Formation C++ 17
Cours n° 04
Retour sur la formation C

Mattéo ROSSILLOL–LARUELLE

20 mars 2024

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.

Différences avec le C
Surcharge de fonction

Surcharge de fonction

Définition

Si le nom d'une fonction réfère à plus d'une entité, alors cette fonction est dite surchargée (ou *overloaded*).

Remarque

Le compilateur doit alors déterminer quelle surcharge (ou *overload*) appeler. En d'autres termes, la surcharge avec les paramètres correspondant le mieux est appelée.

Remarque

En C, cette notion n'existe pas : il est impossible de surcharger des fonctions. Ainsi, deux fonctions distinctes doivent avoir nécessairement des noms distincts.

Surcharge de fonction

Définition

Si le nom d'une fonction réfère à plus d'une entité, alors cette fonction est dite surchargée (ou *overloaded*).

Remarque

Le compilateur doit alors déterminer quelle surcharge (ou *overload*) appeler. En d'autres termes, la surcharge avec les paramètres correspondant le mieux est appelée.

Remarque

En C, cette notion n'existe pas : il est impossible de surcharger des fonctions. Ainsi, deux fonctions distinctes doivent avoir nécessairement des noms distincts.

Surcharge de fonction

Définition

Si le nom d'une fonction réfère à plus d'une entité, alors cette fonction est dite surchargée (ou *overloaded*).

Remarque

Le compilateur doit alors déterminer quelle surcharge (ou *overload*) appeler. En d'autres termes, la surcharge avec les paramètres correspondant le mieux est appelée.

Remarque

En C, cette notion n'existe pas : il est impossible de surcharger des fonctions. Ainsi, deux fonctions distinctes doivent avoir nécessairement des noms distincts.

Différences avec le C

Surcharge de fonction

Exemples

Un petit exemple

Exemple

```
1  #include <iostream>
2
3  int foo(bool) { return 0; }
4  int foo(int) { return 1; }
5
6  int main()
7  {
8      int a = 42;
9
10     std::cout << foo(a) << '\n';
11 }
```

Dans l'exemple ci-dessus que sera-t-il imprimé en console ?

Un petit exemple

Exemple

```
1  #include <iostream>
2
3  int foo(bool) { return 0; }
4  int foo(int) { return 1; }
5
6  int main()
7  {
8      int a = 42;
9
10     std::cout << foo(a) << '\n';
11 }
```

Dans l'exemple ci-dessus que sera-t-il imprimé en console ?

Il sera imprimé « 1 ».

Un petit rappel

Attention

On rappelle cependant que le C++ est un langage faiblement typé.

Exemple

```
1  #include <iostream>
2
3  int foo(bool) { return 0; }
4
5  int main()
6  {
7      int a = 42;
8
9      std::cout << foo(a) << '\n';
10 }
```

Dans l'exemple ci-dessus que sera-t-il imprimé en console ?

Un petit rappel

Attention

On rappelle cependant que le C++ est un langage faiblement typé.

Exemple

```
1  #include <iostream>
2
3  int foo(bool) { return 0; }
4
5  int main()
6  {
7      int a = 42;
8
9      std::cout << foo(a) << '\n';
10 }
```

Dans l'exemple ci-dessus que sera-t-il imprimé en console ?

Il sera imprimé « 0 ». En effet, un entier peut être implicitement converti en booléen.

Différences avec le C
Déclaration implicite de fonction

Déclaration implicite de fonction

Définition

Si une fonction n'a pas été déclaré avant son premier appel, celle-ci est déclaré implicitement si le compilateur déduit une signature de cet appel.

Remarque

Même si cela est à proscrire, en C (du moins, pour les standards antérieurs à C 99), il est possible d'avoir des déclarations implicites.

Remarque

En C++, il n'existe pas de déclaration implicite.

Déclaration implicite de fonction

Définition

Si une fonction n'a pas été déclaré avant son premier appel, celle-ci est déclaré implicitement si le compilateur déduit une signature de cet appel.

Remarque

Même si cela est à proscrire, en C (du moins, pour les standards antérieurs à C 99), il est possible d'avoir des déclarations implicites.

Remarque

En C++, il n'existe pas de déclaration implicite.

Déclaration implicite de fonction

Définition

Si une fonction n'a pas été déclaré avant son premier appel, celle-ci est déclaré implicitement si le compilateur déduit une signature de cet appel.

Remarque

Même si cela est à proscrire, en C (du moins, pour les standards antérieurs à C 99), il est possible d'avoir des déclarations implicites.

Remarque

En C++, il n'existe pas de déclaration implicite.

Un petit exemple

Exemple

```c
// On n'inclue pas stdio.h

int main()
{
    printf("Hello world!");

    return 0;
}
```

Dans l'exemple (écrit en C) ci-dessus, le compilateur pourrait n'imprimer sur la console qu'un avertissement et non se terminer avec une erreur.

Différences avec le C
Allocation dynamique

Différences avec le C

Allocation dynamique

Allocation dynamique en C

Allocation dynamique en C

En C, pour allouer ou libérer dynamiquement une ressource, on utilise les fonctions suivantes se trouvant dans l'entête *stdlib.h* :

```
void* malloc(size_t size);
void* calloc(size_t num, size_t size);
void* realloc(void* ptr, size_t new_size);
void* aligned_alloc(size_t alignment, size_t size);
```

Figure 1 – Allocation dynamique en C

```
void free(void* ptr);
```

Figure 2 – Libération dynamique en C

Allocation dynamique en C

En C, pour allouer ou libérer dynamiquement une ressource, on utilise les fonctions suivantes se trouvant dans l'entête *stdlib.h* :

```
void* malloc(size_t size);
void* calloc(size_t num, size_t size);
void* realloc(void* ptr, size_t new_size);
void* aligned_alloc(size_t alignment, size_t size);
```

Figure 1 – Allocation dynamique en C

```
void free(void* ptr);
```

Figure 2 – Libération dynamique en C

Allocation dynamique en C

En C, pour allouer ou libérer dynamiquement une ressource, on utilise les fonctions suivantes se trouvant dans l'entête *stdlib.h* :

```
void* malloc(size_t size);
void* calloc(size_t num, size_t size);
void* realloc(void* ptr, size_t new_size);
void* aligned_alloc(size_t alignment, size_t size);
```

Figure 1 – Allocation dynamique en C

```
void free(void* ptr);
```

Figure 2 – Libération dynamique en C

Une petite observation

Remarque

Le type de retour des différentes fonctions est, à chaque fois, un `void*`. En effet, ce type, comme expliqué dans le cours précédent, correspond à un type pointeur générique.
Grossièrement, comme la taille en mémoire caractérise la complétude d'un type et que tous les pointeurs font la même taille, `void*` est bien un type complet même si `void` ne l'est pas.
De plus, il existe une conversion implicite de `type*` vers `void*`.

Différences avec le C

Allocation dynamique

Allocation dynamique en C++

Allocation dynamique en C++

En C++, on préfère utiliser les deux opérateurs suivants :

```
new
new[]
```

Figure 3 – Allocation dynamique en C++

```
delete
delete[]
```

Figure 4 – Libération dynamique en C++

Allocation dynamique en C++

En C++, on préfère utiliser les deux opérateurs suivants :

```
new
new[]
```

Figure 3 – Allocation dynamique en C++

```
delete
delete[]
```

Figure 4 – Libération dynamique en C++

Une petite mise en pratique

Exemple

```cpp
#include <iostream>

int main()
{
    int* a = new int(10);

    std::cout << "a = " << *a << '\n';

    delete a;
}
```

Exemple

```cpp
#include <cstddef>

int main()
{
    int* a = new int[10];

    for (std::size_t i = 0; i != 10; ++i) {
        a[i] = i;
    }

    delete[] a;
}
```

Utilisation avancée de ces opérateurs

Remarque

Il existe des utilisations plus avancées de ces deux opérateurs que l'on détaillera peut-être plus tard dans la formation.
On peut citer :

- allocation dans un tampon avec le *placement new*,
- allocation sans exception,
- allocation avec contrainte d'alignement,
- *handler*.

Pour les curieux, voir `https://en.cppreference.com/w/cpp/memory/new`.

Utilisation avancée de ces opérateurs

Remarque

Il existe des utilisations plus avancées de ces deux opérateurs que l'on détaillera peut-être plus tard dans la formation.
On peut citer :

- allocation dans un tampon avec le *placement new*,
- allocation sans exception,
- allocation avec contrainte d'alignement,
- *handler*.

Pour les curieux, voir `https://en.cppreference.com/w/cpp/memory/new`.

Utilisation avancée de ces opérateurs

Remarque

Il existe des utilisations plus avancées de ces deux opérateurs que l'on détaillera peut-être plus tard dans la formation.
On peut citer :

- allocation dans un tampon avec le *placement new*,
- allocation sans exception,
- allocation avec contrainte d'alignement,
- *handler*.

Pour les curieux, voir `https://en.cppreference.com/w/cpp/memory/new`.

Utilisation avancée de ces opérateurs

Remarque

Il existe des utilisations plus avancées de ces deux opérateurs que l'on détaillera peut-être plus tard dans la formation.
On peut citer :

- allocation dans un tampon avec le *placement new*,
- allocation sans exception,
- allocation avec contrainte d'alignement,
- *handler*.

Pour les curieux, voir `https://en.cppreference.com/w/cpp/memory/new`.

Utilisation avancée de ces opérateurs

Remarque

Il existe des utilisations plus avancées de ces deux opérateurs que l'on détaillera peut-être plus tard dans la formation.
On peut citer :

- allocation dans un tampon avec le *placement new*,
- allocation sans exception,
- allocation avec contrainte d'alignement,
- *handler*.

Pour les curieux, voir `https://en.cppreference.com/w/cpp/memory/new`.

Différences avec le C
Opérateurs de conversion

Différences avec le C

Opérateurs de conversion

Les conversions explicites en C++

Les conversions explicites en C++

Le C++ est doté de plusieurs opérateurs de conversion différents ayant chacun un objectif propre.

```
const_cast<<type cible>>(<expression>)
static_cast<<type cible>>(<expression>)
dynamic_cast<<type cible>>(<expression>)
reinterpret_cast<<type cible>>(<expression>)
```

Figure 5 – Différents opérateurs de conversions en C++

const_cast

```
const_cast<<type cible>>(<expression>)
```

Figure 6 – Utilisation de `const_cast`

Il permet la conversion de `expression` en un type `type cible` de constance différente.

Exemple

```cpp
#include <iostream>

int main()
{
    const int* p_a = new int(5);

    std::cout << *p_a << '\n';

    *const_cast<int*>(p_a) = 10;

    std::cout << *p_a << '\n';
}
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?

```
const_cast<<type cible>>(<expression>)
```

Figure 6 – Utilisation de `const_cast`

Il permet la conversion de `expression` en un type `type cible` de constance différente.

Exemple

```
1  #include <iostream>
2
3  int main()
4  {
5      const int* p_a = new int(5);
6
7      std::cout << *p_a << '\n';
8
9      *const_cast<int*>(p_a) = 10;
10
11     std::cout << *p_a << '\n';
12 }
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?

```
const_cast<<type cible>>(<expression>)
```

Figure 6 – Utilisation de `const_cast`

Il permet la conversion de `expression` en un type `type cible` de constance différente.

Exemple

```
1  #include <iostream>
2
3  int main()
4  {
5      const int* p_a = new int(5);
6
7      std::cout << *p_a << '\n';
8
9      *const_cast<int*>(p_a) = 10;
10
11     std::cout << *p_a << '\n';
12 }
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?

Il sera imprimé « 5 », puis « 10 ».

static_cast

```
static_cast<<type cible>>(<expression>)
```

Figure 7 – Utilisation de `static_cast`

Il permet la conversion de `expression` en un type `type cible` en utilisant une combinaison de règles de conversion implicites et d'autres fournies par l'utilisateur.

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     double a = 1.5;
6
7     std::cout << static_cast<int>(a) << '\n';
8 }
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?

static_cast

```
static_cast<<type cible>>(<expression>)
```

Figure 7 – Utilisation de `static_cast`

Il permet la conversion de `expression` en un type `type cible` en utilisant une combinaison de règles de conversion implicites et d'autres fournies par l'utilisateur.

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     double a = 1.5;
6
7     std::cout << static_cast<int>(a) << '\n';
8 }
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?

static_cast

```
static_cast<<type cible>>(<expression>)
```

Figure 7 – Utilisation de `static_cast`

Il permet la conversion de `expression` en un type `type cible` en utilisant une combinaison de règles de conversion implicites et d'autres fournies par l'utilisateur.

Exemple

```
1 #include <iostream>
2
3 int main()
4 {
5     double a = 1.5;
6
7     std::cout << static_cast<int>(a) << '\n';
8 }
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?
Il sera imprimé « 1 ».

reinterpret_cast

```
reinterpret_cast<<type cible>>(<expression>)
```

Figure 8 – Utilisation de `reinterpret_cast`

Il permet la conversion de `expression` en un type `type cible` en réinterprétant la représentation binaire sous-jacente.

Exemple

```
1 #include <cstdint>
2 #include <iostream>
3
4 int main()
5 {
6     std::uint32_t a = 0x00544942;
7
8     std::cout << reinterpret_cast<char*>(&a) << '\n';
9 }
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?

```
reinterpret_cast<<type cible>>(<expression>)
```

Figure 8 – Utilisation de `reinterpret_cast`

Il permet la conversion de `expression` en un type `type cible` en réinterprétant la représentation binaire sous-jacente.

Exemple

```
1 #include <cstdint>
2 #include <iostream>
3
4 int main()
5 {
6     std::uint32_t a = 0x00544942;
7
8     std::cout << reinterpret_cast<char*>(&a) << '\n';
9 }
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?

```
reinterpret_cast<<type cible>>(<expression>)
```

Figure 8 – Utilisation de `reinterpret_cast`

Il permet la conversion de `expression` en un type `type cible` en réinterprétant la représentation binaire sous-jacente.

Exemple

```
1 #include <cstdint>
2 #include <iostream>
3
4 int main()
5 {
6     std::uint32_t a = 0x00544942;
7
8     std::cout << reinterpret_cast<char*>(&a) << '\n';
9 }
```

Dans l'exemple ci-dessus que sera-t-il imprimer sur la ligne de commande ?
En supposant que l'on travaille sur une machine petit-boutiste, il sera imprimé « BIT ».

Différences avec le C

Opérateurs de conversion

Les conversions explicites en C

Les conversions explicites en C

```
(<type cible>) <expression>
<type cible> (<expression>)
```

Figure 9 – Conversion explicite en C

Le comportement est équivalent à une tentative de conversion dans l'ordre suivant :

1. `const_cast`
2. `static_cast`
3. `dynamic_cast`
4. `reinterpret_cast`

Les conversions explicites en C

```
(<type cible>) <expression>
<type cible> (<expression>)
```

Figure 9 – Conversion explicite en C

Le comportement est équivalent à une tentative de conversion dans l'ordre suivant :

1. `const_cast`
2. `static_cast`
3. `dynamic_cast`
4. `reinterpret_cast`

Les conversions explicites en C

```
(<type cible>) <expression>
<type cible> (<expression>)
```

Figure 9 – Conversion explicite en C

Le comportement est équivalent à une tentative de conversion dans l'ordre suivant :

1. `const_cast`
2. `static_cast`
3. `dynamic_cast`
4. `reinterpret_cast`

Les conversions explicites en C

```
(<type cible>) <expression>
<type cible> (<expression>)
```

Figure 9 – Conversion explicite en C

Le comportement est équivalent à une tentative de conversion dans l'ordre suivant :

1. `const_cast`
2. `static_cast`
3. `dynamic_cast`
4. `reinterpret_cast`

Les conversions explicites en C

```
(<type cible>) <expression>
<type cible> (<expression>)
```

Figure 9 – Conversion explicite en C

Le comportement est équivalent à une tentative de conversion dans l'ordre suivant :

1. `const_cast`
2. `static_cast`
3. `dynamic_cast`
4. `reinterpret_cast`

Les conversions explicites en C

```
(<type cible>) <expression>
<type cible> (<expression>)
```

Figure 9 – Conversion explicite en C

Le comportement est équivalent à une tentative de conversion dans l'ordre suivant :

1. `const_cast`
2. `static_cast`
3. `dynamic_cast`
4. `reinterpret_cast`

Différences avec le C
Mots clefs

Mots clefs

Bien que le C++ descends du C et que nombreux mots-clefs du C se retrouve également en C++, le C a suivi sa propre évolution : par conséquent, certains mots clefs existent en C et non en C++.

Remarque

Cependant, certains des mots clefs qui ont été introduit en C l'ont également été en C++ parallèlement même s'ils ne donc suivent pas nécessairement la même syntaxe.

Mots clefs

Bien que le C++ descends du C et que nombreux mots-clefs du C se retrouve également en C++, le C a suivi sa propre évolution : par conséquent, certains mots clefs existent en C et non en C++.

Remarque

Cependant, certains des mots clefs qui ont été introduit en C l'ont également été en C++ parallèlement même s'ils ne donc suivent pas nécessairement la même syntaxe.

Différences avec le C

Mots clefs

`restrict`

```
<type>* restrict
```

Figure 10 – Utilisation de `restrict`

Marquer un pointeur comme `restrict` indique au compilateur que l'entité visée par ce pointeur ne peut être lu ou modifié, directement ou indirectement que par ledit pointeur. Ce mot clef permet au compilateur certaines optimisations, en plus d'indiquer au lecteur le comportement de l'entité spécifiée.

Attention

Si les contraintes définies ci-dessus ne sont pas respectées, alors le comportement est indéfini.

Remarque

`restrict` caractérise le type : c'est-à-dire que `type* restrict` et `type*` sont deux types distincts.

```
<type>* restrict
```

Figure 10 – Utilisation de `restrict`

Marquer un pointeur comme `restrict` indique au compilateur que l'entité visée par ce pointeur ne peut être lu ou modifié, directement ou indirectement que par ledit pointeur. Ce mot clef permet au compilateur certaines optimisations, en plus d'indiquer au lecteur le comportement de l'entité spécifiée.

Attention

Si les contraintes définies ci-dessus ne sont pas respectées, alors le comportement est indéfini.

Remarque

`restrict` caractérise le type : c'est-à-dire que `type* restrict` et `type*` sont deux types distincts.

```
<type>* restrict
```

Figure 10 – Utilisation de `restrict`

Marquer un pointeur comme `restrict` indique au compilateur que l'entité visée par ce pointeur ne peut être lu ou modifié, directement ou indirectement que par ledit pointeur. Ce mot clef permet au compilateur certaines optimisations, en plus d'indiquer au lecteur le comportement de l'entité spécifiée.

Attention

Si les contraintes définies ci-dessus ne sont pas respectées, alors le comportement est indéfini.

Remarque

`restrict` caractérise le type : c'est-à-dire que `type* restrict` et `type*` sont deux types distincts.

```
<type>* restrict
```

Figure 10 – Utilisation de `restrict`

Marquer un pointeur comme `restrict` indique au compilateur que l'entité visée par ce pointeur ne peut être lu ou modifié, directement ou indirectement que par ledit pointeur. Ce mot clef permet au compilateur certaines optimisations, en plus d'indiquer au lecteur le comportement de l'entité spécifiée.

Attention

Si les contraintes définies ci-dessus ne sont pas respectées, alors le comportement est indéfini.

Remarque

`restrict` caractérise le type : c'est-à-dire que `type* restrict` et `type*` sont deux types distincts.

Une petite mise en contexte

Exemple

```
1  void copy(int n, int* restrict p, int* restrict q)
2  {
3      while (n--> 0) {
4          *p++ = *q++;
5      }
6  }
7
8  int main()
9  {
10     int d[100];
11
12     for (size_t i = 0; i != 50; ++i) {
13         d[i] = i;
14     }
15
16     copy(50, d + 50, d);
17     copy(50, d + 1, d);
18 }
```

Dans l'exemple ci-dessus que va-t-il se passer lors des deux appels à `copy()` ?

Une petite mise en contexte

Exemple

```
1  void copy(int n, int* restrict p, int* restrict q)
2  {
3      while (n--- > 0) {
4          *p++ = *q++;
5      }
6  }
7
8  int main()
9  {
10     int d[100];
11
12     for (size_t i = 0; i != 50; ++i) {
13         d[i] = i;
14     }
15
16     copy(50, d + 50, d);
17     copy(50, d + 1, d);
18 }
```

Dans l'exemple ci-dessus que va-t-il se passer lors des deux appels à `copy()` ?

Le premier valide car `p` et `q` ne référeront jamais au même élément ; cependant, ce n'est pas le cas du second appel.

Différences avec le C

Mots clefs

D'autres mots clefs

D'autres mots clefs

```
_Alignas
_Alignof
```

Figure 11 – Mots clefs relatifs à l'alignement

```
_Atomic
```

Figure 12 – Spécifieur pour l'atomicité

```
_Generic
```

Figure 13 – Mot clef pour la sélection générique

```
_Bool
```

Figure 14 – Type booléen

D'autres mots clefs

```
_Alignas
_Alignof
```

Figure 11 – Mots clefs relatifs à l'alignement

```
_Atomic
```

Figure 12 – Spécifieur pour l'atomicité

```
_Generic
```

Figure 13 – Mot clef pour la sélection générique

```
_Bool
```

Figure 14 – Type booléen

D'autres mots clefs

```
_Alignas
_Alignof
```

Figure 11 – Mots clefs relatifs à l'alignement

```
_Atomic
```

Figure 12 – Spécifieur pour l'atomicité

```
_Generic
```

Figure 13 – Mot clef pour la sélection générique

```
_Bool
```

Figure 14 – Type booléen

D'autres mots clefs

```
_Alignas
_Alignof
```

Figure 11 – Mots clefs relatifs à l'alignement

```
_Atomic
```

Figure 12 – Spécifieur pour l'atomicité

```
_Generic
```

Figure 13 – Mot clef pour la sélection générique

```
_Bool
```

Figure 14 – Type booléen

D'autres mots clefs

```
_Complex
_Imaginary
```

Figure 15 – Mots clefs relatifs aux nombres complexes

```
_Thread_local
```

Figure 16 – Spécifieur de durée de stockage

```
_Static_assert
```

Figure 17 – Mot clef pour les assertions statiques

D'autres mots clefs

```
_Complex
_Imaginary
```

Figure 15 – Mots clefs relatifs aux nombres complexes

```
_Thread_local
```

Figure 16 – Spécifieur de durée de stockage

```
_Static_assert
```

Figure 17 – Mot clef pour les assertions statiques

D'autres mots clefs

```
_Complex
_Imaginary
```

Figure 15 – Mots clefs relatifs aux nombres complexes

```
_Thread_local
```

Figure 16 – Spécifieur de durée de stockage

```
_Static_assert
```

Figure 17 – Mot clef pour les assertions statiques

Notions manquantes
Notion de structure

Notion de structure

```
struct [identifier];
```

Figure 18 – Déclaration d'une structure

```
struct [identifier] { [corps] };
```

Figure 19 – Définition d'une structure

Définition

Une structure est un type consistant en un agrégat de données.

Notion de structure

```
struct [identifier];
```

Figure 18 – Déclaration d'une structure

```
struct [identifier] { [corps] };
```

Figure 19 – Définition d'une structure

Définition

Une structure est un type consistant en un agrégat de données.

Un petit exemple

Exemple

```cpp
#include <cstdint>
#include <iostream>

struct Player { char* name; std::uint8_t exp; };

int main()
{
    Player matteo = { "Mattéo", 255 }; // initialisation aggrégat

    std::cout << matteo.name << '\n';
    std::cout << matteo.exp << '\n';
}
```

Que sera-t-il imprimé sur la console ?

Un petit exemple

Exemple

```cpp
#include <cstdint>
#include <iostream>

struct Player { char* name; std::uint8_t exp; };

int main()
{
    Player matteo = { "Mattéo", 255 }; // initialisation aggrégat

    std::cout << matteo.name << '\n';
    std::cout << matteo.exp << '\n';
}
```

Que sera-t-il imprimé sur la console ?

Il sera imprimé « Mattéo », puis « 255 ».

Notions manquantes
L'union fait la force

L'union fait la force

```
union [identifier] { [corps] };
```

Figure 20 – Déclaration d'un union

Définition

Un union est un type spécial de structure où seul un membre vit à la fois.

L'union fait la force

```
union [identifier] { [corps] };
```

Figure 20 – Déclaration d'un union

Définition

Un union est un type spécial de structure où seul un membre vit à la fois.

Un petit exemple

Exemple

```cpp
#include <cstdint>
#include <iostream>

union Union { std::uint32_t integer; char character; };

int main()
{
    Union u = {0x00544942};

    std::cout << &u.character << '\n';
}
```

Dans l'exemple ci-dessus que sera-t-il imprimé sur la ligne de commande ?

Un petit exemple

Exemple

```
1  #include <cstdint>
2  #include <iostream>
3
4  union Union { std::uint32_t integer; char character; };
5
6  int main()
7  {
8      Union u = {0x00544942};
9
10     std::cout << &u.character << '\n';
11 }
```

Dans l'exemple ci-dessus que sera-t-il imprimé sur la ligne de commande ?

Il sera imprimé « BIT ».

Notions manquantes

Nommer des constantes avec les énumérations

Nommer des constantes avec les énumérations

```
<enum> [identifier] : <base>;
```

Figure 21 – Déclaration d'une énumération

```
<enum> [identifier] { [corps] };
<enum> [identifier] : <base> { [corps] };
```

Figure 22 – Définition d'une énumération

- `<enum>` peut prendre les valeurs `enum`, `enum class` ou `enum struct`.
- Le type entier des valeurs définies est `<base>`.

Définition

Une énumération est un type distincts définissant une liste de valeurs nommées dans une plage donnée.

Nommer des constantes avec les énumérations

```
<enum> [identifier] : <base>;
```

Figure 21 – Déclaration d'une énumération

```
<enum> [identifier] { [corps] };
<enum> [identifier] : <base> { [corps] };
```

Figure 22 – Définition d'une énumération

- `<enum>` peut prendre les valeurs `enum`, `enum class` ou `enum struct`.
- Le type entier des valeurs définies est `<base>`.

Définition

Une énumération est un type distincts définissant une liste de valeurs nommées dans une plage donnée.

Nommer des constantes avec les énumérations

```
<enum> [identifier] : <base>;
```

Figure 21 – Déclaration d'une énumération

```
<enum> [identifier] { [corps] };
<enum> [identifier] : <base> { [corps] };
```

Figure 22 – Définition d'une énumération

- `<enum>` peut prendre les valeurs `enum`, `enum class` ou `enum struct`.
- Le type entier des valeurs définies est `<base>`.

Définition

Une énumération est un type distincts définissant une liste de valeurs nommées dans une plage donnée.

Nommer des constantes avec les énumérations

```
<enum> [identifier] : <base>;
```

Figure 21 – Déclaration d'une énumération

```
<enum> [identifier] { [corps] };
<enum> [identifier] : <base> { [corps] };
```

Figure 22 – Définition d'une énumération

- `<enum>` peut prendre les valeurs `enum`, `enum class` ou `enum struct`.
- Le type entier des valeurs définies est `<base>`.

Définition

Une énumération est un type distincts définissant une liste de valeurs nommées dans une plage donnée.

Un petit exemple

Exemple

```cpp
#include <cstdint>
#include <iostream>

enum Enum : uint8_t { first, second = 10, third };
enum class EnumClass { first, second = 10, third };

int main()
{
    int a = first;
    Enum b = 10;
    EnumClass c = EnumClass::first;

    std::cout << a << ", " << b << ", " << c << '\n';
}
```

Que va-t-il se passer et pourquoi ?

Merci pour votre écoute.