**Logiciel de base**
**Première année par alternance**

Responsable : Christophe RIPPERT
Christophe.Rippert@Grenoble-INP.fr

# Structures de données chainées

## Introduction

Le but de cette séance est de pratiquer la manipulation de structures et de pointeurs en assembleur : il est essentiel d'être à l'aise sur ses notions pour l'examen de TP final.
Commencez par récupérer les sources utiles sur la page principale du cours.

## Rappel préliminaire

En C, les structures de données (déclarées avec le mot-clé `struct`) sont stockées en mémoire selon des règles définies dans la spécification du langage et sur lesquelles on doit s'appuyer pour accéder aux différents champs en assembleur.
Soit par exemple la structure suivante :

```
struct structure_t {
    int64_t entier;
    char *ptr;
};
```

En mémoire, si on suppose que la structure est placée à l'adresse A, on aura :
- le champ `entier` également à l'adresse A ;
- le champ `ptr` à l'adresse A + 8, puisque ce champ est le deuxième de la structure, et que la taille du premier est 8 octets.

La spécification du langage C impose donc que les champs d'une structure soit stockés en mémoire dans le même ordre que leur déclaration.

## Exemple détaillé

Dans le fichier `struct.c`, on trouve le programme suivant :

```
struct structure_t {
    int64_t entier;
    char *ptr;
};

void affiche(int64_t entier, char *ptr)
{
    printf("entier = %" PRId64 ", ptr = 0x%" PRIX64 "\n", entier, (uint64_t)ptr);
}

void affichage(struct structure_t);

void modification(uint64_t, char*, struct structure_t*);

int main(void)
```

```
{
    struct structure_t s = {-1, (char*)0xBADC0FFEE0DDF00D};
    affichage(s);
    modification(5, (char*)0xDEADC0DED15EA5E, &s);
    affichage(s);
    return 0;
}
```

Ce programme défini le type `structure_t` comme plus haut, puis affecte les champs d'une variable `s` de ce type avec les valeurs initiales :

— * entier = -1

— ptr = 0xBADC0FFEE0DDF00D

La fonction `affichage` prend en paramètre la structure `s`. Cette fonction est définie dans le fichier `fct_struct.s` par :

```
    .globl affichage
    // void affichage(struct structure_t s)
    // s : %rdi:%rsi
affichage:
    enter $0, $0
    // affiche(s.entier, s.ptr);
    call affiche
    leave
    ret
```

En fait, cette fonction reçoit simplement la structure en paramètre, puis appelle une autre fonction `affiche` (définie dans `struct.c`) pour réaliser l'affichage. La fonction `affiche` prend en paramètre un entier et un pointeur, ce qui veut dire que `affichage` ne sert qu'à découper la structure en deux champs. En pratique, ce découpage est déjà fait par le compilateur C : l'ABI impose en effet que les champs sur 64 bits d'une structure sont passés en paramètres dans les registres habituels (`%rdi`, `%rsi`, `%rdx`, etc.). Le champ `entier` est donc passé dans `%rdi` et le champ `ptr` est passé dans `%rsi`.

On a conseillé à la séance précédente de sauvegarder systématiquement les paramètres lorsque la fonction qu'on écrit appelle elle-même une autre fonction : ici, il est visiblement inutile de sauvegarder `%rdi` et `%rsi` avant d'appeler `affiche`, car la fonction `affichage` se termine tout de suite après l'appel : on n'aura plus besoin des valeurs initiale des paramètres (mais un compilateur C non-optimisant les sauvegarderait tout de même).

L'appel à `affiche` est donc immédiat, vu qu'on a déjà les bons champs dans les bons registres.

Ensuite, dans le programme principal, on appelle une fonction servant à modifier les champs de la structure : `modification` est définie dans le fichier `fct_struct.s` par :

```
    .globl modification
    // void modification(int64_t e, char *p, struct structure_t *ps)
    // e : %rdi
    // p : %rsi
    // ps : %rdx
modification:
    enter $0, $0
    // ps->entier = e;
    movq %rdi, (%rdx)
    // ps->ptr = p;
    movq %rsi, 8(%rdx)
    leave
    ret
```

Pour pouvoir modifier les champs de la structure `s`, on doit passer un pointeur vers cette structure à la fonction `modification` : c'est son troisième paramètre `ps`. Les deux premiers paramètres sont les nouvelles valeurs qu'on veut affecter aux champs de la structure.

En C, on peut écrire : `ps->entier = e` pour affecter la valeur de l'entier `e` au champ `entier` de la structure pointée par `ps` : cette notation est un raccourci équivalent à `(*ps).entier = e`.
Dans le code assembleur, on doit donc :

- affecter la valeur du paramètre `e` qui est dans `%rdi` (premier paramètre) ;
- dans la case pointée par le pointeur `ps` qui est dans `%rdx` (troisième paramètre), ce qui s'écrit simplement `movq %rdi, (%rdx)`. On rappelle que le premier champ d'une structure est situé à la même adresse que la structure elle-même.

Pour le champs `ptr`, le principe est le même sauf que le champ `ptr` ne se situe pas dans la case pointée par `ps` mais 8 octets plus loin (c'est le deuxième champ de la structure, et le premier champ prend 8 octets). On écrira donc `movq %rsi, 8(%rdx)` puisque le deuxième paramètre (p) est dans `%rsi`.

# Exercices

## Utilisation de Valgrind

L'outil Valgrind est très utile pour détecter les erreurs d'accès ou les fuites mémoire dans un programme (écrit en C, assembleur, Ada, etc.).
Pour vous entrainer à son utilisation, on fourni un fichier `fct_progbug.s` qui contient deux fonctions volontairement fausses. Pour rendre les choses plus amusantes, elles ne sont pas commentées. Le fichier `progbug.c` contient un programme principal qui ne fait qu'appeler ces deux fonctions. Pour commencer, on a commenté la fonction `bug2` afin de traquer chaque problème un par un.
Commencez par compiler en tapant `make progbug` puis exécuter le programme : `./progbug` : aucun message d'erreur n'apparait. Vous pouvez si vous le souhaitez exécuter pas à pas le programme dans GDB, ce qui ne devrait pas beaucoup vous aider non plus.
Executez maintenant la commande `valgrind --leak-check=full ./progbug` : le message affiché doit vous permettre de comprendre l'origine du problème (oui, c'est en anglais !).
Ouvrez `progbug.c`, commentez l'appel à `bug1` et décommentez l'appel à `bug2` et répétez le processus pour comprendre le deuxième problème.

## Manipulation de listes chaînees

Dans cet exercice, on va manipuler des listes chainées de cellules définies dans le fichier `liste.c` par :

```
struct cellule_t {
    int64_t val;
    struct cellule_t *suiv;
};
```

Ce fichier contient un programme de test qui génère un tableau de 10 entiers aléatoires puis le convertit en liste chaînée et appelle les fonctions à implanter en assembleur.
La première fonction à implanter est `inverse` qui prend en paramètre un pointeur sur une liste chaînée (donc un pointeur sur un pointeur sur une cellule), inverse la liste pointée, et met à jour le pointeur passé en paramètre pour qu'il pointe sur la liste inversée.
On vous demande comme toujours de traduire littéralement le code C donné ci-dessous et de respecter les conventions de gestion de la pile et des registres.

```
void inverse(struct cellule_t **l)
{
   struct cellule_t *res, *suiv;
   res = NULL;
   while (*l != NULL) {
       suiv = (*l)->suiv;
       (*l)->suiv = res;
       res = *l;
       *l = suiv;
   }
   *l = res;
}
```

Ensuite, vous devez implanter la fonction `decoupe` qui prend en paramètre une liste chaînée (donc un pointeur sur une cellule), ainsi que deux pointeurs sur listes : ces deux paramètres correspondent aux deux listes résultat qu'on doit obtenir après le découpage (le critère de découpage choisi ici étant la parité de la valeur d'une cellule).

```
struct cellule_t *decoupe(struct cellule_t *l,
                          struct cellule_t **l1,
                          struct cellule_t **l2)
{
    struct cellule_t fictif1, fictif2;
    *l1 = &fictif1;
    *l2 = &fictif2;
    while (l != NULL) {
        if (l->val % 2 == 1) {
            (*l1)->suiv = l;
            *l1 = l;
        } else {
            (*l2)->suiv = l;
            *l2 = l;
        }
        l = l->suiv;
    }
    (*l1)->suiv = NULL;
    (*l2)->suiv = NULL;
    *l1 = fictif1.suiv;
    *l2 = fictif2.suiv;
    return l;
}
```

La fonction `decoupe` a besoin de deux variables locales `fictif1` et `fictif2` qui sont de type cellule : c'est à dire que chacune de ces variables occupe 16 octets et contient deux champs, la valeur et le pointeur vers la cellule suivante (comme il s'agit d'éléments fictifs placés en tête de liste pour faciliter les insertions, on ne se servira pas de leurs champs `val`, mais on est obligé de l'allouer quand même si on traduit littéralement le type `cellule_t`).
On rappelle que `lea` est l'instruction permettant de calculer l'adresse d'une variable.

## Si vous avez fini en avance

Reprenez votre module de gestion d'une table de hachage de la partie C, et traduisez les fonctions de création, d'insertion et de suppression dans la table en assembleur x86_64.