

↳ si la liste est vide : NULL

```
typedef int Tdata ; // type des éléments mis dans liste

struct maillon {
    Tdata  data ;
    struct maillon * next ; } ;
typedef struct maillon * Liste ;
```

champ data

champ next : @ maillon suiv.

Liste ≡ (struct maillon *) car 1 liste ↝ @ maillon de tête

si on veut déclarer un maillon ↝ son type : struct zmaillon ...

long ...

struct zmaillon → maillon

type ≠ maillon

```
typedef int bool ; // par confort on définit le type bool
typedef int Tdata ; // type des éléments mis dans liste

struct zmaillon {
    Tdata  data ;
    struct zmaillon * next ; } ;

typedef struct zmaillon maillon ;
typedef maillon * Liste ;
```

prototypes du TAD → toutes les fonctions renvoient le résultat

↓

ne font pas la modif sur l

↳ ds main l = ...

```
Liste  creer_liste_vide ( void ) ;
bool   est_liste_vide ( Liste) ;
Liste  ajouter (Tdata e , Liste l) ;
Tdata  tete ( Liste l) ;
Liste  queue ( Liste l) ;
```

[1;2;3;4] renvoie 1 ← Tdata

[1;2;3;4] → [2;3;4] ← Liste

fct ajouter

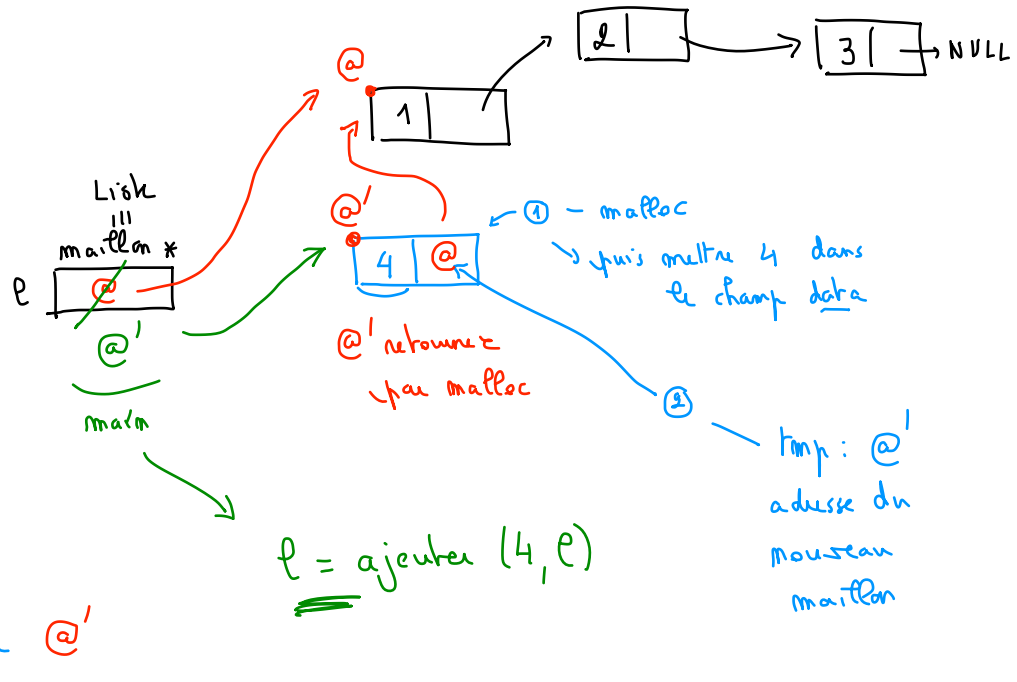
4 [1;2;3]

ajouter (4, l)

① créer 1 maillon pour 4
→ malloc

② remplir le champ next de ce maillon

③ renvoyer la nouvelle liste
↓
renvoyer @'



l = ajouter (4, l)

tmp ~> (adresse d'une structure / ptr sur une structure

accès aux champs →

ici

(*tmp).champ

data
next

↓ compact

tmp → champ

```c
#include <stdio.h>
#include <stdlib.h>

typedef int bool ; // par confort on définit le type bool
typedef int Tdata ; // type des éléments mis dans liste

struct zmaillon {
    Tdata  data ;
    struct zmaillon * next ; } ;

typedef struct zmaillon maillon ;
typedef maillon * Liste ;

// Fonctions du TAD

// Prototypes -> .h
Liste creer_liste_vide () ;
bool est_liste_vide (Liste l) ;
Liste ajouter (Tdata e, Liste l) ;
// .../...

// Implémentations -> .c

Liste creer_liste_vide ()
{
  return NULL ;
}


bool est_liste_vide (Liste l)
{
  /*
  if(l == NULL) // liste vide
  { return 1 ; } // VRAI
  else
  { return 0 ; } // FAUX
  */
  return (l==NULL) ;
}

Liste ajouter (Tdata e, Liste l)
{
  // allouer un maillon
  maillon * tmp ;
  tmp = malloc (sizeof(maillon)) ; // renvoie @' -> stocké dans variable tmp
  // remplir la partie donnée du maillon avec e -> maillon.data
  // tmp : adresse du maillon donc *tmp : maillon
  (*tmp).data = e ;
  // idem : tmp->data
  tmp->next = l ;
  return tmp ;
}

int main ()
{
  Liste l = creer_liste_vide(), l1 ;
  l = ajouter(1,l) ;
  l1 = ajouter(2,l) ;

  return 0 ;
}
```