Affectation de variables d'un type structuré

Considérons le code

```
1  typedef struct {
2      char *t;
3      int n;
4  } T;
5  ...
6  T v1, v2;
7  v1.t = (char *)
8      malloc(sizeof(char) * 3);
9  v1.n = 3;
10 v1.t[0] = 'a';
11 v1.t[1] = 'b';
12 v1.t[2] = 'c';
13 v2 = v1;
14 v2.n = 2;
15 v2.t[0] = 'g';
```

Dessiner le contenu de v1 *et* v2 *aux l. 12, 13 et 15.*

Observation : l'affectation ne recopie pas les tableaux dynamiques. Seule l'adresse d'un tableau dynamique est recopiée. C'est une copie de surface.

Affectation de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré `X`, on définit une fonction de prototype

```
void copier_X(const X *v1, X *v2);
```

qui copie en profondeur les champs de `v1` dans les champs de `v2`.

Par exemple, la définition du type `T` précédent s'accompagne de la définition de la fonction

```
1  void copier_T(const T *v1, T *v2) {
2      int i;
3      assert(v1 != NULL);
4      assert(v2 != NULL);
5      v2->n = v1->n;
6      v2->t = (char *) malloc(sizeof(char) * v1->n);
7      if (v2->t == NULL) exit(EXIT_FAILURE);
8      for (i = 0 ; i < v1->n ; ++i)
9          v2->t[i] = v1->t[i];
10 }
```

Il est possible de munir cette fonction du mécanisme habituel de gestion d'erreurs.

Comparaison de variables d'un type structuré

Considérons le code

```
1  typedef struct {
2      int a;
3      int b;
4  } A;
5  ...
```

```
6  A v1, v2;
7  ...
8  if (v1 == v2) {...}
9  ...
10 if (v1 != v2) {...}
```

Ce code est incorrect (il ne compile pas).

Le compilateur n'accepte pas la comparaison de variables d'un type structuré.

```
invalid operands to binary == (have 'A' and 'A')
```

```
invalid operands to binary != (have 'A' and 'A')
```

Comparaison de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré `X`, on définit deux fonctions de prototypes

```
int sont_ega_X(const X *v1, const X *v2);
int sont_dif_X(const X *v1, const X *v2);
```

qui testent l'égalité et l'inégalité entre `v1` et `v2`.

Par exemple, la définition du type `A` précédent s'accompagne de la définition des fonctions

```
1  int sont_ega_A(A *v1, A *v2) {
2      assert(v1 != NULL);
3      assert(v2 != NULL);
4      return (v1->a == v2->a)
5          && (v1->b == v2->b);
6  }
```

```
7  int sont_dif_A(A *v1, A *v2) {
8      assert(v1 != NULL);
9      assert(v2 != NULL);
10     return !sont_ega_A(v1, v2);
11 }
```

Attention : si `X` est composé d'un champ qui est un type structuré `Y`, il faut appeler dans `sont_ega_X` la fonction de comparaison `sont_ega_Y`.

Destruction de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré `X`, on définit une fonction de prototype

```
void detruire_X(X *v);
```

qui libère l'espace mémoire adressé par `v`.

Par exemple, la déclaration du type `B` suivant s'accompagne de la définition de la fonction

```
1  typedef struct {
2      int *tab;
3      int n;
4  } B;
5
```

```
6  void detruire_B(B *v) {
7      assert(v != NULL);
8      free(v->tab);
9      *v = NULL;
10 }
```

Attention : si `X` est composé d'un champ qui est un type structuré `Y`, il faut appeler dans `detruire_X` la fonction de comparaison `detruire_Y`.

Résumé

Voici en résumé la bonne marche à suivre lors de la manipulation de types structurés :

1. on utilise l'**alias** lors de la déclaration de types structurés **récursifs** et/ou **mutuellement récursifs** ;
2. on ne **renvoie jamais** de valeur d'un type structuré ;
3. on passe les **paramètres** d'un type structuré **par adresse** ;
4. toute **déclaration d'un type structuré** s'accompagne de la définition des quatre fonctions suivantes :
   - une fonction de **copie** ;
   - une fonction de **test d'égalité** ;
   - une fonction de **test d'inégalité** ;
   - une fonction de **destruction**.

Axe 3 : utiliser quelques techniques avancées

Plan

Plan

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. 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 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 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 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 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 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 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 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 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 de gestion 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

Plan

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

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 $0 \leqslant r \leqslant b - 1$ 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 $0 \leqslant r \leqslant b - 1$ 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.

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;
    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;
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).

Les opérateurs d'incrémentation et de décrémentation

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

P.ex., 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

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.

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 juste `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 juste `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

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

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`),

`x`	0	1	1	1	0	1	0	1
`x << 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 << 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

Décalage bit à bit

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

`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

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 << 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

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

`x`	0	1	1	1	0	1	0	1
`x << 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 << 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

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

`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

Décalage bit à bit

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

`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

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

`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

Décalage bit à bit

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

`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

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

`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.

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,

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

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 — 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.

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

$$00010000100000000000001100000101$$

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

```
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 `Mot64` définit 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 `Mot64` définit par

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

1re **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 Mot64 définit par

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

1re **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 Mot64 définit par

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

1re **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 Mot64 définit par

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

1re **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 Mot64 définit par

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

1re **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;
}
```