

# Plan

## Types structurés

Déclaration et initialisation

Affectation et comparaison

Dans les fonctions

**Alignement en mémoire**

# Alignement en mémoire

L'**alignement en mémoire** d'une donnée est la façon dont celle-ci est organisée dans la mémoire.

# Alignement en mémoire

L'**alignement en mémoire** d'une donnée est la façon dont celle-ci est organisée dans la mémoire.

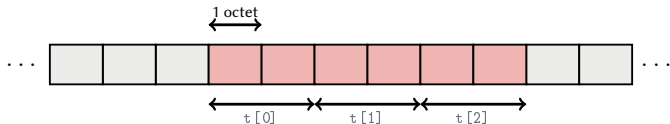
Par exemple, nous avons déjà vu que les **tableaux** de taille  $n$  d'éléments d'un type  $T$  sont organisés en un segment contigu de `sizeof(T) * n` octets.

# Alignement en mémoire

L'**alignement en mémoire** d'une donnée est la façon dont celle-ci est organisée dans la mémoire.

Par exemple, nous avons déjà vu que les **tableaux** de taille  $n$  d'éléments d'un type  $T$  sont organisés en un segment contigu de `sizeof(T) * n` octets.

Ainsi, un tableau `t` de 3 éléments de type `short` est organisé en

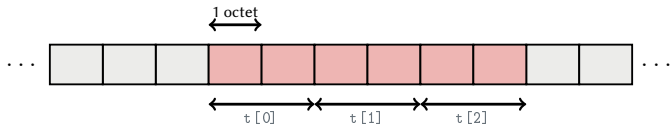


# Alignement en mémoire

L'**alignement en mémoire** d'une donnée est la façon dont celle-ci est organisée dans la mémoire.

Par exemple, nous avons déjà vu que les **tableaux** de taille  $n$  d'éléments d'un type  $T$  sont organisés en un segment contigu de `sizeof(T) * n` octets.

Ainsi, un tableau `t` de 3 éléments de type `short` est organisé en



On peut se poser de la même manière la question de l'**alignement mémoire** des variables d'un **type structuré**.

# Alignement en mémoire des variables d'un type structuré

Considérons les déclarations de types

```
1  typedef struct {
2      short x;
3      short y;
4      int z;
5  } A;
6  typedef struct {
7      short x;
8      int z;
9      short y;
10 } B;
```

A et B sont des types structurés composés des mêmes champs. Il n'y a que l'ordre de leur déclaration qui diffère.

Cependant,

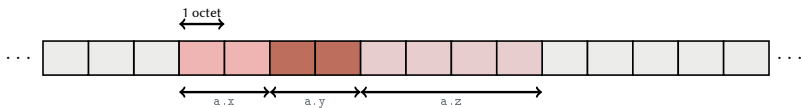
```
1  printf("%lu_%lu\n", sizeof(A), sizeof(B));
```

affiche 8 12.

Le fait que les tailles des variables de type A et B diffèrent est dû à leur **alignements en mémoire** respectifs qui ne sont pas les mêmes.

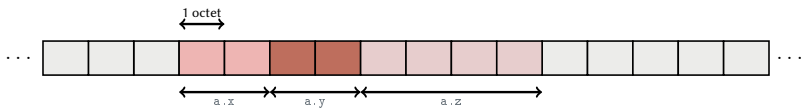
# Alignement en mémoire des variables d'un type structuré

Soit  $a$  une variable de type  $A$ . Cette variable est organisée en mémoire en

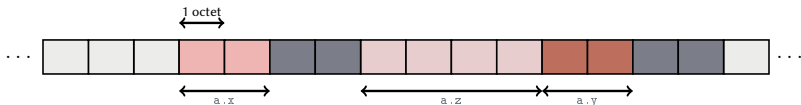


# Alignement en mémoire des variables d'un type structuré

Soit  $a$  une variable de type  $A$ . Cette variable est organisée en mémoire en



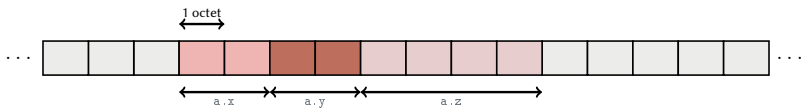
Soit  $b$  une variable de type  $B$ . Cette variable est organisée en mémoire en



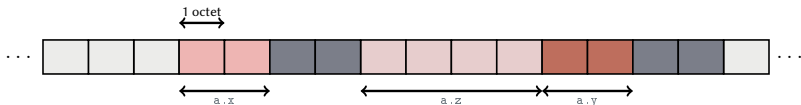


# Alignement en mémoire des variables d'un type structuré

Soit  $a$  une variable de type  $A$ . Cette variable est organisée en mémoire en



Soit  $b$  une variable de type  $B$ . Cette variable est organisée en mémoire en



Les octets en gris intervenant dans l'alignement mémoire de  $b$  sont des **octets de complétion**.

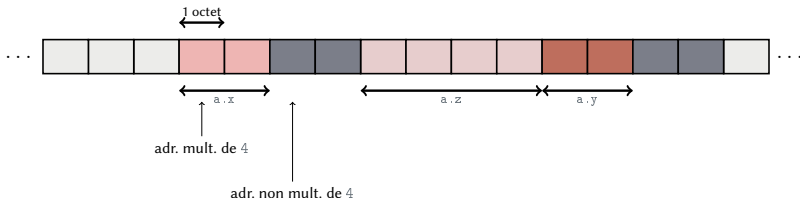
## Octets de complétion

Des octets de complétion sont introduits pour que chaque champ  $c$  d'une variable d'un type structuré **commence à une adresse multiple d'un entier** dépendant du type de  $c$ .

# Octets de complétion

Des octets de complétion sont introduits pour que chaque champ  $c$  d'une variable d'un type structuré **commence à une adresse multiple d'un entier** dépendant du type de  $c$ .

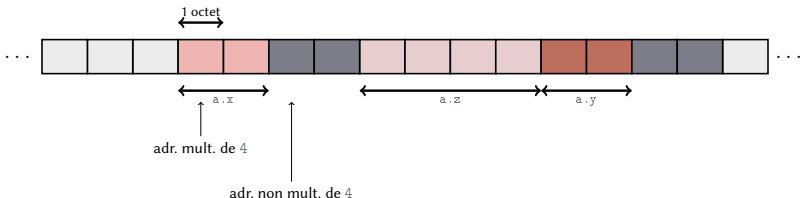
Dans notre exemple, en sachant que tout champ de type `short` (resp. `int`) doit commencer à une adresse multiple de 2 (resp. 4), on explique l'alignement en mémoire de  $b$  précédent :



# Octets de complétion

Des octets de complétion sont introduits pour que chaque champ  $c$  d'une variable d'un type structuré **commence à une adresse multiple d'un entier** dépendant du type de  $c$ .

Dans notre exemple, en sachant que tout champ de type `short` (resp. `int`) doit commencer à une adresse multiple de 2 (resp. 4), on explique l'alignement en mémoire de  $b$  précédent :



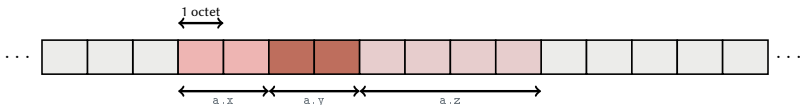
Les derniers octets de complétion sont introduits pour que les tableaux de variables de type  $B$  puissent être représentés en vérifiant cet alignement en mémoire.

# Accès manuel aux champs

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

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

Cette variable est organisée en mémoire en



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

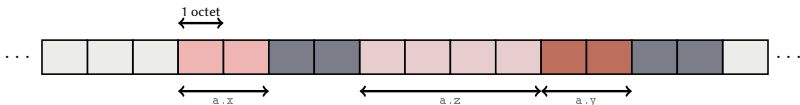
```
1  short x, y;  
2  int z;  
3  void *p;  
4  p = &a;  
5  x = *((short *) p); /* Equivalent a x = a.x; */  
6  p += 2; /* p est de type void * : l'adresse p est incrementee de 2 */  
7  y = *((short *) p); /* Equivalent a y = a.y; */  
8  p += 2;  
9  z = *((int *) p); /* Equivalent a z = a.z; */
```

# Accès manuel aux champs

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

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

Cette variable est organisée en mémoire en



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

```
1  short x, y;  
2  int z;  
3  void *p;  
4  p = &b;  
5  x = *((short *) p); /* Equivalent a x = a.x; */  
6  p += 4;  
7  z = *((int *) p); /* Equivalent a z = a.z; */  
8  p += 4;  
9  y = *((short *) p); /* Equivalent a y = a.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.

Par exemple, avec le type structuré `B` défini par

```
1 typedef struct {
2     short x;
3     int z;
4     short y;
5 } 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;



# 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.

# 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**.

# 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

Opérateurs

Pointeurs de fonction

Génération aléatoire

Mémoïsation

# Plan

## Opérateurs

- Généralités

- Opérateurs d'accès

- Opérateurs de calcul

- Opérateurs d'affectation

- Autres opérateurs

# Plan

## Opérateurs

- Généralités

- Opérateurs d'accès

- Opérateurs de calcul

- Opérateurs d'affectation

- Autres opérateurs

# Caractéristiques d'un opérateur

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

# Caractéristiques d'un opérateur

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

1. son **arité**, qui désigne le nombre d'opérandes sur lesquelles il agit ;



# 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 ;

# 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  $3 * 2 + 1$ .

# Précédence et associativité des opérateurs

Considérons l'expression  $3 * 2 + 1$ .

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

# Précédence et associativité des opérateurs

Considérons l'expression  $3 * 2 + 1$ .

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

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

# Précédence et associativité des opérateurs

Considérons l'expression  $3 * 2 + 1$ .

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

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

# Précédence et associativité des opérateurs

Considérons l'expression  $3 * 2 + 1$ .

Suivant les priorités relatives des opérateurs  $*$  et  $+$ , il y a 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  $4 - 3 - 2 - 1$ .

# Précédence et associativité des opérateurs

Considérons l'expression  $3 * 2 + 1$ .

Suivant les priorités relatives des opérateurs  $*$  et  $+$ , il y a 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 :



## Précédence et associativité des opérateurs

Considérons l'expression  $3 * 2 + 1$ .

Suivant les priorités relatives des opérateurs  $*$  et  $+$ , il y a 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**;

## Précédence et associativité des opérateurs

Considérons l'expression  $3 * 2 + 1$ .

Suivant les priorités relatives des opérateurs  $*$  et  $+$ , il y a 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**.

## Précédence et associativité des opérateurs

Considérons l'expression  $3 * 2 + 1$ .

Suivant les priorités relatives des opérateurs  $*$  et  $+$ , il y a 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**.

# Plan

## Opérateurs

Généralités

Opérateurs d'accès

Opérateurs de calcul

Opérateurs d'affectation

Autres opérateurs

# Opérateurs de gestion la mémoire

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
------------	-------------	-------------	---------------	------------------

# Opérateurs de gestion la mémoire

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
&	référencement	1	-	une variable

# Opérateurs de gestion la mémoire

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
&	référencement	1	–	une variable
*	déréférencement	1	–	un pointeur

# Opérateurs de gestion la mémoire

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
&	référencement	1	-	une variable
*	déréférencement	1	-	un pointeur
[ ]	élément d'un tableau	2	→	un pointeur et un entier



# Opérateurs de gestion la mémoire

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
&	référencement	1	–	une variable
*	déréférencement	1	–	un pointeur
[ ]	élément d'un tableau	2	→	un pointeur et un entier
.	valeur d'un champ	2	→	une var. d'un type struct. et un id. de champ

# Opérateurs de gestion la mémoire

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

# Plan

## Opérateurs

Généralités

Opérateurs d'accès

**Opérateurs de calcul**

Opérateurs d'affectation

Autres opérateurs

# Opérateurs arithmétiques

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
------------	-------------	-------------	---------------	------------------

# Opérateurs arithmétiques

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
+, -, *, /	opérations arith.	2	→	deux val. numériques

# Opérateurs arithmétiques

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
+, -, *, /	opérations arith.	2	→	deux val. numériques
%	modulo	2	→	deux entiers

# Opérateurs arithmétiques

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
+, -, *, /	opérations arith.	2	→	deux val. numériques
%	modulo	2	→	deux entiers
+, -	signe	1	-	une val. numérique

# Opérateurs arithmétiques

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
+, -, *, /	opérations arith.	2	→	deux val. numériques
%	modulo	2	→	deux entiers
+, -	signe	1	-	une val. numérique
++, --	incr./décr.	1	-	une var. d'un type numérique



# L'opérateur modulo

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

# 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  $0 \leq r \leq b - 1$  et  $q$  est un entier.  $q$  est le **quotient** et  $r$  est le **reste**, **toujours positif**.

# 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  $0 \leq r \leq b - 1$  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.

# 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  $0 \leq r \leq b - 1$  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;
    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 :

# 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 ;

# 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`.

# 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`.

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

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



# 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`.

```
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.

# Les opérateurs d'incrémentation et de décrémentation

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

P.ex., les instructions

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

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

# Les opérateurs d'incrémentation et de décrémentation

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

P.ex., les instructions

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

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
------------	-------------	-------------	---------------	------------------

## Opérateurs relationnels

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
<, >	comparaison stricte	2	→	deux val. numériques

## Opérateurs relationnels

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
<, >	comparaison stricte	2	→	deux val. numériques
<=, >=	comparaison large	2	→	deux val. numériques

## Opérateurs relationnels

Op.	Rôle	Ari.	Assoc.	Opérandes
<, >	comparaison stricte	2	→	deux val. numériques
<=, >=	comparaison large	2	→	deux val. numériques
==	égalité	2	→	deux val. numériques



## Opérateurs relationnels

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

## Opérateurs relationnels

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

Toutes les expressions de la forme

$$v1 \text{ CMP } v2$$

où  $v1$  et  $v2$  sont des valeurs numériques et  $\text{CMP}$  est un opérateur de comparaison produisant 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.

## Opérateurs relationnels

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

```
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.

## Opérateurs relationnels

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

```
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

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
------------	-------------	-------------	---------------	------------------

# Opérateurs logiques

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
&&	et logique	2	→	deux val. numériques

# Opérateurs logiques

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
&&	et logique	2	→	deux val. numériques
	ou logique	2	→	deux val. numériques



# Opérateurs logiques

Op.	Rôle	Ari.	Assoc.	Opérandes
&&	et logique	2	→	deux val. numériques
	ou logique	2	→	deux val. numériques
!	non logique	1	-	une val. numérique

# Opérateurs logiques

Op.	Rôle	Ari.	Assoc.	Opérandes
&&	et logique	2	→	deux val. numériques
	ou logique	2	→	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

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
------------	-------------	-------------	---------------	------------------

# Opérateurs bit à bit

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
&	et bit à bit	2	→	deux val. entières

# Opérateurs bit à bit

<b>Op.</b>	<b>Rôle</b>	<b>Ari.</b>	<b>Assoc.</b>	<b>Opérandes</b>
&	et bit à bit	2	→	deux val. entières
	ou bit à bit	2	→	deux val. entières

# Opérateurs bit à bit

Op.	Rôle	Ari.	Assoc.	Opérandes
&	et bit à bit	2	→	deux val. entières
	ou bit à bit	2	→	deux val. entières
^	xor bit à bit	2	→	deux val. entières

# Opérateurs bit à bit

Op.	Rôle	Ari.	Assoc.	Opérandes
&	et bit à bit	2	→	deux val. entières
	ou bit à bit	2	→	deux val. entières
^	xor bit à bit	2	→	deux val. entières
~	non bit à bit	1	-	une val. entière

# Opérateurs bit à bit

Op.	Rôle	Ari.	Assoc.	Opérandes
&	et bit à bit	2	→	deux val. entières
	ou bit à bit	2	→	deux val. entières
^	xor bit à bit	2	→	deux val. entières
~	non bit à bit	1	-	une val. entière
<<, >>	déc. g./d. bit à bit	2	→	deux val. entières



# Et/ou/xor/non bit à bit

Quelques exemples d'opérations bit à bit :

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

# Et/ou/xor/non bit à bit

Quelques exemples d'opérations bit à bit :

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

# Et/ou/xor/non bit à bit

Quelques exemples d'opérations bit à bit :

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	0	1	1	0	0	1

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

# Et/ou/xor/non bit à bit

Quelques exemples d'opérations bit à bit :

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	0	1	1	0	0	1

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
~x	1	0	0	0	1	0	1	0

## Et/ou/xor/non bit à bit

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.

## Et/ou/xor/non bit à bit

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.

Le signe d'un entier signé est lu sur son bit de poids fort :

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

## Et/ou/xor/non bit à bit

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.

Le signe d'un entier signé est lu sur son bit de poids fort :

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

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

x	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
y	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
x   y	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

## Et/ou/xor/non bit à bit

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.

Le signe d'un entier signé est lu sur son bit de poids fort :

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

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

x	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
y	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
x   y	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

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

x	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
y	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0
x   y	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1



# Décalage bit à bit

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

# Décalage bit à bit

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

<code>x</code>	0	1	1	1	0	1	0	1
<code>x &lt;&lt; 3</code>	1	0	1	0	1	0	0	0

# Décalage bit à bit

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

# Décalage bit à bit

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

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

# Décalage bit à bit

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

# Décalage bit à bit

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

# Décalage bit à bit

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

$x$	1	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

# Décalage bit à bit

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

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

$x$	0	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	0	1	1	1	0	1	0	1
$x \gg 3$	0	0	0	0	1	1	1	0

$x$	1	1	1	1	0	1	0	1
$x \ll 3$	1	0	1	0	1	0	0	0

$x$	1	1	1	1	0	1	0	1
$x \gg 3$	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.

## 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,

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

## 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,

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

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

## 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,

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

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

P.ex., l'entier dont l'écriture binaire est

000100001000000000000001100000101

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

```
1 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.

Ainsi,

```
1 int appartient_e_i(SousEnsemble s, int i) {  
2     assert(0 <= i);  
3     assert(i <= 31);  
4  
5     return (1 << i) & s;  
6 }
```

## 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  $e_i \in S_2$ .

Ainsi,

```
1  SousEnsemble union(SousEnsemble s_1, SousEnsemble s_2) {  
2      return s_1 | s_2;  
3  }
```

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  $e_i \in S_2$ .

Ainsi,

```
1  SousEnsemble intersection(SousEnsemble s_1, SousEnsemble s_2) {  
2      return s_1 & s_2;  
3  }
```

## Exemple — Compter le nombre de bits à un

**But** : écrire une fonction qui renvoie le nombre de bits à un de son paramètre.

## Exemple — Compter le nombre de bits à un

**But** : écrire une fonction qui renvoie le nombre de bits à un de son paramètre.

On travaille sur des variables de 64 bits. On considère pour cela le type alias `Mot64` défini par

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



## Exemple — Compter le nombre de bits à un

**But** : écrire une fonction qui renvoie le nombre de bits à un de son paramètre.

On travaille sur des variables de 64 bits. On considère pour cela le type alias `Mot64` défini par

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

1<sup>re</sup> **méthode** : attraper le bit de poids faible et le pousser à droite.

## Exemple — Compter le nombre de bits à un

**But** : écrire une fonction qui renvoie le nombre de bits à un de son paramètre.

On travaille sur des variables de 64 bits. On considère pour cela le type alias `Mot64` défini par

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

1<sup>re</sup> **méthode** : attraper le bit de poids faible et le pousser à droite.

```
int compter_un_1(Mot64 x) {
```

```
}
```

## Exemple — Compter le nombre de bits à un

**But** : écrire une fonction qui renvoie le nombre de bits à un de son paramètre.

On travaille sur des variables de 64 bits. On considère pour cela le type alias `Mot64` défini par

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

1<sup>re</sup> **méthode** : attraper le bit de poids faible et le pousser à droite.

```
int compter_un_1(Mot64 x) {  
    int res, i;  
    res = 0;  
  
    return res;  
}
```

## Exemple — Compter le nombre de bits à un

**But** : écrire une fonction qui renvoie le nombre de bits à un de son paramètre.

On travaille sur des variables de 64 bits. On considère pour cela le type alias `Mot64` défini par

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

1<sup>re</sup> **méthode** : attraper le bit de poids faible et le pousser à droite.

```
int compter_un_1(Mot64 x) {
    int res, i;
    res = 0;
    for (i = 0 ; i < 64 ; ++i) {

        x = x >> 1;
    }
    return res;
}
```

## Exemple — Compter le nombre de bits à un

**But** : écrire une fonction qui renvoie le nombre de bits à un de son paramètre.

On travaille sur des variables de 64 bits. On considère pour cela le type alias `Mot64` défini par

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

1<sup>re</sup> **méthode** : attraper le bit de poids faible et le pousser à droite.

```
int compter_un_1(Mot64 x) {
    int res, i;
    res = 0;
    for (i = 0 ; i < 64 ; ++i) {
        if ((x & 1) == 1)
            res += 1;
        x = x >> 1;
    }
    return res;
}
```

## Exemple — Compter le nombre de bits à un

2<sup>e</sup> **méthode** : on constate que pour tout entier  $x$  non nul (avec au moins un bit à un), l'expression  $x \& -x$  est l'entier qui contient un unique bit à un, le plus à droite de  $x$ .

## Exemple — Compter le nombre de bits à un

2<sup>e</sup> **méthode** : on constate que pour tout entier  $x$  non nul (avec au moins un bit à un), l'expression  $x \& -x$  est l'entier qui contient un unique bit à un, le plus à droite de  $x$ .

Ainsi, l'instruction

$$x = x \wedge (x \& -x);$$

transforme le bit à un le plus à droite de  $x$  en un zéro.

## Exemple — Compter le nombre de bits à un

2<sup>e</sup> **méthode** : on constate que pour tout entier  $x$  non nul (avec au moins un bit à un), l'expression  $x \& -x$  est l'entier qui contient un unique bit à un, le plus à droite de  $x$ .

Ainsi, l'instruction

$$x = x \wedge (x \& -x);$$

transforme le bit à un le plus à droite de  $x$  en un zéro.

L'exploitation de cette idée donne



## Exemple — Compter le nombre de bits à un

2<sup>e</sup> **méthode** : on constate que pour tout entier  $x$  non nul (avec au moins un bit à un), l'expression  $x \& -x$  est l'entier qui contient un unique bit à un, le plus à droite de  $x$ .

Ainsi, l'instruction

$$x = x \wedge (x \& -x);$$

transforme le bit à un le plus à droite de  $x$  en un zéro.

L'exploitation de cette idée donne

```
int compter_un_2(Mot64 x) {
```

```
}
```

## Exemple — Compter le nombre de bits à un

2<sup>e</sup> **méthode** : on constate que pour tout entier  $x$  non nul (avec au moins un bit à un), l'expression  $x \& -x$  est l'entier qui contient un unique bit à un, le plus à droite de  $x$ .

Ainsi, l'instruction

$$x = x \wedge (x \& -x);$$

transforme le bit à un le plus à droite de  $x$  en un zéro.

L'exploitation de cette idée donne

```
int compter_un_2(Mot64 x) {
    int res;
    res = 0;

    return res;
}
```

## Exemple — Compter le nombre de bits à un

2<sup>e</sup> **méthode** : on constate que pour tout entier  $x$  non nul (avec au moins un bit à un), l'expression  $x \& -x$  est l'entier qui contient un unique bit à un, le plus à droite de  $x$ .

Ainsi, l'instruction

$$x = x \wedge (x \& -x);$$

transforme le bit à un le plus à droite de  $x$  en un zéro.

L'exploitation de cette idée donne

```
int compter_un_2(Mot64 x) {
    int res;
    res = 0;
    while (x != 0) {
        x = x ^ (x & -x);
        res += 1;
    }
    return res;
}
```