Opérations bit à bit et techniques de masquage

C.Defrance
version v3.1

Table des matières

Présentation de l'activité

Objectifs

- Se familiariser avec l'utilisation des opérateurs bit à bit du langage C
- S'approprier la technique du masquage de bits
- Manipuler des nombres dans différentes bases (2, 16)

Durée

4h

Compte rendu

- Faire les exercices n°1 à 8 sur le cahier de TP.
- Rendre le code source pour les exercices n°9 et 10.

Pré-requis

- Notions sur le codage binaire et hexadécimal.
- Langage C : instructions de boucles et de test, fonctions

Ressources

Matériels :

- PC Windows 7

Logiciels :

- Calculatrice Windows
- Compilateur C (mingw)

Documents :

Aucun

Travail demandé

1. Introduction

Le langage C est un langage relativement «bas niveau» c'est-à-dire proche de la machine.

À travers les fonctionnalités standards du langage — dont les opérateurs bit à bit mais aussi les pointeurs — il est ainsi possible, dans certains cas, d'accéder directement au matériel.

Ces fonctionnalités font que ce langage est très utilisé en **informatique industrielle**.

Comme leur nom l'indique, les opérateurs bit à bit ou ***bitwise*** en anglais vont permettre de lire ou de modifier certains bits d'une valeur stockée dans une variable.

Jusqu'à présent, vous avez appris qu'une variable était un emplacement mémoire dans lequel était stocké une valeur — codée en binaire — qui représente une information en relation avec le problème à traiter par un programme informatique. Ex. : une température, une adresse mail, le résultat d'un calcul...

Vous avez aussi appris que, selon le type de la variable (nombre entier ou décimal, caractère...), l'emplacement mémoire qui lui était attribué pouvait s'étendre sur 1 ou plusieurs **octets** constitués chacun de 8 **bits** (contraction de *Binary digIT*).

Or, on peut rencontrer des situations dans lesquelles on désire mémoriser des informations pouvant être codées sur un nombre restreint de bits (<8).

Dès lors, pourquoi «gâcher» un octet de mémoire pour mémoriser seulement 1 ou 2 bits d'information ?

Bien sûr, les ordinateurs récents disposent d'énormément de mémoire et le fait de «gâcher» quelques octets n'est pas génant. Cependant, cette mémoire a un coût et occasionne une consommation de courant. Il est donc fréquent de trouver des systèmes informatiques — même modernes — dans lesquels les ressources mémoire sont très limitées.

Dans ces systèmes, on va alors optimiser l'utilisation de la mémoire en stockant dans un seul octet plusieurs informations codées sur quelques bits.

Toutefois, le langage C standard ne propose pas d'accès direct aux bits d'un octet [1]. Dans ces conditions, il va falloir «jongler» avec les opérateurs bit à bit mis à notre disposition pour pouvoir lire ou modifier ces bits.

C'est ce que nous allons étudier dans ce cours.

Si l'optimisation de la mémoire et donc le regroupement de plusieurs informations dans un seul octet est parfois superflu, l'accès au niveau bit est par contre obligatoire lorsqu'on désire accéder à certaines zones mémoire spéciales, appelées **registres**. situées dans le **microprocesseur** ou certains de ses circuits périphériques.

L'écriture d'un ou plusieurs bits dans ces **registres** provoque souvent une action au niveau du système informatique contrairement à l'écriture dans une mémoire conventionnelle dont le rôle se limite exclusivement à la mémorisation de l'information.

2. Les opérateurs bits à bit

Le langage C propose 6 opérateurs qui travaillent au niveau du bit :

1. le NON bit à bit noté **~**
2. le ET bit à bit noté **&**
3. le OU bit à bit noté **|**
4. le OU EXCLUSIF bit à bit noté **^**
5. le DÉCALAGE À GAUCHE noté **<<**
6. le DÉCALAGE À DROITE noté **>>**

Notez que les 4 premiers **opérateurs bit à bit** portent le nom de **fonctions logiques booléennes**. Bien qu'ils présentent des similitudes, leur rôle est **totalement différent** : agir sur 1 bit (0 ou 1) n'a rien à voir avec le fait d'évaluer une expression booléenne (VRAI ou FAUX).

Souvenez-vous en !!

La confusion entre les 2 types d'opérateur est d'autant plus facile que leur notation est très proche en langage C.

Dans la suite du document, des exemples d'opérations bit à bit vont être présentés.

Ces opérations vont agir sur des nombres binaires.

Or, le langage C n'accepte que les nombres saisis en décimal (base 10), hexadécimal (base 16) ou octal (base 8). Je suis d'accord avec vous, c'est un peu «ballot» !

Ainsi, un nombre précédé de **0x** (chiffre `0` suivie de la lettre `x`) sera considéré comme de l'hexadécimal. Ex. : `0x1E`.

De même, un nombre précédé de **0** (zéro) sera considéré comme de l'octal. Ex. : `07`.

Un nombre en base 10 s'exprime quant à lui sans préfixe. Ex. : `96`.

On veillera dans le cas d'un nombre en base 10 à ne pas mettre de 0 non significatifs à sa gauche sous peine

qu'il soit interprété comme de l'octal.

Ex. : `017` en base 8 donne `15` en base 10 (`017` → $1 \times 8^1 + 7 \times 8^0$ → $8 + 7$ → `15`).

La notation choisie dans le reste du document pour désigner un nombre exprimé en binaire (base 2) consiste à le préfixer par **`0b`**.

Utiliser cette notation dans une instruction en langage C provoquera une erreur de compilation puisque, je vous le rappelle, le compilateur C ne comprend pas la notation binaire dans un code source.

Vous utiliserez donc majoritairement l'hexadécimal pour représenter les nombres exprimés au départ en binaire.

La méthode consiste à :

1. constituer des groupes de 4 bits dans la valeur binaire de départ
2. coder en hexadécimal chacun des groupes de 4 bits avec les correspondances suivantes :

binaire	hexadécimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

3. préfixer le nombre obtenu par **`0x`**

 **Exercice n° 1 :**

1. Exprimer en hexadécimal puis en décimal les nombres binaires suivants :
   `0b00000100, 0b00001010, 0b00010101, 0b00100000, 0b00111100, 0b10110110`
2. Exprimer en binaire puis en hexadécimal les nombres décimaux suivants :
   5, 10, 13, 15, 16, 48, 127, 166, 255
3. Exprimer en binaire et en décimal les nombres hexadécimaux suivants :
   0x10, 0x0F, 0x1F, 0xA4, 0x4A

Vous vérifierez vos résultats avec la calculatrice de Windows configurée avec un affichage de type «Programmeur».

2.1. Le NON bit à bit

Cet opérateur inverse les bits de son opérande unique : les '0' sont transformés en '1' et les '1' en '0'

En langage C, il est noté avec un 'tilde' : **`~`**

b	~b
0	1
1	0

On l'appelle aussi le **complément à 1**.

Exemple :

```
// définition d'un octet initialisé avec la valeur binaire 0b10101111
unsigned char octet = 0xAF;
// définition d'une variable qui contient le complément à 1 de 'octet'
unsigned char resultat = ~octet;

// => 'resultat' contient la valeur binaire 0b01010000 (0x50 en hexadécimal)
// ~ 10101111
// ----------
//   01010000
```

Lorsqu'un opérateur n'admet qu'un seul opérande — comme pour l'opérateur **~** — il est qualifié d'opérateur **Unaire**.

Lorsqu'il en admet 3, c'est un opérateur **TERnaire**. Il n'en existe qu'un de ce type en langage C (`a ? b : c`).

Lorsqu'un opérateur admet 2 opérandes, comme dans `a + b`, c'est un opérateur ... **BInaire**. Remarquez toutefois que dans ce contexte, «binaire» ne signifie pas que l'opérateur agit directement au niveau des bits (*bitwise*) mais simplement qu'il est «BI-opérandes».

 **Exercice n° 2 :**

1. Exprimer en binaire et en hexadécimal le **complément à 1** des valeurs suivantes :
   a. `0b00001111`
   b. `0b00111100`
   c. `0x5A`
   d. `0xF0`
   e. `0x41`

2.2. Le ET bit à bit

Cet opérateur effectue un ET logique entre les bits de même rang de ses 2 opérandes c'est-à-dire entre le bit 0 du 1^{ier} opérande et le bit 0 du 2^{ième} opérande PUIS entre le bit 1 du 1^{ier} opérande et le bit 1 du 2^{ième} opérande etc...

En langage C, il est noté avec **1 seul** «et commercial» : **&**

b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

Exemple :

```
unsigned char octet1 = 0xA5;   // 0b10100101
unsigned char octet2 = 0xF0;   // 0b11110000

unsigned char resultat = octet1 & octet2;

// => 'resultat' contient la valeur binaire 0b10100000 (0xA0 en hexadécimal)
//   10100101
// & 11110000
// ----------
//   10100000
```

Une même variable peut constituer le résultat et l'opérande d'un opérateur bit à bit à 2 opérandes. C'est par exemple le cas lorsqu'on désire la mettre à jour à partir de sa valeur précédente. Dans ce cas, il existe une forme abrégée pour noter l'opération.

Ainsi, pour l'opérateur ET binaire, les 2 écritures suivantes sont équivalentes :

```
resultat = resultat & 0xF0;
resultat &= 0xF0;  // Écriture abrégée
```

Exercice n° 3 :

1. Poser les opérations suivantes et exprimer leurs résultats en binaire et en hexadécimal :
   a. 0xBB & 0x44
   b. 0x12 & 0x34
   c. 0xAA & 0xFF
   d. 0x5C & 0x0F
   e. 0x61 & 0xDF
   f. 0x61 & ~0x20
   g. 0xC5 & ~0x0F

2.3. Le OU bit à bit

Cet opérateur effectue un OU logique entre les bits de même rang de ses 2 opérandes.
En langage C, il est noté avec **1 seule** barre verticale : **|**

b1	b2	b1 | b2
0	0	0
0	1	1
1	0	1
1	1	1

Exemple :

```
unsigned char octet1 = 0xA5;    // 0b10100101
unsigned char octet2 = 0xF0;    // 0b11110000

unsigned char resultat = octet1 | octet2;

// => 'resultat' contient la valeur binaire 0b11110101 (0xF5 en hexadécimal)
//   10100101
// | 11110000
// ----------
//   11110101
```

Exercice n° 4 :

1. Poser les opérations suivantes en binaire et exprimer leurs résultats en binaire et en hexadécimal :
   a. 0xBB | 0x44
   b. 0x12 | 0x34
   c. 0xAA | 0xFF
   d. 0x5C | 0x0F
   e. 0x41 | 0x20
   f. 0x61 | ~0x20
   g. 0xC5 | ~0x0F
   h. 0x29 | (~0x29 & 0x0F)

2.4. Le OU EXCLUSIF bit à bit

Cet opérateur effectue un OU EXCLUSIF logique entre les bits de même rang de ses 2 opérandes.
En langage C, il est noté avec l'accent circonflexe : **^**

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

Exemple :

```
unsigned char octet1 = 0xA5;   // 0b10100101
unsigned char octet2 = 0xF0;   // 0b11110000

unsigned char resultat = octet1 ^ octet2;

// => 'resultat' contient la valeur binaire 0b01010101 (0x55 en hexadécimal)
//   10100101
// ^ 11110000
// ----------
//   01010101
```

Exercice n° 5 :

1. Poser les opérations suivantes et exprimer leurs résultats en binaire et en hexadécimal :
   a. 0xBB ^ 0x44
   b. 0x12 ^ 0x34
   c. 0xAA ^ 0xFF
   d. 0x55 ^ 0xFF
   e. 0x41 ^ 0x20
   f. 0x61 ^ ~0x20
   g. 0xC5 ^ ~0x0F
   h. 0x29 ^ (~0x29 & 0x0F)

2.5. le DÉCALAGE À GAUCHE

Cet opérateur permet de faire glisser vers la gauche tous les bits d'une valeur binaire. Le 1^{ier} opérande spécifie la valeur sur laquelle le décalage doit être effectué et le 2^{ième} opérande indique son amplitude.

En langage C, il est noté avec un double chevrons orienté vers la gauche : `<<`

Le «vide» de droite occasionné par le déplacement est comblé de `0`. Les bits de gauche qui «sortent» de la valeur sont, quant à eux, perdus.

Exemple :

```
unsigned char octet1 = 0xA5;   // 0b10100101

unsigned char resultat = octet1 << 3;

// => 'resultat' contient la valeur binaire 0b00101000 (0x28 en hexadécimal)
//
//   10100101 << 3 provoque :
//         <- [10100101] <- 000
//       1 <- [01001010] <- 00
//      10 <- [10010100] <- 0
//     101 <- [00101000] <-
//     \ /           +-+--> 3 bits 0 insérés pour combler le vide
//      v
//3 bits perdus
```

Noter que dans le code précédent, la variable `octet1` n'est pas modifiée suite au décalage.
Pour la modifier, il faudrait écrire : `octet1 = octet1 << 3`.

Exercice n° 6 :

1. Déterminer le résultat des opérations suivantes en se basant sur le fait que les valeurs sont toutes codées sur un seul octet. Les résultats seront exprimés dans la base utilisée pour spécifier la valeur à décaler.
   a. 1 << 0
   b. 1 << 1
   c. 1 << 2
   d. 0x11 << 3
   e. 0xF0 << 4
   f. 0x78 << 8
   g. 0x41 | (0x01 << 5)
   h. 0x81 & ~(0x01 << 5)
2. À quelle opération arithmétique peut-on assimiler un décalage à gauche d'un bit ?
3. Est-ce que cette opération est juste quelque soit la valeur à décaler ?

4. Comment pourrait-on résumer l'action de l'opération `octet & ~(1 << n)` avec `n` compris entre 0 et 7 ?

2.6. le DÉCALAGE À DROITE

Cet opérateur permet de faire glisser vers la droite tous les bits d'une valeur binaire. Le 1ier opérande spécifie la valeur sur laquelle le décalage doit être effectué et le 2ième opérande indique son amplitude.

En langage C, il est noté avec un double chevrons orienté vers la droite : `>>`

Le «vide» de gauche occasionné par le déplacement est comblé **soit par des 0 soit par des 1** selon le signe (+/-) de la valeur de départ. Les bits de droite qui «sortent» de la valeur sont, quant à eux, perdus.

Lorsque la valeur à décaler est **signée** — c'est-à-dire qu'elle peut représenter une valeur positive ou négative — et que cette valeur est **positive** (bit de poids fort à `0`), le vide de gauche est **comblé par des `0`**.

Quand la valeur à décaler est **signée et négative** (bit de pois fort à `1`), le vide de gauche est **comblé par des `1`**. La valeur du signe est donc conservée.

Quand la valeur à décaler n'est **pas signée** (spécificateur `unsigned` devant le type d'une variable en langage C), le vide de gauche est **toujours comblé par des 0**.

```
unsigned char octet1;
signed char octet2;  // ('signed' est implicite pour un 'char'
                     // sauf mention contraire dans les options
                     // du compilateur)

/* Décalage à droite d'une valeur non signée */
octet1 = 0xA5;  // 0b10100101
octet1 >>= 3;   // rappel : équivalent à 'octet1 = octet1 >> 3'

// => 'octet1' contient la valeur binaire 0b00010100 (0x14 en hexadécimal)
//
//   10100101 >> 3 provoque :
//   000 -> [10100101] ->
//    00 -> [01010010] -> 1
//     0 -> [00101001] -> 01
//       -> [00010100] -> 101
//           \ /          +-+--> 3 bits perdus
//            v
//     3 bits 0 insérés

/* Décalage à droite d'une valeur signée positive (bit de poids fort à 0) */
octet2 = 0x5A;   // 0b01011010
octet2 >>= 3;

// => 'octet2' contient la valeur binaire 0b00001011 (0x0B en hexadécimal)
//
//   01011010 >> 3  provoque :
//   000 -> [01011010] ->
//    00 -> [00101101] -> 0
//     0 -> [00010110] -> 10
//       -> [00001011] -> 010
//           \ /          +-+--> 3 bits perdus
//            v
//     3 bits 0 insérés

/* Décalage à droite d'une valeur signée négativee (bit de poids fort à 1) */
octet2 = 0xA5;   // 0b10100101
                 // ici 'octet2' représente une valeur négative (-91 en décimal)
                 // car le bit de poids fort est à 1.
octet2 >>= 3;

// => 'octet2' contient la valeur binaire 0b1111100 (0xF4 en hexadécimal)
//
//   10100101 >> 3  provoque :
//   111 -> [10100101] ->
//    11 -> [11010010] -> 1
//     1 -> [11101001] -> 01
//       -> [11110100] -> 101
//           \ /          +-+--> 3 bits perdus
//            v
//     3 bits 1 insérés
```

Exercice n° 7 :

1. Déterminer le résultat des opérations suivantes en se basant sur le fait que les valeurs sont toutes codées sur un seul octet **signé**. Les résultats seront exprimés dans la base utilisée pour spécifier la valeur à décaler.
   a. 0x60 >> 2
   b. 0x78 >> 3
   c. 0xF0 >> 4

d. 1 >> 0

e. 1 >> 1

f. 127 >> 2

g. -100 >> 3

2. À quelle opération arithmétique peut-on assimiler un décalage à droite d'un bit ?
3. Est-ce que cette opération est juste quelque soit la valeur à décaler ?
4. À quelle opération correspond `octet >> 3` pour des valeurs de `octet` multiples de 8 et comprises entre -128 et +120 ?

   Que peut-on dire des résultats pour des valeurs d' `octet` positives et négatives non multiples de 8 (`-128 < octet <= 127`) ?

---

3. Les masques

Le **masquage de bits** est une technique qui va permettre d'isoler un ou plusieurs bits dans une valeur binaire,

Ceux-ci peuvent être vus comme des pochoirs qui permettent de peindre une zone précise sans risque de débordement. Selon le point de vue, ces «pochoirs» peuvent aussi être considérés comme un moyen de protéger une zone.

Les masques de bits seront ainsi utilisés à chaque fois que l'on désire intervenir sur une seule portion d'une valeur binaire.

Les domaines d'utilisation des masques de bit sont nombreux :

- le forçage à 1 ou à 0 d'un ou plusieurs bits
- le test de la valeur d'un ou plusieurs bits
- l'inversion de la valeurs d'un ou plusieurs bits
- la détermination du nombre de bits à 0 ou à 1 d'une valeur binaire
- l'inversion de l'ordre des bits d'une valeur binaire
- ...

Nous allons décrire ci-après les utilisations les plus courantes.

Il s'avère que le masquage de bits est souvent mal maitrisé par les étudiants.

Cette technique n'est pourtant pas si compliquée que cela si on se donne la peine de la pratiquer un petit peu,

De plus, il faut savoir qu'elle est très souvent utilisée en informatique :

- programmation bas niveau → accès aux registres d'un microprosseur, accès aux broches d'entrée/sortie du système
- réseau → masques de sous-réseau
- imagerie → retouche photo, traitement d'images
- jeux vidéo → *sprites*, *bit blit*

3.1. Forçage de bits à 0 ou à 1

3.1.1. Forçage d'un seul bit

Forçage à 1

On utilise le OU binaire et le DÉCALAGE À GAUCHE :

valeur = valeur | (1 << numbit)

La couleur rouge ainsi que la taille des caractères dans cette formule ainsi que dans celles à venir devraient vous faire prendre conscience qu'elles constituent quelque chose de très important à savoir par cœur de chez par cœur ;-).

Quant aux formules en bleu, elle constituent un «plus».

Ex. :

```
// Forçage à 1 du bit de rang 6 d'un octet (le 7ième bit en partant de la droite)
unsigned char octet = 0xA5; // 0b10100101

octet = octet | (1 << 6);

// 'octet' contient 0b11100101 suite aux 2 étapes :
// 1/ (1 << 6) -> 0b00000001 << 6 -> 0b01000000
// 2/   10100101
//    | 01000000
//    ----------
//      11100101
```

```
//       +-------> seul le 7ième bit a été impacté par le forçage à 1
```

Forçage à 0

On utilise le ET binaire, le DÉCALAGE À GAUCHE, le NON binaire:

$$valeur = valeur \ \& \sim(1 << rang\_bit)$$

Ex. :

```
// Forçage à 0 du bit de rang 5 d'un octet (le 6ième bit)
unsigned char octet = 0xA5; // 0b10100101

octet = octet & ~(1 << 5);

// 'octet' contient 0b11100101 suite aux 3 étapes :
// 1/ (1 << 5) -> 0b00000001 << 6 -> 0b00100000
// 2/ ~(1 << 5) -> 0b11011111
// 3/   10100101
//    & 11011111
//    ----------
//      10000101
//        +-------> seul le 6ième bit a été impacté par le forçage à 0
```

3.1.2. Forçage de plusieurs bits consécutifs

On utilise la même technique que précédemment mais avec un masque plus complexe.

Comme mentionné dans le titre, la technique présentée ci-dessous fonctionne UNIQUEMENT si les bits à forcer sont consécutifs.

Forçage à 1

$$valeur = valeur \mid (((1 << nb\_bit) - 1) << rang\_bit)$$

avec :

- `nb_bit` le nombre de bits à forcer
- `rang_bit` la position (0...n) dans l'octet, en partant de la droite, à partir de laquelle on désire réaliser le forçage.

Forçage à 0

$$valeur = valeur \ \& \sim(((1 << nb\_bit)-1) << rang\_bit)$$

3.2. Inversion de la valeur de bits

On utilise la même technique que le forçage à 1 mais en utilisant cette fois-ci le OU EXCLUSIF binaire.

Inversion d'un seul bit

$$valeur = valeur \ \hat{} \ (1 << numbit)$$

Inversion de plusieurs bits

$$valeur = valeur \ \hat{} \ (((1 << nb\_bit) - 1) << rang\_bit)$$

Rappel

Si on souhaite inverser tous les bits d'un valeur binaire, il suffit d'utiliser :

```
valeur = ~valeur
```

Exercice n° 8 :

Donner 3 exemples pour chacune des «formules» présentées dans cette partie du TP :

- Forçage d'un seul bit à 1 et à 0
- Forçage à 1 et à 0 de plusieurs bits
- Inversion de 1 et plusieurs bits

(donc 6*3=18 exemples)

3.3. Test de la valeur d'un bit

Il suffit d'isoler la valeur du bit dans la condition d'une instruction de test du langage. En langage C :

- `if(condition) ... else ...`
- `do ... while(condition);`
- `while(condition) ...`
- `for( ...; condition; ...)`
- `(condition) ? ... : ...;`

En effet, il faut savoir qu'une condition est considérée comme fausse si elle vaut 0 et vrai dans les autres cas (i.e différente de 0).

Exemple :

```
if( (valeur & (1 << rang_bit)) != 0 ) {
   // bit de rang 'rang_bit' est à 1
} else {
   // bit de rang 'rang_bit' est à 0
}
```

 **Exercice n° 9 :**

 Cet exercice est plus long que les autres !!

On se propose de simuler un chenillard à l'aide d'un programme.

Le chenillard sera modélisé par un octet dans lequel un bit à 1 représentera une lampe allumée.

Cet octet sera affiché à l'écran de façon à ce qu'un bit à 0 soit représenté par un point (`.`) et un bit à 1 par une étoile (`*`).

Ex. : `.......*`

1. Coder une fonction `void affiche(unsigned char octet)` qui affiche les 8 caractères images des 8 bits du paramètre `octet` (`.` = 0, `*` = 1) suivis d'un retour à la ligne.

   Info : L'affichage d'un seul caractère à l'écran peut être obtenu avec `putc('A', stdout);` après inclusion de l'entête `<stdio.h>`.
2. Tester la fonction `affiche()` avec plusieurs valeurs de paramètre.
3. Coder une fonction 'void chenillard()` qui simule l'allumage `tournant' vers la gauche d'une seule lampe.

   Ex. :

```
.......*
......*.
.....*..
....*...
...*....
..*.....
.*......
*.......
.......*
......*.
etc
```

Info : Une pause entre les différents affichages peut être obtenue à l'aide de la fonction `Sleep(delai_en_millisecondes)` sous Windows après inclusion de l'entête `<windows.h>`.

1. Tester votre fonction `chenillard()`
2. Faire évoluer la fonction `chenillard()` de manière à ce qu'elle prenne en paramètre un nombre de lampes voisines à allumer simultanément.

Info : un exemple de ce qui est attendu vous est fourni dans cet exécutable.

3.4. Lire/Ecrire une valeur sur plusieurs bits depuis/dans un fragment d'une variable entière.

Dans ce qui précède, les opérations portaient sur un ou plusieurs bits qui représentaient chacun une information booléenne (1 information = 1 bit).

Cependant, on rencontre des cas où on doit lire/écrire plusieurs informations codées sur plus d'1 bit dans une seule variable.

Ex. : stocker dans un seul octet les couleurs de premier plan (*foreground*) et d'arrière plan (*background*) d'un texte.

Dans ces situations, les techniques précédentes (forçage, inversion, test d'1 bit) ne sont plus adaptées car l'ensemble des bits qui représente l'information doit être considéré comme un tout et non comme une séquence de bits ayant chacun une signification.

Les opérations de lecture/écriture sur ce type d'information vont alors devoir se faire en plusieurs étapes.

3.4.1. Lecture d'une information de plusieurs bits

3 étapes sont nécessaires :

1. Lire l'intégralité de la variable dans laquelle l'information à lire est stockée
2. Mettre à 0 les bits ne faisant pas partie de l'information à lire
3. Faire glisser totalement à droite l'ensemble des bits de l'information.

Exemple : reprenons le cas de l'octet de couleurs évoqué plus haut :

- les 4 bits de poids fort représentent une couleur de 1^{ier} plan
- les 4 bits de poids faible codent une couleur d'arrière plan.

Admettons que la valeur de cet octet code l'information «magenta sur fond jaune» et que la palette des couleurs disponibles soit codée ainsi :

0 → Noir, 1 → Rouge, 2 → Vert, 3 → Jaune, 4 → Bleu, 5 → Magenta, 6 → Cyan, 7 → Blanc

On aura donc : `color = 0x53;`

La lecture de la couleur de 1^{ier} plan se codera en langage C de la manière suivante :

```
unsigned char color = 0x53;

unsigned char foreground;

// Étape n°1 : Lecture de l'intégralité des couleurs (1ier plan et arrière plan)
foreground = color;

// Étape n°2 : Forçage à 0 des 4 bits associés au codage de l'arrière plan
foreground &= 0xF0;

// Étape n°3 : Décalage à droite de la couleur de 1ier plan
foreground >>= 4;

// => 'foreground' contient la valeur 5 qui code effectivement la couleur magenta
```

1. Le code précédent peut également s'écrire en une seule ligne :

   ```
   foreground = (color & 0xF0) >> 4;
   ```

2. Il faut veiller à ce que les variables utilisées soit non signées.

   En effet, si une information utilise le bits de poids fort (*MSB : Most Significant Bit*) de la variable englobante , le décalage à droite d'une telle valeur avec un *MSB* à 1 depuis une variable signée (char, int, long) provoquera un bourrage à gauche avec des 1. La valeur finale sera ainsi faussée.

   Exemple : admettons que les 2 bits de poids fort d'une variable code les 4 valeurs possibles que peut prendre un style de texte : 0 → normal, 1 → gras, 2 → souligné, 3 → italique.

   Admettons que cette variable ait la valeur 0b10xxxxxx (texte gras).

   La lecture du style dans une variable signée donnera, suite au décalage de 6 bits vers la droite, la valeur 0b11111110 c'est-à-dire 0xFE soit -2 en décimal et non +2 comme attendu.

3. Dans l'exemple précédent, l'étape n°2 n'est pas obligatoire puisque le codage de la couleur de 1^{ier} plan occupe l'ensemble des bits de poids fort de l'octet. Cependant, dans un premier temps, je vous conseille de toujours réaliser les 3 étapes.

3.4.2. Écriture d'une information sur plusieurs bits

3 étapes sont encore nécessaires :

1. Décaler la valeur de l'information à écrire de façon à ce que ses bits soient correctement positionnés vis à vis de la variable englobante.
2. Mettre à 0 l'ensemble des bits de l'information initiale dans la variable englobante
3. Écrire la nouvelle information dans la variable englobante avec un simple OU bit à bit de façon à ne pas impacter les bits environnants.

La mise à 0 de l'information initiale de l'étape n°2 est indispensable car le OU binaire final peut forcer à 1 les bits d'origine quelque soient leurs valeurs (0 ou 1). Par contre, il ne peut pas les forcer à 0 s'ils étaient auparavant à 1.

Exemple : Modifions la couleur de texte de l'exemple précédent pour qu'elle devienne «cyan sur fond jaune» (bon, d'accord, c'est pas très lisible mais c'est pour l'exemple...)

```
unsigned char color = 0x53; // couleur initiale : magenta sur fond jaune

unsigned char foreground = 6; // code de la nouvelle couleur de 1ier plan : cyan
```

```
// Étape n°1 : Décalage de la couleur de 1ier plan au niveau des 4 bits de poids fort
// de la variable englobante
foreground <<= 4;

// Étape n°2 : Forçage à 0 des 4 bits de poids fort associés à la couleur de
// 1ier plan dans la variable englobante.
color &= 0x0F;  // ou color &= ~(((1 << 4)-1) << 4)

// Étape n°3 : stockage de l'information
color |= foreground;

// => 'color' contient la valeur 0x63 qui code effectivement cyan sur fond jaune.
```

Exercice n° 10 :

On vous demande de coder un programme qui permet de gérer la couleur et le style d'un texte à l'aide d'un seul octet constitué comme suit :

b7	b6	b5	b4	b3	b2	b1	b0
style		**couleur 1^{ier} plan**			**couleur arrière plan**		

Le codage du style et des couleurs correspond à celui utilisé dans les exemples précédents :

- style : 0 → normal, 1 → gras, 2 → souligné, 3 → italique.
- couleur : 0 → Noir, 1 → Rouge, 2 → Vert, 3 → Jaune, 4 → Bleu, 5 → Magenta, 6 → Cyan, 7 → Blanc

1. Déclarer une variable globale : `static unsigned char attributsTexte;`.
   Celle-ci contiendra les attributs du texte tels que décrits ci-dessus.

   1. Une variable globale est une variable déclarée en dehors de toute fonction (fonction `main()` y compris)
   2. Une variable globale est directement accessible par l'ensemble des fonctions
   3. Le spécificateur `static` pour une variable globale indique que cette variable ne sera visible que par les fonctions codées dans le même fichier où elle est déclarée.

2. Coder une fonction `int getStyle()` qui retourne le style codé dans `attributsTexte`.
3. Coder une fonction 'int getColor(char ForB)` qui retourne le code de la couleur du 1$^{ier}$ plan ou de l'arrière plan suivant la valeur du paramètre (`F' pour le 1^{ier} plan et 'B' pour l'arrière plan).
4. Coder le pendant des fonctions précédentes à savoir celles qui permettent de modifier le style et les couleurs du texte :
   a. `void setStyle(int styleCode)`
   b. `void setColor(char ForB, colorCode)`

---

4. Conclusion

Vous avez appris dans ce cours comment agir directement sur le contenu de valeurs binaires à l'aide des opérateurs bit à bit.

Vous avez également découvert ou redécouvert la technique du masquage de bits pour isoler les bits sur lesquels on souhaite intervenir.

La maîtrise de ces concepts implique une aisance dans les conversions binaire, hexadécimale et décimale.

Cette habileté à manipuler les nombres dans différents bases est un savoir-faire **indispensable** pour un programmeur et tout particulièrement pour un programmeur en informatique industrielle.

Entrainez-vous et noubliez pas le vieil adage suivant :

> Il existe 10 types de personnes, ceux qui comprennent le binaire et ... les autres ! ;-)
>
> — anonyme

1. En fait, le langage C propose bien une fonctionnalité — nommée **champ de bits (*bitfield*)** qui permet d'accéder à des fragments d'un octet mais celle-ci dépasse le cadre de ce cours et doit être utilisée avec précaution car non portable d'une plateforme à l'autre

Version v3.1
Dernière mise à jour 2013-10-07 17:20:38 Paris, Madrid (heure d'été)