Chap 2 : Représentation et stockage de l'information

Table des matières

1 Mémoire d'un programme compilé :

1.1 Organisation de la mémoire

1.1.1 Distinction statique dynamique

Le stockage des objets manipulés par un programme est géré de deux manières différentes selon leur nature :

– Statiquement : lorsque le compilateur dispose de l'information suffisante pour prévoir de la place pour ces données, elle peut être réservée à l'avance ;

– Dynamiquement : lorsque la taille des objets n'est connue que lors de l'éxécution du programme, le programme doit faire une demande d'allocation pour obtenir l'espace en mémoire

Ex : les variables globales / locales constantes ou dont la taille est fixée (ex type `int`, `char`) sont gérées statiquement. Les tableau dont la taille dépend de l'état du programme au moment de l'éxécution (ex : `Array.make n 0` : `n` est variable) sont gérés dynamiquement.

• Allocation statique : plusieurs zones de mémoire sont utilisées selon la nature de l'objet :

– Variables globales *initialisées* : stockées dans le binaire donc chargé en mémoire avec le binaire dans une zone spécifique qui l'accompagne.

– Les variables globales *non initialisées* sont allouées, placées en mémoire au moment du chargement du binaire qui ne contient que leur déclaration.

– Les variables locales et dont les paramètres des fonctions sont placées dans une zone mémoire spécifique appelée pile, utilisée dynamiquement afin de ne réserver l'espace qu'au moment de l'appel de fonction ou de l'éxécution du bloc

• Allocation dynamique : on utilise une zone spécifique appelée *tas*

Les détails spécifiques aux langages C et OCaml seront vus en 1.2.

Schéma :

Code binaire
Variables globales initialisées
variables globales non initialisées
tas (heap) ↓
↑ Pile (stack)

Commande : `objdump` Ex : `gcc -c prog.c` puis `objdump -x prog.o`

1.1.2 Portée syntaxique d'un identificateur

Def : la portée syntaxique d'un identificateur est la zone de texte d'un programme dans laquelle il est possible d'y faire référence sans erreur à la compilation

Ex :

```
1  int a = 1;
2  int f(int x) {
3      int y = x + a;
4      return y;
5  }
6  int g() {
7      int z = 3;
8      return z + f(z);
9  }
10
```

Identificateur	Portée
a	1-9
x	2-5
y	3-5
f	2-9
z	7-9
g	6-9

Attention, la portée est associée aux identificateurs, pas aux variables.

Ex :

```
1  int f() {
2      int i = 3;
3      return i;
4  }
5
6  int g() {
7      int i = 5;
8      return i + 1;
9  }
10
```

Portée de `i` : 2-4 et 6-8

Même identificateur donc même portée pour deux variables bien distinctes.

```
1  int f() {
2      int i = 3;
3      for (int j = 0 ; j < 3 ; j ++) {
4          int i = 4;
5          i = i + j;
6      }
7      return i;
8  }
9
```

Portée de `i` : 2-8

Mais phénomène de masquage dans les lignes 4-6.

Rq : les mêmes principes s'appliquent en OCaml sauf qu'il est impossible de faire référence à une variable non initialisée.

Pour ces deux langages, la portée est statique. En Python, la portée est dynamique.

1.1.3 Durée de vie d'une variable

Def : la durée de vie d'une variable correspond à la période de l'éxécution du programme depuis son allocation jusqu'a la libération de la mémoire associée.

Rq : la plupart du temps, la durée de vie d'une variable correspond à la portée de son identificateur.

En C, il est possible de définir des variables, dites statiques, dont la durée de vie dépasse la portée (H.P).

1.1.4 Gestion de la pile

La pile sert au stockage des variables locales et des paramètres des fonctions. Son remplissage évolue en fonction de l'éxécution du programme en "empilant" les zones crées, et en "dépilant" les zones libérées, à la manière de la structure de données appelée pile (cf plus tard)

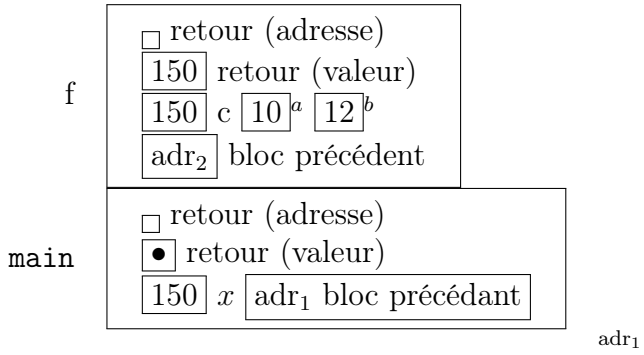
Rq : on peut utiliser les registres du processeur pour stocker certaines données. On suppose ici que ce n'est pas le cas.

Lors d'un appel de fonction, on ajoute sur la pile d'éxécution un bloc de mémoire (stack frame) correspondant au bloc d'activation de la fonction, qui contient :

- Les paramètres de la fonction ;
- L'adresse de retour, indiquant la position à laquelle doit reprendre l'éxécution du programme après l'éxécution de la fonction ;
- Un espace pour le résultat de la fonction, qui sera transmis ;
- De l'espace pour les variables locales ;
- L'adresse du bloc précédant, permettant de dépiler le bloc d'activation après l'éxécution de la fonction.

Ex :

```
1  int f(int a, int b) {
2      int c = 3;
3      c = c + b;
4      c = c * a;
5      return c;
6  }
7
8  int main() {
9      int x = f(10, 12);
10     return 0;
11 }
12
```

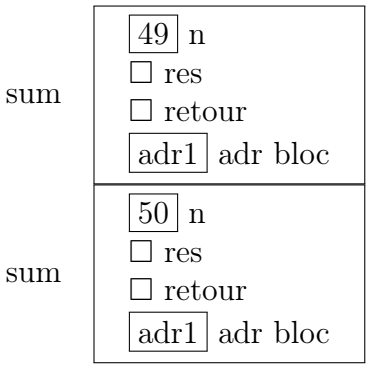


Attention, la pile est limitée : s'il y a trop d'appels de fonction imbriqués, l'empilement des blocs d'activation dépassera la capacité de la pile (Stack overflow)

Ex : une fonction récursive sans cas de base (boucle infinie) ou qui fait trop d'appels récursifs.

```
let rec sum (n : int) : int =
    match n with
    | 0 -> 0
    | _ -> n + sum (n - 1)

```




```
let sum (n : int) =
    let rec aux (n : int) (acc : int) =
        match n with
        | 0 -> acc
        | _ -> aux (n - 1) (acc + n) (*The calculation is in the
    args : the stack frame will not be kept.*)
    in aux n 0;;

```

Optimisation pour les fonctions récursives terminales en OCaml. Pas garantit en C. Pas faite en Python.

1.2 Gestion de la mémoire en OCaml et en C

1.2.1 Gestion de la mémoire en OCaml

En OCaml, la gestion de la mémoire est gérée automatiquement, les allocations dynamiques sont réalisées de manière transparente (comme toutes les déclarations). La mémoire est également libérée automatiquement grâce à un programme appelé *garbage*

collector (ramasse-miette / glaneur de cellules) qui détecte quand un bloc de mémoire peut être libéré et qui réorganise le tas pour le compacter.

1.2.2 Principe de gestion de la mémoire en C

Contrairement à OCaml, les allocations dynamiques en C sont gérées par le programmeur qui doit réclamer explicitement l'allocation d'un bloc mémoire sur le tas d'une taille donnée et qui doit signaler plus tard au système que la mémoire peut être libérée. Il faut impérativement libérer toute mémoire allouée.

Dans le cas contraire, on parle de *fuite mémoire* et le système peut devenir innutilisable faute de mémoire disponible.

En C, on fait référence aux zones de mémoire par leur adresse.

1.2.3 Pointeurs

Les pointeurs sont des objets qui représentent des adresses en mémoire. À la manière des références en OCaml, le type des pointeurs est paramétré par le type de la valeur contenue dans le bloc mémoire qu'ils adressent / sur laquelle ils pointent.

Syntaxe : `<type>*` est le type des pointeurs qui pointent sur ine valeur de type `<type>`.

Déclaration : Comme pour les autres variables.

Ex :

```
int *p;
double **q;
```

Rq : on peut placer l'étoile contre le type ou contre le nom de la variable ou même séparée des deux par des espaces.

Attention : `int* p, a;` déclare un pointeur sur entier `p` et un entier `a`.

• Opérateur d'adressage : c'est `&`.

Rappel : `scanf` prend une adresse en argument.

Ex :

```
int a = 2;
int* p = &a;
```

• Opérateur de déréférencement : c'est `*`.

Ex : après l'ex précédent, `*p` vaut 2.

On peut voir le déréférencement de `p` comme une variable dont on peut modifier la valeur.

Ex :

```
*p = 3;
printf("%d\n", a); //affiche 3
```

On peut utiliser les pointeurs pour écrire des fonctions qui renvoient plusieurs résultats. Idée : on écrit une fonction de type `void` qui prend en paramètres des pointeurs sur les cases mémoires où l'on souhaite récupérer les résultats.

Ex :

```
void minx_max(int x, int y, int *min, int *max) {
    if (x <= y) {
        *min = x;
        *max = y;
    }
    else {
        *min = y;
        *max = x;
    }
}

int x = 5, y = -8;
min_max(x, y, &x, &y);
```

Pour afficher un pointeur avec `printf`, on utilise `%p`.

1.2.4 Allocation dynamique en C

On dispose de deux fonctions :

– Allocation : on utilise la fonction `malloc`, de prototype `void* malloc(size_t size)`.

`void*` : pointeur sur "rien" (on ne connait pas le type de la valeur sur laquelle le pointeur pointe) ;

`size_t` : type représentant une taille.

On utilise le transtypage pour choisir le type du pointeur.

Ex : `char *t = (char*) malloc(n);`

En cas d'échec de l'allocation, `malloc` renvoit un pointeur particulier, noté `NULL`

Pour déterminerle nombre d'octets à allouer, on utilise la fonction `sizeof` qui prend en paramètre un type ou un objet et qui renvoit sa taille.

Ex :

```
int *t = (int*) malloc(n * sizeof(int));
```

– Libération : on utilise la fonction `free`, de prototype `void free(void* ptr)`.

`void` : c'est une instruction ;

`void* ptr` : on peut prendre n'importe quel pointeur (par transtypage).

Ex : `free(t);`

Attention :

```
int a;
free(&a);
//et
char *a = malloc(8);
free(a);
free(a);
```

provoquent des erreurs.

Protection mémoire : un programme s'éxécute dans un environement mémoire constitué de plusieurs zones appelées *segments*, et ne peut écrire que dans certaines d'entre elles.
Le système protège l'intégrité de la mémoire grâce à une erreur (`segmentation fault` ou `segmentation violation`) en cas d'accès interdit.
Ex : on ne peut pas écrire à l'adresse 0 ni dans le segment du code :

```
int *a = 0;
printf("%d", *a);

int main() {
    int *a = (int*) (&main);
    *a = 0;
    return 0;
}
```

Lèvent des erreurs.

1.2.5 Tableaux

En C, on peut voir un tableau comme un pointeur sur sa première case.
Ex :

```
int *p, tab[5];
p = tab;

```

est équivalent à

```
int *p, tab[5];
p = &tab[0];

```

H.P : dans ce contexte, `p + i` (ou `tab + i`) est l'adresse da la case d'indice `i`.
Ex :

```
int t[5] = {1, 2, 3, 4, 5};
for (int k = 0 ; i < 5 ; i++) {
    printf("%d\n", *(t + i));
}
```

Allocation dynamique d'un tableau : Ex :

```
int *t = (int*) malloc(n * sizeof(int));

```

alloue un tableau de `n` entiers.

Passage de tableau en paramètre d'une fonction :
Attention : la taille n'est pas vérifiée
Ex :

```
void f(int t[6]) {
    t[0]++;
    t[1]++;
}

```

Fonctionnne aussi pour un tableau de taille 2.

En fait, on pourrait écrire `int t[]`.

Il est recommandé de passer la taille du tableau en argument.

Ex :

```
void print_tab(int *t, const int n) {
    for (int i = 0 ; i < n ; i++) {
        printf("%d\n", t[i]);
    }
}
```

Allocation d'une matrice à n lignes et m colones : 2 possibilités :

- On alloue un tableau de tableaux ;
- On linéarise la matrice.

```
int **mat = (int**) malloc(n * sizeof(int*));

for (int i = 0 ; i < n ; i++) {
    mat[i] = (int*) malloc (m * sizeof(int));
}
```

À la libération, appeler `free` sur chaque ligne avant de l'appeler sur `mat`.

```
int *mat = (int*) malloc(n * m * sizeof(int));
```

Accès :

```
mat[j + n*i]
//au lieu de
mat[i][j]
```

Libération : un seul appel à `free`

1.2.6 Arguments de la fonction `main`

On peut passer des arguments au programme *via* la ligne de commande éxécutée dans le terminal pour le lancer (cela permet par exemple de mettre en place un système d'options ou passer un nom de fichier en paramètre pour que le programme traite les données du ficher). Ces arguments sont obtenus comme paramètres de la fonction `main` sous la forme d'un tableau de chaîne de caractères.

Prototype de `main` :

```
int main(int argc, char *argv[])
```

- `int argc` : nombre d'éléments écrits sur la ligne de commande *avec* le nom du programme ;
- `char *argv[]` : tableau des éléments.

`argv[0]` est le nom du programme et `argv[1]`, ..., `argv[argc - 1]` sont les arguments du programme.

En OCaml, le point d'entrée du programme n'est pas une fonction `main` prenant des arguments, mais une expression de type `unit`. On récuprère les arguments du programme grâce au module `Sys`.

`Sys.argv` est un objet de type `string array` équivalent au `argv` du langague C.

2 Système de fichiers

2.1 Organisation

2.1.1 Arborescence de fichiers

Les données stockées sur l'ordinateur sont réparties dans des suites cohérentes appelées fichiers. Les fichiers ne sont pas en vrac dans la mémoire mais organisés grâce aux répertoires / dossiers, représantant un groupe de fichers. Cette structure organisée est appelée l'arborescence des fichiers.

Ex :

- `bin` : éxécutables permettant de démarrer le système et commandes de terminal ;
- `dev` : interfaces vers des périphériques ;
- `etc` : les fichiers de configuration du système ;
- `home` : répertoires personnels des utilisateurs ;
- `media` : les données des périphériques de stockage que l'on doit "monter" (brancher).

Chaque élément de l'arborescence est décrit par un chemin indiquant comment naviguer dans l'arborescence pour accéder au ficher.

Un chemin peut être

- Absolu : c'est la suite des répertoires à traverser depuis la racine.

  Ex : `/home/alice/toto.txt`

- Relatif : c'est la suite des répertoires à traverser depuis le répertoire courant.

  Ex : depuis `home`, `alice/toto.txt`

  Depuis `bin` : `../home/alice/toto.txt`

Symboles :

`.` désigne le répertoire courant ;

`..` désigne le répertoire parent ;

$\sim$ désigne le répertoire personnel de l'utilisateur (répertoire courant par défaut)

2.1.2 Manipulation de ficher depuis un programme

Chaque langage a sa propre syntaxe pour les lectures et écritures dans des fichiers.

• En OCaml, il faut ouvrir un canal de communication / un flux de données avant de pouvoir lire ou écrire dans le ficher. Il faut penser à refermer ce canal.

– Type :

`in_channel` pour la lecture ;

`out_channel` pour l'écriture.

– Ouverture d'un canal : on utilise les fonctions :

`open_in : string -> in_channel` ;

`open_out : string -> out_channel`

La chaîne de caractères en argument est un chemin (relatif / absolu) vers le ficher.

– Fermeture d'un canal : on utilise les fonctions :

`close_in : in_channel -> unit` ;

`close_out : out_channel -> unit`

– Écriture / lecture :

On écrit dans un ficher grâce à la fonction

`output_string : out_channel -> string -> unit`

Ex :

```
let oc = open_out "/home/alice/toto.txt" in
    output_string oc "bonjour\n";
    close_out oc
```

On lit *via* la fonction `input_line : in_channel -> string` qui lit une ligne du fichier et la renvoie sans le caractère de retour à la ligne. S'il n'y a plus de ligne à lire, la fonction soulève l'exception `End_of_file`.

Ex :

```
let ic = open_int "/home/alice/toto.txt" in
try
    while true do
        let s = input_line ic in
        Printf.printf "%s\n" s
    done
with
| End_of_file -> close_in ic
```

• En C : les flux de données sont manipulés *via* des pointeurs.

– Ouverture :

On utilise la fonction `fopen`, de prototype

`FILE* fopen(char path[], char mode[])`

où `path` désigne un chemin vers le fichier et `mode` est le mode d'ouverture : `"r"` pour la lecture ou `"w"` pour l'écriture (autres modes H.P).

Il faut vérifier que le résultat n'est pas `NULL`.

– Fermeture :

on utilise la fonction `fclose`, de prototype

`void fclose(FILE* file)`

– Écriture / lecture :

Pour l'écriture, in utilise la fonction `fprintf`, équivalente à `printf` avec un argument supplémentaire de type `FILE*` (en première position).

Ex :

```
fprintf(file, "%p\n", file);
fprintf(file, "%d\n", x + y);
```

On lit grâce à la fonction `fscanf`, qui fonctionne comme `scanf` avec un argument supplémentaire que `fprintf`.

On peut tester la valeur de retour de `fscanf` pour savoir si on a atteint la fin du fichier (valeur spéciale `EOF`)

Ex :

```
FILE* file = fopen("toto.txt", "r");
if (file == NULL) {
    printf("open failed\n");
    assert false;
}

char s[81];

while (fscanf(file, "%s", s) != EOF)
    printf("%s\n", s);
fclose(file);
```

Rq : `printf` et `scanf` sont les équivalents de `fprintf` et `fscanf` où l'on utilise le flux standard :

- `stdin` : flux d'entrée standard ;
- `stdout` : flux de sortie standard ;
- `stderr` : flux d'erreur standard, souvent égal à `stdout`.

OCaml dispose aussi des flux standards, de mêmes noms. Il y a aussi une fonction `Printf.printf` en OCaml.

2.1.3 Implémentation de l'arborescence de fichiers

Dans les exemples précédants, on a accédé au contenu des fichiers de manière séquentielle. En pratique, la mémoire est organisée en blocs auxquels on peut accéder :

- directement, *via* leur adresse en mémoire ;
- séquentiellement : on parcourt les blocs successifs à partir d'un bloc départ.

Les accès séquentiels sont en général plus rapides que les accès directs. Dans l'idéal, les fichiers seraient donc stockés dans des blocs contigus. Seulement, il y a un problème de fragmentation de la mémoire : les modifications successives de l'arborescence (création / suppression / redimmentionnement des fichiers) laissent des plages de blocs innutilisables. De plus, la gestion des blocs dépend du système de fichiers utilisé : le système d'exploitation n'applique pas la même stratégie pour éviter la fragmentation selon le système de fichier.

Nous allons voir le principe de l'allocation indexée, utilisée dans les systèmes type UNIX.

Un fichier est en fait un nom associé à un noeud d'index (inode) qui est une structure codée dans un / plusieurs blocs de mémoire et contenant les métadonnées du fichier (*cf* 2.1.4) et les adresses des différents blocs contenant les données du fichier.

Cela résout le problème de fragmentation (tout bloc devient utilisable en mettant à jour le noeud d'index du fichier concerné) sans trop réduire les performances des accès aux fichiers.

Un répertoire est alors une liste de noeuds d'index associés à des noms. La séparation du nom des autres métadonnées est importante pour partager les blocs de mémoire associés à un fichier entre plusieurs noms, vus comme des "raccourcis".

Il y a deux manières de procéder :

- Le lien physique (hard link) : on crée deux noms pour le *même* noeud d'index.

  Il y a des restrictions : pas de lien physique vers des répertoires, ni vers un autre système de fichier.

- Le lien symbolique (symbolic link) : un lien symbolique est un nouveau fichier, donc un noeud d'index différent, ne contenant dans ses données que le chemin vers le fichier auquel il fait référence. Les restrictions des liens physiques ne s'appliquent pas aux liens symboliques.

2.1.4 Attributs des fichiers

Les attributs ou métadonnées, qui sont stockés dans le noeud d'index d'un fichier fournissent des informations sur le fichier.

Dans la norme POSIX, on y trouve en particulier :

- La taille du fichier (en octets) ;
- Le nombre de blocs alloués ;
- Le nombre de liens physiques vers le noeud d'index ;
- Le numéro de l'index ;
- Les identifiants de l'utilisateur propriétaires et du groupe du fichier ;
- Des données d'horodatage concernant les accès et modifications du fichier ;
- Les droits d'accès au fichier, identifiant les opérations autorisées sur ce fichier selon l'utilisateur.

Il y a trois catégories d'utilisateur :

- Le proriétaire du fichier ;
- Les membres du groupe du fichier ;

– Les autres.

Pour chaque catégorie, il est possible d'autoriser ou d'interdire 3 opérations :

– La lecture du fichier ;

– L'écriture du fichier ;

– L'exécution du fichier en tant que programme exécutable.

Cas particulier pour les répertoires :

– L'accès en lecture permet de lister le contenu du répertoire ;

– L'accès en écriture permet de créer / supprimer / renommer les fichiers contenus dans le répertoire (nécessite l'accès en exécution) ;

– L'accès en exécution permet de traverser le répertoire, i.e d'accéder au contenu du répertoire pour les fichiers pour lesquels on a des droits d'accès.

2.2 Manipulations depuis le terminal

2.2.1 Navigation dans l'arborescence de fichiers

• Se repérer : l'environement du terminal est placé dans un répertoire *courant*. On utilise la commande `pwd` (*print working directory*) pour déterminer le répertoire courant.

Pour connaître le contenu d'un répertoire, on utilise la commande `ls` (*list*).

Deux options

– Sans argument, la commande liste le contenu du répertoire courant ;

– On peut passer en argument un chemin relatif / absolu vers un répertoire et la commande liste alors le contenu de ce répertoire.

Ex :

```
ls /home/Alice
ls Bureau #Or Desktop
ls ../tp
```

• Se déplacer : on utilise la commande `cd` (*change directory*) pour changer de répertoire courant.

Sans argument, cette commande déplace le répertoire courant vers le répertoire utilisateur (∼ : `/home/user`).

On peut choisir le répertoire courant en donnant un chemin vers un répertoire.

Ex :

```
cd ../..
cd ~
cd Bureau
cd ~/git/coursCPGE
```

2.2.2 Modifications de l'arborescence de fichiers

- Création de fichiers

  – Fichiers : On peut utiliser un éditeur de texte (ex : commande `vim`, ou `emacs`).

  On peut également créer un fichier vide à l'aide de la commande `touch`, qui prend en argument un chemin vers le fichier à créer.

  Ex :

```
touch DS/ds_03.tex
touch toto.txt
```

  Si le fichier existe déjà, la commande modifie les métadonnées d'horodatage.

  – Répertoire : on crée un répertoire à l'aide de la commande `mkdir` (*make directory*).

  Ex :

```
mkdir /tmp/test
```

  – Copie : on peut copier un fichier ou un répertoire à l'aide de la commande `cp` (*copy*) qui prend en arguments un chemin vers le fichier / répertoire source et un chemin vers la destination.

  Ex :

```
cp toto.txt ~/Bureau
cp toto.txt ~/Bureau/bla.txt
cp bla.txt bli.txt
```

  Pour les répertoires, on doit utilier l'option `-r` (*recursive*).

  Ex :

```
cp -r /tmp ~/Documents
cp -r /tmp ~/Documents/copie_tmp #If copie_tmp don't exists,
    copy create copie_tmp in ~/Documents and /tmp/* in copie_tmp
```

- Suppression :

  – Fichiers : on supprime un fichier grâce à la commande `rm` (*remove*) qui prend en argument un chemin vers le fichier à supprimer.

  Ex :

```
rm /tmp/toto.txt
```

  On peut donner plusieurs chemins pour supprimer plusieurs fichiers, et on dispose du symbole `*`, qui représente n'importe quelle chaîne de caractères.

```
rm *.ml #Delete all files in current working directory with the
     .ml extension.
rm tp_*.c
```

  – Répertoires : on supprime un répertoire *vide* à l'aide de la commande `rmdir` (*remove directory*). Plutôt que de vider un répertoire pour utiliser `rmdir`, on utilise plutôt l'option `-r` de `rm` : `rm -r [DIR...]`.

Rq : on peut aussi utiliser `rm -d [empty_dir...]` pour supprimer un répertoire vide.

• Déplacement :

On déplace un fichier à l'aide de la commande `mv` (*move*) prenant les mêmes arguments que `cp`.

Pas d'option `-r`.

2.2.3 Gestion des systèmes de fichier

• Ajout d'un système de fichier : on utilise la commande `mount` (on parle de montage), qui prend en arguments un chemin vers le périphérique, et un chemin vers le répertoire qui contiendra l'arborescence du prériphérique. Sans arguments, la commande `mount` liste les points de montage.

Ex :

```
mount /dev/sdc1 Document/my_USB_key
```

• Démontage : on utilise la commande `umount`, qui prend en argument un chemin vers le point de montage.

2.2.4 Gestion des droits

• Exécuter une commande en tant qu'administrateur : on préfixe la commande par la commande `sudo` (*super user do*).

Ex :

```
sudo apt upgrade
```

L'utilisateur doit faire partie du groupe sudo, *aka* les *sudoers*.

• Gérer les droits des fichiers / répertoires : on représente les droits d'accès par une chaîne de caractères `rwxrwxrwx`, en plaçant des `-` pour les droits non attribués. Les trois premiers caractères représentent les droits de l'utilisateur, les trois suivants le groupe, et les trois derniers les autres.

Ex : `rwxrw-r-`

On peut changer cette chaîne à l'aide de la commande `chmod` (*change mode*) qui peut être utilisée de 2 manières :

– On modifie les droits d'une catégorie (`u` pour utilisateur, `g` pour groupe, `o` pour les autres) par ajout (`+`), suppression (`-`) ou définition (`=`) de droits.

Ex :

```
chmod u+rw <path>
chmod o=r <path>
```

– On utilise une représenation numérique des droits (4 pour la lecture, 2 pour l'écriture, et 1 pour l'exécution) que l'on somme pour chaque catégorie.

Ex :

```
chmod 751 <path> # 7 for u : 4+2+1 ; 5 for g : 4+1 ; 1 for o.
```

2.2.5 Création de liens

On utilise la commande `ln` (*link*) qui prend les mêmes arguments que `cp` et `mv`. Par défaut, le lien est physique. Pour obtenir un lien symbolique, on utilise l'option `-s`.

Ex :

```
ln -s /dev/sdb ~/Documents/data
ln tp/macros.tex td/
```

2.2.6 Redirection

On peut rediriger les flux d'entrée / sortie des programmes.

• Rediriger la sortie vers un fichier : on utilise `>` pour la sortie standard et `2>` pour la sortie d'erreur.

Ex :

```
./prog 2> erreur.log
echo bonjour > /tmp/bonjour.txt
```

On utilise `»` et `2»` pour ajouter à la fin sans écraser le contenu.

• Rediriger l'entrée vers un fichier : on utilise `<`

Ex :

```
ls > contenu.txt
sort < contenu.txt > sorted_content.txt
```

• Rediriger la sortie d'un programme vers l'entrée d'un autre : on utilise un *pipe* (tube en français), dénoté par `|`.

Ex :

```
ls | sort > sorted_content.txt
ls | grep ".ml$"
```

3 Représentation de l'information : types numériques

3.1 Représentation des entiers

3.1.1 Intro

En machine, les données sont stockées sous forme binaire, *i.e* sous la forme de suites de 0 et de 1. Cela fonctionne car de nombreux types d'information peuvent être encodés par des entiers, qui eux-mêmes ont plusieurs représentations, dont la représentation binaire.

3.1.2 Bases de numération

Théorème :

Soit $b \in \mathbb{N} \setminus \{0\ ;\ 1\}$.

Alors

$$\forall n \in \mathbb{N},\ \exists!\ (n_k)_{k\in\mathbb{N}} \quad \text{tq} \begin{cases} \forall k \in \mathbb{N},\ n_k \in [\![0\ ;\ b-1]\!] \\ (n_k)_{k\in\mathbb{N}} \text{ stationne ultimement en } 0 \\ n = \displaystyle\sum_{k\in\mathbb{N}} n_k b^k = \sum_{k=0}^{p} n_k b^k \quad \text{avec } p \text{ tq } \forall k > p,\ n_k = 0 \end{cases}$$

On appelle représentation en base b la suite (finie) $n_p, \ldots, n_0$ et on note $n = \langle n_p,\ \ldots, n_0 \rangle_b$

□

• Unicité :

On suppose l'existence d'un tel développement.

$$n = \sum_{k=0}^{p} n_k b^k$$

On remarque que $n = \left(\displaystyle\sum_{k=1}^{p} n_k b^{k-1}\right) b + n_0$, avec $0 \leq n_0 < b$.

Donc n_0 est nécessairement le reste de la division euclidienne de n par b.

On note $N_1 = \displaystyle\sum_{k=1}^{p} n_k b^{k-1}$ le quotient (aussi uniquement déterminé) et on recommence.

On montre par récurrence que les n_k sont uniquement déterminés.

• Existence

Si $n = 0$: $(0)_{k\in\mathbb{N}}$ convient.

Si $n > 0$: on définit les suites $(n_k)_{k\in\mathbb{N}}$ et $(N_k)_{k\in\mathbb{N}}$ par

$N_0 = n$

$$\forall k \in \mathbb{N},\ \begin{cases} n_k \text{ est le reste dans la division euclidienne de } N_k \text{ par } b \\ N_{k+1} \text{ est le quotient dans la division euclidienne de } N_k \text{ par } b \end{cases}$$

Par définition, $\forall k \in \mathbb{N},\ n_k \in [\![0\ ;\ b-1]\!]$.

$$\begin{aligned} n &= N_0 \\ &= N_1 \cdot b + n_0 \\ &= (N_2 \cdot b + n_1) + n_0 = N_2 \cdot b^2 + n_1 \cdot b + n_0 \end{aligned}$$

Une récurrence (finie) permettra de démontrer $n = \displaystyle\sum_{k\in\mathbb{N}} n_k b^k$ quand on aura montré que $(n_k)_{k\in\mathbb{N}}$ stationne en 0 à partir d'un certain rang.

Soit $k \in \mathbb{N}$.

$N_k = bN_{k+1} + n_k$ et $b > 1$
Donc $2N_{k+1} \leq bN_{k+1} \leq N_k$
Donc par récurrence, $2^k N_k \leq N_0 = n$
Donc, pour k tq $2^k > n$, $N_k = 0$ et $\forall p \geq k+1,\ n_p = 0$
■

Ex :
$b = 10$: représentation binaire ;
$b = 2$: représentation binaire ;
$b = 16$: représentation hexadécimale : on utilise comme "chiffres" les symboles `0123456789abcdef`.

décimal	binaire	hexadécimal
17	10001	11
15	1111	f
0	0	0

3.1.3 Remarques

– Il y a aussi une représentation unaire (base 1).

– Les aglorithmes vu en primaire s'adaptent à toutes les bases.

– Mais en pratique, on ne peut pas représenter tous les entiers : il y a un nombre maximal de chiffres. Si le résultat d'une opération doit utiliser plus de chiffres, il est tronqué (ce qui revient à faire des calculs modulos b^p où p est la taille maximale). On appelle cela un dépassement de capacité (*overflow* en anglais).

– La gestion des entiers négatifs nécessite un choix de représentation. Types en C :
• Taille fixée explicitement : `int8_t`, `int32_t`, `int64_t` pour les entiers relatifs ; `uint8_t`, `uint32_t`, `uint64_t` pour les entiers naturels.
On parle d'entiers non signés (unsigned).

• Si la taille importe peu : `int` et `unsigned int` (dont la taille est généralement 32 bits).

3.1.4 Entiers signés

– Solution naïve : on utilise un bit de signe (0 pour positif, 1 pour négatif).
Pb : on a 2 représentations de 0 ;
Les aglorithmes usuels ne fonctionnent plus :
$4 + (-2) = 2$ mais $\underbrace{\langle 0000\ 0100\rangle_2}_{4} + \underbrace{\langle 1000\ 0010\rangle_2}_{-2} = \underbrace{\langle 1000\ 0110\rangle_2}_{-6}$

– Complément à 2 : sur p bits, les entiers entre -2^{p-1} et $2^{p-1} - 1$, représentés de

manière unique.
Idée : on note $\hat{n}$ le complémentaire de n (en inversant les bits)
Alors $n + \hat{n} = \langle 1 \dots 1 \rangle_2 = 2^p - 1$
donc $\hat{n} + 1 = 2^p - n \equiv -n \ [2^p]$

Si $n \geq 0$, on le représente normalement ;
Si $n < 0$, on le représente par $\hat{n} + 1$, *i.e* $2^p - |n|$
Ex : $6 + (-6) = 0$
et $\langle 0000\ 0110 \rangle_2 + \langle 1111\ 1010 \rangle_2 = \langle 1\ 0000\ 0000 \rangle_2 = \langle 0000\ 0000 \rangle_2$ par *overflow*.

3.2 Représentation des réels

3.2.1 Idée

On s'inspire des bases de numération par les entiers et on représente séparément les parties entière et décimale :

$$\langle x_p \dots x_0 \ . \ x_{-1} \dots x_{-q} \rangle_b = \sum_{k=-q}^{p} x_k b^k$$

Ex : $\langle 4.35 \rangle_{10} = 4 \cdot 10^0 + 3 \cdot 10^{-1} + 5 \cdot 10^{-2}$
$\langle 10.11 \rangle_2 = 2^1 + 2^{-1} + 2^{-2} = \langle 2.75 \rangle_{10}$
Rq : une telle représentation n'est pas nécessairement finie.

On appelle nombre décimal (resp. dyadique) un nombre dont la représentation en base 10 (resp. 2) est finie.

Calcul de la représentation :
Soit $x \in]0\ ;\ 1[$. On veut écrire x sous la forme $\langle 0.x_{-1}x_{-2} \dots \rangle_b$.
On remarque que $b \cdot x = \langle x_{-1} \ . \ x_{-2}x_{-3} \dots \rangle_b$ donc $x_{-1} = \lfloor bx \rfloor$
Comme $x_{-1} - \lfloor bx \rfloor = \langle 0 \ . \ x_{-2}x_{-3} \dots \rangle_b$, on peut recommencer.

Exemple :
$\langle 0.7 \rangle_{10}$ en base 2 ?
$2 \times 0.7 = 1.4 \to 1$
$2 \times 0.4 = 0.8 \to 0$
$2 \times 0.8 = 1.6 \to 1$
$2 \times 0.6 = 1.2 \to 1$
$2 \times 0.2 = 0.4 \to 0$
$\langle 0.10110\ 0110\ 0110 \dots \rangle_2$

Rq : pour utiliser cette représentation, on fixe les nombres de chiffres avant et après la virgule. On parle de représentation de virgule fixe.

3.2.2 Représentation en virgule flottante

On s'inspire de la représentation scientifique des nombres.

$x = (-1)^s \cdot m \cdot 2^e$ où :
$s \in \{0\ ;\ 1\}$ est le bit de signe ;
$m \in [1\ ;\ 2[$ est la mantisse ;
$e \in \mathbb{Z}$ est l'exposant.

Rq : on ne peut pas représenter 0.
La représentation binaire de m est de la forme $\langle 1.m_{-1}m_{-2}\ldots\rangle_2$.
On fait des erreurs d'arrondi si m n'est pas dyadique.

Norme `IEEE-754` : établie en 1985, elle définit une convention sur la représentation des nombres en virgule flottante.

	Nb bits	Signe	Exposant	Mantisse	Type
Simple précision	32	1	8	23	`float` en C
Double précision	64	1	11	52	`double` en C, `float` en OCaml

Convention : on ne stocke pas la partie entière de la mantisse
l'exposant e est représenté par l'écriture binaire de $e + 2^{p-1} - 1$ où p est le nombre de bits d'exposant.
Attention, pas de complément à 2.
Ex : en simple précision : `0 1000 0010 1100 0000 0000 0000 0000 000`
$s = 0$ nombre positif
$e = \langle 1000\ 0010\rangle_2 - 2^7 + 1 = 130 - 127 = 3$
$m = \langle 1.11\rangle_2 = \langle 1.75\rangle_{10}$
Donc $x = +1.75 \cdot 2^3 = 14$

Rq : En raison des erreurs d'approximation, on ne teste jamais l'égalité de 2 flottants. On compare plutôt la valeur absolue de leur différence à l'erreur liée à leur précision.
Ex : en simple précision l'erreur est de l'ordre de $2^e \cdot 2^{-23}$ donc on compare à 10^{-7}
En double précision, on compare à 10^{-16}

Cas particuliers de la norme `IEEE-754` :

- Si les bits d'exposant ne sont pas $0\ldots0$ ou $1\ldots1$, on parle de forme normalisée ;
- Si les bits d'exposant sont $1\ldots1$:
  - + Si les bits de la mantisse sont $0\ldots0$: cela encode $\pm\infty$ selon le signe ;
  - + Si au moins un bit de mantisse vaut 1, cela encode `NaN` (Not a Number) ;
- Si les bits d'exposant sont $0\ldots0$, on parle de forme dénormalisée.

  Le nombre représenté est alors, avec p bits d'exposant et q bits de mantisse :

$$(-1)^s \cdot m \cdot 2^e$$

où $e = 2 - 2^{p-1}$ et $m = \langle 0.m_{-1}\ldots m_{-q}\rangle_2$.