Les tableaux de bits

Vous avez déjà rencontrer 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() {
    std::cout << "0b" << std::bitset<8>(0b101010) << std::
endl;
    std::cout << "0b" << std::bitset<8>(42) << 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 utilisees, n'hesitez pas a 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 deja rencontré des classes qui ont des tailles determinées a la compilation ou l'exécution : respectivement `std::array` et `std::vector`).

Le nombre de bits geres par `std::bitset` est un argument template (donc qui s'écrit entre `<>`) de type entier.

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

Il existe differentes methodes pour initialiser un `std::bitset` :

- avec une valeur entiere ;
- 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` a `9` et `a` a `f`, en minuscule **ou** majuscule). Si aucune valeur n'est fournie, `std::bitset` est initialisé avec la valeur nulle.

main.cpp
```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 etre differente selon si vous utiliser une litterale chaine ou un type `std::string`.

Le cas le plus simple est d'initialiser un `std::bitset` a partir d'une chaine complete, constituee des caracteres `0` et `1`. Dans ce cas, la syntaxe est identique pour une litterale et un `std::string` :

main.cpp
```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 prefixe dans la chaine de caracteres.

Un `std::bitset` peut egalement etre initialise a partir d'une sous-chaine de caracteres (c'est a dire une partie d'une chaine de caracteres). La syntaxe est differentes entre une litterale et un `st::string` :

- pour une litterale, l'initialisation de `std::bitset` ne peut prendre qu'un seul argument optionnel supplementaire : le nombre de caracteres a conserver ;
- pour un `std::string`, `std::bitset` peut prendre deux arguments optionnels supplementaires : la position du premier caractere et le nombre de caracteres a conserver.

Un exemple concret avec une litterale chaine :

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 chaines de caracteres, puisqu'elles peuvent etre considerees comme des tableaux de caracteres) commencent a l'indice 0. Donc l'indice 4 correspond au cinquieme caractere :

```
chaine : 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 caracteres que `0` et `1`. Pour cela, il faut fournir deux arguments supplementaires, correspondent respectivement aux caracteres a utiliser a la place de `0` et de `1`.

main.cpp

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

```cpp
#include <iostream>

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

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

affiche :

```
10110110
10110110
```

Notez que les caracteres utilisees pour representer le `std::bitset` ne sont utilises 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 identifies 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 :

```cpp
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 :

```cpp
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 defaut 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 un std::bitset

Lors de l'affichage d'un `std::bitset`, les bits sont affichés en utilisant les caractères 0 et 1 et le nombre de caractères correspondra à la taille du `std::bitset`, quelque soit les caractères et leur nombre, utilisés pour initialiser le `std::bitset`.

Avec to_sting: utiliser d'autres caractères.

Mask et flag

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

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

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

Accès a un élément : [] 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