A rediger: cast vers d'autres types, bit shift

[Aller plus loin] 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 une classe qui a une taille déterminée à la compilation : `std::array`).

Le nombre de bits gérés par `std::bitset` est un argument *template* (donc qui s'écrit entre chevrons `<>`) 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` 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 différente selon si vous utiliser 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

```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 différentes 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 a 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 caractères = "1010"
    std::cout << b1 << std::endl;

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

affiche :

```
00001010
```

```
10101010
```

Avec un `std::string` :

main.cpp

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

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

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

    const std::bitset<8> b2 { s, 4, 2 }; // commence à l'indice 4 et conserve
                                         // 2 caractères = "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 caractères, puisqu'elles peuvent être considérées comme des tableaux de caractères) commencent à l'indice 0. Donc l'indice 4 correspond au cinquième caractère :

```
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 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 { "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 caractères utilisés pour représenter le `std::bitset` ne le sont que pour l'initialisation. La représentation en mémoire sera toujours identique et l'affichage utilisera par défaut les caractères `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 représente la position du premier caractère et `taille` qui correspond au nombre de caractères à 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

caractères et `position` pour la position du premier caractère (donc prendre 4 caractères à partir de la position 2), mais va prendre 2 caractères à partir de la position 4.

Pour la même raison, si on fournit un argument optionnel, les arguments optionnels qui le précédent 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 considérer que l'argument `position` prend sa valeur par défaut 0 (donc prendre 4 caractères a partir de la position 0), mais va utiliser `taille` comme position (donc prendre tous les caractères après la position 4).

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

Notez qu'il est possible d'affecter une valeur entière à un `std:bitset` en utilisant l'opérateur `=`, mais ce n'est pas possible directement avec une chaîne de caractères.

```
std::bitset<8> b {};
b = 0b110011;   // ok
b = "110011";   // erreur
```

Construction explicite et implicite

En fait, la classe `std::bitset` ne propose pas d'opérateur d'affectation `=` pour utiliser une valeur entière. Pourquoi est-il possible dans ce cas de pouvoir utiliser `=` ?

La classe `std::bitset` est copiable, ce qui signifie qu'il est possible d'affecter un `std::bitset` à un autre `std::bitset`.

```
std::bitset<8> b1 {};
std::bitset<8> b2 {};
b2 = b1;  // ok
```

Dans le cas d'une valeur entière, le compilateur est autorisé à créer automatiquement et de façon transparente un `std::bitset` à partir de cette valeur. C'est-à-dire à remplacer :

```
b = 0b110011;   // ok
```

Par :

```
b = std::bitset { 0b110011 };   // ok
```

Le compilateur réalise une conversion *implicite*.

Dans le cas des chaînes de caractères, cette conversion implicite est interdite, le compilateur ne peut réaliser que des conversions *explicites* (c'est à dire que le développeur doit explicitement écrire cette conversion).

```
std::bitset<8> b;
b = 0b110011;                    // ok, conversion implicite
b = std::bitset<8> { "110011" }; // ok, conversion explicite
```

Les conversion explicites et implicites seront détaillées dans la partie sur la création de classes.

Afficher et convertir en chaîne de caractères

Comme vous l'avez vu dans les codes précédents, un `std::bitset` peut être 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 caractères `0` et `1`, avec autant de caractères que définie dans l'argument *template* de `std::bitset` (donc en complétant avec `0` si nécessaire).

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

Vous voyez ici l'intérêt d'avoir une interface cohérente : beaucoup de classe de la bibliothèque standard représentant un tableau proposent la fonction `size` pour connaitre la taille du tableau (`std::vector`, `std::array`, `std::string`, etc). Avoir un 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> b8 {};
    std::cout << b8.size() << std::endl;
    const std::bitset<16> b16{};
    std::cout << b16.size() << std::endl;
}
```

affiche :

```
8
16
```

L'opérateur d'indexation

Comme pour les autres tableaux de la bibliothèque standard (`std::array`, `std::vector`, etc.), l'opérateur d'indexation `[]` permet d'accéder aux élément d'un `std::bitset`. Comme toujours, l'indice doit avoir une valeur comprise entre 0 et `SIZE-1`.

main.cpp

```
#include <iostream>
```

```
#include <bitset>
#include <cassert>

int main() {
    const std::bitset<8> b8 { 0b110111 };
    size_t index { 0 };
    assert(index < b8.size());
    std::cout << b8[index] << std::endl;
    index = 6;
    assert(index < b8.size());
    std::cout << b8[index] << std::endl;
}
```

affiche :

```
1
0
```

Notez bien que le bit correspondant à l'indice 0 est celui le plus à droite dans la représentation binaire (le bit de poids faible), le second bit est le second en partant de la droite et ainsi de suite.

main.cpp

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

int main() {
    const std::bitset<8> b8 { 0b110111 };
    std::cout << b8[0] << ' ' << b8[1] << ' ' << b8[2] << '
' << b8[3] << ' ' <<
                 b8[4] << ' ' << b8[5] << ' ' << b8[6] << '
' << b8[7] << std::endl;
```

affiche :

```
1 1 1 0 1 1 0 0
```

L'opérateur d'indexation `[]` est également utilisé pour modifier la valeur d'un bit en particulier, utilisant simplement l'opérateur d'affectation `=`, comme vous le feriez avec n'importe quelle variable.

main.cpp

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

int main() {
    std::bitset<8> b8 {};
    std::cout << b8 << std::endl;
    b8[0] = true;
    b8[5] = true;
    std::cout << b8 << std::endl;
}
```

affiche :

```
00000000
00100001
```

Les fonctions test, set, reset et flip

En complément de l'opérateur d'indexation `[]`, qui permet d'accéder aux bits d'un `std::bitset` en lecture ET en écriture, la classe `std::bitset` propose également plusieurs fonctions plus spécialisées, pour accéder aux bits en lecture OU en écriture.

Une seconde différence importante est que ces fonctions vérifient les accès en dehors des limites de `std::bitset` et lancer une exception de type `std::out_of_range` si vous essayez d'utiliser un index invalide.
La gestion des erreurs et les exceptions seront vu dans la partie sur la création de bibliothèque, mais retenez que la vérification systématique des accès est assimilée à de la programmation défensive, ce qui n'est pas forcément une bonne pratique.

Les fonctions sont les suivantes :

- `test` pour lire une valeur ;

- `set` pour modifier une valeur (par défaut pour la mettre à `true`) ;
- `reset` pour mettre une valeur à `false` ;
- `flip` pour inverser une valeur (c'est-à-dire que les bits valant `true` passent à `false` et vice-versa).

Ces fonctions prennent comme argument l'indice du bit à modifier. les fonctions `set`, `reset` et `flip` peuvent également être appelée sans argument, ce qui a pour effet de modifier tous les bits. Pour terminer, la fonction `set` peut prendre deux arguments : l'indice du bit à modifier et la valeur booléenne à définir.

main.cpp

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

int main() {
    std::bitset<8> b8 {};
    std::cout << b8 << std::endl;
    b8.set(3);
    std::cout << b8 << std::endl;
    b8.set();
    std::cout << b8 << std::endl;
    b8.set(5, false);
    std::cout << b8 << std::endl;
    b8.reset(3);
    std::cout << b8 << std::endl;
    b8.reset();
    std::cout << b8 << std::endl;
    b8.flip(3);
    std::cout << b8 << std::endl;
    b8.flip();
    std::cout << b8 << std::endl;
    std::cout << b8.test(2) << std::endl;
    std::cout << b8.test(3) << std::endl;
}
```

affiche :

```
00000000
00001000
```

```
11111111
11011111
11010111
00000000
00001000
11110111
1
0
```

Tester plusieurs bits

La classe `std::bitset` permet de tester plusieurs bits :

- `all` permet de tester si tous les bits ont la valeur `true` ;
- `any` permet de tester si au moins un bit à la valeur `true` ;
- `none` permet de tester si aucun bit n'a la valeur `true` ;
- `count` permet de compter le nombre de bits qui ont la valeur `true`.

main.cpp

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

int main() {
    const std::bitset<4> b1("0000");
    const std::bitset<4> b2("0101");
    const std::bitset<4> b3("1111");

    std::cout << "bitset\t" << "all\t" << "any\t" << "none\t"
<<
        "count" << std::endl;
    std::cout << b1 << '\t' << b1.all() << '\t' << b1.any()
<< '\t' <<
        b1.none() << '\t' << b1.count() << std::endl;
    std::cout << b2 << '\t' << b2.all() << '\t' << b2.any()
<< '\t' <<
        b2.none() << '\t' << b2.count() << std::endl;
    std::cout << b3 << '\t' << b3.all() << '\t' << b3.any()
<< '\t' <<
```

```
        b3.none() << '\t' << b3.count() << std::endl;
}
```

affiche :

```
bitset  all    any    none    count
0000    0    0    1    0
0101    0    1    0    2
1111    1    1    0    4
```

Masquage et drapeau

Il est également possible d'utiliser les énumérations pour représenter des drapeaux.

Les drapeaux

Il arrive souvent de devoir représenter l'état d'un concept en utilisant un nombre important de valeurs booléennes. Par exemple, une fenêtre dans une interface graphique sera visible ou non, pourra être déplaçable ou non, pourra avoir une barre de titre ou non, etc. Un véhicule pourra avoir trois ou cinq portes, pourra avoir un coffre ou non, pourra avoir ses phares allumés ou non, etc.

Une première possibilité pour représenter cela est de créer autant de variables booléennes que vous avez d'états à représenter.

```
// représentation de l'état d'une fenêtre
bool is_visible {};
bool is_movable {};
bool has_title {};
```

Il existe de nombreuses habitudes de programmation. Par exemple comme dans le code précédent, les préfixes “is” et “has”

sont utilisés respectivement les états “est” ou “n'est pas” et les états “a” ou “n'a pas”. Ainsi, `is_visible` signifie littéralement “est visible” et sera vraie quand la fenêtre est visible et fausse dans le cas contraire.
Cependant, vous avez vu qu'un booléen en mémoire n'est pas représenté par un bit, mais par un octet en général (soit 8 bits). Cela signifie donc une consommation mémoire inutilement plus importante.

main.cpp
```cpp
#include <iostream>

int main() {
    std::cout << sizeof(bool) << " octet(s)" << std::endl;
}
```

affiche :

```
1 octet(s)
```

De plus, chaque variable est indépendante. Si vous voulez réinitialiser une fenêtre par exemple, il faudra modifier chaque booléen un par un, il n'est pas possible de les remettre tous à faux en une instruction.

Pour améliorer cela, vous l'aurez surement deviné, il est possible d'utiliser `std::bitset` pour représenter un ensemble d'états booléens de façon compact. Chaque variable booléenne est maintenant replacée par un bit en particulier dans le `std::bitset`.

Par exemple, pour la fenêtre, il est possible d'utiliser un `std::bitset<3>`, dans lequel le premier bit représente `is_visible`, le second représente `is_movable` et le dernier `has_title`. Il est alors possible d'utiliser les syntaxes vues précédemment pour manipuler l'état de la fenêtre.

```cpp
using window_state = std::bitset<3>;

const window_state ws1 { 0b001 };    // fenêtre visible, non déplaçable et sans barre de titre
const window_state ws2 { 0b110 };    // fenêtre masquée, déplaçable et avec une barre de titre
```

```
window_state ws3;
ws3.flip(0);                          // inverse is_visible,
si la fenêtre était visible,
                                      // elle devient
invisible et réciproquement.
const bool is_visible = ws3.test(0); // teste si la fenêtre
est visible ou non
```

Vous voyez ici un effet indésirable des optimisation : le code perd en lisibilité. Sans la documentation, il est impossible de savoir que `0b001` représente une fenêtre visible, non déplaçable et sans barre de titre, alors que `is_visible`, `is_movable` et `has_title` sont suffisamment compréhensibles.
Et plus généralement, lorsque vous concevrez un code, vous devrez faire des compromis entre vos objectifs de qualité logicielle (lisibilité, évolutivité, fiabilité, etc) et vos contraintes (temps de développement, moyens humains, etc).

Le masquage

Le masquage est un concept générique en informatique, qui consiste à définir un sous-ensemble sur lequel sera appliqué une modification (vous pouvez par exemple retrouver ce concept dans les masques de calque dans un logiciel de dessin tel que Gimp).

Pour prendre un exemple concret, imaginez que vous avez la valeur `0b11001110` et que vous souhaitez modifier que les quatre premiers bits (en partant de la droite). En appliquant les opérations précédentes, vous obtiendrez :

Operation	Resultat	Commentaire
set	`0b11001111`	les quatre derniers bits sont mit a 1
reset	`0b11000000`	les quatre derniers bits sont mit a 0

flip	`0b11000001`	les quatre derniers bits sont inverses
all	`false`	les quatre derniers bits ne valent pas tous 1
any	`true`	les quatre derniers bits contiennent au moins 1
none	`false`	les quatre derniers bits ne valent pas 0
count	`3`	il y a 3 bits a 1 dans les quatres derniers bits

Les bits sur les quels est appliqués ou non les modifications sont appelés le masque. En pratique, il est possible de définir n'importe quelle séquence de bits dans un masque, vous n'êtes pas limités à définir une séquence continue comme dans l'exemple précédent.

Définir un masque est assez simple : il faut pouvoir définir pour chaque bit si celui-ci sera modifié (`true`) ou non (`false`). Un masque est donc en fait un `std::bitset`, pour lequel vous affectez `true` aux bits qui seront pourront être modifiés.

Par exemple, dans l'exemple précédent, le masque est le `std::bitset` suivant : `0b00001111`.

Pour appliquer le masque sur un `std::bitset`, vous utilisez ensuite les opérateurs logiques bit à bit que vous avez déjà vu dans le chapitre Logique binaire et calcul booléen (en particulier "ET" et "OU") :

- "ET" `&` ("AND") ;
- "OU" `|` ("OR") ;
- "OU Exclusif" `^` ("XOR") ;
- "Negation" `~` ("NOT").

Pour rappel, voici le tableau récapitulatif de ces opérateurs :

a	b	~a	a & b	a | b	a ^ b
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

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

Exercices avancés (nécessite de connaitre les classes) :

- créer une classe bitset_view
- créer une classe bitset_array_view

Chapitre précédent	Sommaire principal	Chapitre suivant