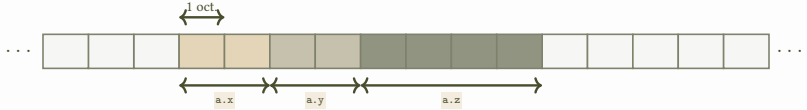
Accès manuel aux champs

Soit `a` une variable de type `A` initialisée par

```
A a = {1000, 2000, 3000};
```

Cette variable est organisée en mémoire en



On peut accéder aux champs de `a` de la manière suivante :

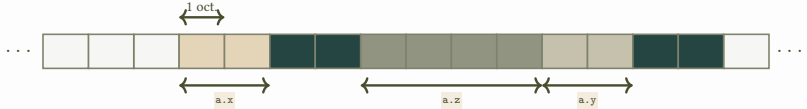
```
short x, y;
int z;
void *p;
p = &a;
x = *((short *) p); /* Equivalent a x = a.x; */
p += 2; /* p est de type void * : l'adresse p est incrementee de 2 */
y = *((short *) p); /* Equivalent a y = a.y; */
p += 2;
z = *((int *) p); /* Equivalent a z = a.z; */
```

Accès manuel aux champs

Soit `b` une variable de type `B` initialisée par

```
B b = {1000, 2000, 3000};
```

Cette variable est organisée en mémoire en



On peut accéder aux champs de `b` de la manière suivante :

```
short x, y;
int z;
void *p;
p = &b;
x = *((short *) p); /* Equivalent a x = b.x; */
p += 4;
z = *((int *) p); /* Equivalent a z = b.z; */
p += 4;
y = *((short *) p); /* Equivalent a y = b.y; */
```

L'option `Wpadded`

L'option du compilateur `-Wpadded` permet d'obtenir un avertissement sanctionnant la déclaration d'un type structuré nécessitant des octets de complétion.

– Exemple –

Avec le type structuré `B` défini par

```
typedef struct {
    short x;
    int z;
    short y;
} B;
```

on obtient l'avertissement

```
Prog.c:3:9: warning: padding struct to align ‘z’ [-Wpadded]
    int z;
        ^
Prog.c:5:1: warning: padding struct size to alignment boundary [-Wpadded]
 } B;
 ^
```

Alignement en mémoire — ce qu'il faut retenir

L'alignement mémoire d'une variable d'un type structuré dépend de beaucoup de paramètres, notamment :

- de l'architecture de la machine exécutant ou ayant compilé le programme ;
- du compilateur ayant compilé le programme.

Il est donc important de savoir que le calcul de la **taille** d'une variable d'un **type structuré** n'est pas immédiat et **dépend du contexte**.

Dans la pratique, on ne cherchera pas à déclarer des types structurés qui ne nécessitent pas d'octet de complétion. Au contraire : il est de loin préférable de déclarer les champs dans un ordre logique favorisant la relecture et la maintenance.

Axe 3 : utiliser quelques techniques avancées

9. Opérateurs

10. Pointeurs de fonction

11. Génération aléatoire

12. Mémoïsation

9. Opérateurs

Plan

9. Opérateurs

Caractéristiques d'un opérateur

Un **opérateur** dispose des caractéristiques structurelles suivantes :

1. son arité, qui désigne le nombre d'opérandes sur lesquelles il agit ;
2. sa précédence, qui permet de savoir, dans une expression, dans quel ordre appliquer les différents opérateurs qui la composent ;
3. pour les opérateurs binaires (d'arité 2), son sens d'associativité, qui permet de savoir, dans une expression, dans quel sens appliquer des mêmes opérateurs qui la composent.

Précédence et associativité des opérateurs

- Considérons l'expression `3 * 2 + 1`.

  Suivant les priorités relatives des opérateurs `*` et `+`, il y deux manières de l'évaluer :

  1. `(3 * 2) + 1`, si `*` est **plus prioritaire** que `+` ;
  2. `3 * (2 + 1)`, si `+` est **plus prioritaire** que `*`.

- Considérons l'expression `4 - 3 - 2 - 1`.

  Suivant le sens d'associativité de `-`, il y a deux manières de l'évaluer :

  1. `((4 - 3) - 2) - 1`, si `-` est **associatif de gauche à droite** ;
  2. `4 - (3 - (2 - 1))`, si `-` est **associatif de droite à gauche**.

Tout ceci peut être rendu explicite par l'**utilisation de parenthèses**.

9. Opérateurs

Opérateurs de gestion de la mémoire

Op.	Rôle	Ari.	Assoc.	Opérandes
`&`	référencement	1	–	une variable
`*`	déréférencement	1	–	un pointeur
`[]`	élément d'un tableau	2	$\longrightarrow$	un pointeur et un entier
`.`	valeur d'un champ	2	$\longrightarrow$	une var. d'un type struct. et un id. de champ
`->`	valeur d'un champ	2	$\longrightarrow$	une pointeur sur une var. d'un type struct. et un id. de champ

9. Opérateurs

Opérateurs arithmétiques

Op.	Rôle	Ari.	Assoc.	Opérandes
`+`, `-`, `*`, `/`	opérations arith.	2	$\longrightarrow$	deux val. numériques
`%`	modulo	2	$\longrightarrow$	deux entiers
`+`, `-`	signe	1	–	une val. numérique
`++`, `--`	incr. / décr.	1	–	une var. d'un type numérique ou un pointeur

L'opérateur modulo

L'opérateur **modulo** `%` calcule le reste de la division euclidienne de son premier opérande par son second.

D'un point de vue mathématique, si a et b sont deux entiers, on a $a = b \times q + r$, où $0 \leqslant r \leqslant b - 1$ et q est un entier. q est le quotient et r est le reste, **toujours positif**.

Cependant, `%` peut produire des valeurs négatives, dans le cas où l'un des deux opérandes est négatif.

Solution pour un modulo qui respecte la définition mathématique :

```
int vrai_modulo(int a, int b) {
    int r;
    assert(b != 0);
    r = a % b;
    if (r < 0)
        return r + b;
    return r;
}
```

Les opérateurs d'incrémentation et de décrémentation

Les opérateurs `++` et `--` existent en deux versions, suivant qu'ils soient préfixes ou suffixes :

1. `a++`, incrémente (de un) la valeur de la variable `a` et est une expression dont la valeur est l'**ancienne valeur** de `a` ;

2. `++a`, incrémente (de un) la valeur de la variable `a` et est une expression dont la valeur est la **nouvelle valeur** de `a`.

– Exemples –

```
int a = 5, b;
b = 3 + a++;
```

`b` vaut `8` et `a` vaut `6`.

```
int a = 5, b;
b = 3 + ++a;
```

`b` vaut `9` et `a` vaut `6`.

Les opérateurs d'incrémentation et de décrémentation

Attention au **pièges d'utilisation** de ces opérateurs.

– Exemples –

Les instructions

```
int a = 5, b;
b = a++ + ++a;
```

```
int a = 5;
a = a++;
```

```
int a = 5;
a = ++a;
```

ne sont pas évaluables (l'effet produit par les lignes 2 dépend du compilateur et de ses options).

Règle : pour éviter ce type de piège, on s'interdit de réaliser plus d'une modification d'une même variable dans une même expression.

Opérateurs relationnels

Op.	Rôle	Ari.	Assoc.	Opérandes
`<`, `>`	comparaison stricte	2	$\longrightarrow$	deux val. numériques
`<=`, `>=`	comparaison large	2	$\longrightarrow$	deux val. numériques
`==`	égalité	2	$\longrightarrow$	deux val. numériques
`!=`	différence	2	$\longrightarrow$	deux val. numériques

Toutes les expressions de la forme

```
v1 CMP v2
```

où `v1` et `v2` sont des valeurs numériques et `CMP` est un opérateur de comparaison produisent une valeur :

- `1` si la comparaison est vraie ;
- `0` sinon.

Opérateurs relationnels

Un pointeur étant une adresse, et donc une valeur numérique, il est possible de comparer deux pointeurs.

– Exemple –

```
char *ptr1, *ptr2;
char c;
c = 'a';
ptr1 = &c;
ptr2 = &c;
if (ptr1 == ptr2)
    printf("ok1\n");
if (&ptr1 == &ptr2)
    printf("ok2\n");
```

Ceci affiche seulement `ok1`.

En effet, les deux pointeurs `ptr1` et `ptr2` pointent vers le même emplacement en mémoire.

Le second test est faux car les adresses des variables `ptr1` et `ptr2` sont différentes.

```
int t1[2], t2[2];
t1[0] = 1;
t1[1] = 2;
t2[0] = 1;
t2[1] = 2;
if (t1 == t2)
    printf("ok\n");
```

Ceci compare les **adresses** de `t1` et `t2` et non pas les valeurs de leurs cases.

Rien n'est donc affiché car les tableaux `t1` et `t2` sont à des adresses différentes.

Opérateurs logiques

Op.	Rôle	Ari.	Assoc.	Opérandes
`&&`	et logique	2	$\longrightarrow$	deux val. numériques
`||`	ou logique	2	$\longrightarrow$	deux val. numériques
`!`	non logique	1	–	une val. numérique

Toutes les expressions formées d'opérateurs logiques produisent une valeur, `0` ou bien `1`.

Cette valeur est

- `1` si l'expression logique est vraie ;
- `0` sinon.

Opérateurs bit à bit

Op.	Rôle	Ari.	Assoc.	Opérandes
`&`	et bit à bit	2	$\longrightarrow$	deux val. entières
`|`	ou bit à bit	2	$\longrightarrow$	deux val. entières
`^`	xor bit à bit	2	$\longrightarrow$	deux val. entières
$\sim$	non bit à bit	1	–	une val. entière
`<<`, `>>`	déc. g./d. bit à bit	2	$\longrightarrow$	deux val. entières

Et / ou / xor /non bit à bit

– Exemples –

`x`	0	1	1	1	0	1	0	1
`y`	1	0	1	0	1	1	0	0
`x & y`	0	0	1	0	0	1	0	0

`x`	0	1	1	1	0	1	0	1
`y`	1	0	1	0	1	1	0	0
`x | y`	1	1	1	1	1	1	0	1

`x`	0	1	1	1	0	1	0	1
`y`	1	0	1	0	1	1	0	0
`x ^ y`	1	1	0	1	1	0	0	1

`x`	0	1	1	1	0	1	0	1
`~x`	1	0	0	0	1	0	1	0

Opérateurs bit à bit et taille des opérandes

Si les deux opérandes n'ont pas la même taille (en nombre de bits), le plus petit est complété à gauche par des

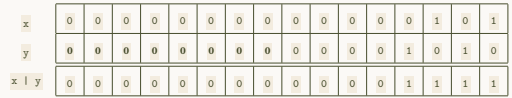
- **zéros** s'il est non signé ou bien positif ;
- **uns** s'il est négatif et signé.

Une conséquence du **codage en complément à deux** est que le signe d'un entier signé est lu sur son bit de poids fort :

- `0` s'il est positif ;
- `1` s'il est négatif.

– Exemples –

```
short x = 5;
char y = 10;
x = x | y;
```




```
short x = 5;
char y = -10;
x = x | y;
```

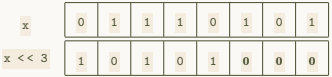
Décalages bit à bit de valeurs non signées

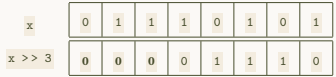
Soit `x` une variable de type entier (**`char`**, **`short`**, **`int`** ou **`long`**).

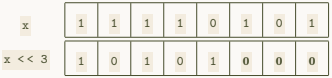
Si `x` est non signé (déclaré avec **`unsigned`**),

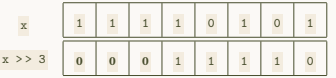
- le décalage à gauche insère des bits `0` ;
- le décalage à droite insère des bits `0`.

– Exemples –









Décalages bit à bit de valeurs signées

Soit `x` une variable de type entier (**`char`**, **`short`**, **`int`** ou **`long`**).

Si `x` est signé (déclaré sans **`unsigned`**),

- le décalage à gauche insère des bits `0` ;
- le décalage à droite insère des bits **égaux au bit de poids fort** de `x`.

– Exemples –

`x`	0	1	1	1	0	1	0	1
`x << 3`	1	0	1	0	1	**0**	**0**	**0**

`x`	0	1	1	1	0	1	0	1
`x >> 3`	**0**	**0**	**0**	0	1	1	1	0

`x`	1	1	1	1	0	1	0	1
`x << 3`	1	0	1	0	1	**0**	**0**	**0**

`x`	1	1	1	1	0	1	0	1
`x >> 3`	**1**	**1**	**1**	1	1	1	1	0

Exemple — ensembles finis

Les opérateurs bit à bit sont adaptés pour représenter des **ensembles finis** et réaliser des opérations ensemblistes de manière simple et efficace.

Soit E un ensemble à 32 éléments.

On considère que E est muni d'une relation d'ordre totale de sorte que ses éléments puissent être indexés de 0 à 31. Ainsi,

$$E = \{e_0, e_1, e_2, \ldots, e_{31}\}.$$

Cette indexation permet de représenter tout **sous-ensemble** S de E par un mot de 32 bit dont le i^e bit code la présence (`1`) ou l'absence (`0`) de e_i dans S.

– Exemple –

L'entier dont l'écriture binaire est

```
00010000 10000000 00000011 00000101
```

code le sous ensemble $\{e_0, e_2, e_8, e_9, e_{23}, e_{28}\}$ de E.

Exemple — ensembles finis

Ceci mène à la déclaration du type alias

```
typedef unsigned int SousEnsemble;
```

Pour tester si e_i **appartient** à S, il suffit de réaliser un et bit à bit entre l'entier représentant S et le mot binaire constitué d'un unique `1` en i^e position.

Cette expression vaut `0` si $e_i \notin S$ et une valeur non nulle sinon.

Ceci est implanté par la fonction

```
int appartient_e_i(SousEnsemble s, int i) {
    assert(0 <= i);
    assert(i <= 31);

    return (1 << i) & s;
}
```

Exemple — ensembles finis

Pour réaliser l'**union** de deux sous-ensembles S_1 et S_2 de E, il suffit de réaliser un ou bit à bit des deux entiers représentant S_1 et S_2. En effet, pour tout i, $e_i \in S_1 \cup S_2$ si $e_i \in S_1$ ou $e_i \in S_2$.

Ceci est implanté par la fonction

```
SousEnsemble union(SousEnsemble s_1, SousEnsemble s_2) {
    return s_1 | s_2;
}
```

Pour réaliser l'**intersection** de deux sous-ensembles S_1 et S_2 de E, il suffit de réaliser un et bit à bit des deux entiers représentant S_1 et S_2. En effet, pour tout i, $e_i \in S_1 \cap S_2$ si $e_i \in S_1$ et $e_i \in S_2$.

Ceci est implanté par la fonction

```
SousEnsemble intersection(SousEnsemble s_1, SousEnsemble s_2) {
    return s_1 & s_2;
}
```

Exemple — compter le nombre de bits `1`

Objectif : écrire une fonction `compter_un` qui

- prend en entrée un nombre `x` de 64 bits ;
- renvoie le **nombre de bits** `1` dans `x`.

Informations et rappels :

- le type `unsigned long long` est adapté pour représenter des nombres de 64 bits ;
- le `unsigned` est utile pour s'assurer que lors de décalages à droite bit à bit, ce sont toujours des `0` qui sont insérés.

Dans la suite, on suppose déclaré le type `Mot64` par

```
typedef unsigned long long Mot64;
```

Méthode 1 : décalages successifs

L'idée consiste à observer par `x & 1` si le bit de poids faible de `x` est à un, à en tenir compte, puis à décaler `x` vers la droite pour poursuivre l'observation sur le bit suivant.

– Exemple –

Pour simplifier, on se place sur 5 bits. On obtient successivement

x	x & 1	x >> 1
0110**1**	**1**	00110
0011**0**	**0**	00011
0001**1**	**1**	00001
0000**1**	**1**	00000
0000**0**	**0**	00000

Voici une implantation possible :

```
int compter_un_v1(Mot64 x) {
    int res, i;

    res = 0;
    for (i = 0 ; i < 64 ; ++i) {
        if (x & 1)
            res += 1;
        x = x >> 1;
    }
    return res;
}
```

Méthode 2 : extinctions successives

L'idée consiste à utiliser le fait que, quand `x` $\neq$ `0 ... 0`, l'expression `x & -x` est l'entier qui contient comme unique bit à un le `1` le plus à droite de `x`.

Ainsi, l'instruction

```
x = x ^ (x & -x);
```

transforme le bit `1` le plus à droite de `x` par un `0`.

– Exemple –

Pour simplifier, on se place sur 5 bits. On obtient successivement

x	-x	x & -x	x ^ (x & -x)
01101	10011	00001	01100
01100	10100	00100	01000
01000	11000	01000	00000

Voici une implantation possible :

```
int compter_un_v2(Mot64 x) {
    int res;

    res = 0;
    while (x != 0) {
        x = x ^ (x & -x);
        res += 1;
    }
    return res;
}
```

Méthode 3 : par table

Cette méthode n'utilise **pas de boucle** mas une **table pré-calculée**.

On commence par construire un tableau qui associe à tout entier de huit bits (un `char`) le nombre de bits à un qu'il contient, en utilisant au choix l'une des deux méthodes précédentes.

```
int nombre_un_v3[256]; /* Table en variable globale. */

void initialiser_nombre_un_v3() {
    int i;
    Mot64 x;
    for (i = 0 ; i < 256 ; ++i) {
        x = (Mot64) i;
        nombre_un_v3[i] = compter_un_v2(x);
    }
}
```

À ce stade, après avoir effectué ce pré-calcul, nous avons par exemple que

- `nombre_un_v3[0xA1]` a pour valeur `3` car `0xA1` est l'octet `10100001` ;
- `nombre_un_v3[0x78]` a pour valeur `4` car `0x78` est l'octet `01111000`.

Méthode 3 : par table

On isole l'octet d'indice `i` d'une variable `x` de type `Mot64` par l'expression

```
0xFF & (x >> (8 * i))
```

Comme le nombre de bits à un de `x` est la somme du nombre de bits à un de chacun des huit octets qui le constituent, on obtient

```
int compter_un_v3(Mot64 x) {
    int res;
    res = 0;
    res += nombre_un_v3[0xFF & x];
    res += nombre_un_v3[0xFF & (x >> 8)];
    res += nombre_un_v3[0xFF & (x >> 16)];
    res += nombre_un_v3[0xFF & (x >> 24)];
    res += nombre_un_v3[0xFF & (x >> 32)];
    res += nombre_un_v3[0xFF & (x >> 40)];
    res += nombre_un_v3[0xFF & (x >> 48)];
    res += nombre_un_v3[0xFF & (x >> 56)];
    return res;
}
```

Méthode 4 : par table plus grande

Nous pouvons pousser plus loin l'idée précédente en considérant des blocs de deux octets (au lieu d'un seul).

```
int nombre_un_v4[65536];

void initialiser_nombre_un_v4() {
    int i;
    Mot64 x;
    for (i = 0 ; i < 65536 ; ++i) {
        x = (Mot64) i;
        nombre_un_v4[i] = compter_un_v2(x);
    }
}
```

Taille mémoire occupée par le tableau `nombre_un` :

$$2^{16} \times \texttt{sizeof(int)}\ \text{o} \;=\; 2^{18}\ \text{o} \;=\; \frac{2^{18}}{2^{10}}\ \text{Kio} \;=\; 256\ \text{Kio}.$$

Méthode 4 : par table plus grande

Ceci fournit la solution suivante.

```
int compter_un_v4(Mot64 x) {
    int res;
    res = 0;
    res += nombre_un_v4[0xFFFF & x];
    res += nombre_un_v4[0xFFFF & (x >> 16)];
    res += nombre_un_v4[0xFFFF & (x >> 32)];
    res += nombre_un_v4[0xFFFF & (x >> 48)];
    return res;
}
```

Elle ne demande que quatre lectures dans le tableau `nombre_un`, au lieu des huit de la méthode précédente.

À l'extrême, il est impossible de maintenir un tableau `nombre_un` qui ne demanderait qu'une seule lecture. En effet, sa taille mémoire serait de

$$2^{64} \times \texttt{sizeof(int)}\ \text{o} \;=\; 2^{66}\ \text{o} \;=\; \frac{2^{66}}{2^{30}}\ \text{Gio} \;=\; 2^{36}\ \text{Gio}.$$

Méthode 5 : par masques bit à bit

Cette méthode est la plus compliquée. Elle se base sur une compréhension très fine du fonctionnement des opérateurs bit à bit.

```
int compter_un_v5(Mot64 x) {
    x = x - ((x >> 1) & 0x5555555555555555LLU);
    x = (x & 0x3333333333333333LLU) + ((x >> 2) & 0x3333333333333333LLU);
    x = (x + (x >> 4)) & 0x0F0F0F0F0F0F0F0FLLU;
    x = (x * 0x0101010101010101LLU) >> 56;
    return (int) x;
}
```

Mesure du temps d'exécution

Mesurons maintenant les efficacités des cinq méthodes.

On utilise pour cela la fonction

```
clock_t clock();
```

de `time.h`.

Schéma général pour **mesurer le temps d'exécution** d'une suite d'instructions :

```
clock_t debut, fin;
double temps;

debut = clock();
/* Instructions a mesurer ici. */
fin = clock();

/* Affichage du temps. */
temps = (double) (fin - debut) / CLOCKS_PER_SEC;
printf("%g s\n", temps);
```

Génération aléatoire d'entrées

On mesure le temps d'exécution des cinq méthodes en leur donnant en entrée des nombres de 64 bits générés de manière aléatoire.

Pour générer un nombre de 32 bits de manière aléatoire, on utilise la fonction `int rand();` de `stdlib.h`.

Le nombre de 64 bits est construit en générant aléatoirement ses quatre octets de droite, puis ses quatre octets de gauche et en les **associant avec un ou bit à bit** :

```
Mot64 mot64_alea() {
    Mot64 gauche, droite;

    gauche = rand();
    droite = rand();
    return (gauche << 32) | droite;
}
```

– Exemple –

En supposant pour simplifier que nous souhaiter générer un nombre de 10 bits à partir de deux nombres de 5 bits :

`gauche`	00000	10011
`gauche << 5`	10011	00000
`droite`	00000	01010
`(gauche << 5) | droite`	10011	01010

Comparaison des performances

Voici les temps réalisés par chacune des cinq méthodes sur 84000000 nombres de 64 bits générés aléatoirement :

Méthode	Caractéristique	Temps (s)
1	Décalages successifs	27.31
2	Extinctions successives	6.36
3	Par petite table	2.53
4	Par grande table	2.37
5	Par masques	2.33

La 2e méthode est plus de **4 fois plus rapide** que la 1re et la 5e est presque **12 fois plus rapide** que la 1re.

Les 3e et 4e méthodes demandent un **pré-calcul** et une occupation mémoire (par la table).

La 5e méthode est la plus rapide et ne demande aucun pré-calcul. Elle est en revanche difficile à comprendre et très difficile à imaginer.