Formation $C++17 n^{\circ} 02$

Portées, opérateurs, structures de contrôle et fonctions

ROSSILLOL-LARUELLE Mattéo

24 janvier 2024

- Avant-propos
- 2 Portées
 - Une vision générale de la chose
 - Les espaces de noms
- 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-propos
- 2 Portées
 - Une vision générale de la chose
 - Les espaces de noms
- 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-propos
- 2 Portées
 - Une vision générale de la chose
 - Les espaces de noms
- 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

R.–L. Mattéo Formation C++ 17 24 janvier 2024 2 /

- Avant-propos
- 2 Portées
 - Une vision générale de la chose
 - Les espaces de noms
- Opérateurs
 - Similitudes et différences avec Python
 - Les opérateurs bit-à-bit
- Structures de contrôle
 - Avant de commencer
 - Les instructions de conditionnement
 - Les boucles
- 5 Fonctions

- Avant-propos
- 2 Portées
 - Une vision générale de la chose
 - Les espaces de noms
- Opérateurs
 - Similitudes et différences avec Python
 - Les opérateurs bit-à-bit
- Structures de contrôle
 - Avant de commencer
 - Les instructions de conditionnement
 - Les boucles
- 5 Fonctions



Avant-propos

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

4/57

Portées

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 par }.

R.–L. Mattéo Formation C++17

Portées

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 par }.

5 / 57

Quelques exemples

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.

Quelques exemples

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
```

6 / 57

Portées Les espaces de noms

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.

R.–L. Mattéo Formation C++ 17

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.

8 / 57

R.–L. Mattéo Formation C++ 17

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.

Déclaration d'un espace de noms

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

Figure 1 – Déclaration d'un espace de noms

On déclare un espace de noms portant le nom <u>identifiant</u> 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

Déclaration d'un espace de noms

```
namespace <identifiant> { <corps> }
Figure 1 - Déclaration d'un espace de noms
```

On déclare un espace de noms portant le nom <u>identifiant</u> 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

Un petit cas concret

Exemple

```
#include <cstdint>

using namespace std;

namespace lib

uint8_t un = 1;

bool est_pair(uint8_t __a)

return ((_a % 2) == 0);

}

int main()

bool un_booleen = lib ::est_pair(2 + lib ::un);

bool un_booleen = lib ::est_pair(2 + lib ::un);

bool un_booleen
```

Utilisation d'un espace de noms

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>
```

11 / 57

R.-L. Mattéo Formation C++ 17

Utilisation d'un espace de noms

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>

11 / 57

R.–L. Mattéo Formation C++ 17

Utilisation d'un espace de noms

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>
```

R.-L. Mattéo

Un exemple

Exemple

En reprenant le code précédent, on se ramène à :

```
1 #include <cstdint>
   using namespace std;
  namespace lib
       uint8_t un = 1;
       bool est_pair(uint8_t __a)
           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
       // si 'using namespace lib;' avait été ici, on aurait eu une erreur à la compilation
23 }
```

Une petite précision

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>

13 / 57

Une petite précision

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>

13 / 57

Une petite précision

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>

13 / 57

Exemple

Avec cette nouvelle syntaxe, l'exemple précédent donnerait :

```
1 #include <cstdint>
   using namespace std;
5 namespace lib
       uint8\_t un = 1;
       bool est_pair(uint8_t __a)
           return ((__a % 2) == 0);
13 }
15 int main()
       using lib ::est_pair;
       using lib::un;
19
20
       bool un_booleen = est_pair(2 + un);
21 }
```

La révélation

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;

15 / 57

La révélation

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;

15 / 57

La révélation

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;.

15 / 57

Les espaces de noms inline

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

16 / 57

Un petit exemple

Exemple

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

```
1 #include <cstdint>
   namespace lib
       inline namespace variables
           inline namespace nombres { std::uint8_t un = 1; }
       inline namespace fonctions
           bool est_pair(std::uint8_t __a) { return ((__a % 2) == 0); }
14
15
   int main()
       using lib::un;
19
       bool un_booleen = lib ::est_pair(2 + un);
21 }
```

Les alias

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>

18 / 57

Les alias

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>

18 / 57

Une facilité syntaxique

Les deux codes suivant sont strictement équivalent :

19/57

24 janvier 2024

Pour faire le point sur tout ça

Exemple

```
1 #include <cstdint>
 2 #include <iostream>
4 namespace lib
       namespace a::variables::cardinaux { std::uint16 t premier = 0; }
       namespace b { namespace variables::cardinaux { std::uint16 t premier = 1: } }
       inline namespace c { namespace variables ::cardinaux { std ::uint16_t premier = 2; } }
9
   using lib::variables::cardinaux::premier;
12 namespace a = lib::b;
14 int main(int argc, char* argv[])
15
16
       using namespace lib::a;
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';</pre>
22 }
```

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

20 / 57

Pour faire le point sur tout ça

Exemple

```
1 #include <cstdint>
 2 #include <iostream>
4 namespace lib
       namespace a::variables::cardinaux { std::uint16 t premier = 0; }
       namespace b { namespace variables::cardinaux { std::uint16 t premier = 1: } }
       inline namespace c { namespace variables ::cardinaux { std ::uint16_t premier = 2; } }
9
   using lib ::variables ::cardinaux ::premier;
   namespace a = lib ::b;
14 int main(int argc, char* argv[])
15
16
       using namespace lib::a;
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';</pre>
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

R.–L. Mattéo Formation C++ 17 24 janvier 2024 21 / 57

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 a pour valeur b.

Exemple

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

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

R.–L. Mattéo Formation C++ 17 24 janvier 2024 22/57

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 a pour valeur b.

Exemple

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

22 / 57

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 a pour valeur b.

Exemple

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

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

22 / 57

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 a pour valeur b.

Exemple

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

Il sera imprimé « 5 ».

22 / 57

Opérateurs étudiés en Python

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

Figure 8 – Opérateurs unitaires

$$a = b$$

Figure 9 – Opérateur de copie

23 / 57

Opérateurs étudiés en Python

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

Figure 11 – Opérateurs d'affectation associés

23 / 57

24 janvier 2024

R.–L. Mattéo Formation C++ 17

Opérateurs étudiés en Python

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

R.–L. Mattéo

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.

1 #include <cstdint> 2 #include <iostream> 3 4 int main() 5 { 6 std::uint32_t a = 0; 7

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

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

24 / 57

R.–L. Mattéo Formation C++ 17

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';
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 }</pre>
```

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

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

24 / 57

Opérateurs d'incrémentation et de décrémentation

Figure 13 – Opérateurs de préincrémentation et de prédécrémentation

Figure 14 – Opérateurs de postincrémentation et de postdécrémentation

Les expressions précédentes sont équivalentes à :

Figure 16 – Opérateurs de postincrémentation et de postdécrémentation (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.

25 / 57

Opérateurs d'incrémentation et de décrémentation

Figure 13 – Opérateurs de préincrémentation et de prédécrémentation

Figure 14 – Opérateurs de postincrémentation et de postdécrémentation

Les expressions précédentes sont équivalentes à :

Figure 15 – Opérateurs de préincrémentation et de Figure 16 – Opérateurs de postincrémentation et de prédécrémentation (équivalent)

postdécrémentation (équivalent)

25 / 57

Opérateurs d'incrémentation et de décrémentation

Figure 13 – Opérateurs de préincrémentation et de prédécrémentation

Figure 14 – Opérateurs de postincrémentation et de postdécrémentation

Les expressions précédentes sont équivalentes à :

Figure 15 – Opérateurs de préincrémentation et de Figure 16 – Opérateurs de postincrémentation et de prédécrémentation (équivalent)

postdécrémentation (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.

25 / 57

Pour bien saisir la différence

Exemple

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

26 / 57

R.-L. Mattéo Formation C++ 17

Pour bien saisir la différence

Exemple

1 #include <cstdint>

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

R.–L. Mattéo Formation C++ 17 24 janvier 2024 27/57

Opérateurs Les opérateurs bit-à-bit Éléments de syntaxe

Opérateurs bit-à-bit

~a

Figure 17 – Opérateur unitaire

a & b		
a b		
a ^ b		
a << b		
a >> b		

Figure 18 – Opérateurs arithmétiques

a &= b a |= b a ^= b a <<= b a >>= b

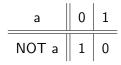
Figure 19 – Opérateurs d'affectation associés

R.-L. Mattéo

Opérateurs Les opérateurs bit-à-bit Tables de vérité

R.–L. Mattéo Formation C++ 17 24 janvier 2024 30 / 57

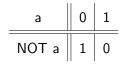
NOT



Exemple

Que sera-t-il imprimé en console?

NOT



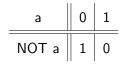
Exemple

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

Que sera-t-il imprimé en console?

R.-L. Mattéo Formation C++ 17

NOT



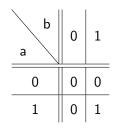
Exemple

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

Que sera-t-il imprimé en console?

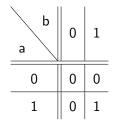
Il sera imprimé « 254 ».





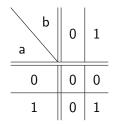
```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a & b) << '\n';
9 }</pre>
```

Que sera-t-il imprimé en console?



```
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 }</pre>
```

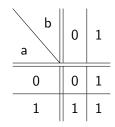
Que sera-t-il imprimé en console?



```
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 }</pre>
```

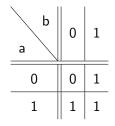
Que sera-t-il imprimé en console?

Il sera imprimé « 1 ».

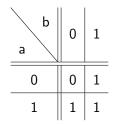


```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7     std::cout << static_cast<int>(a | b) << '\n';
9 }</pre>
```

Que sera-t-il imprimé en console?



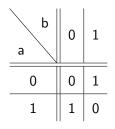
Que sera-t-il imprimé en console?



```
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 }</pre>
```

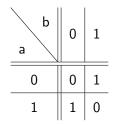
Que sera-t-il imprimé en console?

Il sera imprimé « 255 ».



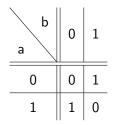
```
1 #include <iostream>
2
3 int main()
4 {
5     std::uint8_t a = 0b11111111;
6     std::uint8_t b = 0b00000001;
7
8     std::cout << static_cast<int>(a ^ b) << '\n';
9 }</pre>
```

Que sera-t-il imprimé en console?



```
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 }</pre>
```

Que sera-t-il imprimé en console?



```
1 #include <iostream>
2
3 int main()
4 {
5     std ::uint8_t a = 0b111111111;
6     std ::uint8_t b = 0b00000001;
7
8     std ::cout << static_cast<int>(a ^ b) << '\n';
9 }</pre>
```

Que sera-t-il imprimé en console?

Il sera imprimé « 254 ».

Décalage de bits

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 .

35 / 57

Un petit exemple

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 }</pre>
```

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

Un petit exemple

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 }</pre>
```

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

Il sera imprimé « 32 », puis « 0 ».

Structures de contrôle Avant de commencer

Structures de contrôle

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.

38 / 57

Structures de contrôle Les instructions de conditionnement

if, else if et else

Figure 20 – Instructions en C++

Figure 21 – Instructions en Python

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

40 / 57

if, else if et else

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

Figure 20 - Instructions en C++
if <condition>: [corps]
elif <condition>: [corps]

Figure 21 - Instructions en Python
```

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

40 / 57

Trop de else if

On peut transformer le code

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

12 }

41 / 57

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

Figure 22 - Le statement switch
```

- initialisation est une expression ou une déclaration dont la porté 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>

42 / 57

- initialisation est une expression ou une déclaration dont la porté 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 porté 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>

R.–L. Mattéo

- initialisation est une expression ou une déclaration dont la porté 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 porté 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 porté 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>

42 / 57

Une petite précision

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*.

R.-L. Mattéo Formation C++ 17 24 janvier 2024 43/57

Une petite précision

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*.

R.-L. Mattéo Formation C++ 17 24 janvier 2024 43/57

Une petite précision

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*.

<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?

44 / 57

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

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

44 / 57

```
<condition> ? <a> : <b>
```

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 \ll ((argc = 1) : argv[0] : argv[1]) \ll '\n';
```

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

R.-L. Mattéo Formation C++ 17 44 / 57

```
<condition> ? <a> : <b>
```

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

Quelques boucles déjà connues grâce à Python

46 / 57

La boucle do-while

```
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.

24 janvier 2024

```
for ([déclaration ou expression]; [condition]; [expression]) { [corps] }
Figure 27 - La boucle for
```

Le fonctionnement est le suivant

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

48 / 57

Le fonctionnement est le suivant :

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

48 / 57

Le fonctionnement est le suivant :

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

48 / 57

Le fonctionnement est le suivant :

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

48 / 57

Le fonctionnement est le suivant :

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

48 / 57

Un petit exemple pour la route

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()?

49 / 57

Un petit exemple pour la route

Exemple

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

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

Il s'agit de la fonction factorielle.

Comment écrire une boucle infini

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++

50 / 57

24 janvier 2024

Comment écrire une boucle infini

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++

50 / 57

Déclaration et définition de fonction

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

 ([__p0], ..., n> [__p
$$n$$
]);

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.

51 / 57

24 janvier 2024

Déclaration et définition de fonction

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

 ([__p0], ..., n> [__p
$$n$$
]);

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.

51 / 57

Déclaration et définition de fonction

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.

Un exemple de déclaration de fonction

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.

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

52 / 57

Un exemple de déclaration de fonction

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);
```

52 / 57

24 janvier 2024

Un exemple de déclaration de fonction

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);
```

52 / 57

R.–L. Mattéo Formation C++ 17

Exemple

```
1 #include <iostream>
 2 #include <string>
4 void print(unsigned int ___n)
       std::cout << __n << "! = " << factorielle(__n) << '\n';
9 int factorielle (unsigned int __n)
10 {
       int ret = 1;
       for (; _{n} > 1; _{n} 
           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?

Exemple

```
1 #include <iostream>
 2 #include <string>
4 void print(unsigned int ___n)
       std::cout << __n << "! = " << factorielle(__n) << '\n';
9 int factorielle (unsigned int ___n)
10
11
       int ret = 1;
       for (; __n > 1; --_n) {
           ret *= n;
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?

24 janvier 2024

R.–L. Mattéo Formation C++ 17

Exemple

Comment corriger le problème?

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

Paramètre par défaut

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

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

54 / 57

R.-L. Mattéo Formation C++ 17 24 janvier 2024

Paramètre par défaut

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

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

54 / 57

R.-L. Mattéo Formation C++ 17 24 janvier 2024

Paramètre 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.

54 / 57

24 janvier 2024

R.-L. Mattéo Formation C++ 17

Un exemple pour la route

Exemple

```
1 char mystere(char, char = 3);
3 char mystere(char __c, char __o)
        constexpr char lenght = 26; // équivalent à 'const' en Rust
        char begin = (__c >= 'a') ? 'a' : 'A';
char end = (__c <= 'Z') ? 'Z' : 'z';</pre>
10
        __c += __o;
11
        if (__c > end) {
          __c -= lenght;
        } 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);
   char mystere(char __c, char __o)
         constexpr char lenght = 26; // équivalent à 'const' en Rust
        char begin = (__c >= 'a') ? 'a' : 'A';
char end = (__c <= 'Z') ? 'Z' : 'z';
10
        __c += __o;
        if (__c > end) {
             __c -= lenght;
         } else if ( c < begin) {</pre>
15
             \underline{\phantom{a}}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.

Une petite précision avant de finir

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);
```

56 / 57

24 janvier 2024

R.-L. Mattéo Formation C++ 17

Une petite précision avant de finir

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);
```

56 / 57

Merci pour votre écoute.

R.–L. Mattéo Formation C++17 24 janvier 2024 57/57