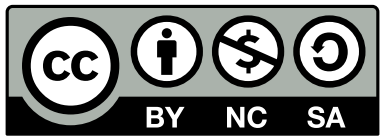
1001 Structure et données

Structure, tas, liste chainée

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur
v241



Rappel

Données simples

- Nombre entier
- Caractère
- Pointeur
- Nombre flottant $\rightarrow$ plus tard
- Instruction machine

Données complexes

- Tableaux (dont chaînes de caractères)
- Tableaux à plusieurs dimensions

Stockage et manipulation

Dans des registres

- Rapides et accessible en tout temps
- Taille limitée
- Règles d'ABI pour les routines

Allocation en mémoire

- Données globales
  - Segments data (et rodata et bss)
  - Durée de vie: tout le programme
- Données locales
  - Segment pile
  - Durée de vie: tant que la routine est active

Données complexes

Principe

- Tout n'est que bits
- Le sens des tas de bits est la manipulation qui en est faite

Et mes données complexes alors ?

- Structure (enregistrement) ?
- Objet (instance de classes) ?
- Au programmeur assembleur de se débrouiller

Plan

Structure

Structure

- Synonyme: enregistrement, agrégat, *struct*, *record*, *aggregate*
- Un agrégat de données possiblement hétérogène: **champ**
- Le nombre et les tailles des champs sont fixés

Exemples dans les langages de haut niveau

- Un struct C
- Un tuple de base de donnée
- Un objet Java (sans méthode)

Exemple: classe produit (en Java)

```
class Produit {
    int code; // code interne du produit
    int prix; // prix en cents
    String nom; // courte description
}
```

Structures en assembleur

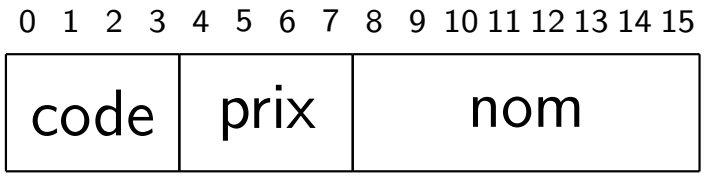
En mémoire

- Les champs sont stockés en mémoire
- En séquence, l'un après l'autre
- La taille de la structure vaut (au moins) la somme de la taille des éléments

Accès aux champs

- On connaît l'adresse mémoire de la structure
  - Dans un registre: **base**
- On connaît la position relative du champ
  - Valeur fixe immédiate: **décalage**
- Donc, on somme
  - L'adressage peut faire la somme pour nous

Exemple: structure produit en assembleur



- Taille totale d'un produit: 16 octets
- Le code: entier 32 bits
  - Taille: 4 octets
  - Décalage: +0
- Le prix en cents: entier 32 bits
  - Taille: 4 octets
  - Décalage: +4
- Le nom: pointeur vers une chaîne de caractères
  - Taille: 8 octets (en RV64)
  - Décalage: +8

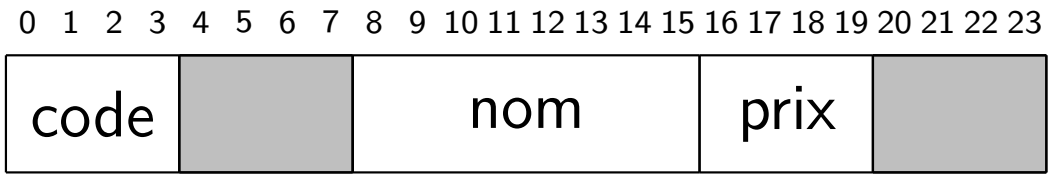
Alignement

Les champs doivent (généralement) être **naturellement alignés**

- Aligner la structure sur l'alignement du plus grand champ
- Taille de la structure multiple de celle du plus grand champ
- Ça peut laisser des trous (*padding*)

Exemple

Si on voulait les champs dans l'ordre `code`, `nom`, `prix`



- `code` sur un multiple de 4
- `nom` sur un multiple de 8
- `prix` sur un multiple de 4
- taille totale multiple de 8

Structures en RISC-V

Déclarer un produit

```
.data
prod1:  # Un produit
    .word 1234 # code
    .word 1995 # 19.95$
    .dword sourisStr
sourisStr: .string "Souris sans fil"
```

- Attention aux alignements
  - RARS aligne pour vous
  - GNU as n'aligne pas (`.align`)

Structures en RISC-V

Nommer les champs (avec .eqv)

```
    # Structure produit
    .eqv    prCode, +0 # champ `code`
    .eqv    prPrix, +4 # champ `prix` (en cents)
    .eqv    prNom,  +8 # champ `nom`
```

Accéder aux champs (*load*/*store*)

```
.text
    la s0, prod1 # adresse du prodit
    lw a0, prPrix(s0) # prix du produit
    call printInt
```

Exercice

Écrire un programme `produit.s`

- Afficher le prix d'un produit initialisé
- Afficher un produit en entier (routine `printProduit`)
  « 1234: Souris sans fil (19.95$) »
- Initialiser un produit interactivement (routine `readProduit`)
  $\rightarrow$ en lab

Tableaux et structures

Un tableau

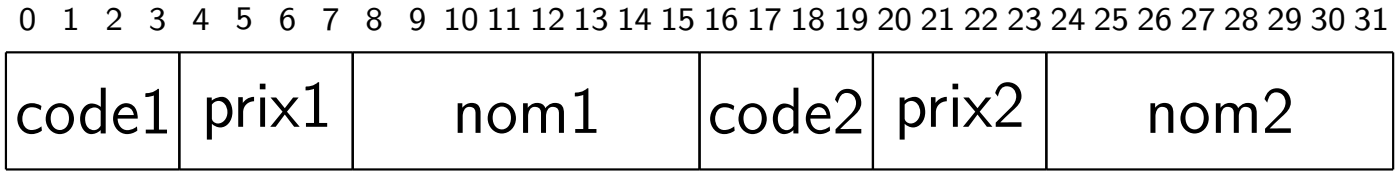
- Des données concaténées

Un tableau de structures

- Des structures concaténées

Accéder à un champ

- C'est de l'arithmétique d'adresse



Tas (*heap*)

Allocation dynamique des langages de haut niveau

Non gérée par le programmeur

- Gérée par une bibliothèque (`malloc` de C)
- Gérée par le compilateur (`new` de C++)
- Gérée par la machine virtuelle/interpréteur (`new` de Java)

Et les détails?

- Cachés au programmeur

Tas (*heap*)

Définition

- Segment mémoire réservée aux allocations **dynamiques**
- Utilisé par `malloc`, `new`, etc.
- Habituellement située après le segment `data`
- Croit vers les adresse supérieures
- Pas de registre dédiée
  - L'allocation est (relativement) rare
  - Les données de gestion sont en mémoire

Tas manuel

Do It Yourself

- On se déclare un bloc d'octets dans le segment `data`
- On garde un pointeur sur le premier octet non alloué du bloc
- On se fait une routine `myalloc` qui retourne un morceau du bloc

Pas de support

- Aucun support de l'environnent d'exécution
- Aucun registre ou instruction dédiée de l'architecture

Exercice

- Allouer un produit dynamiquement (routine `allocProduit`) et l'afficher

Tas manuel

```
.data
# Adresse du premier octet non alloué (limite du tas)
maPtr:   .dword maHeap
# 1K devrait suffire pour tout le monde
maHeap: .space 0x400
.text
# myalloc:
#  IN: a0, nombre d'octets demandés
#  OUT: a0, adresse de la zone mémoire allouée
myalloc:
    la a1, maPtr      # Adresse du pointeur en mémoire
    ld a2, 0(a1)      # Lit l'ancienne limite du tas
    add a3, a2, a0    # Nouvelle fin = ancienne + demande
    sd a3, 0(a1)      # Met à jour le pointeur en mémoire
    mv a0, a2         # Valeur de retour = ancienne fin
    ret
```

Alignement

- `myalloc` retourne une adresse
- Mais sans savoir à quoi elle va servir
- L'adresse n'est donc pas forcément alignée pour l'usage

Solution

- On peut forcer un alignement sur la taille de l'architecture
- Exemple: 8 octets en RV64

Inconvénient: fragmentation interne

- On alloue seulement des multiples de 8 octets
- Ça peut gaspiller un peu de mémoire

Problème de l'allocation dynamique manuelle

- Quelle quantité de mémoire considérer?
- Si trop peu, on va manquer
- Si trop, on gaspille
  Moins de mémoire disponible pour les autres programmes

L'environnent d'exécution

L'environnent d'exécution est responsable de l'organisation mémoire

- Les adresses des segments (text, data, stack)
- Les règle d'accès éventuelles (lecture, écriture, exécution)
- Le plus simple est de lui demander...

Tas en RARS

Appel système `Sbrk`

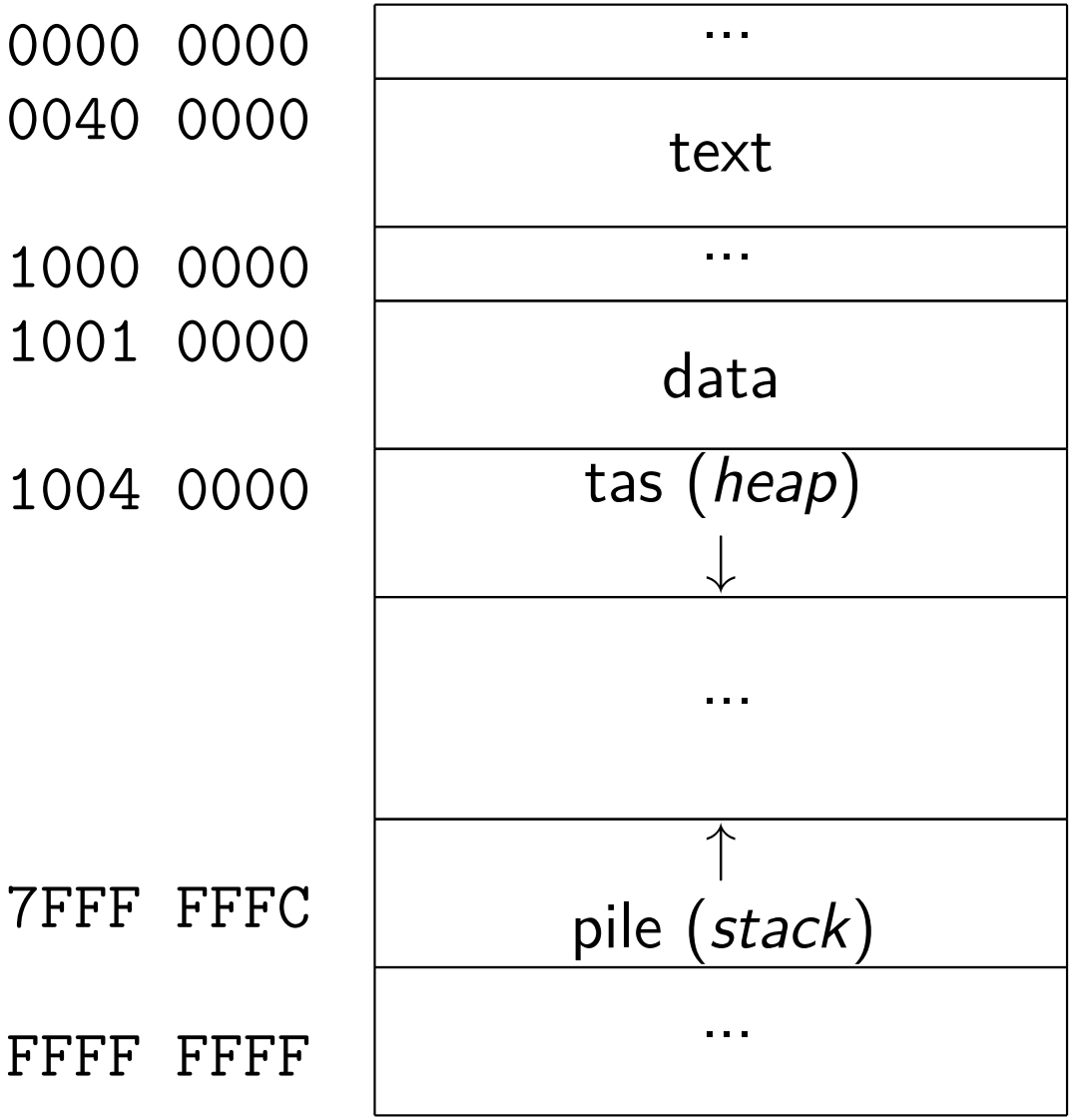
- `a7`: 9
- `a0`: nombre d'octets à allouer
- Résultat: `a0`: adresse du bloc alloué

```
li a7, 9 # Sbrk
li a0, 10
ecall
```

Alignement

- `Sbrk` force un alignement à 4 octets
  Même en mode 64 bit (bug)

Tas en RARS


0000 0000
0040 0000
1000 0000
1001 0000
1004 0000
7FFF FFFC
FFFF FFFF
...
text
...
data
tas (heap)
...
pile (stack)
...

Exercice

Modifier `produit.s`

- Écrire `allocProduit2` pour allouer dans le tas de RARS

Tas chez Linux

Appel système `brk`

- `a7`: 214
- `a0`: adresse voulue de la limite du tas
- `a0`: nouvelle adresse de la limite du tas
  - Inchangée si la demande n'est pas raisonnable
- Attention: `brk` n'aligne pas
- Disponible aussi dans le fork de RARS

Étymologie

- `brk` = *break*
- *program break*: la limite (la fin) du segment data
- Augmenter `brk` = allouer plus de mémoire au programme

Très bas niveau

- L'appel système Linux `brk` n'est pas portable
- Des détails supplémentaires en INF3173

Désallocation

Manuelle

- On fait une routine `free`
- Plus une structure de données pour savoir ce qui est libre ou pas
- `alloc` et `free` mettent à jour la structure de donnée

Automatique (Ramasse-miettes)

- Quelque chose fait des `free` tout seul
- Les détails en INF7741

En INF2171

- Laissons tomber la désallocation

Dans la vrai vie

Les langages et bibliothèques gèrent l'organisation interne de leur tas

- Demande un gros bloc de tas au système (via `brk`)
- Découpage « maison » de ce bloc de tas,
  pour répondre aux allocations des programmes (`malloc`, `new`)
- Si pas assez de tas disponible,
  on demande à augmenter la taille du tas au système (via `brk`)
- Structures de données et algorithmes +/- complexes
  pour la gestion, l'allocation et la libération

En INF2171

- On se contente de `Sbrk` de RARS
- Pas besoin de gestion de tas supplémentaire

Liste chainée

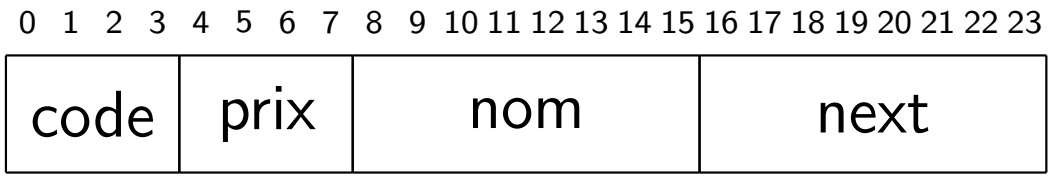
Liste chainée

Définition

- Des structures **chaînés** ensembles
- Chaque structure de la chaîne est un **maillon**
- Une structure contient l'**adresse** de la structure suivante
  Voire de la précédente (liste doublement chainée)

Exemple

```
class Produit {
    int code;
    int prix;
    String nom;
    Produit next; // Pointeur vers l'élément suivant
}
```



Pointeur nul

Problème

- Le dernier élément d'une liste n'a pas de suivant
- Que mettre pour `next`?

Convention

- On se décide d'une valeur arbitraire « pas de pointeur »
- On peut nommer et réutiliser cette valeur

  `NULL`, `null`, `None`, `nil`, etc.

ABI RISC-V

- Un pointeur nul a la valeur 0

Gestion de liste chainée

Tête de liste

- Pointeur vers le premier élément
- Ou `null` si liste vide

Ajouter en tête de liste

- Faire un nouveau maillon
- Son suivant est la tête de liste
- Mettre à jour la tête de liste

Ajouter en queue de liste

- Se rendre au dernier maillon
- Y accrocher le nouveau maillon
- Attention au cas de la liste vide!

Exercice

Des listes de produits

- Allouer et initialiser une liste de produits
- Les afficher tous mais dans l'ordre inverse
- Afficher seulement ceux dont le prix est $>$ 10\$

Structures avancées

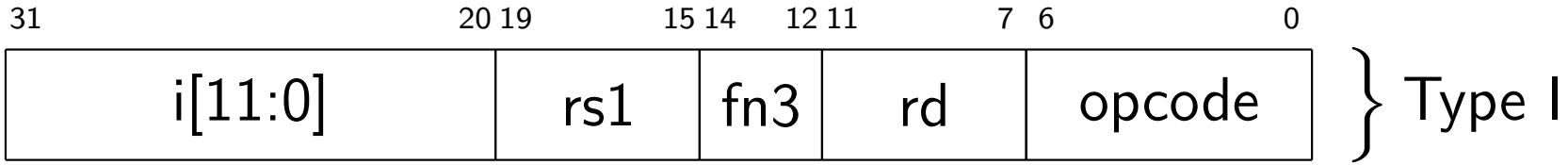
Champs de bits

Idée

- Plusieurs informations dans un seul entier
- Utiliser opérations bit à bit pour extraire les champs

Exemple

- Codage des instructions RISC-V
- Chaque élément (opcode, fonction, opérande) fait seulement quelques bits



Unions

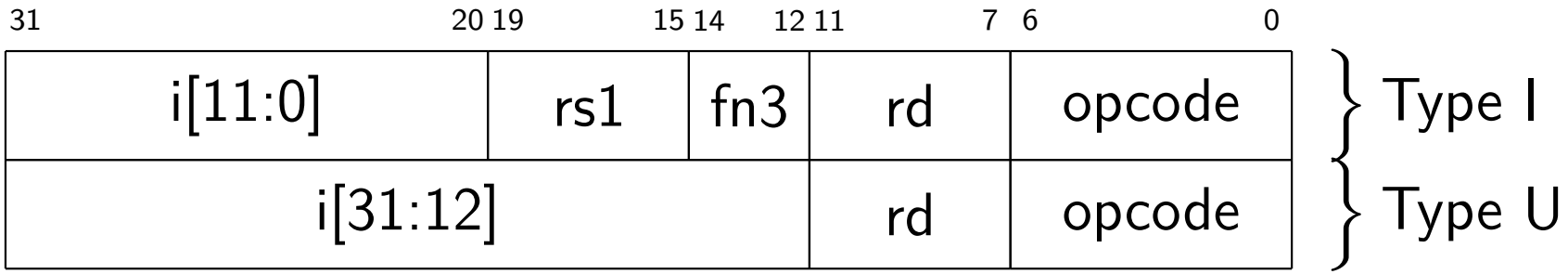
Idée

- Utiliser un même espace mémoire dans une structure
- Pour de l'information alternative
- C'est l'un ou l'autre, et non les deux en même temps
- **Important**

  Il faut un moyen de déterminer la représentation utilisée

Exemple

- Codage des instructions RISC-V
- L'opcode permet de savoir comment interpréter les autre bits



Passage de structure en paramètre

Passage par référence

- L'argument utilisée est un pointeur vers la structure
- Ce que l'on a fait dans les exemples
- Fonctionne quelque soit la taille de la donnée
- Mais nécessite de manipuler des pointeurs

Passage par valeur

- On décompose les champs dans des registres
- Efficace pour les petites structures
- Exemple: un point avec un champ `x` et un champ `y`

Conclusion

Résumé

Structure

- Agrégat de champs hétérogènes fixé
- Adressage basé pour accéder aux champs en mémoire

Tas (*heap*)

- Espace d'allocation dynamique
- Géré programmativement

Liste chainée

- Exemple d'utilisation de structures allouée dans le tas
- Structure de donnée de base mais très expressive

La prochaine fois

Nombres flottants

- Représentation des nombre réels
- IEEE 754
- Instructions flottantes RISC-V