Types de données

Durant la première semaine, nous avons abordé quelques types de données de base dont les `int` et les `char`. Pour utiliser ces types de données à bon escient, il est important de comprendre en détail la façon dont ils sont supportés par le compilateur et leurs limitations. Celles-ci dépendent souvent de leur représentation en mémoire et durant cette semaine nous allons commencer à analyser de façon plus détaillée comment la mémoire d'un ordinateur est structurée.

Nombres entiers

Toutes les données stockées sur un ordinateur sont représentées sous la forme de séquences de bits. Ces séquences de bits peuvent d'abord permettre de représenter des nombres entiers. Un système informatique peut travailler avec deux types de nombres entiers :

les nombres entiers signés (`int` notamment en C)
les nombres entiers non-signés (`unsigned int` notamment en C)

Une séquence de n bits $b_0 \dots b_i \dots b_{n-1}$ peut représenter le nombre entier $\sum_{i=0}^{n-1} b_i \times 2^i$. Par convention, le bit b_{n-1}, associé au facteur du plus grand indice 2^{n-1}, est appelé le **bit de poids fort** tandis que le bit b_0, associé à 2^0, est appelé le **bit de poids faible**. Les suites de bits sont communément écrites dans l'ordre descendant des indices $b_{n-1} \dots b_i \dots b_0$. À titre d'exemple, la suite de bits `0101` correspond à l'entier non signé représentant la valeur cinq. Le bit de poids fort (resp. faible) de cette séquence de quatre bits (ou **nibble**) est `0` (resp. `1`). La table ci-dessous reprend les différentes valeurs décimales correspondant à toutes les séquences de quatre bits consécutifs.

binaire	octal	hexadécimal	décimal
0000	00	0	0
0001	01	1	1
0010	02	2	2
0011	03	3	3
0100	04	4	4
0101	05	5	5
0110	06	6	6
0111	07	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15

Écrire une séquence de bits sous la forme d'une suite de `0` et de `1` peut s'avérer fastidieux. La représentation décimale traditionnelle n'est pas pratique (optimale) non plus car il faut un ou deux chiffres pour représenter une séquence de quatre bits (ou **nibble**) en fonction de la valeur de ces bits. En pratique, de nombreux systèmes informatiques utilisent une représentation hexadécimale pour afficher des séquences de bits. Cette notation hexadécimale est définie sur base de la table ci-dessus en utilisant des lettres pour représenter chaque séquence de quatre bits dont la valeur numérique est supérieure à `9`. Tout comme avec la représentation décimale habituelle, il est possible d'utiliser la représentation hexadécimale pour de longues séquences de bits. La notation octale est parfois utilisée et est supportée par les compilateurs C. Elle utilise un chiffre pour représenter trois bits consécutifs. A titre d'exemple, voici quelques conversions de nombres en notation décimale vers les notations hexadécimales et binaires.

L'entier décimal `123` s'écrit `0x7b` en notation hexadécimale et `0b00000000000000000000000001111011` en notation binaire
L'entier décimal `7654321` s'écrit `0x74cbb1` en notation hexadécimale et `0b00000000001110100110010111011001` en notation binaire

Dans ces exemples, nous avons pris la convention de représentation des nombres en langage C. En C, un nombre décimal s'écrit avec la représentation standard. Un nombre entier en notation hexadécimale est par convention préfixé par `0x` tandis qu'un nombre entier en notation binaire est préfixé par `0b`. Ainsi, les déclarations ci-dessous correspondent toutes à la même valeur.

```
int i;
i = 123;   // décimal
i = 0x7b;  // hexadécimal
i = 0173;  // octal !!
//  i = 0b1111011; // binaire, seulement certains compilateurs
```

Certains compilateurs permettent d'entrer des constantes en binaire directement en préfixant le nombre avec `0b` comme dans `i=0b1111011;`. Cependant, cette notation n'est pas portable sur tous les compilateurs. Elle doit donc être utilisée avec précaution, contrairement à la notation hexadécimale qui fait partie du langage.

Note

Notation octale

La notation octale peut poser des surprises désagréables au programmeur C débutant. En effet, pour des raisons historiques, les compilateurs C considèrent qu'un entier dont le premier chiffre est `0` est écrit en représentation octale et non en représentation décimale.

Ainsi, le fragment de code ci-dessous affichera à l'écran le message `65 et 53 sont différents` car le compilateur C interprète la ligne `j=065;` comme contenant un entier en notation octale et non décimale.

```
int i, j;
i = 65;   // décimal
j = 065;  // octal !!!
if (i == j)
  printf("%d et %d sont égaux\n", i, j);
else
  printf("%d et %d sont différents\n", i, j);
```

Le langage C supporte différents types de données qui permettent de représenter des nombres entiers non signés. Les principaux sont repris dans le tableau ci-dessous.

Type	Explication
`unsigned short`	Nombre entier non signé représenté sur au moins 16 bits
`unsigned int`	Nombre entier non signé représenté sur au moins 16 bits
`unsigned long`	Nombre entier non signé représenté sur au moins 32 bits
`unsigned long long`	Nombre entier non signé représenté sur au moins 64 bits

Le nombre de bits utilisés pour stocker chaque type d'entier non signé peut varier d'une implémentation à l'autre. Le langage C permet de facilement déterminer le nombre de bits utilisés pour stocker un type de données particulier en utilisant l'expression `sizeof`. Appliquée à un type de données, celle-ci retourne le nombre d'octets que ce type occupe. Ainsi, sur de nombreuses plateformes, `sizeof(int)` retournera la valeur `4`.

Les systèmes informatiques doivent également manipuler des nombres entiers négatifs. Cela se fait en utilisant des nombres dits signés. Au niveau binaire, il y a plusieurs approches possibles pour représenter des nombres signés. La première est de réserver le bit de poids fort dans la représentation du nombre pour stocker le signe et stocker la valeur absolue du nombre dans les bits de poids faible. Mathématiquement, un nombre de *n* bits utilisant cette notation pourrait se convertir via la formule $(-1)^{b_{n-1}} \times \sum_{i=0}^{n-2} b_i \times 2^i$.

En pratique, cette notation est rarement utilisée pour les nombres entiers car elle rend l'implémentation des circuits électroniques de calcul plus compliquée. Un autre inconvénient de cette notation est qu'elle utilise deux séquences de bits différentes pour représenter la valeur zéro (`00...0` et `10...0`). La représentation la plus courante pour les nombres entiers signés est la notation en *complément à 2*. Avec cette notation, une séquence de *n* bits correspond au nombre entier $-(b_{n-1}) \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$. Avec cette notation, le nombre négatif de 4 bits le plus petit correspond à la valeur `-8`. En notation en complément à deux, il n'y a qu'une seule représentation pour le nombre zéro, la séquence dont tous les bits valent `0`. Par contre, il existe toujours un nombre entier négatif qui n'a pas d'équivalent positif.

binaire	décimal signé
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

En C, les types de données utilisés pour représenter des entiers sont signés par défaut. Ils ont la même taille que leurs équivalents non signés et sont repris dans la table ci-dessous.

Type	Explication
`short`	Nombre entier signé représenté sur au moins 16 bits
`int`	Nombre entier signé représenté sur au moins 16 bits
`long`	Nombre entier signé représenté sur au moins 32 bits
`long long`	Nombre entier signé représenté sur au moins 64 bits

Dans de nombreux systèmes Unix, on retrouve dans le fichier **stdint.h** les définitions des types d'entiers supportés par le système avec le nombre de bits et les valeurs minimales et maximales pour chaque type. La table ci-dessous reprend à titre d'exemple l'information relative aux types `short` (16 bits) et `unsigned int` (32 bits).

Type	Bits	Minimum	Maximum
`short`	16	-32768	32767
`unsigned int`	32	0	4294967295

Le fichier **stdint.h** contient de nombreuses constantes qui doivent être utilisées lorsque l'on a besoin des valeurs minimales et maximales pour un type donné. Voici à titre d'exemple quelques unes de ces valeurs :

```
#define INT8_MAX         127
#define INT16_MAX        32767
#define INT32_MAX        2147483647
#define INT64_MAX        9223372036854775807LL

#define INT8_MIN         -128
#define INT16_MIN        -32768
#define INT32_MIN        (-INT32_MAX-1)
#define INT64_MIN        (-INT64_MAX-1)

#define UINT8_MAX        255
#define UINT16_MAX       65535
#define UINT32_MAX       4294967295U
#define UINT64_MAX       18446744073709551615ULL
```

L'utilisation d'un nombre fixe de bits pour représenter les entiers peut causer des erreurs dans certains calculs. Par exemple, voici un petit programme qui affiche les 10 premières puissances de cinq et dix.

```
short int i = 1;
unsigned short j = 1;
int n;

printf("\nPuissances de 5 en notation signée\n");
for (n = 1; n < 10; n++) {
  i = i * 5;
  printf("5^%d=%d\n", n, i);
}

printf("\nPuissances de 10 en notation non signée\n");
for (n = 1; n < 10; n++) {
  j = j * 10;
  printf("10^%d=%d\n", n, j);
}
```

Lorsqu'il est exécuté, ce programme affiche la sortie suivante.

```
Puissances de 5 en notation signée
5^1=5
5^2=25
5^3=125
5^4=625
5^5=3125
5^6=15625
5^7=12589
5^8=-2591
5^9=-12955
```

```
Puissances de 10 en notation non signée
10^1=10
10^2=100
10^3=1000
10^4=10000
10^5=34464
10^6=16960
10^7=38528
10^8=57600
10^9=51712
```

Il est important de noter que le langage C ne contient aucun mécanisme d'exception qui permettrait au programmeur de détecter ce problème à l'exécution. Lorsqu'un programmeur choisit une représentation pour stocker des nombres entiers, il est essentiel qu'il ait en tête l'utilisation qui sera faite de cet entier et les limitations qui découlent du nombre de bits utilisés pour représenter le nombre en mémoire. Si dans de nombreuses applications ces limitations ne sont pas pénalisantes, il existe des applications critiques où un calcul erroné peut avoir des conséquences énormes **[Bashar1997]**.

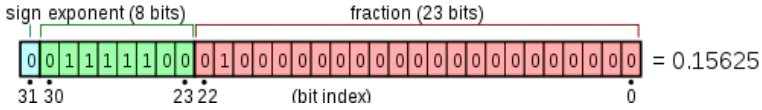
Nombres réels

Outre les nombres entiers, les systèmes informatiques doivent aussi pouvoir manipuler des nombres réels. Ceux-ci sont également représentés sous la forme d'une séquence fixe de bits. Il existe deux formes de représentation pour les nombres réels :

la représentation en **simple précision** dans laquelle le nombre réel est stocké sous la forme d'une séquence de 32 bits;
la représentation en **double précision** dans laquelle le nombre réel est stocké sous la forme d'une séquence de 64 bits.

La plupart des systèmes informatiques qui permettent de manipuler des nombres réels utilisent le standard IEEE-754. Un nombre réel est représenté en virgule flottante et la séquence de bits correspondante est décomposée en trois parties **[2]** :

le bit de poids fort indique le signe du nombre. Par convention, `0` est utilisé pour les nombres positifs et `1` pour les nombres négatifs.
e bits sont réservés pour stocker l'exposant **[1]**.
Les *f* bits de poids faible servent à stocker la partie fractionnaire du nombre réel.



Exemple de nombre en virgule flottante (simple précision). (source : Wikipedia)

En simple (resp. double) précision, *8* (resp. *11*) bits sont utilisés pour stocker l'exposant et *23* (resp. *52*) bits pour la partie fractionnaire. L'encodage des nombres réels en simple et double précision a un impact sur la précision de divers algorithmes numériques. Une analyse détaillée de ces problèmes sort du cadre de ce cours, mais il est important de noter deux propriétés importantes de la notation en virgule flottante utilisée actuellement. Ces deux propriétés s'appliquent de la même façon à la simple qu'à la double précision.

une représentation en virgule flottante sur n bits ne permet jamais de représenter plus de 2^n nombres réels différents;
les représentations en virgule flottante privilégient les nombres réels compris dans l'intervalle $[-1, 1]$. On retrouve autant de nombres réels représentables dans cet intervalle que de nombres dont la valeur absolue est supérieure à *1*.

En C, ces nombres en virgule flottante sont représentés en utilisant les types `float` (simple précision) et `double` (double précision). Les fichiers **float.h** et **math.h** définissent de nombreuses constantes relatives à ces types. Voici, à titre d'exemple, les valeurs minimales et maximales pour les `float` et les `double` ainsi que les constantes associées. Pour qu'un programme soit portable, il faut utiliser les constantes définies dans **float.h** et **math.h** et non leurs valeurs numériques.

```
#define FLT_MIN 1.17549435e-38F
#define FLT_MAX 3.40282347e+38F

#define DBL_MIN 2.2250738585072014e-308
#define DBL_MAX 1.7976931348623157e+308
```

Les tableaux

En langage C, les tableaux permettent d'agréger des données d'un même type. Il est possible de définir des vecteurs et des matrices en utilisant la syntaxe ci-dessous.

```
#define N 10
int vecteur[N];
float matriceC[N][N];
float matriceR[N][2*N];
```

Les premières versions du langage C ne permettaient que la définition de tableaux dont la taille est connue à la compilation. Cette restriction était nécessaire pour permettre au compilateur de réserver la zone mémoire pour stocker le tableau. Face à cette limitation, de nombreux programmeurs définissaient la taille du tableau via une directive `#define` du pré-processeur comme dans l'exemple ci-dessus. Cette directive permet d'associer une chaîne de caractères quelconque à un symbole. Dans l'exemple ci-dessus, la chaîne `10` est associée au symbole `N`. Lors de chaque compilation, le préprocesseur remplace toutes les occurences de `N` par `10`. Cela permet au compilateur de ne traiter que des tableaux de taille fixe.

Un tableau à une dimension peut s'utiliser avec une syntaxe similaire à celle utilisée par Java. Dans un tableau contenant `n` éléments, le premier se trouve à l'indice `0` et le dernier à l'indice `n-1`. L'exemple ci-dessous présente le calcul de la somme des éléments d'un vecteur.

```
int i;
int sum = 0;
for (i = 0; i < N; i++) {
  sum += v[i];
}
```

Le langage C permet aussi la manipulation de matrices carrées ou rectangulaires qui sont composées d'éléments d'un même type. L'exemple ci-dessous calcule l'élément minimum d'une matrice rectangulaire. Il utilise la constante `FLT_MAX` qui correspond au plus grand nombre réel représentable avec un `float` et qui est définie dans **float.h**.

```
#define L 2
#define C 3
float matriceR[L][C] = { {1.0,2.0,3.0},
                         {4.0,5.0,6.0} };
int i, j;
float min = FLT_MAX;
for (i = 0; i < L; i++)
  for (j = 0; j < C; j++)
    if (matriceR[i][j] < min)
      min=matriceR[i][j];
```

Les compilateurs récents qui supportent **[C99]** permettent l'utilisation de tableaux dont la taille n'est connue qu'à l'exécution. Nous en reparlerons ultérieurement.

Caractères et chaînes de caractères

Historiquement, les caractères ont été représentés avec des séquences de bits de différentes longueurs. Les premiers ordinateurs utilisaient des blocs de cinq bits. Ceux-ci permettaient de représenter 32 valeurs différentes. Cinq bits ne permettent pas facilement de représenter à la fois les chiffres et les lettres et les premiers ordinateurs utilisaient différentes astuces pour supporter ces caractères sur 5 bits. Ensuite, des représentations sur six puis sept et huit bits ont été utilisées. Au début des années septante, le code de caractères ASCII sur `7` et `8` bits s'est imposé sur un grand nombre d'ordinateurs et a été utilisé comme standard pour de nombreuses applications et notamment sur Internet **RFC 20**. La table de caractères ASCII définit une correspondance entre des séquences de bits et des caractères. **RFC 20** contient la table des caractères ASCII représentés sur 7 bits. À titre d'exemple, le chiffre *0* correspond à l'octet *0b00110000* et le chiffre *9* à l'octet *0b00111001*. La lettre *a* correspond à l'octet *0b01100001* et la lettre *A* à l'octet *0b01000001*.

Les inventeurs du C se sont appuyés sur la table ASCII et ont choisi de représenter un caractère en utilisant un octet. Cela correspond au type `char` que nous avons déjà évoqué.

Concernant le type `char`, il est utile de noter qu'un `char` est considéré en C comme correspondant à un entier. Cela implique qu'il est possible de faire des manipulations numériques sur les caractères. À titre d'exemple, une fonction **toupper(3)** permettant de transformer un caractère représentant une minuscule dans le caractère représentant la majuscule correspondante peut s'écrire :

```
// conversion de minuscules en majuscules
int toUpper(char c) {
  if (c >= 'a' && c <= 'z')
    return c + ('A' - 'a');
  else
    return c;
}
```

En pratique, l'utilisation de la table ASCII pour représenter des caractères souffre d'une limitation majeure. Avec *7* ou *8* bits il n'est pas possible de représenter exactement tous les caractères écrits de toutes les langues. Une table des caractères sur *7* bits est suffisante pour les langues qui utilisent peu de caractères accentués comme l'anglais. Pour le français et de nombreuses langues d'Europe occidentale, la table sur *8* bits est suffisante et la norme **ISO-8859** contient des tables de caractères *8* bits pour de nombreuses langues. La norme Unicode va plus loin en permettant de représenter les caractères écrits de toutes les langues connues sur Terre. Une description détaillée du support de ces types de caractères sort du cadre de ce cours sur les systèmes informatiques. Il est cependant important que vous soyez conscient de cette problématique pour pouvoir la prendre en compte lorsque vous développerez des applications qui doivent traiter du texte dans différentes langues. À titre d'exemple, la fonction **toupper(3)** qui est implémentée dans les versions actuelles de Linux est nettement plus complexe que celle que nous avons vue ci-dessus. Tout d'abord, la fonction **toupper(3)** prend comme argument un `int` et non un `char`. Cela lui permet d'accepter des caractères dans n'importe quel encodage. Ensuite, le traitement qu'elle effectue dépend du type d'encodage qui a été défini via **setlocale(3)** (voir **locale(7)**).

Dans la suite du cours, nous supposerons qu'un caractère est toujours représentable en utilisant le type `char` permettant de stocker un octet.

En C, les chaînes de caractères sont représentées sous la forme d'un tableau de caractères. Une chaîne de caractères peut s'initialiser de différentes façons reprises ci-dessous.

```
char name1[] = { 'U', 'n', 'i', 'x' };
char name2[] = { "Unix" };
char name3[] = "Unix";
```

Lorsque la taille de la chaîne de caractères n'est pas indiquée à l'initialisation (c'est-à-dire dans les deux dernières lignes ci-dessus), le compilateur C la calcule et alloue un tableau permettant de stocker la chaîne de caractères suivie du caractère `\0` qui par convention termine *toujours* les chaînes de caractères en C. En mémoire, la chaîne de caractères correspondant à `name3` occupe donc cinq octets. Les quatre premiers contiennent les caractères *U*, *n*, *i* et *x* et le cinquième le caractère `\0`. Il est important de bien se rappeler cette particularité du langage C car comme nous le verrons plus tard ce choix a de nombreuses conséquences.

Cette particularité permet d'implémenter facilement des fonctions de manipulation de chaînes de caractères. À titre d'exemple, la fonction ci-dessous calcule la longueur d'une chaîne de caractères.

```
int length(char str[])
{
  int i = 0;
  while (str[i] != 0) // '\0' et 0 sont égaux
    i++;
  return i;
}
```

Contrairement à des langages comme Java, C ne fait aucune vérification sur la façon dont un programme manipule un tableau. En C, il est tout à fait légal d'écrire le programme suivant :

```
char name[5] = "Unix";
printf("%c", name[6]);
printf("%c", name[12345]);
printf("%c", name[-1]);
```

En Java, tous les accès au tableau `name` en dehors de la zone mémoire réservée provoqueraient une `ArrayIndexOutOfBoundsException`. En C, il n'y a pas de mécanisme d'exception et le langage présuppose que lorsqu'un programmeur écrit `name[i]`, il a la garantie que la valeur `i` sera telle qu'il accédera bien à un élément valide du tableau `name`. Ce choix de conception du C permet d'obtenir du code plus efficace qu'avec Java puisque l'interpréteur Java doit vérifier tous les accès à chaque tableau lorsqu'ils sont exécutés. Malheureusement, ce choix de conception du C est aussi à l'origine d'un très grand nombre de problèmes qui affectent la sécurité de nombreux logiciels. Ces problèmes sont connus sous la dénomination **buffer overflow**. Nous aurons l'occasion d'y revenir plus tard.

Les pointeurs

Une différence majeure entre le C et la plupart des langages de programmation actuels est que le C est proche de la machine (langage de bas niveau) et permet au programmeur d'interagir directement avec la mémoire où les données qu'un programme manipule sont stockées. En Java, un programme peut créer autant d'objets qu'il souhaite (ou presque **[3]**). Ceux-ci sont stockés en mémoire et le **garbage collector** retire de la mémoire les objets qui ne sont plus utilisés. En C, un programme peut aussi réserver des zones pour stocker de l'information en mémoire. Cependant, comme nous le verrons plus tard, c'est le programmeur qui doit explicitement allouer et désallouer la mémoire.

Les *pointeurs* sont une des caractéristiques principales du langage C par rapport à de nombreux autres langages. Un **pointeur** est défini comme étant une variable contenant l'adresse d'une autre variable. Pour bien comprendre le fonctionnement des pointeurs, il est important d'avoir en tête la façon dont la mémoire est organisée sur un ordinateur. D'un point de vue abstrait, la mémoire d'un ordinateur peut être vue sous la forme d'une zone de stockage dans laquelle il est possible de lire ou d'écrire de l'information. Chaque zone permettant de stocker de l'information est identifiée par une **adresse**. La mémoire peut être vue comme l'implémentation de deux fonctions C :

> `data read(addr)` est une fonction qui, sur base d'une adresse, retourne la valeur stockée à cette adresse.
> `void write(addr, data)` est une fonction qui écrit la donnée `data` à l'adresse `addr` en mémoire.

Ces adresses sont stockées sur un nombre fixe de bits qui dépend en général de l'architecture du microprocesseur. Les valeurs les plus courantes aujourd'hui sont *32* et *64*. Par convention, les adresses sont représentées sous la forme d'entiers non signés. Sur la plupart des architectures de processeurs, une adresse correspond à une zone mémoire permettant de stocker un octet. Lorsque nous utiliserons une représentation graphique de la mémoire, nous placerons toujours les adresses numériquement basses en bas de la figure et elles croîtront vers le haut.

Considérons l'initialisation ci-dessous et supposons qu'elle est stockée dans une mémoire où les adresses sont encodées sur *3* bits. Une telle mémoire dispose de huit slots permettant chacun de stocker un octet.

```
char name[] = "Unix";
char c = 'Z';
```

Après exécution de cette initialisation et en supposant que rien d'autre n'est stocké dans cette mémoire, celle-ci contiendra les informations reprises dans la table ci-dessous.

Adresse	Contenu
111	0
110	0
101	Z
100	0
011	x
010	i
001	n
000	U

En langage C, l'expression `&var` permet de récupérer l'adresse à laquelle une variable a été stockée. Appliquée à l'exemple ci-dessus, l'expression `&(name[0])` retournerait la valeur `0b000` tandis que `&c` retournerait la valeur `0b101`.

L'expression `&` peut s'utiliser avec n'importe quel type de donnée. Les adresses de données en mémoire sont rarement affichées, mais quand c'est le cas, on utilise la notation hexadécimale comme dans l'exemple ci-dessous.

```
int i = 1252;
char str[] = "sinf1252";
char c = 'c';

printf("i vaut %d, occupe %ld bytes et est stocké à l'adresse : %p\n",
       i, sizeof(i), &i);
printf("c vaut %c, occupe %ld bytes et est stocké à l'adresse : %p\n",
       c, sizeof(c), &c);
printf("str contient \"%s\" et est stocké à partir de l'adresse : %p\n",
       str, &str);
```

L'exécution de ce fragment de programme produit la sortie suivante.

```
i vaut 1252, occupe 4 bytes et est stocké à l'adresse : 0x7fff89f99cbc
c vaut c, occupe 1 bytes et est stocké à l'adresse : 0x7fff89f99caf
str contient "sinf1252" et est stocké à partir de l'adresse : 0x7fff89f99cb0
```

L'intérêt des pointeurs en C réside dans la possibilité de les utiliser pour accéder et manipuler des données se trouvant en mémoire de façon efficace. En C, chaque pointeur a un type et le type du pointeur indique le type de la donnée qui est stockée dans une zone mémoire particulière. Le type est associé au pointeur lors de la déclaration de celui-ci.

```
int i = 1;      // entier
int *ptr_i;     // pointeur vers un entier
char c = 'Z';   // caractère
char *ptr_c;    // pointeur vers un char
```

Grâce aux pointeurs, il est possible non seulement d'accéder à l'adresse où une donnée est stockée, mais aussi d'accéder à la valeur qui est stockée dans la zone mémoire pointée par le pointeur en utilisant l'expression `*ptr`. Il est également possible d'effectuer des calculs sur les pointeurs comme représenté dans l'exemple ci-dessous.

```
int i = 1;           // entier
int *ptr_i;          // pointeur vers un entier
char str[] = "Unix";
char *s;             // pointeur vers un char

ptr_i = &i;
printf("valeur de i : %d, valeur pointée par ptr_i : %d\n", i, *ptr_i);
*ptr_i = *ptr_i + 1252;
printf("valeur de i : %d, valeur pointée par ptr_i : %d\n", i, *ptr_i);
s = str;
for (i = 0; i < strlen(str); i++){
  printf("valeur de str[%d] : %c, valeur pointée par *(s+%d) : %c\n",
         i, str[i], i, *(s+i));
}
```

L'exécution de ce fragment de programme produit la sortie suivante.

```
valeur de i : 1, valeur pointée par ptr_i : 1
valeur de i : 1253, valeur pointée par ptr_i : 1253
valeur de str[0] : U, valeur pointée par *(s+0) : U
valeur de str[1] : n, valeur pointée par *(s+1) : n
valeur de str[2] : i, valeur pointée par *(s+2) : i
valeur de str[3] : x, valeur pointée par *(s+3) : x
```

En pratique en C, les notations `char*` et `char[]` sont équivalentes et l'une peut s'utiliser à la place de l'autre. En utilisant les pointeurs, la fonction de calcul de la longueur d'une chaîne de caractères peut se réécrire comme suit.

```
int length(char *s)
{
  int i = 0;
  while (*(s+i) != '\0')
    i++;
  return i;
}
```

Les pointeurs sont fréquemment utilisés dans les programmes écrits en langage C et il est important de bien comprendre leur fonctionnement. Un point important à bien comprendre est ce que l'on appelle l'"*arithmétique des pointeurs*, c'est-à-dire la façon dont les opérations sur les pointeurs sont exécutées en langage C. Pour cela, il est intéressant de considérer la manipulation d'un tableau d'entiers à travers des pointeurs.

```
#define SIZE 3
unsigned int tab[3];
tab[0] = 0x01020304;
tab[1] = 0x05060708;
tab[2] = 0x090A0B0C;
```

En mémoire, ce tableau est stocké en utilisant trois mots consécutifs de 32 bits comme le montre l'exécution du programme ci-dessous :

```
int i;
for (i = 0; i < SIZE; i++) {
  printf("%X est à l'adresse %p\n", tab[i], &(tab[i]));
}
```

```
1020304 est à l'adresse 0x7fff5fbff750
5060708 est à l'adresse 0x7fff5fbff754
90A0B0C est à l'adresse 0x7fff5fbff758
```

La même sortie est produite avec le fragment de programme suivant qui utilise un pointeur.

```
unsigned int* ptr = tab;
for (i = 0; i < SIZE; i++) {
  printf("%X est à l'adresse %p\n", *ptr, ptr);
  ptr++;
}
```

Ce fragment de programme est l'occasion de réfléchir sur la façon dont le C évalue les expressions qui contiennent des pointeurs. La première est l'assignation `ptr=tab`. Lorsque `tab` est déclaré par la ligne `unsigned int tab[3]`, le compilateur considère que `tab` est une constante qui contiendra toujours l'adresse du premier élément du tableau. Il faut noter que puisque `tab` est considéré comme une constante, il est interdit d'en modifier la valeur en utilisant une assignation comme `tab=tab+1`. Le pointeur `ptr`, par contre, correspond à une zone mémoire qui contient une adresse. Il est tout à fait possible d'en modifier la valeur. Ainsi, l'assignation `ptr=tab` (ou `ptr=&(tab[0])`) place dans `ptr` l'adresse du premier élément du tableau. Les pointeurs peuvent aussi être modifiés en utilisant des expressions arithmétiques.

```
ptr = ptr + 1;  // ligne 1
ptr++;          // ligne 2
ptr = ptr - 2;  // ligne 3
```

Après l'exécution de la première ligne, `ptr` va contenir l'adresse de l'élément `1` du tableau `tab` (c'est-à-dire `&(tab[1])`). Ce résultat peut surprendre car si l'élément `tab[0]` se trouve à l'adresse `0x7fff5fbff750` c'est cette adresse qui est stockée dans la zone mémoire correspondant au pointeur `ptr`. On pourrait donc s'attendre à ce que l'expression `ptr+1` retourne plutôt la valeur `0x7fff5fbff751`. Il n'en est rien. En C, lorsque l'on utilise des calculs qui font intervenir des pointeurs, le compilateur prend en compte le type du pointeur qui est utilisé. Comme `ptr` est de type `unsigned int*`, il pointe toujours vers une zone mémoire permettant de

stocker un entier non signé sur 32 bits. L'expression `ptr+1` revient en fait à calculer la valeur `ptr+sizeof(unsigned int)` et donc `ptr+1` correspondra à l'adresse `0x7fff5fbff754`. Pour la même raison, l'exécution de la deuxième ligne placera l'adresse `0x7fff5fbff758` dans `ptr`. Enfin, la dernière ligne calculera `0x7fff5fbff758-2*sizeof(unsigned int)`, ce qui correspond à `0x7fff5fbff750`.

Il est intéressant pour terminer cette première discussion de l'arithmétique des pointeurs, de considérer l'exécution du fragment de code ci-dessous.

```
unsigned char* ptr_char = (unsigned char *) tab;
printf("ptr_char contient %p\n", ptr_char);
for (i = 0; i < SIZE + 1; i++) {
  printf("%X est à l'adresse %p\n", *ptr_char, ptr_char);
  ptr_char++;
}
```

L'exécution de ce fragment de code produit une sortie qu'il est intéressant d'analyser.

```
ptr_char contient 0x7fff5fbff750
4 est à l'adresse 0x7fff5fbff750
3 est à l'adresse 0x7fff5fbff751
2 est à l'adresse 0x7fff5fbff752
1 est à l'adresse 0x7fff5fbff753
```

Tout d'abord, l'initialisation du pointeur `ptr_char` a bien stocké dans ce pointeur l'adresse en mémoire du premier élément du tableau. Ensuite, comme `ptr_char` est un pointeur de type `unsigned char *`, l'expression `*ptr_char` a retourné la valeur de l'octet se trouvant à l'adresse `0x7fff5fbff750`. L'incrémentation du pointeur `ptr_char` s'est faite en respectant l'arithmétique des pointeurs. Comme `sizeof(unsigned char)` retourne `1`, la valeur stockée dans `ptr_char` a été incrémentée d'une seule unité par l'instruction `ptr_char++`. En analysant les quatre `unsigned char` se trouvant aux adresses `0x7fff5fbff750` à `0x7fff5fbff753`, on retrouve bien l'entier `0x01020304` qui avait été placé dans `tab[0]`.

Les structures

Outre les types de données décrits ci-dessus, les programmes informatiques doivent souvent pouvoir manipuler des données plus complexes. À titre d'exemples, un programme de calcul doit pouvoir traiter des nombres complexes, un programme de gestion des étudiants doit traiter des fiches d'étudiants avec nom, prénom, numéro de matricule,… Dans les langages orientés objet comme Java, cela se fait en encapsulant des données de différents types avec les méthodes permettant leur traitement. C n'étant pas un langage orienté objet, il ne permet pas la création d'objets et de méthodes directement associées. Par contre, C permet de construire des types de données potentiellement complexes.

C permet la définition de structures qui combinent différents types de données simples ou structurés. Contrairement aux langages orientés objet, il n'y a pas de méthode directement associée aux structures qui sont définies. Une structure est uniquement un type de données. Voici quelques exemples de structures simples en C.

```
// structure pour stocker une coordonnée 3D
struct coord {
  int x;
  int y;
  int z;
};

struct coord point = {1, 2, 3};
struct coord p;

// structure pour stocker une fraction
struct fraction {
  int numerator;
  int denominator;
};

struct fraction demi = {1, 2};
struct fraction f;

// structure pour représenter un étudiant
struct student {
  int matricule;
  char prenom[20];
  char nom[30];
};

struct student s = {1, "Linus", "Torvalds"};
```

Le premier bloc définit une structure dénommée `coord` qui contient trois entiers baptisés `x`, `y` et `z`. Dans une structure, chaque élément est identifié par son nom et il est possible d'y accéder directement. La variable `point` est de type `struct coord` et son élément `x` est initialisé à la valeur `1` tandis que son élément `z` est initialisé à la valeur `3`. La variable `p` est également de type `struct coord` mais elle n'est pas explicitement initialisée lors de sa déclaration.

La structure `struct fract` définit une fraction qui est composée de deux entiers qui sont respectivement le numérateur et le dénominateur. La structure `struct student`, elle, définit un type de données qui comprend un numéro de matricule et deux chaînes de caractères.

Les structures permettent de facilement regrouper des données qui sont logiquement reliées entre elles et doivent être manipulées en même temps. C permet d'accéder facilement à un élément d'une structure en utilisant l'opérateur ".". Ainsi, la structure `point` dont nous avons parlé ci-dessus aurait pu être initialisée par les trois expressions ci-dessous :

```
point.x = 1;
point.y = 2;
point.z = 3;
```

Dans les premières versions du langage C, une structure devait nécessairement contenir uniquement des données qui ont une taille fixe, c'est-à-dire des nombres, des caractères, des pointeurs ou des tableaux de taille fixe. Il n'était pas possible de stocker des tableaux de taille variable comme une chaîne de caractères `char []`. Les compilateurs récents **[C99]** permettent de supporter des tableaux flexibles à l'intérieur de structures. Nous ne les utiliserons cependant pas dans le cadre de ce cours.

Les structures sont utilisées dans différentes librairies et appels système sous Unix et Linux. Un exemple classique est la gestion du temps sur un système Unix. Un système informatique contient généralement une horloge dite *temps-réel* qui est en pratique construite autour d'un cristal qui oscille à une fréquence fixée. Ce cristal est piloté par un circuit électronique qui compte ses oscillations, ce qui permet de mesurer le passage du temps. Le système d'exploitation utilise cette horloge *temps réel* pour diverses fonctions et notamment la mesure du temps du niveau des applications.

Un système de type Unix maintient différentes structures qui sont associées à la mesure du temps **[4]**. La première sert à mesurer le nombre de secondes et de microsecondes qui se sont écoulées depuis le 1er janvier 1970. Cette structure, baptisée `struct timeval` est définie dans **sys/time.h** comme suit :

```
struct timeval {
  time_t       tv_sec;   /* seconds since Jan. 1, 1970 */
  suseconds_t  tv_usec;  /* and microseconds */
};
```

Cette structure est utilisée par des appels système tels que **gettimeofday(2)** pour notamment récupérer l'heure courante ou les appels de manipulation de timers tels que **getitimer(2)** / **setitimer(2)**. Elle est aussi utilisée par la fonction **time(3posix)** de la librairie standard et est très utile pour mesurer les performances d'un programme.

Les structures sont également fréquemment utilisées pour représenter des formats de données spéciaux sur disque comme le format des répertoires **[5]** ou les formats de paquets qui sont échangés sur le réseau **[6]**.

La définition de `struct timeval` utilise une fonctionnalité fréquemment utilisée du C : la possibilité de définir des alias pour des noms de type de données existants. Cela se fait en utilisant l'opérateur `typedef`. En C, il est possible de renommer des types de données existants. Ainsi, l'exemple ci-dessous utilise `typedef` pour définir l'alias `Entier` pour le type `int` et l'alias `Fraction` pour la structure `struct fraction`.

```
// structure pour stocker une fraction
typedef struct fraction {
  double numerator;
  double denominator;
} Fraction ;

typedef int Entier;

int main(int argc, char *argv[])
{
  Fraction demi = {1, 2};
  Entier i = 2;
  // ...
  return EXIT_SUCCESS;
}
```

Les types `Entier` et `int` peuvent être utilisés de façon interchangeable à l'intérieur du programme une fois qu'ils ont été définis.

> **Note**
>
> `typedef` en pratique
>
> Le renommage de types de données a des avantages et des inconvénients dont il faut être conscient pour pouvoir l'utiliser à bon escient. L'utilisation de `typedef` peut faciliter la lecture et la portabilité de certains programmes. Lorsqu'un `typedef` est associé à une structure, cela facilite la déclaration de variables de ce type et permet le cas échéant de modifier la structure de données ultérieurement sans pour autant devoir modifier

l'ensemble du programme. Cependant, contrairement aux langages orientés objet, des méthodes ne sont pas directement associées aux structures et la modification d'une structure oblige souvent à vérifier toutes les fonctions qui utilisent cette structure. L'utilisation de `typedef` permet de clarifier le rôle de certains types de données ou valeurs de retour de fonctions. À titre d'exemple, l'appel système **read(2)** qui permet notamment de lire des données dans un fichier retourne le nombre d'octets qui ont été lus après chaque appel. Cette valeur de retour est de type `ssize_t`. L'utilisation de ces types permet au compilateur de vérifier que les bons types de données sont utilisés lors des appels de fonctions.

`typedef` est souvent utilisé pour avoir des identifiants de type de données plus court. Par exemple, il est très courant d'abrévier les types `unsigned` comme ci-dessous.

```
typedef unsigned int u_int_t;
typedef unsigned long u_long_t;
```

Soyez prudent si vous utilisez des `typedef` pour redéfinir des pointeurs. En C, il est tout à fait valide d'écrire les lignes suivantes.

```
typedef int * int_ptr;
typedef char * string;
```

Malheureusement, il y a un risque dans un grand programme que le développeur oublie que ces types de données correspondent à des pointeurs qui doivent être manipulés avec soin. Le **Linux kernel coding style** contient une discussion intéressante sur l'utilisation des `typedef`.

Les pointeurs sont fréquemment utilisés lors de la manipulation de structures. Lorsqu'un pointeur pointe vers une structure, il est utile de pouvoir accéder facilement aux éléments de la structure. Le langage C supporte deux notations pour représenter ces accès aux éléments d'une structure. La première notation est `(*ptr).elem` où `ptr` est un pointeur et `elem` l'identifiant d'un des éléments de la structure pointée par `ptr`. Cette notation est en pratique assez peu utilisée. La notation la plus fréquente est `ptr->elem` dans laquelle `ptr` et `->elem` sont respectivement un pointeur et un identifiant d'élément. L'exemple ci-dessous illustre l'initialisation de deux fractions en utilisant ces notations.

```
struct fraction demi, quart;
struct fraction *demi_ptr;
struct fraction *quart_ptr;

demi_ptr = &demi;
quart_ptr = &quart;

(*demi_ptr).num = 1;
(*demi_ptr).den = 2;

quart_ptr->num = 1;
quart_ptr->den = 4;
```

Les pointeurs sont fréquemment utilisés en combinaison avec des structures et on retrouve très souvent la seconde notation dans des programmes écrits en C.

Les fonctions

Comme la plupart des langages, le C permet de modulariser un programme en le découpant en de nombreuses fonctions. Chacune réalise une tâche simple. Tout comme Java, C permet la définition de fonctions qui ne retournent aucun résultat. Celles-ci sont de type `void` comme l'exemple trivial ci-dessous.

```
void usage()
{
  printf("Usage : ...\n");
}
```

La plupart des fonctions utiles retournent un résultat qui peut être une donnée d'un des types standard ou une structure. Cette utilisation est similaire à ce que l'on trouve dans des langages comme Java. Il faut cependant être attentif à la façon dont le langage C traite les arguments des fonctions. Le langage C utilise le *passage par valeur* des arguments. Lorsqu'une fonction est exécutée, elle reçoit les valeurs de ces arguments. Ces valeurs sont stockées dans une zone mémoire qui est locale à la fonction. Toute modification faite sur la valeur d'une variable à l'intérieur d'une fonction est donc locale à cette fonction. Les deux fonctions ci-dessous ont le même résultat et aucune des deux n'a d'effet de bord.

```
int twotimes(int n)
{
  return 2 * n;
}

int two_times(int n)
{
  n = 2 * n;
  return n;
```

```
}
```

Il faut être nettement plus attentif lorsque l'on écrit des fonctions qui utilisent des pointeurs comme arguments. Lorsqu'une fonction a un argument de type pointeur, celui-ci est passé par valeur, mais connaissant la valeur du pointeur, il est possible à la fonction de modifier le contenu de la zone mémoire pointée par le pointeur. Ceci est illustré par l'exemple ci-dessous.

```
int times_two(int *n)
{
  return (*n) + (*n);
}

int timestwo(int *n)
{
  *n = (*n) + (*n);
  return *n;
}

void f()
{
  int i = 1252;
  printf("i:%d\n", i);
  printf("times_two(&i)=%d\n", times_two(&i));
  printf("après times_two, i:%d\n", i);
  printf("timestwo(&i)=%d\n", timestwo(&i));
  printf("après timestwo, i:%d\n", i);
}
```

Lors de l'exécution de la fonction `f`, le programme ci-dessus affiche à la console la sortie suivante :

```
i:1252
times_two(&i)=2504
après times_two, i:1252
timestwo(&i)=2504
après timestwo, i:2504
```

Cet exemple illustre aussi une contrainte imposée par le langage C sur l'ordre de définition des fonctions. Pour que les fonctions `times_two` et `timestwo` puissent être utilisées à l'intérieur de la fonction `f`, il faut qu'elles aient été préalablement définies. Dans l'exemple ci-dessus, cela s'est fait en plaçant la définition des deux fonctions avant leur utilisation. C'est une règle de bonne pratique utilisable pour de petits programmes composés de quelques fonctions. Pour des programmes plus larges, il est préférable de placer au début du code source la signature des fonctions qui y sont définies. La signature d'une fonction comprend le type de valeur de retour de la fonction, son nom et les types de ses arguments. Généralement, ces déclarations sont regroupées à l'intérieur d'un **fichier header** dont le nom se termine par `.h`.

```
int times_two(int *);
int timestwo(int *);
```

Les fonctions peuvent évidemment recevoir également des tableaux comme arguments. Cela permet par exemple d'implémenter une fonction qui calcule la longueur d'une chaîne de caractères en itérant dessus jusqu'à trouver le caractère de fin de chaîne.

```
int length(char *s)
{
  int i = 0;
  while (*(s+i) != '\0')
    i++;
  return i;
}
```

Tout comme cette fonction peut accéder au ième caractère de la chaîne passée en argument, elle peut également et sans aucune restriction modifier chacun des caractères de cette chaîne. Par contre, comme le pointeur vers la chaîne de caractères est passé par valeur, la fonction ne peut pas modifier la zone mémoire qui est pointée par l'argument.

Un autre exemple de fonctions qui manipulent les tableaux sont des fonctions mathématiques qui traitent des vecteurs par exemple.

```
void plusun(int size, int *v)
{
  int i;
  for (i = 0; i < size; i++)
    v[i]++;
```

```
}

void print_vecteur(int size, int*v) {
  int i;
  printf("v={");
  for (i = 0;i < size - 1; i++)
    printf("%d,", v[i]);

  if (size > 0)
    printf("%d}", v[size - 1]);
  else
    printf("}");
}
```

Ces deux fonctions peuvent être utilisées par le fragment de code ci-dessous :

```
int vecteur[N] = {1, 2, 3, 4, 5};
plusun(N, vecteur);
print_vecteur(N, vecteur);
```

Note

Attention à la permissivité du compilateur C

Certains langages comme Java sont fortement typés et le compilateur contient de nombreuses vérifications, notamment sur les types de données utilisés, qui permettent d'éviter un grand nombre d'erreurs. Le langage C est lui nettement plus libéral. Les premiers compilateurs C étaient très permissifs notamment sur les types de données passés en arguments. Ainsi, un ancien compilateur C accepterait probablement sans broncher les appels suivants :

```
plusun(vecteur,N);
print_vecteur(N,vecteur);
```

Dans ce fragment de programme, l'appel à `print_vecteur` est tout à fait valide. Par contre, l'appel à `plusun` est lui erroné puisque le premier argument est un tableau d'entiers (ou plus précisément un pointeur vers le premier élément d'un tableau d'entiers) alors que la fonction `plusun` attend un entier. Inversement, le second argument est un entier à la place d'un tableau d'entiers. Cette erreur n'empêche pas le compilateur **gcc(1)** de compiler le programme correspondant. Il émet cependant le **warning** suivant :

```
warning: passing argument 1 of ‘plusun’ makes integer from pointer without a cast
warning: passing argument 2 of ‘plusun’ makes pointer from integer without a cast
```

De nombreux programmeurs débutants ignorent souvent les warnings émis par le compilateur et se contentent d'avoir un programme compilable. C'est la source de nombreuses erreurs et de nombreux problèmes. Dans l'exemple ci-dessus, l'exécution de l'appel `plusun(vecteur,N)` provoquera une tentative d'accès à la mémoire dans une zone qui n'est pas allouée au processus. Dans ce cas, la tentative d'accès est bloquée par le système et provoque l'arrêt immédiat du programme sur une **segmentation fault**. Dans d'autres cas, des erreurs plus subtiles mais du même type ont provoqué des problèmes graves de sécurité dans des programmes écrits en langage C. Nous y reviendrons ultérieurement.

Pour terminer, mentionnons que les fonctions écrites en C peuvent utiliser des structures et des pointeurs vers des structures comme arguments. Elles peuvent aussi retourner des structures comme résultat. Ceci est illustré par deux variantes de fonctions permettant d'initialiser une fraction et de déterminer si deux fractions sont égales **[7]**.

```
struct fraction init(int num, int den)
{
  struct fraction f;
  f.numerator = num;
  f.denominator = den;
  return f;
}

int equal(struct fraction f1, struct fraction f2)
{
  return ((f1.numerator == f2.numerator)
          && (f1.denominator == f2.denominator));
}

int equalptr(struct fraction *f1, struct fraction *f2)
{
  return ((f1->numerator==f2->numerator)
          && (f1->denominator==f2->denominator));
}

void initptr(struct fraction *f, int num, int den)
```

```
{
  f->numerator = num;
  f->denominator = den;
}
```

Considérons d'abord les fonctions `init` et `equal`. `init` est une fonction qui construit une structure sur base d'arguments entiers et retourne la valeur construite. `equal` quant à elle reçoit comme arguments les valeurs de deux structures. Elle peut accéder à tous les éléments des structures passées en argument. Comme ces structures sont passées par valeur, toute modification aux éléments de la structure est locale à la fonction `equal` et n'est pas répercutée sur le code qui a appelé la fonction.

Les fonctions `initptr` et `equalptr` utilisent toutes les deux des pointeurs vers des `struct fraction` comme arguments. Ce faisant, elles ne peuvent modifier la valeur de ces pointeurs puisqu'ils sont passés comme valeurs. Par contre, les deux fonctions peuvent bien entendu modifier les éléments de la structure qui se trouvent dans la zone de mémoire pointée par le pointeur. C'est ce que `initptr` fait pour initialiser la structure. `equalptr` par contre se contente d'accéder aux éléments des structures passées en argument sans les modifier. Le fragment de code ci-dessous illustre comment ces fonctions peuvent être utilisées en pratique.

```
struct fraction quart;
struct fraction tiers;
quart = init(1, 4);
initptr(&tiers, 1, 3);
printf("equal(tiers,quart)=%d\n", equal(tiers, quart));
printf("equalptr(&tiers,&quart)=%d\n", equalptr(&tiers, &quart));
```

Les expressions de manipulation de bits

La plupart des langages de programmation sont spécialisés dans la manipulation des types de données classiques comme les entiers, les réels et les chaînes de caractères. Comme nous l'avons vu, le langage C permet de traiter ces types de données. En outre, il permet au programmeur de pouvoir facilement manipuler les bits qui se trouvent en mémoire. Pour cela, le langage C définit des expressions qui correspondent à la plupart des opérations de manipulation de bits que l'on retrouve dans les langages d'assemblage. Les premières opérations sont les opérations logiques.

La première opération logique est la négation **négation** (**NOT** en anglais). Elle prend comme argument un bit et retourne le bit inverse. Comme toutes les opérations logiques, elle peut se définir simplement sous la forme d'une table de vérité. Dans des formules mathématiques, la négation est souvent représentée sous la forme $\neg A$.

A	NOT(A)
0	1
1	0

La deuxième opération est la **conjonction logique** (**AND** en anglais). Cette opération prend deux arguments binaires et retourne un résultat binaire. Dans des formules mathématiques, la conjonction logique est souvent représentée sous la forme $A \wedge B$. Elle se définit par la table de vérité suivante :

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

La troisième opération est la **disjonction logique** (**OR** en anglais). Cette opération prend deux arguments binaires. Dans des formules mathématiques, la disjonction logique est souvent représentée sous la forme $A \vee B$. Elle se définit par la table de vérité suivante.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Enfin, une dernière opération logique intéressante est le **ou exclusif** (**XOR** en anglais). Celle-ci se définit par la table de vérité ci-dessous. Cette opération est parfois représentée mathématiquement comme $A \oplus B$.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Ces opérations peuvent être combinées entre elles. Pour des raisons technologiques, les circuits logiques implémentent plutôt les opérations NAND (qui équivaut à AND suivi de NOT) ou NOR (qui équivaut à OR suivi de NOT). Il est également important de mentionner les lois formulées par De Morgan qui peuvent se résumer par les équations suivantes :

$$\neg(A \wedge B) = \neg A \vee \neg B$$
$$\neg(A \vee B) = \neg A \wedge \neg B$$

Ces opérations binaires peuvent s'étendre à des séquences de bits. Voici quelques exemples qui permettent d'illustrer ces opérations sur des octets.

```
~ 00000000 = 11111111
11111010 & 01011111 = 01011010
11111010 | 01011111 = 11111111
11111010 ^ 01011111 = 10100101
```

En C, ces expressions logiques s'utilisent comme dans le fragment de code suivant. En général, elles s'utilisent sur des representations non signées, souvent des `unsigned char` ou des `unsigned int`.

```
r = ~a;    // négation bit à bit
r = a & b; // conjonction bit à bit
r = a | b; // disjonction bit à bit
r = a ^ b; // xor bit à bit
```

En pratique, les opérations logiques sont utiles pour effectuer des manipulations au niveau des bits de données stockées en mémoire. Une utilisation fréquente dans certaines applications réseaux ou systèmes est de forcer certains bits à prendre la valeur `0` ou `1`. La conjonction logique permet de forcer facilement un bit à zéro tandis que la disjonction logique permet de forcer facilement un bit à un. L'exemple ci-dessous montre comment forcer les valeurs de certains bits dans un `unsigned char`. Il peut évidemment se généraliser à des séquences de bits plus longues.

```
r = c & 0x7E; // 0b01111110 force les bits de poids faible et fort à 0
r = d | 0x18; // 0b00011000 force les bits 4 et 3 à 1
```

L'opération XOR joue un rôle important dans certaines applications. La plupart des méthodes de chiffrement et de déchiffrement utilisent de façon extensive cette opération. Une des propriétés intéressantes de l'opération XOR est que $(A \oplus B) \oplus B = A$. Cette propriété est largement utilisée par les méthodes de chiffrement. La méthode développée par Vernam au début du vingtième siècle s'appuie sur l'opération XOR. Pour transmettre un message M de façon sûre, elle applique l'opération XOR bit à bit entre tous les bits du message M et une clé K doit avoir au moins le même nombre de bits que M. Si cette clé K est totalement aléatoire et n'est utilisée qu'une seule fois, alors on parle de *one-time-pad*. On peut montrer que dans ce cas, la méthode de chiffrement est totalement sûre. En pratique, il est malheureusement difficile d'avoir une clé totalement aléatoire qui soit aussi longue que le message à transmettre. Le programme ci-dessous implémente cette méthode de façon triviale. La fonction **memfrob(3)** de la librairie **GNU** utilise également un chiffrement via un XOR.

```
int main(int argc, char* argv[])
{
  if (argc != 2)
    usage("ce programme prend une clé comme argument");

  char *key = argv[1];
  char c;
  int i = 0;
  while (((c = getchar()) != EOF) && (i < strlen(key))) {
    putchar(c ^ *(key + i));
    i++;
  }
  return EXIT_SUCCESS;
}
```

Note

Ne pas confondre expressions logiques et opérateurs binaires.

En C, les symboles utilisés pour les expressions logiques (`||` et `&&`) sont très proches de ceux utilisés pour représenter les opérateurs binaires (`|` et `&`). Il arrive parfois qu'un développeur confonde `&` avec `&&`. Malheureusement, le compilateur ne peut pas détecter une telle erreur car dans les deux cas le résultat attendu est généralement du même type.

```
0b0100 & 0b0101 = 0b0100
0b0100 && 0b0101 = 0b0001
0b0100 | 0b0101 = 0b0101
0b0100 || 0b0101 = 0b0001
```

Un autre point important à mentionner concernant les expressions logiques est qu'en C celles-ci sont évaluées de gauche à droite. Cela implique que dans l'expression `( expr1 && expr2 )`, le compilateur C va d'abord évaluer l'expression `expr1`. Si celle-ci est évaluée à la valeur `0`, la seconde expression ne sera pas évaluée. Cela peut être très utile lorsque l'on doit exécuter du code si un pointeur est non-`NULL` et qu'il pointe vers une valeur donnée. Dans ce cas, la condition sera du type `((ptr != NULL) && (ptr->den > 0))`.

Pour terminer, le langage C supporte des expressions permettant le décalage à gauche ou à droite à l'intérieur d'une suite de bits non signée.

- `a = n >> B` décale les bits représentants `n` de `B` bits vers la droite et place le résultat dans la variable `a`.
- `a = n << B` décale les bits représentants `n` de `B` bits vers la gauche et place le résultat dans la variable `a`.

Ces opérations de décalage permettent différentes manipulations de bits. À titre d'exemple, la fonction `int2bin` utilise à la fois des décalages et des masques pour calculer la représentation binaire d'un entier non signé et la placer dans une chaîne de caractères.

```c
#define BITS_INT 32
// str[BITS_INT]
void int2bin(unsigned int num, char *str)
{
  int i;
  str[BITS_INT] = '\0';
  for (i = BITS_INT - 1; i >= 0; i--) {
    if ((num & 1) == 1)
      str[i] = '1';
    else
      str[i] = '0';
    num = num >> 1;
  }
}
```

Footnotes

[1] En pratique, le format binaire contient $127 + exp$ en simple précision et non l'exposant exp. Ce choix facilite certaines comparaisons entre nombres représentés en virgule flottante. Une discussion détaillée de la représentation binaire des nombres en virgule flottante sort du cadre de ce cours dédié aux systèmes informatiques. Une bonne référence à ce sujet est **[Goldberg1991]**.

[2] Source : **http://en.wikipedia.org/wiki/Single-precision_floating-point_format**

[3] En pratique, l'espace mémoire accessible à un programme Java est limité par différents facteurs. Voir notamment le paramètre `-Xm` de la machine virtuelle Java **http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/java.html**

[4] Une description plus détaillée des différentes possibilités de mesure du temps via les fonctions de la librairie standard est disponible dans le chapitre **21** du **manuel** de la **libc**.

[5] Voir notamment **fs(5)** pour des exemples relatifs aux systèmes de fichiers. Une analyse détaillée des systèmes de fichiers sort du cadre de ce cours.

[6] Parmi les exemples simples, on peut citer la structure `struct ipv6hdr` qui correspond à l'entête IPv6 et est définie dans **linux/ipv6.h**.

[7] Cette définition de l'égalité entre fractions suppose que les fractions à comparer sont sous forme irréductible. Le lecteur est invité à écrire la fonction générale permettant de tester l'égalité entre fractions réductibles.