Langages de Programmation, Interprétation, Compilation

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

LDD3 Informatique, Mathématiques - Magistère Informatique

2025–26

Structures de données et tas mémoire

Données

Les données manipulées par un programme, selon leur nature, peuvent être représentées de différentes manières en mémoire :

- nombre entier ou booléen : représenté par un unique mot mémoire (par exemple de 4 octets),
- donnée plus complexe (tableau, *n*-uplet, structure, objet ou fermeture) représentée par plusieurs mots consécutifs, formant un bloc, accessible par son adresse
- une structure plus riche (liste chaînée, arbre ou table de hachage) représentée par plusieurs blocs, non consécutifs, reliés par des pointeurs.

Tableaux

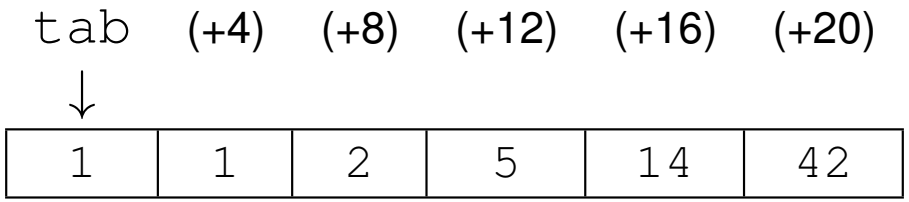
Un tableau est représenté par une séquence de mots consécutifs en mémoire, et identifié par l'adresse de son premier élément.
Une définition C comme

```
int tab[] = {1, 1, 2, 5, 14, 42};
```

ou son équivalent ImpScript

```
let tab = [1, 1, 2, 5, 14, 42];
```

chaque élément du tableau occupe quatre octets.



Tableaux : accès

- le premier élément du tableau est directement à l'adresse donnée par la variable tab,
- le dernier est à l'adresse tab+20,
- on accède à chaque élément en partant de l'adresse de base tab, et en ajoutant un décalage correspondant à l'indice de la case cherchée (par multiples de 4 octets).

Tableaux alloués statiquement

- le tableau tab précédent est global, défini à la racine du programme
- comme les autres variables globales, ce tableau peut être placé dans les données statiques.
- au lieu de lui allouer un unique mot mémoire on en donne six consécutifs : un pour chaque case du tableau.
- On initialise un tel tableau en MIPS avec le fragment suivant.

```
        .data
tab:    .word   1 1 2 5 14 42
```

Tableaux statiques : initialisation

- besoin des valeurs initiales du tableau à la compilation
- besoin de connaître au moins la taille fixe du tableau
- en C

```
int tab[6];
```

- en ImpScript

```
let tab = Array(6);
```

- traduction MIPS : une séquence de six valeurs (choix arbitraire du contenu).

```
        .data
tab:    .word   0 0 0 0 0 0
```

- l'étiquette tab désigne l'adresse de base du tableau statique

Tableaux statiques : accès

- l'étiquette tab sert pour calculer l'adresse de chaque élément.
- accès au quatrième élément : décalage de 12

```
la    $t0, tab
lw    $t1, 12($t0)
```

- Si l'indice auquel chercher est dans un registre $a0, on calcule dynamiquement le bon décalage.
  - multiplier la valeur de $a0 par 4 (décalage de 2 bits vers la gauche, sll),
  - puis l'ajouter à l'adresse de base
  - accès direct ensuite

```
la    $t0, tab
sll   $a0, 2
add   $t0, $t0, $a0
lw    $t1, 0($t0)
```

Tableaux alloués dynamiquement

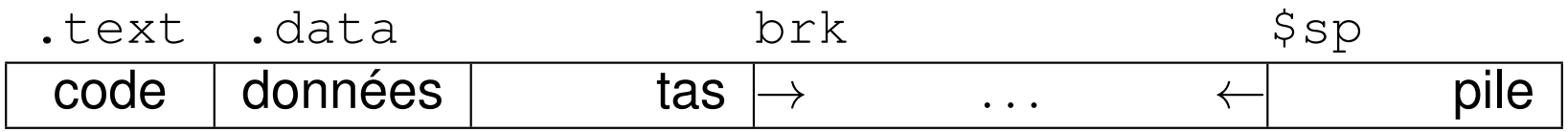
- tableau créé dynamiquement, taille inconnue à la compilation

```
int f(int n) {
  int tab[n];
  ...          }
```

- *n* mots consécutifs en mémoire, identifié par une adresse
- sur la pile (défaut en C) : durée de vie limitée, détruit à la fin de l'appel.
- donnée persistante (ne disparaît pas) : rangée sur le tas

Organisation du tas

le tas se trouve entre les données statiques et la pile.



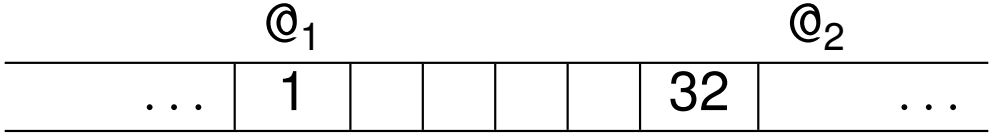
- plus petite adresse : immédiatement après la zone des données statiques.
- pointeur brk (memory break) : fin du tas, (1ère adresse au-delà).
- dernier mot du tas : adresse brk–4.
- espace vide entre le tas brk (inclus) et la pile $sp (exclu)
- si besoin de plus de place : incrémenter le pointeur brk,
- pointeur brk géré par l'OS : déplacement par appel système sbrk (code 9)
- dans MARS :
  - argument $a0 = nombre d'octets dont brk doit être incrémenté.
  - résultat dans $v0 : ancienne valeur de brk, 1ère adresse de la zone ajoutée.

Tableaux alloués : exemple

on alloue sur le tas de l'espace pour un tableau de taille 6, avec 1 dans la première case et 32 dans la sixième.

instruction	$a0	$v0	$t0	brk
				0*x*10040000
`li $a0, 24`	24			0*x*10040000
`li $v0, 9`	24	9		0*x*10040000
`syscall`	24	0*x*10040000		0*x*10040018
`li $t0, 1`	24	0*x*10040000	1	0*x*10040018
`sw $t0, 0($v0)`	24	0*x*10040000	1	0*x*10040018
`li $t0, 32`	24	0*x*10040000	32	0*x*10040018
`sw $t0, 20($v0)`	24	0*x*10040000	32	0*x*10040018

La forme du tas associée à cet exemple est la suivante, où l'adresse $@_1$ est la position d'origine de brk (0x10040000) et $@_2$ est la nouvelle position après appel à sbrk (0x10040018).



Extension du compilateur ImpScript

4 constructions

- accès (en lecture) à un élément d'un tableau
- modification d'une case d'un tableau
- création d'un nouveau tableau (spécifié par une liste explicite d'éléments)
- création d'un tableau spécifié par sa taille

```
type instr = ...
  | ArrSet of expr * expr * expr   (* e1[e2] = e3;    *)

type expr = ...
  | ArrGet of expr * expr          (* e1[e2]          *)
  | Array  of expr list            (* [e1, ..., eN]   *)
  | EArray of expr                 (* Array(e)        *)
```

Extension du compilateur ImpScript

Calcul (dans $t0) de l'adresse de la case e1[e2], (égale à e1 + 4*e2)

```
let rec tr_access e1 e2 =
  tr_expr e1 @@ push t0
 @@ tr_expr e2 @@ sll t0 t0 2
 @@ pop t1 @@ add t0 t1 t0

let tr_expr = ...
  | ArrGet(e1, e2) ->
     tr_access e1 e2 @@ lw t0 0(t0)

let tr_instr = ...
  | ArrSet(e1, e2, e3) ->
     tr_access e1 e2 @@ push t0
    @@ tr_expr e3
    @@ pop t1 @@ sw t0 0(t1)
```

Création d'un tableau non initialisé

- appel à sbrk
- le nombre d'éléments du tableau donné par l'expression e (multiplié par 4 pour obtenir le bon nombre d'octets).
- dans ce modèle, la valeur d'une expression désignant un tableau est un pointeur vers ce tableau (le tableau lui-même n'étant pas une valeur).
  - En C : les tableaux sont des valeurs, si besoin on manipule explicitement des pointeurs
  - En Caml, Java : toute valeur est représentée par un mot (soit une donnée de base comme un entier, soit un pointeur vers un bloc mémoire)
- le « résultat » de la création est un pointeur vers la première case du tableau créé, que l'on transfère dans $t0.

```
let rec tr_expr = ...
  | EArray e ->
     tr_expr e
    @@ sll a0 t0 2 @@ li v0 9 @@ syscall
    @@ move t0 v0
```

Création d'un tableau initialisé

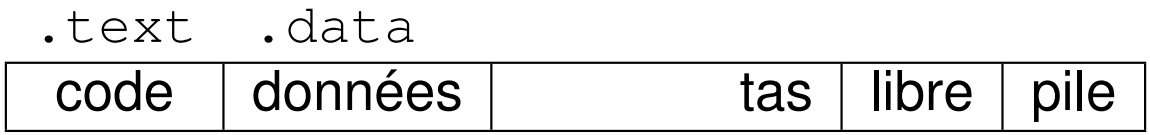
- allocation du tableau : la taille est connue statiquement
- initialisation : évaluer chaque élément et l'écrire dans la bonne case.
- enregistrer sur la pile l'adresse du tableau créé le temps de finir l'initialisation
- utiliser une fonction auxiliaire write_elts

```
    | Array(elts) ->
       let n = List.length elts in
       li a0 (4*n) @@ li v0 9 @@ syscall
      @@ push v0
      @@ write_elts elts 0
      @@ pop t0

and write_elts elts offset = match elts with
    | [] -> nop
    | e::tl ->
       tr_expr e
      @@ lw t1 0(sp)  (* rappel de l'adresse de base *)
      @@ sw t0 offset(t1)
      @@ write_elts tl (offset+4)
```

Partie 3 : Typage, fonctions, mémoire

Organisation du tas



- région de la mémoire adaptée au stockage des données persistantes.
- région divisée en blocs, gérée par un programme dédié qui permet de :
  - demander des blocs de mémoire pour stocker de nouvelles données,
  - libérer des blocs de mémoire affectés à des données inutiles, et les réutiliser pour d'autres données.
- opérations dynamiques, pendant l'exécution du programme principal.

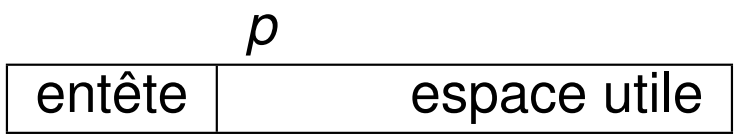
Gestion de la mémoire dynamique.

Différentes formes possibles

- En C, on utilise la bibliothèque malloc : fonctions malloc et free.
- En caml, java, python, un gestionnaire automatique appelé GC (garbage collector, ou glaneur de cellules) est intégré à l'environnement d'exécution.
  Il tourne en parallèle du programme principal et identifie puis libère les parties du tas qui ne sont plus utiles.
- Détail du fonctionnement des opérations manuelles malloc et free, qui peuvent ensuite servir de base à la réalisation d'un gestionnaire automatique.

Bloc mémoire : définition et format

- unité de gestion de la mémoire dynamique,
- zone de mémoire contiguë, qui peut être libre ou utilisée,
- constitué d'un entête et d'un espace utile, identifié par un pointeur.



- l'espace utile : partie effectivement utilisée pour stocker des données,
- entête : informations additionnelles pour le gestionnaire de mémoire.
  - statut libre ou réservé du bloc : un bit d'information (`1` pour un bloc réservé et `0` pour un bloc libre),
  - la taille totale (en octets) du bloc (espace utile + entête)
- accès au contenu d'un bloc via le pointeur *p* (premier octet utile)
  - contraintes sur l'adresse désignée par le pointeur *p* : ici, multiple de 8
  - entête à l'adresse *p* `-4`, mot d'indice *i* à l'adresse *p* `+4*`*i*.

Bloc mémoire : entête

Représentation de la taille + statut dans un seul mot

- taille multiple de 8 : 3 bits de poids faible à 0
- placer le bit de statut à la place du dernier bit de taille
- utilisation du ou bit-à-bit $e = t \mid s$.

t	s	e
32	libre	$0b100000$
24	réservé	$0b011001$

- à l'inverse on récupère le statut ou la taille à partir d'un mot d'entête e à l'aide d'opérations de masquage :
  - statut donné par le dernier bit : $s = e$ `& 0x1`,
  - taille obtenue en masquant les trois derniers bits : $t = e$ `& ~0x7`.

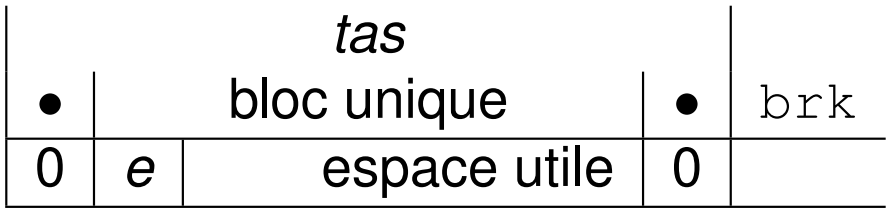
Gestion manuelle : `malloc` et `free`

Deux opérations malloc et free :

- Un appel `malloc(`*n*`)`, avec *n* un entier positif, trouve dans le tas un bloc libre ayant au moins *n* octets utiles, le réserve, et renvoie un pointeur vers ce bloc.
  - si le tas ne contient pas de bloc libre adapté, la fonction peut soit appeller `sbrk`, soit échouer.
- Un appel `free(`*p*`)` libère le bloc de mémoire à l'adresse donnée par le pointeur *p*.
  - suppose que le pointeur *p* désigne un bloc mémoire qui a effectivement été alloué par `malloc`
- On suppose que l'ensemble du tas est formé de blocs consécutifs en mémoire, sans espace libre entre les blocs, et est intégralement géré par `malloc` et `free`.

Initialisation

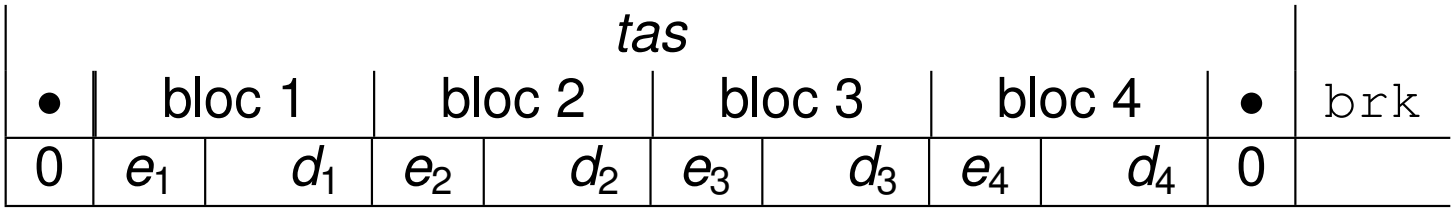
- on demande au système une zone de mémoire à l'aide d'un appel sbrk
- on marque les extrémités de cette zone, par une valeur spéciale (0) dans le premier mot et le dernier mot.
- tout le reste est un gros bloc



- tas de 1024 octets
  - - 2 mots début et fin de tas : bloc 1016 octets
  - - entête de 4 octets : reste 1012 octets utiles
  - bloc vide : statut 0, l'entête *e* vaut 1016.
  - le pointeur *p* vers l'espace utile de cet unique bloc est un multiple de 8 (2 mots après le début du tas - adresse 0x1004000 en MIPS).

Initialisation (suite)

- lorsque le tas est étendu par un nouvel appel à sbrk, on déplace le zéro final pour préserver la forme générale : une séquence de blocs consécutifs, avec extrémités marquées par deux mots nuls.



Fonction `malloc` v1, parcours séquentiel

- appel `malloc(`*n*`)`
- recherche dans le tas un bloc libre suffisamment grand.
- V1 : parcourt séquentiellement tous les blocs du tas
- deux opérations principales :
  1. **Tester un bloc désigné par un pointeur *p*.**
     - décomposer l'entête (*p*`[-1]`), en un bit de statut *s* et une taille *t*,
     - tester si le bloc est libre (`!`*s*),
     - et s'il contient assez d'octets utiles (*n* `<=` *t* `-` `4`).
  2. **Passer au bloc suivant.**
     - ajouter au pointeur *p* sur un bloc la taille *t* de ce bloc : adresse du bloc suivant
     - chaînage implicite des blocs.

Fonction `malloc` v1, stratégies

On peut varier les stratégies :

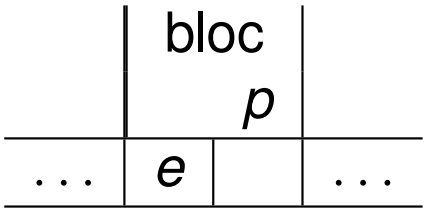
- commencer au 1er et s'arrêter au 1er bloc convenable (first fit),
- commencer au bloc touché le plus récemment et s'arrêter au 1er bloc convenable (next fit),
- chercher le plus petit des blocs convenables (best fit).

si aucun bloc convenable trouvé :

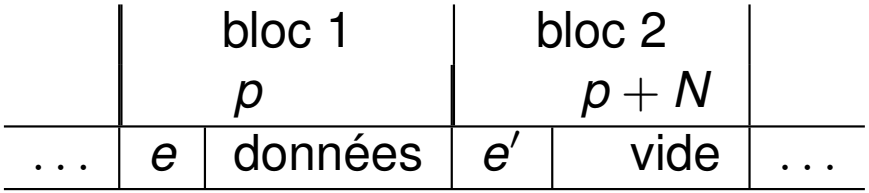
- appeler sbrk pour étendre le tas (tant qu'il reste de l'espace)

Réservation d'un bloc

- identification d'un bloc convenable, pointeur p vers son premier octet utile



- réservation du bloc
  - réserver tout le bloc : modifier le bit de statut ($e \leftarrow e$ `|` `0x1`),
  - si le bloc est grand, le découper en deux blocs, un bloc de taille N, pour $N \geq n + 4$ multiple de 8 qui sera réservé et le reste laissé libre.



Réservation d'un bloc : stratégie

l'allocateur mémoire choisi de découper ou non (et choisit N)

- découper permet d'éviter de sous-employer un bloc,
- amène à terme une fragmentation de la mémoire en de multiples petits blocs difficilement réutilisables.

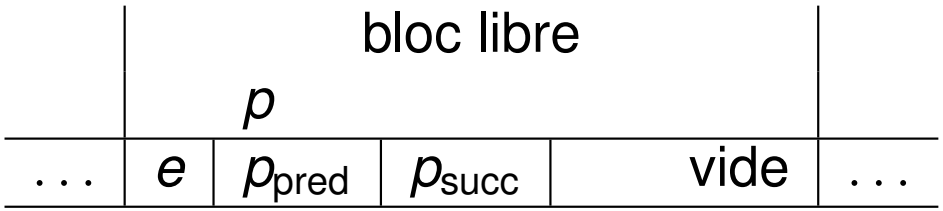
Fonction `free` v1

libérer un bloc désigné par un pointeur p :

- modifier dans son entête le bit de statut, pour le repasser à `0`
- ($e \leftarrow e$ `& ~0x1`).

Fonction `malloc` v2, listes libres

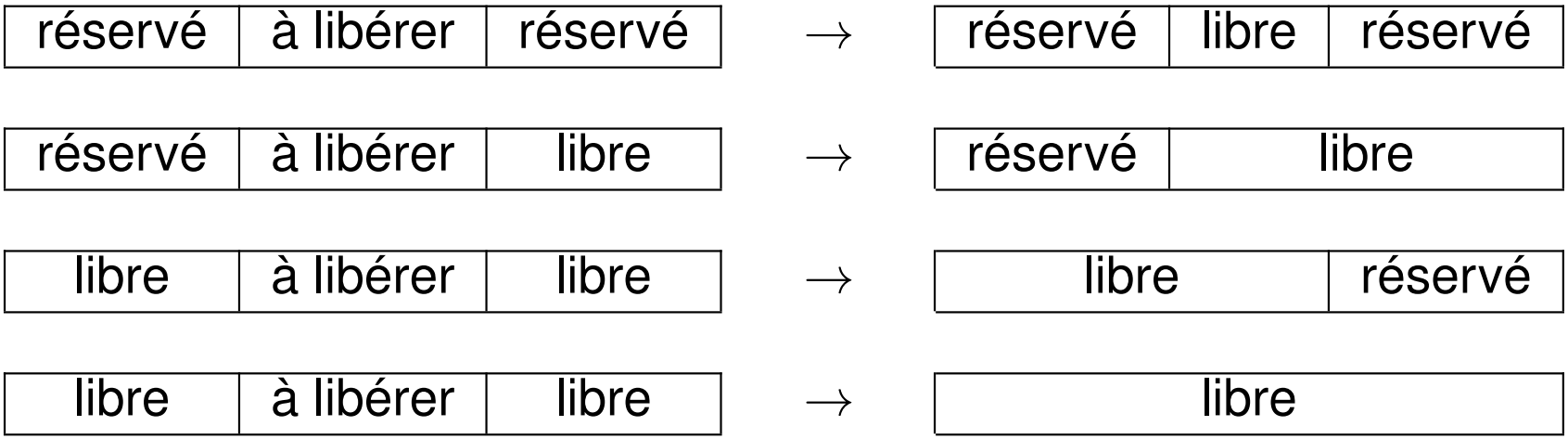
- ne parcourir que les blocs libres
- nécessite de chaîner les blocs libres entre eux
- utiliser les deux premiers champs de l'espace utile du bloc pour les adresses p_{pred} et p_{succ} des blocs libres précédent et suivant



- taille de chaque bloc : au moins 12 octets (en fait 16)
- liste doublement chaînée des blocs libres (free list), mise à jour à chaque changement de statut d'un bloc.
- variable globale pour l'adresse d'un premier bloc libre.
- optimisation : plusieurs listes doublement chaînées de blocs libres, en distinguant plusieurs catégories de tailles (segregated free lists).
- fonction free : réintégrer le bloc libéré dans la liste des blocs libres (par exemple avant le 1$^{\text{er}}$ bloc)

Fonction `free` v2, libérer et fusionner

- défaut : apparition progressive de fragmentation de la mémoire
  - l'allocation coupe les blocs en deux
  - la libération remet en service des plus petits blocs
- solution : fusionner les groupes de blocs libres consécutifs
- intégré à l'opération free : si on trouve un bloc libre juste avant ou juste après le bloc libéré par free, on les fusionne.
  quatre situations possibles

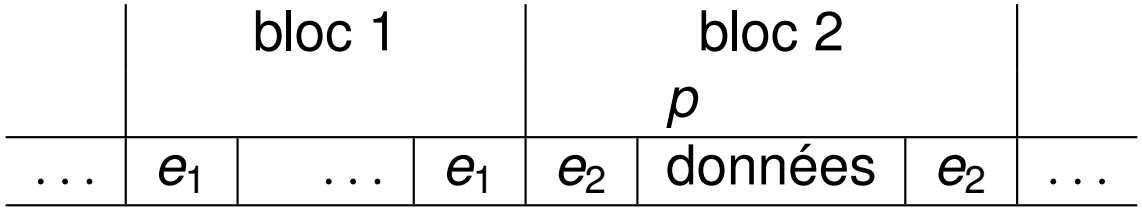


- jamais avoir deux blocs libres consécutifs.

Réaliser les fusions

Consulter les entêtes des blocs situés juste avant et juste après le bloc libéré.

- à partir du pointeur p du bloc à libérer, ajouter la taille du bloc pour trouver l'adresse du bloc suivant
- pour le bloc précédent, on peut dupliquer l'entête en fin de bloc (boundary tags).



On accède alors aux méta-données du bloc précédent en regardant deux mots avant l'adresse p.

- pour éviter de perdre de la place,
  - l'entête n'est dupliquée que pour les blocs libres
  - le statut du bloc précédent (1 bit) est ajouté à l'entête

Bilan

- un bon allocateur de mémoire est un programme assez subtil,
- repose sur des structures de données “cachées” dans les interstices du tas,
- beaucoup de choix de stratégie à faire,
- abondance d’études empiriques sur l’efficacité des différentes stratégies.
- la bibliothèque `malloc.c` de linux contient plus de 5000 lignes de code.

Partie 3 : Typage, fonctions, mémoire

Structures de données composées

Définition de structures de données composées de plusieurs champs, chaque champ étant identifié et accessible par un nom.

- structures en C

```
struct point {  int x;  int y; };
```

- enregistrements en caml (par défaut non mutables)

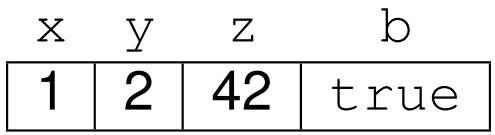
```
type point = { x: int; y: int; }
```

- les objets en java

```
class Point {
  public final int x;
  public final int y;
  public Point(int x, int y)
    { this.x = x; this.y = y; }
}
```

Représentation de structures

- bloc de plusieurs mots mémoire consécutifs,
- valeurs des champs stockées l'une après l'autre
- ordre des champs fixé à la compilation
- chaque champs correspond à une position dans le bloc



$\texttt{x} \mapsto 0$ (décalage 0 octets)
$\texttt{y} \mapsto 1$ (décalage 4 octets)
$\texttt{z} \mapsto 2$ (décalage 8 octets)
$\texttt{b} \mapsto 3$ (décalage 12 octets)

Création de structures en Caml et Java

Réserver un bloc de taille suffisante, et initialiser éventuellement les champs.

- En caml :

```
let p = { x=1; y=2; } in ...
```

  - bloc créé dans le tas (malloc)
  - valeur de p de type point : pointeur vers le bloc
  - initialisation des champs (accès avec la notation p.x)

- En java :

```
Point p = new Point(1, 2);
```

  - mêmes opérations de bas niveau qu'en caml
  - appel d'une fonction de construction pour l'initialisation

Création de structures en C

- En C, allocation explicite dans le tas malloc.

```
point *p = malloc(sizeof(point));
```

notation p->x pour l'accès aux champs :

```
p->x = 1;
p->y = 2;
```

- En C, manipulation par valeur sur la pile

```
point p = { x=1; y=2; };
```

la variable p désigne directement cette structure et non un pointeur
on accède au champ x avec la notation p.x.

Autres formes de types définies par l'utilisateur

Une énumération est un type défini par un ensemble fini de symboles.
enum en C avec le mot-clé enum

```
enum tete {
  valet; dame; roi;
}
```

type algébrique avec constructeurs constants en Caml.

```
type tete =
  Valet | Dame | Roi
```

Traduction : en numérote les symboles

$$\begin{aligned} \texttt{valet} &\mapsto 0 \\ \texttt{dame} &\mapsto 1 \\ \texttt{roi} &\mapsto 2 \end{aligned}$$

chaque occurrence d'un symbole est traduit par son numéro.

Autres formes de types définies par l'utilisateur

Un type union décrit une alternative entre plusieurs formats.

- Type algébrique en Caml

```
type s = | P of point | N of int
```

  - des constructeurs, ici P et N, pour distinguer les différents formats.
  - le filtrage permet alors de tester la forme d'une donnée de type s.

- En C :

```
union s {
  point p;
  int   n;
};
```

  - la représentation d'une donnée ne distingue pas s'il contient un point ou un entier.
  - ajouter un "tag" dans la structure (énumération des différentes formes)
  - possible incohérence entre le tag et la valeur

Synthèse

A retenir

- Le rôle du tas
- L'organisation en blocs
- Les enjeux des fonctions malloc et free
- La manipulation en assembleur de données comme les tableaux et les structures

Savoir faire

- Dérouler une stratégie particulière pour malloc et free
- Implémenter la traduction en assembleur MIPS d'une proposition de représentation de données en mémoire