# 12. Mémoïsation

# Plan

## 12. Mémoïsation

# Notion de variable statique

Une variable statique est une variable dont la **durée de vie** est égale à celle de l'**exécution** du programme. Une telle variable est créée et initialisée à la compilation.

L'instruction

```
static T ID;
```

déclare une variable statique `ID` de type `T`.

Elles sont les contreparties des variables automatiques dont la durée de vie s'étend à l'exécution du bloc dans lesquels elles se trouvent et dont la création se fait à l'exécution le moment voulu.

Les variables automatiques sont les variables habituelles (se déclarent sans le mot clé `static`).

# Persistance des variables statiques

Une variable statique est rémanente : elle **conserve sa dernière valeur** jusqu'à une éventuelle modification.

## – Exemple –

```
void fct(int a) {
    static int s = 3;
    s = s + a;
    printf("%d\n", s);
}
...
fct(1);
fct(5);
fct(2);
```

Produit l'affichage

`4`

`9`

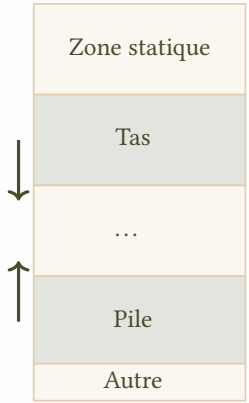
`11`.

Ici, `s` est une variable statique. La valeur `3` (initialisation) ne lui est attribuée qu'une fois, lors de la compilation.

De plus, `s` conserve sa valeur (qui peut être modifiée) au fil de l'exécution, d'appel en appel à `fct`.

# Géographie de la mémoire lors d'une exécution

Lors de l'exécution d'un programme, la mémoire qui lui est consacrée est divisée de la manière suivante.




Les zones de la mémoire réservées à la pile et au tas ont des tailles variables durant l'exécution.

Le tas grandit vers le bas (vers les adresses hautes) et la pile grandit vers le haut (vers les adresses basses).

La zone statique, qui contient les variables statiques et le code du programme, est de **taille constante durant l'exécution**.

Cette taille est connue et calculée lors de la compilation.

# Compter le nombre d'appels à une fonction

Étant donnée une fonction, nous souhaitons **compter** le nombre de fois qu'elle a été **appelée** depuis le lancement du programme.

```
int fact(int n) {
    if (n <= 1)
        return 1;
    return n * fact(n - 1);
}
```

Fonction originale.

```
int fact(int n) {

    /* Mecanisme de comptage */
    static int nb_app = 0;
    nb_app += 1;

    if (n <= 1)
        return 1;
    return n * fact(n - 1);
}
```

Fonction modifiée.

À chaque instant, la variable statique `nb_app` a pour valeur le nombre de fois que `fact` a été appelée.

**Problème** : cette variable a pour portée lexicale le corps de la fonction `fact` et est invisible ailleurs. On ne peut donc pas la lire depuis l'extérieur.

# Compter le nombre d'appels à une fonction

Pour avoir accès à la valeur de `nb_app` hors du corps de `fact`, nous la transmettons à l'aide d'un **passage par adresse**.

```
int fact(int n, int *nb) {

    /* Mecanisme de comptage */
    static int nb_app = 0;
    nb_app += 1;
    *nb = nb_app;

    if (n <= 1)
        return 1;
    return n * fact(n - 1, nb);
}
```

Nous l'utilisons de la manière suivante :

```
int nb;
fact(6, &nb);
printf("%d\n", nb);
```

Ceci affiche `6`.

# Plan

## 12. Mémoïsation

# Principe de la mémoïsation

Soit `f` une fonction **sans effet secondaire** et **déterministe** (sa valeur de retour ne dépend que de ses arguments).

**Observation** : si l'on appelle `f` deux fois avec des mêmes arguments, elle renverra deux fois la même valeur.

**Conséquence** : comme il est inutile (et inefficace) de refaire deux fois un même calcul, il suffit de **mémoriser le résultat** renvoyé par `f` lors du 1er appel et de le renvoyer sans calcul lors du 2e.

**Marche à suivre** : on enregistre toutes les valeurs de retour de `f` au fur et à mesure de ses appels.

Ce procédé s'appelle la mémoïsation de fonction.

# Détails sur la mémoïsation

Les résultats renvoyés par `f` sont mémorisés dans une **table de hachage** `mem` associant à chaque jeu d'arguments avec lequel elle est appelée la valeur de retour de `f`.

Il est nécessaire de disposer d'une **fonction de hachage** `h` associant à chaque jeu d'arguments un entier.

Il ne doit y avoir qu'une seule table `mem` pour `f`, dont la durée de vie est égale à l'exécution du programme et rémanente : `mem` est ainsi une **variable statique**, locale à `f`.

# Procédé général de mémoïsation d'une fonction

Soit `f` une fonction à `n` paramètres.

Nous notons `f'` la version mémoïsée de `f`, définie de la manière suivante.

Lors de l'appel à `f'(a_1, ..., a_n)` :

(1) si `mem` contient une donnée associée à la clé `(a_1, ..., a_n)`,
   (a) renvoyer cette valeur ;

(2) sinon,
   (a) calculer la valeur de retour de `f(a_1, ..., a_n)` ;
   (b) **mémoriser** cette valeur dans `mem` ;
   (c) renvoyer cette valeur.

L'étape (1) est celle de lecture et l'étape (2) est celle de mémorisation.

# Exemple de mémoïsation d'une fonction à un paramètre

## – Exemple (début) –

Nous souhaitons mémoïser la fonction

```
int fibo(int n) {
    if (n <= 1)
        return n;
    return fibo(n - 1) + fibo(n - 2);
}
```

Cette fonction est paramétrée par un entier et renvoie un entier. La table `mem` est ainsi un tableau d'entiers. La valeur de hachage d'un argument `n` est `n`.

Nous l'utilisons de la manière suivante :

1. elle est de taille `T_MEM` où `T_MEM` est une constante suffisamment grande ;
2. pour tout `0` $\leqslant$ `n` $<$ `T_MEM`, la valeur de `mem[n]` est
   - `-1` si `fibo(n)` n'a pas encore été appelée ;
   - la valeur renvoyée par `fibo(n)` si `fibo` a été appelée au moins une fois avec l'argument `n`.

# Exemple de mémoïsation d'une fonction à un paramètre

## – Exemple (suite et fin) –

La fonction `fibo` peut se mémoïser de la façon suivante.

```
int fibo(int n) {
    /* Table */
    static int mem[T_MEM];

    /* Booleen 1er appel */
    static int init = 1;

    int i, res;

    /* Initialisation 1er appel */
    if (init) {
        for (i = 0; i < T_MEM; ++i)
            mem[i] = -1;
        init = 0;
    }

    /* Lecture */
    if (mem[n] != -1)
        return mem[n];

    /* Calcul */
    if (n <= 1)
        res = n;
    else
        res = fibo(n - 1) + fibo(n - 2);

    /* Memorisation */
    mem[n] = res;

    return res;
}
```

# Mémoïsation de fonctions à plusieurs paramètres

Soit `f` une fonction à `n` paramètres.

La table `mem` devient un **tableau de variables d'un type structuré** `E_f` composé

1. d'un champ pour chaque **paramètre** de `f` ;
2. d'un champ pour le **résultat** renvoyé par `f` appelé avec les arguments spécifiés par les précédents champs ;
3. d'un champ **booléen** indiquant si le résultat de `f` appelée avec les arguments spécifiés par les précédents champs a bien été calculé.

Il faut de plus disposer d'une **fonction de hachage** `h_f` de même signature que `f` et qui renvoie un entier positif.

Elle permet d'attribuer à chaque jeu d'arguments une position dans la table `mem`.

# Exemple de mémoïsation d'une fonction à deux paramètres

## – Exemple (début) –

Nous souhaitons mémoïser la fonction

```
int puiss(int n, int p) {
    if (p == 0)
        return 1;
    return n * puiss(n, p - 1);
}
```

Cette fonction est paramétrée par deux entiers. La table `mem` est ainsi un tableau de valeurs de type `E_puiss`.

Le type structuré `E_puiss` est donc défini par

```
typedef struct {
    int n; /* 1er parametre */
    int p; /* 2e parametre */
    int res; /* Resultat */
    int ok; /* Indique si le calcul a deja ete fait */
} E_puiss;
```

On utilise la fonction de hachage suivante :

```
int h_puiss(int n, int p) {
    return n ^ (p << 2);
}
```

# Exemple de mémoïsation d'une fonction à deux paramètres

## – Exemple (suite et fin) –

La fonction `puiss` peut se mémoïser de la façon suivante.

```
int puiss(int n, int p) {
    /* Table */
    static E_puiss mem[T_MEM];

    /* Booleen 1er appel */
    static int init = 1;

    int i, res, h;

    /* Initialisation 1er appel */
    if (init) {
        for (i = 0; i < T_MEM; ++i)
            mem[i].ok = 0;
        init = 0;
    }

    /* Lecture */
    h = h_puiss(n, p) % T_MEM;
    if (mem[h].ok) {
        if (mem[h].n == n && mem[h].p == p)
            return mem[h].res;
    }

    /* Calcul */
    if (p == 0) res = 1;
    else res = n * puiss(n, p - 1);

    /* Memorisation */
    mem[h].n = n;
    mem[h].p = p;
    mem[h].res = res;
    mem[h].ok = 1;

    return res;
}
```