

Formation C++ 17 n° 02

Portées, opérateurs, structures de contrôle et fonctions

ROSSILLOL-LARUELLE Mattéo

24 janvier 2024

1 Avant-propos

2 Portées

- Une vision générale de la chose
- Les espaces de noms

3 Opérateurs

- Similitudes et différences avec Python
- Les opérateurs bit-à-bit

4 Structures de contrôle

- Avant de commencer
- Les instructions de conditionnement
- Les boucles

5 Fonctions

1 Avant-propos

2 Portées

- Une vision générale de la chose
- Les espaces de noms

3 Opérateurs

- Similitudes et différences avec Python
- Les opérateurs bit-à-bit

4 Structures de contrôle

- Avant de commencer
- Les instructions de conditionnement
- Les boucles

5 Fonctions

1 Avant-propos

2 Portées

- Une vision générale de la chose
- Les espaces de noms

3 Opérateurs

- Similitudes et différences avec Python
- Les opérateurs bit-à-bit

4 Structures de contrôle

- Avant de commencer
- Les instructions de conditionnement
- Les boucles

5 Fonctions

1 Avant-propos

2 Portées

- Une vision générale de la chose
- Les espaces de noms

3 Opérateurs

- Similitudes et différences avec Python
- Les opérateurs bit-à-bit

4 Structures de contrôle

- Avant de commencer
- Les instructions de conditionnement
- Les boucles

5 Fonctions

1 Avant-propos

2 Portées

- Une vision générale de la chose
- Les espaces de noms

3 Opérateurs

- Similitudes et différences avec Python
- Les opérateurs bit-à-bit

4 Structures de contrôle

- Avant de commencer
- Les instructions de conditionnement
- Les boucles

5 Fonctions

Avant de commencer, il est important de rappeler que ce cours est réalisé par **un étudiant**. Par conséquent, il n'a pas la même fiabilité qu'un cours dispensé par **un réel enseignant de l'ENSIMAG**.

N'utilisez pas ce cours comme **un argument d'autorité** !

Si un professeur semble, a posteriori, contredire des éléments apportés par ce cours, **il a très probablement raison**.

Ce document est **vivant** : je veillerai à corriger les coquilles ou erreurs plus problématiques.

Portées

Une vision générale de la chose

Définition

Une **portée** (ou *scope*) est une portion, potentiellement discontinue (on verra des exemples concrets lorsque l'on parlera d'**unité de traduction**), dans laquelle des entités données (variables, objets, etc.) *vivent*.

Dans un premier temps, on considérera que, en C++, les *scopes* sont définis par un bloc délimité, d'un côté, par { et, de l'autre, par }.

Définition

Une **portée** (ou *scope*) est une portion, potentiellement discontinue (on verra des exemples concrets lorsque l'on parlera d'**unité de traduction**), dans laquelle des entités données (variables, objets, etc.) *vivent*.

Dans un premier temps, on considérera que, en C++, **les scopes** sont définis par un bloc délimité, d'un côté, par { et, de l'autre, par }.

Exemple

Le code suivant montre l'exemple de la portée définie par la fonction `main()` :

```
1 int main(int argc, char* argv[])
2 {
3     int ma_variable_locale;
4 }
```

Dans l'exemple ci-dessus, `argc` et `argv`, en tant que paramètres, sont des variables qui n'existent que dans la portée de la fonction `main()`, tout comme `ma_variable_locale` qui, hors de cette fonction, cesse également d'exister.

Exemple

On peut définir une portée dans n'importe quelle portée parente.

```
1 int main(int argc, char* argv[])
2 {
3     int ma_variable_locale;
4
5     {
6         int une_variable_dans_une_autre_portee;
7     } // une_variable_dans_une_autre_portee « meure » ici
8
9     int une_autre_variable_locale;
10 } // les autres variables « meurent » là
```

Portées

Les espaces de noms

Définition

Un **espace de nommage** (ou *namespace*) est ce que l'on pourrait voir comme une portée nommée.

Il permet d'éviter de polluer la portée globale et d'avoir donc une meilleure structuration de son code.

Comme première approximation, on pourrait le voir comme l'équivalent d'un module en Python.

Définition

Un **espace de nommage** (ou *namespace*) est ce que l'on pourrait voir comme une portée nommée.

Il permet d'éviter de polluer **la portée globale** et d'avoir donc une meilleure structuration de son code.

Comme première approximation, on pourrait le voir comme l'équivalent d'un **module en Python**.

Définition

Un **espace de nommage** (ou *namespace*) est ce que l'on pourrait voir comme une portée nommée.

Il permet d'éviter de polluer **la portée globale** et d'avoir donc une meilleure structuration de son code.

Comme première approximation, on pourrait le voir comme l'équivalent d'un **module en Python**.


```
namespace <identifiant> { <corps> }
```

Figure 1 – Déclaration d'un espace de noms

On déclare un espace de noms portant le nom `identifiant` et ayant pour corps `corps` : ce dernier peut contenir tout ce que pourrait avoir une portée classique.

Pour accéder à un membre de l'espace de noms, on utilise la syntaxe suivante :

```
<espace de noms>::<membre>
```

Figure 2 – Accès aux membres

```
namespace <identifiant> { <corps> }
```

Figure 1 – Déclaration d'un espace de noms

On déclare un espace de noms portant le nom `identifiant` et ayant pour corps `corps` : ce dernier peut contenir tout ce que pourrait avoir une portée classique.
Pour accéder à `un membre` de l'espace de noms, on utilise la syntaxe suivante :

```
<espace de noms>::<membre>
```

Figure 2 – Accès aux membres

Exemple

```
1 #include <cstdint>
2
3 using namespace std;
4
5 namespace lib
6 {
7     uint8_t un = 1;
8
9     bool est_pair(uint8_t __a)
10    {
11        return ((__a % 2) == 0);
12    }
13 }
14
15 int main()
16 {
17     bool un_booleen = lib::est_pair(2 + lib::un);
18 }
```

Avec la syntaxe suivante, on va pouvoir *polluer* la portée dans laquelle on se trouve, en accédant aux membres de l'espace de noms **espace de noms** sans les préfixer par **<espace de noms>::**.

```
using namespace <espace de noms>;
```

Figure 3 – Utilisation d'un espace de noms

Attention On peut mettre ce *statement* dans n'importe quelle portée. Il doit cependant être avant l'endroit où on utilise son effet. Cette syntaxe serait donc l'équivalente, en **Python**, de

```
import * from <module>
```

Avec la syntaxe suivante, on va pouvoir *polluer* la portée dans laquelle on se trouve, en accédant aux membres de l'espace de noms **espace de noms** sans les préfixer par **<espace de noms>::**.

```
using namespace <espace de noms>;
```

Figure 3 – Utilisation d'un espace de noms

Attention On peut mettre ce *statement* dans n'importe quelle portée. Il doit cependant être avant l'endroit où on utilise son effet.

Cette syntaxe serait donc l'équivalente, en Python, de

```
import * from <module>
```

Avec la syntaxe suivante, on va pouvoir *polluer* la portée dans laquelle on se trouve, en accédant aux membres de l'espace de noms **espace de noms** sans les préfixer par **<espace de noms>::**.

```
using namespace <espace de noms>;
```

Figure 3 – Utilisation d'un espace de noms

Attention On peut mettre ce *statement* dans n'importe quelle portée. Il doit cependant être avant l'endroit où on utilise son effet. Cette syntaxe serait donc l'équivalente, en **Python**, de

```
import * from <module>
```

Exemple

En reprenant le code précédent, on se ramène à :

```
1 #include <cstdlib>
2
3 using namespace std;
4
5 namespace lib
6 {
7     uint8_t un = 1;
8
9     bool est_pair(uint8_t __a)
10    {
11        return ((__a % 2) == 0);
12    }
13 }
14
15 int main()
16 {
17     using namespace lib;
18
19     bool un_booleen = est_pair(2 + un);
20     bool un_autre_booleen = lib::est_pair(2 + lib::un); // cette ligne fonctionne également
21
22     // si 'using namespace lib;' avait été ici, on aurait eu une erreur à la compilation
23 }
```

Au lieu d'*importer* tous les membres d'un espace de noms, on peut n'en sélectionner que quelques-uns :

```
using <espace de noms>::<membre>;
```

Figure 4 – Utilisation d'un membre d'un espace de noms

Cette syntaxe serait donc l'équivalente, en Python, de

```
import <membre> from <module>
```


Au lieu d'*importer* tous les membres d'un espace de noms, on peut n'en sélectionner que quelques-uns :

```
using <espace de noms>::<membre>;
```

Figure 4 – Utilisation d'un membre d'un espace de noms

Cette syntaxe serait donc l'équivalente, en Python, de

```
import <membre> from <module>
```

Au lieu d'*importer* tous les membres d'un espace de noms, on peut n'en sélectionner que quelques-uns :

```
using <espace de noms>::<membre>;
```

Figure 4 – Utilisation d'un membre d'un espace de noms

Cette syntaxe serait donc l'équivalente, en Python, de

```
import <membre> from <module>
```

Exemple

Avec cette nouvelle syntaxe, l'exemple précédent donnerait :

```
1 #include <cstdint>
2
3 using namespace std;
4
5 namespace lib
6 {
7     uint8_t un = 1;
8
9     bool est_pair(uint8_t __a)
10    {
11        return ((__a % 2) == 0);
12    }
13 }
14
15 int main()
16 {
17     using lib::est_pair;
18     using lib::un;
19
20     bool un_booleen = est_pair(2 + un);
21 }
```

Avec ce que l'on a dit précédemment la ligne `using namespace std;` que l'on retrouvait un peu partout n'a plus aucun secret pour vous.

Les *statements* que l'on vient d'introduire sont, dans la grande majorité des cas (mais pas tous), à éviter car ils *polluent* la portée dans laquelle on travaille et déstructurent donc le code : ce qui va à l'encontre de l'intérêt des espace de noms.

Attention Dans la suite du cours, on n'utilisera donc plus `using namespace std;`.

Avec ce que l'on a dit précédemment la ligne `using namespace std;` que l'on retrouvait un peu partout n'a plus aucun secret pour vous.

Les *statements* que l'on vient d'introduire sont, dans la grande majorité des cas (mais pas tous), à éviter car ils *polluent* la portée dans laquelle on travaille et déstructurent donc le code : ce qui va à l'encontre de l'intérêt des espace de noms.

Attention Dans la suite du cours, on n'utilisera donc plus `using namespace std;`.

Avec ce que l'on a dit précédemment la ligne `using namespace std;` que l'on retrouvait un peu partout n'a plus aucun secret pour vous.

Les *statements* que l'on vient d'introduire sont, dans la grande majorité des cas (mais pas tous), à éviter car ils *polluent* la portée dans laquelle on travaille et déstructurent donc le code : ce qui va à l'encontre de l'intérêt des espace de noms.

Attention Dans la suite du cours, on n'utilisera donc plus `using namespace std;`.

Avec la syntaxe suivante, on va pouvoir déclarer un espace de noms qui *pollue* tout seul la portée dans laquelle on se trouve :

```
inline namespace <identifiant> { <corps> }
```

Figure 5 – Déclaration d'un espace de noms inline

Exemple

Toujours en reprenant le code de tout à l'heure, on pourrait se ramener à :

```
1 #include <cstdint>
2
3 namespace lib
4 {
5     inline namespace variables
6     {
7         inline namespace nombres { std::uint8_t un = 1; }
8     }
9
10    inline namespace fonctions
11    {
12        bool est_pair(std::uint8_t __a) { return ((__a % 2) == 0); }
13    }
14 }
15
16 int main()
17 {
18     using lib::un;
19
20     bool un_booleen = lib::est_pair(2 + un);
21 }
```


Comme pour les types, on peut *renommer* les espaces de noms :

```
namespace <autre nom> = <espace de noms>;
```

Figure 6 – Déclaration d'un alias pour un espace de noms

Cette syntaxe serait donc l'équivalente, en Python, de

```
import <module> as <autre nom>
```

Comme pour les types, on peut *renommer* les espaces de noms :

```
namespace <autre nom> = <espace de noms>;
```

Figure 6 – Déclaration d'un alias pour un espace de noms

Cette syntaxe serait donc l'équivalente, en Python, de

```
import <module> as <autre nom>
```

Les deux codes suivant sont strictement équivalent :

```
1 namespace lib
2 {
3     namespace fonctions
4     {
5         bool est_pair(std::uint8_t __a) { return ((__a % 2) == 0); }
6     }
7 }
```

```
1 namespace lib::fonctions
2 {
3     bool est_pair(std::uint8_t __a) { return ((__a % 2) == 0); }
4 }
```

Pour faire le point sur tout ça

Exemple

```
1 #include <cstdint>
2 #include <iostream>
3
4 namespace lib
5 {
6     namespace a::variables::cardinaux { std::uint16_t premier = 0; }
7     namespace b { namespace variables::cardinaux { std::uint16_t premier = 1; } }
8     inline namespace c { namespace variables::cardinaux { std::uint16_t premier = 2; } }
9 }
10
11 using lib::variables::cardinaux::premier;
12 namespace a = lib::b;
13
14 int main(int argc, char* argv[])
15 {
16     using namespace lib::a;
17
18     std::cout << premier << '\n';
19     std::cout << ::a::variables::cardinaux::premier << '\n';
20     std::cout << lib::variables::cardinaux::premier << '\n';
21     std::cout << variables::cardinaux::premier << '\n';
22 }
```

Dans l'exemple ci-dessus que va-t-il se passer et pourquoi ?

Pour faire le point sur tout ça

Exemple

```
1 #include <cstdint>
2 #include <iostream>
3
4 namespace lib
5 {
6     namespace a::variables::cardinaux { std::uint16_t premier = 0; }
7     namespace b { namespace variables::cardinaux { std::uint16_t premier = 1; } }
8     inline namespace c { namespace variables::cardinaux { std::uint16_t premier = 2; } }
9 }
10
11 using lib::variables::cardinaux::premier;
12 namespace a = lib::b;
13
14 int main(int argc, char* argv[])
15 {
16     using namespace lib::a;
17
18     std::cout << premier << '\n';
19     std::cout << ::a::variables::cardinaux::premier << '\n';
20     std::cout << lib::variables::cardinaux::premier << '\n';
21     std::cout << variables::cardinaux::premier << '\n';
22 }
```

Dans l'exemple ci-dessus que va-t-il se passer et pourquoi ?

Il sera imprimé, dans l'ordre, dans la console « 2 », « 1 », « 2 » et « 0 ».

Opérateurs

Similitudes et différences avec Python

Un opérateur bien singulier

Dans ce transparent et les suivants, a et b désignent **deux expressions compatibles**.

a, b

Figure 7 – L'opérateur virgule

L'opérande de gauche est évalué, puis son résultat est écrasé et l'opérande de droite est évalué : ainsi, a, b pour valeur b.

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << (5.5, 5) << '\n';
6 }
```

Que sera-t-il imprimé en console dans l'exemple ci-dessus ?

Un opérateur bien singulier

Dans ce transparent et les suivants, a et b désignent **deux expressions compatibles**.

a, b

Figure 7 – L'opérateur virgule

L'opérande de gauche est évalué, puis son résultat est écrasé et l'opérande de droite est évalué : ainsi, a, b pour valeur b.

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << (5.5, 5) << '\n';
6 }
```

Que sera-t-il imprimé en console dans l'exemple ci-dessus ?

Un opérateur bien singulier

Dans ce transparent et les suivants, a et b désignent **deux expressions compatibles**.

a, b

Figure 7 – L'opérateur virgule

L'opérande de gauche est évalué, puis son résultat est écrasé et l'opérande de droite est évalué : ainsi, a, b pour valeur b.

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << (5.5, 5) << '\n';
6 }
```

Que sera-t-il imprimé en console dans l'exemple ci-dessus ?

Un opérateur bien singulier

Dans ce transparent et les suivants, a et b désignent **deux expressions compatibles**.

a, b

Figure 7 – L'opérateur virgule

L'opérande de gauche est évalué, puis son résultat est écrasé et l'opérande de droite est évalué : ainsi, a, b pour valeur b.

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << (5.5, 5) << '\n';
6 }
```

Que sera-t-il imprimé en console dans l'exemple ci-dessus ?

Il sera imprimé « 5 ».

Certains opérateurs existent en C++ aussi bien qu'en Python, et ont la même forme syntaxique.

+a
-a

Figure 8 – Opérateurs unitaires

a = b

Figure 9 – Opérateur de copie

Certains opérateurs existent en C++ aussi bien qu'en Python, et ont la même forme syntaxique.

```
a + b  
a - b  
a * b  
a / b  
a % b
```

Figure 10 – Opérateurs arithmétiques

```
a += b  
a -= b  
a *= b  
a /= b  
a %= b
```

Figure 11 – Opérateurs d'affectation associés

Certains opérateurs existent en C++ aussi bien qu'en Python, et ont la même forme syntaxique.

```
a == b  
a != b  
a < b  
a > b  
a <= b  
a >= b
```

Figure 12 – Opérateurs de comparaison

Une petite nuance qui a son importance

Contrairement **en Python**, **en C++**, les opérateurs d'affectation ont un retour : l'évaluation des opérations d'affectation a une valeur.

Exemple

```
1 #include <cstdint>
2 #include <iostream>
3
4 int main()
5 {
6     std::uint32_t a = 0;
7
8     std::cout << (a += 1) << '\n';
9     std::cout << (a -= 1) << '\n';
10    std::cout << (a = 5) << '\n';
11    std::cout << (a *= 2) << '\n';
12    std::cout << (a /= 3) << '\n';
13    std::cout << (a %= 3) << '\n';
14 }
```

Que sera-t-il imprimé sur la ligne de commande ?

Une petite nuance qui a son importance

Contrairement **en Python**, **en C++**, les opérateurs d'affectation ont un retour : l'évaluation des opérations d'affectation a une valeur.

Exemple

```
1 #include <cstdlib>
2 #include <iostream>
3
4 int main()
5 {
6     std::uint32_t a = 0;
7
8     std::cout << (a += 1) << '\n';
9     std::cout << (a -= 1) << '\n';
10    std::cout << (a = 5) << '\n';
11    std::cout << (a *= 2) << '\n';
12    std::cout << (a /= 3) << '\n';
13    std::cout << (a %= 3) << '\n';
14 }
```

Que sera-t-il imprimé sur la ligne de commande ?

Une petite nuance qui a son importance

Contrairement **en Python**, **en C++**, les opérateurs d'affectation ont un retour : l'évaluation des opérations d'affectation a une valeur.

Exemple

```
1 #include <cstdlib>
2 #include <iostream>
3
4 int main()
5 {
6     std::uint32_t a = 0;
7
8     std::cout << (a += 1) << '\n';
9     std::cout << (a -= 1) << '\n';
10    std::cout << (a = 5) << '\n';
11    std::cout << (a *= 2) << '\n';
12    std::cout << (a /= 3) << '\n';
13    std::cout << (a %= 3) << '\n';
14 }
```

Que sera-t-il imprimé sur la ligne de commande ?

Il sera imprimé, dans cette ordre, « 1 », « 0 », « 5 », « 10 », « 3 », « 0 ».

Opérateurs d'incrémentation et de décrémentation

`++a`

`--a`

Figure 13 – Opérateurs de préincrémentation et de prédécrémentation

Les expressions précédentes sont équivalentes à :

`a += 1`

`a -= 1`

Figure 15 – Opérateurs de préincrémentation et de prédécrémentation (équivalent)

Les opérateurs de postincrémentation et de postdécrémentation font intervenir une variable temporaire : ils sont donc moins performants.

Dans la majorité des cas, on préférera utiliser les opérateurs de préincrémentation et de prédécrémentation.

`a++`

`a--`

Figure 14 – Opérateurs de postincrémentation et de postdécrémentation

`a += 1, a - 1`

`a -= 1, a + 1`

Figure 16 – Opérateurs de postincrémentation et de postdécrémentation (équivalent)

Opérateurs d'incrémentation et de décrémentation

`++a`

`--a`

Figure 13 – Opérateurs de préincrémentation et de prédécrémentation

Les expressions précédentes sont équivalentes à :

`a += 1`

`a -= 1`

Figure 15 – Opérateurs de préincrémentation et de prédécrémentation (équivalent)

Les opérateurs de postincrémentation et de postdécrémentation font intervenir une variable temporaire : ils sont donc moins performants.

Dans la majorité des cas, on préférera utiliser les opérateurs de préincrémentation et de prédécrémentation.

`a++`

`a--`

Figure 14 – Opérateurs de postincrémentation et de postdécrémentation

Figure 16 – Opérateurs de postincrémentation et de postdécrémentation (équivalent)

`a += 1, a - 1`

`a -= 1, a + 1`

Opérateurs d'incrément et de décrémentation

`++a`

`--a`

Figure 13 – Opérateurs de préincrément et de prédécrément

Les expressions précédentes sont équivalentes à :

`a += 1`

`a -= 1`

Figure 15 – Opérateurs de préincrément et de prédécrément (équivalent)

Les opérateurs de postincrément et de postdécrémentation font intervenir une variable temporaire : ils sont donc moins performants.

Dans la majorité des cas, on préférera utiliser les opérateurs de préincrément et de prédécrément.

`a++`

`a--`

Figure 14 – Opérateurs de postincrément et de postdécrémentation

Figure 16 – Opérateurs de postincrément et de postdécrémentation (équivalent)

`a += 1, a - 1`

`a -= 1, a + 1`

Exemple

```
1 #include <cstdint>
2 #include <iostream>
3
4 int main()
5 {
6     std::uint32_t a = 0;
7
8     std::cout << a++ << '\n';
9     std::cout << ++a << '\n';
10    std::cout << --a << '\n';
11    std::cout << a-- << '\n';
12 }
```

Que sera-t-il imprimé sur la ligne de commande ?

Exemple

```
1 #include <cstdint>
2 #include <iostream>
3
4 int main()
5 {
6     std::uint32_t a = 0;
7
8     std::cout << a++ << '\n';
9     std::cout << ++a << '\n';
10    std::cout << --a << '\n';
11    std::cout << a-- << '\n';
12 }
```

Que sera-t-il imprimé sur la ligne de commande ?

Il sera imprimé, dans cet ordre, « 0 », « 2 », « 1 », « 1 ».

Opérateurs

Les opérateurs bit-à-bit

Opérateurs

Les opérateurs bit-à-bit

Éléments de syntaxe

`~a`

Figure 17 – Opérateur unitaire

`a & b`

`a | b`

`a ^ b`

`a << b`

`a >> b`

`a &= b`

`a |= b`

`a ^= b`

`a <<= b`

`a >>= b`

Figure 18 – Opérateurs arithmétiques

Figure 19 – Opérateurs d'affectation associés

Opérateurs

Les opérateurs bit-à-bit

Tables de vérité

a	0	1
NOT a	1	0

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b00000001;
6
7     std::cout << static_cast<int>(~a) << '\n';
8 }
```

Que sera-t-il imprimé en console ?

a	0	1
NOT a	1	0

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b00000001;
6
7     std::cout << static_cast<int>(~a) << '\n';
8 }
```

Que sera-t-il imprimé en console ?

a	0	1
NOT a	1	0

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b00000001;
6
7     std::cout << static_cast<int>(~a) << '\n';
8 }
```

Que sera-t-il imprimé en console ?

Il sera imprimé « 254 ».

	b	0	1
a			
0		0	0
1		0	1

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a & b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

	b	0	1
a			
0		0	0
1		0	1

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a & b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

	b	0	1
a			
0		0	0
1		0	1

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a & b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

Il sera imprimé « 1 ».

	b	0	1
a			
0		0	1
1		1	1

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a | b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

	b	0	1
a			
0		0	1
1		1	1

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a | b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

	b	0	1
a			
0		0	1
1		1	1

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a | b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

Il sera imprimé « 255 ».

	b	0	1
a			
0		0	1
1		1	0

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a ^ b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

	b	0	1
a			
0		0	1
1		1	0

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a ^ b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

	b	0	1
a			
0		0	1
1		1	0

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a ^ b) << '\n';
9 }
```

Que sera-t-il imprimé en console ?

Il sera imprimé « 254 ».

Il faut comprendre l'effet de cette opération de manière littérale. Pour un **décalage à gauche** (resp. **à droite**), on jette **les bits de poids forts** (resp. **de poids faibles**) et on fait apparaître des zéros au niveau des **bits de poids faibles** (resp. **poids forts**).

Remarque

Un décalage de n bits vers la gauche (resp. vers la droite) correspond à une multiplication (resp. une division) par 2^n .

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b00000001;
6
7     std::cout << static_cast<int>(a <= 5) << '\n';
8     std::cout << static_cast<int>(a > 10) << '\n';
9 }
```

Que sera-t-il imprimé sur la ligne de commande ?

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b00000001;
6
7     std::cout << static_cast<int>(a <= 5) << '\n';
8     std::cout << static_cast<int>(a > 10) << '\n';
9 }
```

Que sera-t-il imprimé sur la ligne de commande ?

Il sera imprimé « 32 », puis « 0 ».

Structures de contrôle

Avant de commencer

Définition

Une **structure de contrôle** est une instruction particulière, dans un langage de programmation impératif, pouvant dévier l'ordre dans lequel sont exécutées certaines instructions du programme.

Structures de contrôle

Les instructions de conditionnement

```
if (<condition>) { [corps] }  
else if (<condition>) { [corps] }  
    else { [corps] }
```

Figure 20 – Instructions en C++

```
if <condition>: [corps]  
elif <condition>: [corps]  
    else: [corps]
```

Figure 21 – Instructions en Python

Attention En C++, la présence des parenthèses est obligatoire.

```
if (<condition>) { [corps] }  
else if (<condition>) { [corps] }  
    else { [corps] }
```

Figure 20 – Instructions en C++

```
if <condition>: [corps]  
elif <condition>: [corps]  
    else: [corps]
```

Figure 21 – Instructions en Python

Attention En C++, la présence des parenthèses est obligatoire.

On peut transformer le code

```
1 if (a == 'a') {
2     std::cout << "Il s'agit de la 1ere lettre de l'alphabet" << '\n';
3     /* *** Tout le reste *** */
4 } else if (a == 'z') {
5     std::cout << "Il s'agit de la 26e lettre de l'alphabet" << '\n';
6 } else {
7     std::cout << "Le caractère n'est pas reconnu" << '\n';
8 }
```

en

```
1 switch (a) {
2     case 'a':
3         std::cout << "Il s'agit de la 1ere lettre de l'alphabet" << '\n';
4         break;
5     /* *** Tout le reste *** */
6     case 'z':
7         std::cout << "Il s'agit de la 26e lettre de l'alphabet" << '\n';
8         break;
9     default:
10        std::cout << "Le caractère n'est pas reconnu" << '\n';
11        break;
12 }
```

```
switch (<initialisation>; <condition>) { [bloc] }  
    switch (<condition>) { [bloc] }
```

Figure 22 – Le *statement* switch

- `initialisation` est une expression ou une déclaration dont la portée des entités est celle de `bloc`.
- `condition` est soit une expression soit une déclaration (la valeur de l'entité déclaré est alors utilisée).

`bloc` est constitué de :

- `case <expression constante>: <statement>`
- `default: <statement>`

```
switch (<initialisation>; <condition>) { [bloc] }  
    switch (<condition>) { [bloc] }
```

Figure 22 – Le *statement* switch

- **initialisation** est une expression ou une déclaration dont la portée des entités est celle de **bloc**.
- **condition** est soit une expression soit une déclaration (la valeur de l'entité déclarée est alors utilisée).

bloc est constitué de :

- **case** <expression constante>: <statement>
- **default**: <statement>


```
switch (<initialisation>; <condition>) { [bloc] }  
    switch (<condition>) { [bloc] }
```

Figure 22 – Le *statement* switch

- **initialisation** est une expression ou une déclaration dont la portée des entités est celle de **bloc**.
- **condition** est soit une expression soit une déclaration (la valeur de l'entité déclaré est alors utilisée).

bloc est constitué de :

- case <expression constante>: <statement>
- default: <statement>

```
switch (<initialisation>; <condition>) { [bloc] }  
    switch (<condition>) { [bloc] }
```

Figure 22 – Le *statement* switch

- **initialisation** est une expression ou une déclaration dont la portée des entités est celle de **bloc**.
- **condition** est soit une expression soit une déclaration (la valeur de l'entité déclaré est alors utilisée).

bloc est constitué de :

- `case <expression constante>: <statement>`
- `default: <statement>`

```
switch (<initialisation>; <condition>) { [bloc] }  
    switch (<condition>) { [bloc] }
```

Figure 22 – Le *statement* switch

- **initialisation** est une expression ou une déclaration dont la portée des entités est celle de **bloc**.
- **condition** est soit une expression soit une déclaration (la valeur de l'entité déclaré est alors utilisée).

bloc est constitué de :

- case **<expression constante>**: **<statement>**
- default: **<statement>**

```
switch (<initialisation>; <condition>) { [bloc] }  
    switch (<condition>) { [bloc] }
```

Figure 22 – Le *statement* switch

- **initialisation** est une expression ou une déclaration dont la portée des entités est celle de **bloc**.
- **condition** est soit une expression soit une déclaration (la valeur de l'entité déclaré est alors utilisée).

bloc est constitué de :

- case **<expression constante>**: *<statement>*
- default: *<statement>*

Attention

Le mot-clef `break`, dans l'exemple précédent, a son importance ! Sans celui-ci, si `a` valait `'c'`, on aurait eu dans la console :

```
Il s'agit de la 3e lettre de l'alphabet
```

```
...
```

```
Il s'agit de la 26e lettre de l'alphabet
```

```
Le caractère n'est pas reconnu
```

Remarque

Les `cases` définissent en réalité des *labels* qui sont donc utilisables avec un `goto`.

Remarque

Le conditionnement avec `switch` est *plus rapide* car le compilateur le retraduit sous la forme d'une *jump table*.

Attention

Le mot-clef `break`, dans l'exemple précédent, a son importance ! Sans celui-ci, si `a` valait `'c'`, on aurait eu dans la console :

```
Il s'agit de la 3e lettre de l'alphabet
```

```
...
```

```
Il s'agit de la 26e lettre de l'alphabet
```

```
Le caractère n'est pas reconnu
```

Remarque

Les `cases` définissent en réalité des *labels* qui sont donc utilisables avec un `goto`.

Remarque

Le conditionnement avec `switch` est *plus rapide* car le compilateur le retraduit sous la forme d'une *jump table*.

Attention

Le mot-clef `break`, dans l'exemple précédent, a son importance ! Sans celui-ci, si `a` valait `'c'`, on aurait eu dans la console :

```
Il s'agit de la 3e lettre de l'alphabet
```

```
...
```

```
Il s'agit de la 26e lettre de l'alphabet
```

```
Le caractère n'est pas reconnu
```

Remarque

Les `cases` définissent en réalité des *labels* qui sont donc utilisables avec `un goto`.

Remarque

Le conditionnement avec `switch` est *plus rapide* car le compilateur le retraduit sous la forme d'une *jump table*.

L'opérateur ternaire

`<condition> ? <a> : `

Figure 23 – L'opérateur ternaire

Si `condition` est vraie (resp. fausse), alors `condition ? a : b` est évalué en `a` (resp. `b`).

Attention

`a` et `b` doivent être deux types compatibles.

Exemple

```
1 std::cout << ((argc == 1) : argv[0] : argv[1]) << '\n';
```

Dans l'exemple ci-dessus que va-t-il être imprimé en console et dans quelle condition ?

`<condition> ? <a> : `

Figure 23 – L'opérateur ternaire

Si `condition` est vraie (resp. fausse), alors `condition ? a : b` est évalué en `a` (resp. `b`).

Attention

`a` et `b` doivent être deux types compatibles.

Exemple

```
1 std::cout << ((argc == 1) : argv[0] : argv[1]) << '\n';
```

Dans l'exemple ci-dessus que va-t-il être imprimé en console et dans quelle condition ?

`<condition> ? <a> : `

Figure 23 – L'opérateur ternaire

Si `condition` est vraie (resp. fausse), alors `condition ? a : b` est évalué en `a` (resp. `b`).

Attention

`a` et `b` doivent être deux types compatibles.

Exemple

```
1 std::cout << ((argc == 1) : argv[0] : argv[1]) << '\n';
```

Dans l'exemple ci-dessus que va-t-il être imprimé en console et dans quelle condition ?

`<condition> ? <a> : `

Figure 23 – L'opérateur ternaire

Si `condition` est vraie (resp. fausse), alors `condition ? a : b` est évalué en `a` (resp. `b`).

Attention

`a` et `b` doivent être deux types compatibles.

Exemple

```
1 std::cout << ((argc == 1) : argv[0] : argv[1]) << '\n';
```

Dans l'exemple ci-dessus que va-t-il être imprimé en console et dans quelle condition ?

Il sera imprimé le nom du programme si aucun argument n'est passé en ligne de commande ;
sinon, il sera imprimé le premier argument.

Structures de contrôle

Les boucles

```
while (<condition>) { [corps] }  
for (<déclaration> : <expression>) { [corps] }
```

Figure 24 – Instructions en C++

```
while <condition>: [corps]  
for <déclaration> in <expression>: [corps]
```

Figure 25 – Instructions en Python

```
do { [corps] } while (<condition>);
```

Figure 26 – La boucle do-while

Contrairement à la boucle `while`, la vérification de `condition` se fait à la fin de l'itération : il y a donc au moins une itération effectuée.

```
for ([déclaration ou expression]; [condition]; [expression]) { [corps] }
```

Figure 27 – La boucle for

Le fonctionnement est le suivant :

- 1 `déclaration` ou `expression` est évalué avant la première itération ;
- 2 `condition` est évalué avant chaque itération et si, et seulement si, elle est fausse alors la boucle cesse ;
- 3 `expression` est évalué à la fin de chaque itération.

```
for ([déclaration ou expression]; [condition]; [expression]) { [corps] }
```

Figure 27 – La boucle for

Le fonctionnement est le suivant :

- 1 `déclaration` ou `expression` est évalué avant la première itération ;
- 2 `condition` est évalué avant chaque itération et si, et seulement si, elle est fausse alors la boucle cesse ;
- 3 `expression` est évalué à la fin de chaque itération.


```
for ([déclaration ou expression]; [condition]; [expression]) { [corps] }
```

Figure 27 – La boucle for

Le fonctionnement est le suivant :

- 1 `déclaration ou expression` est évalué avant la première itération ;
- 2 `condition` est évalué avant chaque itération et si, et seulement si, elle est fausse alors la boucle cesse ;
- 3 `expression` est évalué à la fin de chaque itération.

```
for ([déclaration ou expression]; [condition]; [expression]) { [corps] }
```

Figure 27 – La boucle for

Le fonctionnement est le suivant :

- 1 **déclaration ou expression** est évalué avant la première itération ;
- 2 **condition** est évalué avant chaque itération et si, et seulement si, elle est fausse alors la boucle cesse ;
- 3 **expression** est évalué à la fin de chaque itération.

```
for ([déclaration ou expression]; [condition]; [expression]) { [corps] }
```

Figure 27 – La boucle for

Le fonctionnement est le suivant :

- 1 **déclaration ou expression** est évalué avant la première itération ;
- 2 **condition** est évalué avant chaque itération et si, et seulement si, elle est fausse alors la boucle cesse ;
- 3 **expression** est évalué à la fin de chaque itération.

Exemple

```
1 int mystere(unsigned int __n)
2 {
3     int ret = 1;
4
5     for (; __n > 1; --__n) {
6         ret *= __n;
7     }
8
9     return ret;
10 }
```

Que peut-on dire de la fonction `mystere()` ?

Exemple

```
1 int mystere(unsigned int __n)
2 {
3     int ret = 1;
4
5     for (; __n > 1; --__n) {
6         ret *= __n;
7     }
8
9     return ret;
10 }
```

Que peut-on dire de la fonction `mystere()` ?

Il s'agit de la fonction factorielle.

Définition

Une **boucle infini** est une boucle dont l'expression de condition reste vraie à chaque itération : le seul moyen d'en sortir est d'utiliser le mot-clef `break`.

```
while (true) { [corps] }  
for (;;) { [corps] }
```

Figure 28 – Boucles infinies en C++

Définition

Une **boucle infini** est une boucle dont l'expression de condition reste vraie à chaque itération : le seul moyen d'en sortir est d'utiliser le mot-clef `break`.

```
while (true) { [corps] }  
for (;;) { [corps] }
```

Figure 28 – Boucles infinies en C++

Soit $n \in \mathbb{N}$.

```
<type de retour> <nom>(<type de __p1> [__p0], ..., <type de __pn> [__pn]);
```

Figure 29 – Déclaration d'une fonction

Définition

On appelle cette portion de la fonction **sa signature**.

Pour **définir** une fonction (la rendre utilisable), il faut lui ajouter un corps.

Soit $n \in \mathbb{N}$.

```
<type de retour> <nom>(<type de __p1> [__p0], ..., <type de __pn> [__pn]);
```

Figure 29 – Déclaration d'une fonction

Définition

On appelle cette portion de la fonction **sa signature**.

Pour **définir** une fonction (la rendre utilisable), il faut lui ajouter un corps.

Soit $n \in \mathbb{N}$.

```
<type de retour> <nom>(<type de __p1> [__p0], ..., <type de __pn> [__pn]);
```

Figure 29 – Déclaration d'une fonction

Définition

On appelle cette portion de la fonction **sa signature**.

Pour **définir** une fonction (la rendre utilisable), il faut lui ajouter un corps.

Exemple

```
1 int pow(unsigned int, unsigned int);
```

Dans ce contexte, le nom des paramètres est optionnel : en effet, on n'utilise pas ces derniers ici.

Cependant, celui-ci porte une information sur la nature du paramètre auquel il est associé. Ainsi, la déclaration suivante est préférable :

```
1 int pow(unsigned int __n, unsigned int __exp);
```

Exemple

```
1 int pow(unsigned int, unsigned int);
```

Dans ce contexte, le nom des paramètres est optionnel : en effet, on n'utilise pas ces derniers ici.

Cependant, celui-ci porte une information sur la nature du paramètre auquel il est associé.

Ainsi, la déclaration suivante est préférable :

```
1 int pow(unsigned int __n, unsigned int __exp);
```

Exemple

```
1 int pow(unsigned int, unsigned int);
```

Dans ce contexte, le nom des paramètres est optionnel : en effet, on n'utilise pas ces derniers ici.

Cependant, celui-ci porte une information sur la nature du paramètre auquel il est associé. Ainsi, la déclaration suivante est préférable :

```
1 int pow(unsigned int __n, unsigned int __exp);
```

Pour bien comprendre l'intérêt d'une déclaration

Exemple

```
1 #include <iostream>
2 #include <string>
3
4 void print(unsigned int __n)
5 {
6     std::cout << __n << "! = " << factorielle(__n) << '\n';
7 }
8
9 int factorielle(unsigned int __n)
10 {
11     int ret = 1;
12
13     for (; __n > 1; --__n) {
14         ret *= __n;
15     }
16
17     return ret;
18 }
19
20 int main()
21 {
22     print(5);
23 }
```

Que va-t-il se passer et pourquoi ?

Pour bien comprendre l'intérêt d'une déclaration

Exemple

```
1 #include <iostream>
2 #include <string>
3
4 void print(unsigned int __n)
5 {
6     std::cout << __n << "! = " << factorielle(__n) << '\n';
7 }
8
9 int factorielle(unsigned int __n)
10 {
11     int ret = 1;
12
13     for (; __n > 1; --__n) {
14         ret *= __n;
15     }
16
17     return ret;
18 }
19
20 int main()
21 {
22     print(5);
23 }
```

Que va-t-il se passer et pourquoi ?

Le compilateur traite le programme dans l'ordre de lecture et `factorielle()` n'est pas déclaré.

Exemple

Comment corriger le problème ?

Exemple

Comment corriger le problème ?

```
1 #include <iostream>
2 #include <string>
3
4 int factorielle(unsigned int); // on rajoute cette ligne
5
6 void print(unsigned int __n)
7 {
8     std::cout << __n << " != " << factorielle(__n) << '\n';
9 }
10
11 int factorielle(unsigned int __n)
12 {
13     int ret = 1;
14
15     for (; __n > 1; --__n) {
16         ret *= __n;
17     }
18
19     return ret;
20 }
21
22 int main()
23 {
24     print(5);
25 }
```

Il est possible de spécifier une valeur par défaut pour un paramètre.

```
<type> [identificateur] = <valeur par défaut>
```

Figure 30 – Paramètre par défaut

Ainsi, lors de l'appel de la fonction, si aucune valeur pour `identificateur` n'est spécifiée, ce dernier prend pour valeur `valeur par défaut`.

Il est possible de spécifier une valeur par défaut pour un paramètre.

```
<type> [identificateur] = <valeur par défaut>
```

Figure 30 – Paramètre par défaut

Ainsi, lors de l'appel de la fonction, si aucune valeur pour `identificateur` n'est spécifiée, ce dernier prend pour valeur `valeur par défaut`.

Il est possible de spécifier une valeur par défaut pour un paramètre.

```
<type> [identificateur] = <valeur par défaut>
```

Figure 30 – Paramètre par défaut

Ainsi, lors de l'appel de la fonction, si aucune valeur pour `identificateur` n'est spécifiée, ce dernier prend pour valeur `valeur par défaut`.

Exemple

```
1 char mystere(char, char = 3);
2
3 char mystere(char __c, char __o)
4 {
5     constexpr char lenght = 26; // équivalent à 'const' en Rust
6
7     char begin = (__c >= 'a') ? 'a' : 'A';
8     char end = (__c <= 'Z') ? 'Z' : 'z';
9
10    __c += __o;
11
12    if (__c > end) {
13        __c -= lenght;
14    } else if (__c < begin) {
15        __c += lenght;
16    }
17
18    return __c;
19 }
```

Que peut-on dire de la fonction `mystere()` ?

Un exemple pour la route

Exemple

```
1 char mystere(char, char = 3);
2
3 char mystere(char __c, char __o)
4 {
5     constexpr char lenght = 26; // équivalent à 'const' en Rust
6
7     char begin = (__c >= 'a') ? 'a' : 'A';
8     char end = (__c <= 'Z') ? 'Z' : 'z';
9
10    __c += __o;
11
12    if (__c > end) {
13        __c -= lenght;
14    } else if (__c < begin) {
15        __c += lenght;
16    }
17
18    return __c;
19 }
```

Que peut-on dire de la fonction `mystere()` ?

À condition que `__c` soit une lettre (majuscule ou minuscule), la fonction retourne le `__o`-ième (potentiellement négatif) caractère suivant celui-ci. Si `__o` n'est pas spécifié, il s'agit de l'encodage d'un caractère avec le chiffre de César.

Attention

Il n'est pas possible d'avoir de paramètre sans valeur par défaut à droite du premier paramètre ayant une valeur par défaut.

Exemple

Ainsi, une fonction ayant la signature suivante est invalide.

```
1 void foo(char, char = 'a', char, char);
```

Attention

Il n'est pas possible d'avoir de paramètre sans valeur par défaut à droite du premier paramètre ayant une valeur par défaut.

Exemple

Ainsi, une fonction ayant la signature suivante est invalide.

```
1 void foo(char, char = 'a', char, char);
```


Merci pour votre écoute.