Plan

Types entier

On se place sur une machine 64 bits.

Nom	Taille (octets)	Plage
`char`	1	-128 à 127
`short`	2	-32768 à 32767
`int`	4	-2^{31} à $2^{31}-1$
`long`	8	-2^{63} à $2^{63}-1$

Types entier

On se place sur une machine 64 bits.

Nom	Taille (octets)	Plage
`char`	1	-128 à 127
`short`	2	-32768 à 32767
`int`	4	-2^{31} à $2^{31}-1$
`long`	8	-2^{63} à $2^{63}-1$

Chacun de ces types peut être précédé de `unsigned` pour faire en sorte de ne représenter que des entiers positifs. On a ainsi les plages suivantes :

Nom	Plage
`unsigned char`	0 à 255
`unsigned short`	0 à 65535
`unsigned int`	0 à $2^{32}-1$
`unsigned long`	0 à $2^{64}-1$

Entiers non signés

Dans le cas où l'on a besoin de représenter uniquement des valeurs entières positives, on utilisera les versions non signées des types entiers.

Entiers non signés

Dans le cas où l'on a besoin de représenter uniquement des valeurs entières positives, on utilisera les versions non signées des types entiers.

Quelques avantages de ce procédé :

1. possibilité de représenter des entiers plus grands ;
2. gain de lisibilité du programme.

Entiers non signés

Dans le cas où l'on a besoin de représenter uniquement des valeurs entières positives, on utilisera les versions non signées des types entiers.

Quelques avantages de ce procédé :

1. possibilité de représenter des entiers plus grands ;
2. gain de lisibilité du programme.

Attention : les instructions

```
unsigned int i;
for (i = 8 ; i >= 0 ; --i) {
    ...
}
```

produisent une boucle infinie. En effet, `i` étant non signé, il est toujours positif et donc la condition `i >= 0` est toujours vraie.

Constantes entières

Il existe plusieurs manières d'exprimer des constantes entières :

- en base dix : `0`, `29`, `-322`, …

Constantes entières

Il existe plusieurs manières d'exprimer des constantes entières :

- en base dix : `0`, `29`, `-322`, …
- en octal : `01`, `0145`, `-01234567`, …

Constantes entières

Il existe plusieurs manières d'exprimer des constantes entières :

- en base dix : `0`, `29`, `-322`, ...
- en octal : `01`, `0145`, `-01234567`, ...
- en hexadécimal : `0x1`, `0x5555FFFF`, `-0x98879AFA`, ...

Constantes entières

Il existe plusieurs manières d'exprimer des constantes entières :

- en base dix : `0`, `29`, `-322`, ...
- en octal : `01`, `0145`, `-01234567`, ...
- en hexadécimal : `0x1`, `0x5555FFFF`, `-0x98879AFA`, ...
- par un caractère : `'a'`, `'9'`, `'*'`, `'\n'`, ...

Constantes entières

Il existe plusieurs manières d'exprimer des constantes entières :

- en base dix : `0`, `29`, `-322`, ...
- en octal : `01`, `0145`, `-01234567`, ...
- en hexadécimal : `0x1`, `0x5555FFFF`, `-0x98879AFA`, ...
- par un caractère : `'a'`, `'9'`, `'*'`, `'\n'`, ...

Un entier peut être représenté par un caractère car tout caractère est représenté par son code ASCII (qui est un entier compris entre `0` et `127`).

Constantes entières

Il existe plusieurs manières d'exprimer des constantes entières :

- en base dix : `0`, `29`, `-322`, ...
- en octal : `01`, `0145`, `-01234567`, ...
- en hexadécimal : `0x1`, `0x5555FFFF`, `-0x98879AFA`, ...
- par un caractère : `'a'`, `'9'`, `'*'`, `'\n'`, ...

Un entier peut être représenté par un caractère car tout caractère est représenté par son code ASCII (qui est un entier compris entre `0` et `127`).

Attention : ne pas confondre les caractères chiffres avec les entiers (l'entier `'1'` vaut `49` et non pas `1`).

Types flottant

On se place sur une machine 64 bits.

Nom	Taille (octets)	Valeur absolue maximale
`float`	4	3.40282×10^{38}
`double`	8	1.79769×10^{308}
`long double`	16	1.18973×10^{4932}

Le fichier d'en-tête `float.h` contient des constantes donnant d'autres renseignements sur les types flottant.

Danger des types flottant

```
float x = 10000001.0;
printf("%f\n", x);
```

Ces instructions affichent, de manière attendue, `10000001.000000`.

Danger des types flottant

```
float x = 10000001.0;
printf("%f\n", x);
```

Ces instructions affichent, de manière attendue, `10000001.000000`.

```
float x = 100000001.0;
printf("%f\n", x);
```

En revanche, ces instructions affichent, de manière inattendue, `100000000.000000`.

Danger des types flottant

```
float x = 10000001.0;
printf("%f\n", x);
```

Ces instructions affichent, de manière attendue, `10000001.000000`.

```
float x = 100000001.0;
printf("%f\n", x);
```

En revanche, ces instructions affichent, de manière inattendue, `100000000.000000`.

Les nombres flottants sont représentés de manière approchée.

Comme ces exemples le montrent, même certains entiers, représentables de manière exacte par des types entier, ne le sont pas par des types flottant.

Danger des types flottant : une solution partielle

Les types flottant présentent divers désavantages par rapport aux types entier :

1. représentation non exacte des nombres ;
2. opérations arithmétiques beaucoup moins efficaces.

Danger des types flottant : une solution partielle

Les types flottant présentent divers désavantages par rapport aux types entier :

1. représentation non exacte des nombres ;
2. opérations arithmétiques beaucoup moins efficaces.

Pour ces raisons, **il est recommandé de ne jamais utiliser de types flottant**.

Danger des types flottant : une solution partielle

Les types flottant présentent divers désavantages par rapport aux types entier :

1. représentation non exacte des nombres ;
2. opérations arithmétiques beaucoup moins efficaces.

Pour ces raisons, **il est recommandé de ne jamais utiliser de types flottant**.

Solution partielle : on représente par l'entier $x \times 10^k$ tout nombre x qui dispose de $k \geqslant 0$ chiffres (en base dix) après la virgule, k étant fixé.

Danger des types flottant : une solution partielle

Les types flottant présentent divers désavantages par rapport aux types entier :

1. représentation non exacte des nombres ;
2. opérations arithmétiques beaucoup moins efficaces.

Pour ces raisons, **il est recommandé de ne jamais utiliser de types flottant**.

Solution partielle : on représente par l'entier $x \times 10^k$ tout nombre x qui dispose de $k \geqslant 0$ chiffres (en base dix) après la virgule, k étant fixé.

P.ex., si l'on a besoin de manipuler des nombres à $k := 2$ chiffres après la virgule, les nombres 0.15 et 331.9 sont respectivement représentés par les entiers 15 et 33190.

Opérations sur les types scalaires

Les valeurs d'un type scalaire (entier ou flottant) forment un ensemble totalement ordonné : étant donné deux valeurs, il est toujours possible de les comparer. On utilise pour cela les opérateurs relationnels

`==`, `!=`, `<=`, `>=`, `<`, `>`.

Opérations sur les types scalaires

Les valeurs d'un type scalaire (entier ou flottant) forment un ensemble totalement ordonné : étant donné deux valeurs, il est toujours possible de les comparer. On utilise pour cela les opérateurs relationnels

`==`, `!=`, `<=`, `>=`, `<`, `>`.

Il est possible de mélanger des comparaisons de valeurs de types entier et de types flottant. Dans ce cas, les entiers sont convertis implicitement en une valeur de type flottant avant d'effectuer la comparaison.

Opérations sur les types scalaires

Les valeurs d'un type scalaire (entier ou flottant) forment un ensemble totalement ordonné : étant donné deux valeurs, il est toujours possible de les comparer. On utilise pour cela les opérateurs relationnels

`==`, `!=`, `<=`, `>=`, `<`, `>`.

Il est possible de mélanger des comparaisons de valeurs de types entier et de types flottant. Dans ce cas, les entiers sont convertis implicitement en une valeur de type flottant avant d'effectuer la comparaison.

Sur des variables de type scalaire sont définis les opérateurs arithmétiques

`+`, `-`, `*`, `/`, `++`, `--`.

Les opérateurs `++` et `--` servent à additionner ou à retrancher de 1 la valeur des variables sur lesquels ils sont appliqués.

Plan

Types structurés

La syntaxe

```
typedef struct ALIAS {
    TYPE_1 CHAMP_1;
    TYPE_2 CHAMP_2;
    ...
} NOM;
```

permet de déclarer un type structuré `NOM`, constitué des **champs** `CHAMP_1`, `CHAMP_2`, ... L'**alias** `ALIAS` est facultatif.

Types structurés

La syntaxe

```
typedef struct ALIAS {
    TYPE_1 CHAMP_1;
    TYPE_2 CHAMP_2;
    ...
} NOM;
```

permet de déclarer un type structuré `NOM`, constitué des **champs** `CHAMP_1`, `CHAMP_2`, ... L'**alias** `ALIAS` est facultatif.

C'est un **amalgame** de types.

Types structurés

La syntaxe

```
typedef struct ALIAS {
    TYPE_1 CHAMP_1;
    TYPE_2 CHAMP_2;
    ...
} NOM;
```

permet de déclarer un type structuré `NOM`, constitué des **champs** `CHAMP_1`, `CHAMP_2`, ... L'**alias** `ALIAS` est facultatif.

C'est un **amalgame** de types.

P.ex.,

```
typedef struct {
    int x;
    int y;
} Couple;
```

déclare un type structuré `Couple` qui permet de représenter des couples d'entiers.

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

```
x.ch
```

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

```
x.ch
```

Si `adr_x` est une adresse sur une variable de type `T`, on accède à ce même champ par la syntaxe

```
adr_x->ch
```

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

```
x.ch
```

Si `adr_x` est une adresse sur une variable de type `T`, on accède à ce même champ par la syntaxe

```
adr_x->ch
```

Cette syntaxe est un raccourci pour

```
(*adr_x).ch
```

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

```
x.ch
```

Si `adr_x` est une adresse sur une variable de type `T`, on accède à ce même champ par la syntaxe

```
adr_x->ch
```

Cette syntaxe est un raccourci pour

```
(*adr_x).ch
```

P.ex., les trois suites d'instructions suivantes sont équivalentes :

```
Couple *c;
...
c->x = c->x + 1;
```

```
Couple *c;
...
*(c).x = c->x + 1;
```

```
Couple *c;
...
c->x = (*c).x + 1;
```

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur `==`. Il faut tester l'égalité de chacun des champs qui les constituent.

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur ==. Il faut tester l'égalité de chacun des champs qui les constituent.

En revanche, l'**opérateur d'affectation** = est compatible avec les types structurés.

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur `==`. Il faut tester l'égalité de chacun des champs qui les constituent.

En revanche, l'**opérateur d'affectation** `=` est compatible avec les types structurés.

```
typedef struct {
    char nom[32];
    char prenom[32];
    int age;
} Personne;
...
Personne p1, p2;
scanf(" %s", p1.nom);
scanf(" %s", p1.prenom);
p1.age = 30;
p2 = p1;
```

L'affectation en dernière ligne fait en sorte que tous les champs de `p2` contiennent les mêmes valeurs que ceux de `p1`.

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur `==`. Il faut tester l'égalité de chacun des champs qui les constituent.

En revanche, l'**opérateur d'affectation** `=` est compatible avec les types structurés.

```
typedef struct {
    char nom[32];
    char prenom[32];
    int age;
} Personne;
...
Personne p1, p2;
scanf(" %s", p1.nom);
scanf(" %s", p1.prenom);
p1.age = 30;
p2 = p1;
```

L'affectation en dernière ligne fait en sorte que tous les champs de `p2` contiennent les mêmes valeurs que ceux de `p1`.

Il y a recopie des tableaux statiques `p1.nom` et `p1.prenom` dans `p2.nom` et `p2.prenom`.

Ce phénomène va être étudié en détail plus loin.

Types énumérés

La syntaxe

```
typedef enum {
    ENU_1,
    ENU_2,
    ...
} NOM;
```

permet de déclarer un type énuméré `NOM`, constitué des **énumérateurs** `ENU_1`, `ENU_2`, ... (Attention, on utilise des `,` et non pas des `;`.)

Types énumérés

La syntaxe

```
typedef enum {
    ENU_1,
    ENU_2,
    ...
} NOM;
```

permet de déclarer un type énuméré `NOM`, constitué des **énumérateurs** `ENU_1`, `ENU_2`, ... (Attention, on utilise des `,` et non pas des `;`.)

Une valeur de ce type prend pour valeur exactement un des énumérateurs qui le constituent.

Types énumérés

La syntaxe

```
typedef enum {
    ENU_1,
    ENU_2,
    ...
} NOM;
```

permet de déclarer un type énuméré `NOM`, constitué des **énumérateurs** `ENU_1`, `ENU_2`, ... (Attention, on utilise des `,` et non pas des `;`.)

Une valeur de ce type prend pour valeur exactement un des énumérateurs qui le constituent.

P.ex.,

```
typedef enum {
    FAUX,
    VRAI
} Booleen;
```

est un type qui permet de représenter des booléens.

Une valeur de type `Booleen` est soit `FAUX`, soit `VRAI`.

Types énumérés

```
typedef enum {
    LUNDI,     /* = 0 */
    MARDI,     /* = 1 */
    MERCREDI,  /* = 2 */
    JEUDI,     /* = 3 */
    VENDREDI,  /* = 4 */
    SAMEDI,    /* = 5 */
    DIMANCHE   /* = 6 */
} Jour;
...
printf("%d\n", MERCREDI);
```

Les énumérateurs sont des **expressions entières**. Leur valeur est déterminée par leur ordre de déclaration dans le type.

Ces instructions affichent 2. En effet, `LUNDI` vaut `0` car il est le 1er énumérateur déclaré et les valeurs des suivants s'incrémentent selon leur ordre de déclaration.

Types énumérés

```
typedef enum {
    LUNDI,     /* = 0 */
    MARDI,     /* = 1 */
    MERCREDI,  /* = 2 */
    JEUDI,     /* = 3 */
    VENDREDI,  /* = 4 */
    SAMEDI,    /* = 5 */
    DIMANCHE   /* = 6 */
} Jour;
...
printf("%d\n", MERCREDI);
```

Les énumérateurs sont des **expressions entières**. Leur valeur est déterminée par leur ordre de déclaration dans le type.

Ces instructions affichent 2. En effet, `LUNDI` vaut `0` car il est le 1er énumérateur déclaré et les valeurs des suivants s'incrémentent selon leur ordre de déclaration.

```
typedef enum {
    LA = 0,
    SI = 2,
    DO,        /* = 3 */
    RE = 5,
    MI = 7,
    FA = 8,
    SOL = 10
} Note;
```

Il est possible de spécifier manuellement les valeurs des énumérateurs avec la syntaxe `ENU = VAL` où `ENU` est un énumérateur et `VAL` une constante entière.

Si une valeur n'est pas spécifiée, elle est déduite de la précédente en l'incrémentant.

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

```
Note note;

scanf(" %d", &note);

switch (note) {
  case LA : printf("A"); break;
  case SI : printf("B"); break;
  case DO : printf("C"); break;
  case RE : printf("D"); break;
  case MI : printf("E"); break;
  case FA : printf("F"); break;
  case SOL : printf("G");break;
  default : printf(
    "%d non note", note);
}
```

Ces instructions lisent une valeur entière (représentant une `Note`) sur l'entrée standard et l'affichent (en notation internationale).

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

```
Note note;

scanf(" %d", &note);

switch (note) {
  case LA : printf("A"); break;
  case SI : printf("B"); break;
  case DO : printf("C"); break;
  case RE : printf("D"); break;
  case MI : printf("E"); break;
  case FA : printf("F"); break;
  case SOL : printf("G");break;
  default : printf(
    "%d non note", note);
}
```

Ces instructions lisent une valeur entière (représentant une `Note`) sur l'entrée standard et l'affichent (en notation internationale).

Remarque : une variable d'un type énuméré peut prendre comme valeur n'importe quel entier. Ceci explique la présence de la clause `default`.

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

```
Note note;

scanf(" %d", &note);

switch (note) {
  case LA : printf("A"); break;
  case SI : printf("B"); break;
  case DO : printf("C"); break;
  case RE : printf("D"); break;
  case MI : printf("E"); break;
  case FA : printf("F"); break;
  case SOL : printf("G");break;
  default : printf(
    "%d non note", note);
}
```

Ces instructions lisent une valeur entière (représentant une `Note`) sur l'entrée standard et l'affichent (en notation internationale).

Remarque : une variable d'un type énuméré peut prendre comme valeur n'importe quel entier. Ceci explique la présence de la clause `default`.

L'intérêt de l'utilisation des types énumérés est principalement **sémantique** : un programme qui les utilise est plus facile à lire et à maintenir.

Opérations sur les types énumérés

Contrairement aux types structurés, il est possible de comparer les éléments d'un type énuméré au moyen des **opérateurs relationnels**.

Ceci est une conséquence du fait que les énumérateurs sont des entiers.

```
printf("%d\n", SOL == SOL);     affiche 1
printf("%d\n", SOL == FA);      affiche 0
printf("%d\n", SI <= RE);       affiche 1
```

Opérations sur les types énumérés

Contrairement aux types structurés, il est possible de comparer les éléments d'un type énuméré au moyen des **opérateurs relationnels**.

Ceci est une conséquence du fait que les énumérateurs sont des entiers.

```
printf("%d\n", SOL == SOL);     affiche 1
printf("%d\n", SOL == FA);      affiche 0
printf("%d\n", SI <= RE);       affiche 1
```

De même, l'**opérateur d'affectation** = est compatible avec les types énumérés.

Opérations sur les types énumérés

Contrairement aux types structurés, il est possible de comparer les éléments d'un type énuméré au moyen des **opérateurs relationnels**.

Ceci est une conséquence du fait que les énumérateurs sont des entiers.

```
printf("%d\n", SOL == SOL);     affiche 1
printf("%d\n", SOL == FA);      affiche 0
printf("%d\n", SI <= RE);       affiche 1
```

De même, l'**opérateur d'affectation** `=` est compatible avec les types énumérés.

L'**opérateur de taille** `sizeof` renvoie `4` sur les valeurs d'un type énuméré. C'est la taille occupée par le type `int`.

Axe 3 : utiliser quelques techniques avancées

Opérateurs

Pointeurs de fonction

Génération aléatoire

Mémoïsation

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. 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 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 de la mémoire

Op.	Rôle	Ari.	Assoc.	Opérandes

Opérateurs de gestion de la mémoire

Op.	Rôle	Ari.	Assoc.	Opérandes
&	référencement	1	–	une variable

Opérateurs de gestion de la mémoire

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

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

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

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

Plan

Opérateurs arithmétiques

Op.	Rôle	Ari.	Assoc.	Opérandes

Opérateurs arithmétiques

Op.	Rôle	Ari.	Assoc.	Opérandes
`+`, `-`, `*`, `/`	opérations arith.	2	$\longrightarrow$	deux val. numériques

Opérateurs arithmétiques

Op.	Rôle	Ari.	Assoc.	Opérandes
`+`, `-`, `*`, `/`	opérations arith.	2	$\longrightarrow$	deux val. numériques
`%`	modulo	2	$\longrightarrow$	deux entiers

Opérateurs arithmétiques

Op.	Rôle	Ari.	Assoc.	Opérandes
+, -, *, /	opérations arith.	2	$\longrightarrow$	deux val. numériques
%	modulo	2	$\longrightarrow$	deux entiers
+, -	signe	1	–	une val. numérique

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

Opérateurs relationnels

Op.	Rôle	Ari.	Assoc.	Opérandes
$<$, $>$	comparaison stricte	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

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

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 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

Op.	Rôle	Ari.	Assoc.	Opérandes

Opérateurs logiques

Op.	Rôle	Ari.	Assoc.	Opérandes
&&	et logique	2	$\longrightarrow$	deux val. numériques

Opérateurs logiques

Op.	Rôle	Ari.	Assoc.	Opérandes
&&	et logique	2	$\longrightarrow$	deux val. numériques
||	ou logique	2	$\longrightarrow$	deux val. numériques

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

Opérateurs bit à bit

Op.	Rôle	Ari.	Assoc.	Opérandes
&	et bit à bit	2	$\longrightarrow$	deux val. entières

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

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

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
~	non bit à bit	1	–	une val. entière

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 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;
```

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 alias `Mot64` défini 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 alias `Mot64` défini 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 alias `Mot64` défini 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 alias `Mot64` défini 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;
}
```

Exemple — Compter le nombre de bits à un

2e **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

2e **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 ^ (x & -x);
```

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

Exemple — Compter le nombre de bits à un

2e **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 ^ (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

2e **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 ^ (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

2e **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 ^ (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

2e **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 ^ (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;
}
```