Les tableaux de bits

Vous avez déjà rencontré la classe `std::bitset` dans le chapitre [Logique binaire et calcul booléen](#), pour afficher une séquence de bits.

main.cpp

```
#include <iostream>
#include <bitset>

int main() {
    const std::bitset<8> b1 { 0b101010 };
    std::cout << "0b" << b1 << std::endl;
    const std::bitset<8> b2 { 42 };
    std::cout << "0b" << b2 << std::endl;
}
```

affiche :

```
0b00101010
0b00101010
```

Ce chapitre détaille l'utilisation de cette classe `std::bitset` et les notions de *flag* et *mask*. Les notions vues dans les chapitres [Logique binaire et calcul booléen](#) et [\[Aller plus loin\] L'algèbre de Boole](#) seront utilisées, n'hésitez pas à les relire si nécessaire.

Créer un `std::bitset`

La classe `std::bitset` est une classe template prenant en argument le nombre de bits. La taille d'un `std::bitset` est donc déterminée à la compilation (Pour rappel, vous avez déjà rencontré des classes qui ont des tailles déterminées à la compilation ou l'exécution : respectivement `std::array` et `std::vector`).

Le nombre de bits gérés par `std::bitset` est un argument template

(donc qui s'écrit entre `<>`) de type entier.

```
std::bitset<TAILLE>
```

Il existe différentes méthodes pour initialiser un `std::bitset` :

- avec une valeur entière ;
- avec une chaîne de caractères.

La méthode la plus simple pour initialiser un `std::bitset` est de lui fournir une valeur entière lors de l'initialisation, de préférence en utilisant une représentation binaire (avec le préfixe `0b` et les chiffres `0` et `1`) ou hexadécimale (avec le préfixe `0x` et les symboles `0` à `9` et `a` à `f`, en minuscule **ou** majuscule). Si aucune valeur n'est fournie, `std::bitset` est initialisé avec la valeur nulle.

main.cpp

```
#include <bitset>

int main() {
    const std::bitset<8> b;
    const std::bitset<8> b8 { 0b101010 };
    const std::bitset<16> b16 { 0xA1B2 };
}
```

Pour faciliter la création de `std::bitset` à partir d'une entrée utilisateur (flux standard, fichier, etc), il est également possible d'initialiser un `std::bitset` à partir d'une chaîne de caractères. La syntaxe peut être différente selon si vous utilisez une littérale chaîne ou un type `std::string`.

Le cas le plus simple est d'initialiser un `std::bitset` à partir d'une chaîne complète, constituée des caractères `0` et `1`. Dans ce cas, la syntaxe est identique pour une littérale et un `std::string` :

main.cpp

```
#include <bitset>
#include <string>

int main() {
```

```

const std::bitset<8> b1 { "101010" };

const std::string s { "101010" };
const std::bitset<8> b2 { s };
}

```

Notez bien qu'il ne faut pas ajouter de préfixe dans la chaîne de caractères.

Un `std::bitset` peut également être initialisé à partir d'une sous-chaîne de caractères (c'est à dire une partie d'une chaîne de caractères). La syntaxe est différente entre une littérale et un `std::string` :

- pour une littérale, l'initialisation de `std::bitset` ne peut prendre qu'un seul argument optionnel supplémentaire : le nombre de caractères à conserver ;
- pour un `std::string`, `std::bitset` peut prendre deux arguments optionnels supplémentaires : la position du premier caractère et le nombre de caractères à conserver.

Un exemple concret avec une littérale chaîne :

main.cpp

```

#include <bitset>
#include <iostream>

int main() {
    const std::bitset<8> b1 { "1010101011", 4}; // les 4
    premiers caracteres = "1010"
    std::cout << b1 << std::endl;

    const std::bitset<8> b2 { "1010101011", 8}; // les 8
    premiers caracteres = "10101010"
    std::cout << b2 << std::endl;
}

```

affiche :

```

00001010
10101010

```

Avec un `std::string` :

main.cpp

```
#include <bitset>
#include <string>
#include <iostream>

int main() {
    const std::string s { "1010101011" };

    const std::bitset<8> b1 { s, 4 }; // commence a
l'indice 4 = "101011"
    std::cout << b1 << std::endl;

    const std::bitset<8> b2 { s, 4, 2 }; // commence a
l'indice 4 et conserve // 2 caracteres =
"10"
    std::cout << b2 << std::endl;
}
```

affiche :

```
00101011
00000010
```

N'oubliez pas qu'en C++, les indices dans les tableaux (et donc dans les chaînes de caractères, puisqu'elles peuvent être considérées comme des tableaux de caractères) commencent à l'indice 0. Donc l'indice 4 correspond au cinquième caractère :

```
chaîne : 1 0 1 0 1 0 1 0 1 1
indice : 0 1 2 3 4 5 6 7 8 9
           ^
```

Pour terminer, il est possible d'utiliser d'autres caractères que `0` et `1`. Pour cela, il faut fournir deux arguments supplémentaires, correspondent respectivement aux caractères à utiliser à la place de `0` et de `1`.

main.cpp

```

#include <bitset>
#include <string>
#include <iostream>

int main() {
    const std::bitset<8> b1 { "BABBBABBA", 8, 'A', 'B' };
    std::cout << b1 << std::endl;

    const std::string s { "YXYXYXYX" };
    const std::bitset<8> b2 { s, 0, 8, 'X', 'Y' };
    std::cout << b2 << std::endl;
}

```

affiche :

```

10110110
10110110

```

Notez que les caracteres utilisees pour represente le `std::bitset` ne sont utilisees que pour l'initialisation. En memoire et lors de l'affichage, un `std::bitset` sera representes par defaut par une suite de 0 et 1.

Ordres des arguments et arguments optionnels

En C++, les arguments sont identifiés par leur position dans l'appel d'une fonction. Par exemple pour initialiser `std::bitset` (avec `position` qui represente la position du premier caractere et `taille` qui correspond au nombre de caracteres a conserver :

```

const size_t position { 2 };
const size_t taille { 4 };
const std::bitset<8> b { s, position, taille };

```

Cela implique qu'il n'est pas possible de changer l'ordre des arguments dans une fonction :

```

const std::bitset<8> b { s, taille, position };

```

Dans ce code, le compilateur ne va pas utiliser `taille` pour le nombre de caracteres et `position` pour la position du premier caractere (donc

prendre 4 caracteres a partir de la position 2), mais va prendre 2 caracteres a partir de la position 4.

Pour la meme raison, si on fournit un argument optionnel, les arguments optionnels qui le precedent ne sont plus optionnels.

```
const std::bitset<8> b { s, taille };
```

Dans ce code, le compilateur ne va pas utiliser l'argument `taille` fournit et considerer que l'argument `position` prend sa valeur par default 0 (donc prendre 4 caracteres a partir de la position 0), mais va utiliser `taille` comme position (donc prendre tous les caracteres apres la position 4).

Les arguments de fonction, en particulier l'ordre des arguments et les arguments optionnels seront vu en detail dans les chapitres sur la creation de fonctions.

Afficher et convertir en chaine de caracteres

Comme vous l'avez vu dans les codes precedents, un `std::bitset` peut etre affiche directement avec `std::cout`.

main.cpp

```
#include <iostream>
#include <bitset>

int main() {
    const std::bitset<8> b1 { 0b101010 };
    std::cout << "0b" << b1 << std::endl;
}
```

affiche :

```
0b00101010
```

Dans ce cas, `std::bitset` sera affiche en utilisant les caracteres `0` et `1`, avec autant de caracteres que definie dans l'argument template de `std::bitset` (donc en completant avec `0` si necessaire).

La fonction `to_string` permet de transformer un `std::bitset` en une chaîne de caractères, en utilisant par défaut les caractères `0` et `1` (c'est l'opération inverse de l'initialisation avec une chaîne). Cette fonction peut prendre deux arguments optionnels, correspondant aux caractères à utiliser respectivement pour `0` et `1`.

La chaîne de caractères produite peut être conservée dans une variable ou être directement affichée avec `std::cout`.

main.cpp

```
#include <iostream>
#include <bitset>

int main() {
    const std::bitset<8> b { 0b101010 };
    std::cout << b.to_string() << std::endl;
    std::cout << b.to_string('.', '.') << std::endl;
    std::cout << b.to_string('A', 'B') << std::endl;
}
```

affiche :

```
00101010
..1.1.1.
AABABABA
```

Manipuler individuellement les bits

Conceptuellement, `std::bitset` est un tableau compact de booléens. Il est donc possible de manipuler directement chaque bit comme une valeur booléenne, de la lire ou de la modifier directement dans `std::bitset`.

Pour connaître la taille d'un `std::bitset` (c'est à dire connaître la valeur de l'argument template `TAILLE` utilisé pour initialiser le `std::bitset`), vous pouvez utiliser la fonction `size`.

Vous voyez ici l'intérêt d'avoir une interface cohérente : beaucoup de classes de la bibliothèque standard représentant un tableau proposent la fonction `size` pour connaître la taille du tableau (`std::vector`, `std::array`, `std::string`, etc). Avoir une interface cohérente simplifie la mémorisation et évite les erreurs. Conservez cette idée en tête lorsque vous créerez vos propres interfaces.

main.cpp

```
#include <iostream>
#include <bitset>

int main() {
    const std::bitset<8> b1{};
    std::cout << b1.size() << std::endl;
    const std::bitset<16> b2{};
    std::cout << b2 << std::endl;
}
```

affiche :

```
8
16
```

<note>La fonction `test`

`std::bitset` propose également la fonction `test` pour accéder en lecture à un bit spécifique. La différence avec l'opérateur `[]` est que la fonction `test` vérifie si

Mask et flag

tester un bit : `mask`, `flag`, opérateur ET bit à bit, `test()` forcer un bit : OU bit à bit

tester plusieurs bits : `count`, `all`, `any`, `none` (cf algo)

Plusieurs fois des données de même type. Accès avec un indice, partant de 0. Taille fixée à la compilation ou à l'exécution : `bitset` à la compilation (`vector<bool>` à l'exécution).

Validation taille : assert connaitre la taille : size

Exercices

A partir de la représentation binaire d'un nombre (42 = 0b0000000000101010)

- Compter le nombre de 1 dans la représentation
- trouver la plus longue chaîne de 1 dans la représentation

Chapitre précédent	Sommaire principal	Chapitre suivant
------------------------------------	------------------------------------	----------------------------------