



Programmation C++ moderne

Dr. Eng. Chiheb Ameer ABID

[in /in/chiheb-ameur-abid](#)
✉ chiheb.abid@gmail.com

Les directives =default, et =delete

La directive =default

- La directive `=default` permet de générer automatiquement des fonctions membres spéciales
 - Le constructeur par défaut, le constructeur de copie, le constructeur de déplacement, l'opérateur d'affectation de copie, l'opérateur d'affectation de déplacement et le destructeur

```

1 class Fraction {
2 public:
3     Fraction() = default; // Initialiser aussi les attributs de la classe
4 private:
5     int m_numerator; // Valeur indéterminée
6     int m_denominator = 1;
7 };
    
```

- Le constructeur généré avec `=default`
 - Initialise les membres avec leurs initialisateurs respectifs
 - Appelle les constructeurs par défaut des membres objets

Plan

- Les nouveautés au niveau des classes en C++11/C++17
- Les conversions
- Programmation fonctionnelle en C++11/C++17

Les directives =default, et =delete

La directive =delete

- La directive `=delete` permet de supprimer une fonction membre spéciale
 - Le constructeur par défaut, le constructeur de copie, le constructeur de déplacement, l'opérateur d'affectation de copie, l'opérateur d'affectation de déplacement et le destructeur

```

1 struct NonCopiable { // Cette classe n'est pas copiable
2     ...
3     NonCopiable(NonCopiable const & copie) = delete;
4     NonCopiable& operator=(NonCopiable const & copie) = delete;
5 };
    
```

Héritage et délégation des constructeurs

Héritage des constructeurs

- ↳ L'héritage des constructeurs est un mécanisme permettant simplifier l'écriture des constructeurs d'une classe dérivée

☛ En C++ traditionnel

```

1 class Base {
2     Dependency* myDependency;
3 public:
4     Base(Dependency* dep) : myDependency(dep) {}
5 };
6
7 class Derived : public Base {
8 public:
9     //constructor does nothing except forwarding to base constructor
10    Derived(Dependency* dep) : Base(dep) {}
11 };

```

- ↳ En C++ moderne (depuis C++11)

```

1 class Derived : public Base {
2     using Base::Base;
3 };

```



Héritage et délégation des constructeurs

Délégation des constructeurs

- ↳ L'objectif de la délégation est d'éviter la duplication du code dans les constructeurs d'une classe
 - ☛ Éviter de violer le principe **DRY** (Don't Repeat Yourself)
- ↳ Le principe est de permettre l'appel d'un constructeur depuis un autre constructeur dans la même classe

```

1 class Point {
2     int x, y;
3 public:
4     // Default constructor
5     Point(): Point(0, 0) {} // delegate to another constructor
6
7     // Parameterized constructor
8     Point(int x, int y): x(x), y(y) {} // initialize members
9
10    // Copy constructor
11    Point(const Point& p): Point(p.x, p.y) {} // delegate to another constructor
12 };

```



Overriding

Overriding

- ↳ Introduit en C++11, **override** exprime clairement qu'une méthode est une redéfinition de la méthode de la classe mère
- ↳ Dans le programme suivant on a fait une surcharge et non pas une redéfinition

```

1 #include <iostream>
2 struct A {
3     virtual char getValue() const {
4         return 'a';
5     }
6     virtual ~A() {}
7 };
8 struct B : public A {
9     virtual char getValue() {
10        return 'b';
11    }
12 };
13 int main() {
14     A* a = new B();
15     std::cout << a->getValue() << std::endl; // Affiche 'a'
16     delete a;
17 }

```

⚠ Le compilateur ne signale aucun avertissement



Overriding

Overriding

- ↳ En utilisant le mot clé **override**, le compilateur signale une erreur

```

1 #include <iostream>
2 struct A {
3     virtual char getValue() const {
4         return 'a';
5     }
6     virtual ~A() {}
7 };
8
9 struct B : public A {
10    virtual char getValue() override {
11        return 'b';
12    }
13 };

```



Les méthodes et les classes final

Les méthodes et les classes final

- Appliqué à une méthode, le mot clé **final** interdit sa redéfinition dans une classe dérivée

```

1 struct Base {
2     virtual void f() {
3         std::cout << "Base class default behaviour\n";
4     }
5 };
6
7 struct Derived : public Base {
8     void f() final // Version finale
9     {
10        std::cout << "Derived class overridden behaviour\n";
11    }
12 };
    
```

- Appliqué à une classe, **final** empêche sa dérivation

```

1 class X final {
2     // ...
3 };
    
```

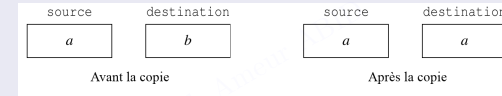


Copie et déplacement

Présentation

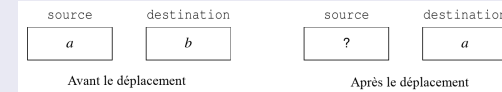
- Une opération de **copie** réalise la propagation de la valeur de l'objet source vers l'objet de destination en la copiant

- ☛ Une opération de copie ne modifie pas la valeur de l'objet source.



- L'opération de **déplacement** réalise la propagation de la valeur de l'objet source vers l'objet de destination en la déplaçant

- ☛ Une opération de déplacement **ne garantit pas la préservation de la valeur de l'objet source**
- ☛ Après l'opération de déplacement, la valeur de l'objet source est inconnue (c'est-à-dire non spécifiée mais valide).
- ☛ La sémantique de déplacement a été introduite en C++11



Constructeur/Opérateur de déplacement

Présentation

- Les opérations de déplacement ont été introduites en C++11
- ☛ Le constructeur de déplacement permet de transférer la propriété des ressources d'un objet existant vers un nouvel objet, sans les copier

```
class_name (class_name && other);
```

- ☛ L'opérateur d'affectation de déplacement permet de transférer la propriété des ressources d'un objet existant à un autre objet existant

```
class_name& operator= (class_name && other);
```

📢 Un constructeur de déplacement et un opérateur de déplacement permettent de transférer les ressources détenues par un objet **rvalue** vers un objet **lvalue** sans les copier



Exemple de constructeur et opérateur de déplacement (1/2)

```

1 struct SimpleString {
2     SimpleString(const char* str) : data(new char[std::strlen(str) + 1]) {
3         std::strcpy(data, str);
4     }
5     ~SimpleString() {
6         delete[] data;
7     }
8     SimpleString(SimpleString&& other) noexcept : data(other.data) {
9         other.data = nullptr; // L'objet 'other' ne pointe plus vers la mémoire
10    }
11    SimpleString& operator=(SimpleString&& other) { // Assignment de déplacement
12        if (this != &other) {
13            delete[] data; // Libérer la mémoire actuelle
14            data = other.data; // Transférer la propriété de la mémoire
15            other.data = nullptr; // L'objet 'other' ne pointe plus vers la mémoire
16        }
17        return *this;
18    }
19    void print() const { // Méthode pour afficher la chaîne
20        if (data) {
21            std::cout << data << std::endl;
22        } else {
23            std::cout << "Object is empty" << std::endl;
24        }
25    }
26    private:
27        char* data; // Pointeur vers les données
28 };
    
```



Exemple de constructeur et opérateur de déplacement (2/2)

```

1 // Utilisation
2 int main() {
3     SimpleString a {"Hello"};
4     SimpleString b(std::move(a)); // Utilise le constructeur de déplacement
5
6     b.print(); // Affiche "Hello"
7     a.print(); // Affiche "Object is empty"
8
9     a=std::move(b); // Opérateur de déplacement
10    a.print(); // Affiche "Hello"
11    b.print(); // Affiche "Object is empty"
12    return 0;
13 }

```

Modèle de classe `std::initializer_list`

Présentation

- ↳ `std::initializer_list` (C++11) est un modèle de classe en C++ qui permet de représenter une liste d'éléments de manière légère
- ↳ Déclarée dans `<initializer_list>`

```
template<class T> class initializer_list;
```

- ↳ Il est possible de requérir le nombre d'éléments et d'obtenir des itérateurs pour accéder à ces éléments
- ↳ Souvent utilisé comme type de paramètre pour les constructeurs, permettant l'initialisation d'objets avec une liste d'élément

Modèle de classe `std::initializer_list`

Suppression de copie garantie en C++17

Constructeur avec `std::initializer_list`

```

1 #include <iostream>
2 #include <vector>
3
4 struct Sequence {
5     Sequence(std::initializer_list<int> list) {
6         for (const auto &elt : list)
7             elements_.push_back(elt);
8     }
9     void print() const {
10        for (const auto& i : elements_)
11            std::cout << i << '\n';
12        }
13 private:
14     std::vector<int> elements_;
15 };
16
17 int main() {
18     Sequence seq{1, 2, 3, 4, 5, 6};
19     seq.print();
20 }

```

Présentation

- ↳ Une optimisation qui évite la création d'objets temporaires inutiles.
 - ↳ Permet de construire des objets directement à leur emplacement final sans passer par une copie ou un déplacement temporaire
- ↳ En C++17, cette optimisation est garantie par le standard
 - ↳ Les constructeurs de copie et de déplacement peuvent être complètement omis
 - ↳ Simplifie le code
 - ↳ Améliore les performances

Suppression de copie garantie en C++17

Exemple de suppression de copie

```

1 #include <iostream>
2 #include <array>
3
4 struct NonDeplacable {
5     NonDeplacable(int x) : v(x) {}
6     NonDeplacable(const NonDeplacable&) = delete;
7     NonDeplacable(NonDeplacable&&) = delete;
8     std::array<int, 1024> arr;
9     int v;
10 };
11
12 NonDeplacable fabriquer(int val) {
13     return NonDeplacable{val > 0 ? val : -val};
14 }
15
16 int main() {
17     auto grandObjetNonDeplacable {fabriquer(90)}; // L'objet est construit sur place
18 }

```

Les conversions

Présentation

Deux types de conversions

- 1 Implicite
 - Peuvent avoir lieu dans le calcul d'une expression quand on passe directement un argument à une fonction ou lors du retour d'une valeur par une fonction.
- 2 Explicite
 - On peut demander explicitement une conversion d'un opérande dans un type désiré.

Plan

- 1 Les nouveautés au niveau des classes en C++11/C++17
- 2 Les conversions
 - Conversions implicites
 - Conversions explicites
 - Conversion via constructeurs ou l'opérateur de conversion
- 3 Programmation fonctionnelle en C++11/C++17

Plan

- 1 Les nouveautés au niveau des classes en C++11/C++17
- 2 Les conversions
 - Conversions implicites
 - Conversions explicites
 - Conversion via constructeurs ou l'opérateur de conversion
- 3 Programmation fonctionnelle en C++11/C++17
 - Pointeurs de fonctions
 - Foncteurs
 - Fonctions lambdas
 - La classe modèle std::function

Les conversions

Conversion implicite

- ↳ Les conversions implicites facilitent la tâche du programmeur.
- ↳ Elles risquent d'être potentiellement dangereuses.
- ↳ Elles peuvent générer des bogues lors de l'exécution d'un programme. Des bogues qui sont parfois difficiles à cerner.
- ↳ Les conversions implicites sont effectuées chaque fois qu'une expression d'un type T1 est utilisée dans un contexte qui n'accepte pas ce type, mais accepte un autre type T2
- ↳ Situations courantes de conversions
 - ☞ Appel de fonction qui attend un paramètre de type T2
 - ☞ Opérateur qui attend un opérande de type T2
 - ☞ L'initialisation d'un nouvel objet de type T2, y compris dans l'instruction return d'une fonction renvoyant un T2
 - ☞ Lorsque l'expression est utilisée dans une instruction switch, où T2 est un type entier.
 - ☞ Si l'expression est utilisée dans une instruction if ou une boucle, et que T2 est de type booléen.



Les conversions explicites

Présentation

- ↳ Les conversions explicites permettent de contrôler précisément les transformations et d'éviter les erreurs involontaires
 - ☞ On définit explicitement la conversion souhaitée : meilleur contrôle
 - ☞ Le compilateur vérifie la validité de la conversion : meilleure sécurité
 - ☞ Le code est plus clair et plus facile à comprendre pour les autres développeurs : meilleure lisibilité
- ↳ En C++ moderne, l'objectif est de rendre ces conversions plus sûres, lisibles et moins sujettes aux erreurs par rapport aux casts de style C tel que `(int)x`



Plan

- 1 Les nouveautés au niveau des classes en C++11/C++17
- 2 Les conversions
 - Conversions implicites
 - Conversions explicites
 - Conversion via constructeurs ou l'opérateur de conversion
- 3 Programmation fonctionnelle en C++11/C++17
 - Pointeurs de fonctions
 - Foncteurs
 - Fonctions lambdas
 - La classe modèle `std::function`



Les conversions explicites

Présentation

- ↳ C++ offre 5 types de conversions
 - 1 `const_cast()` : modifier la constance d'un objet (avec précaution!).
 - 2 `static_cast()` : convertir entre types primitifs
 - 3 `reinterpret_cast()` : une conversion risquée entre types sans tenir compte de leur structure interne
 - 4 `dynamic_cast()` : réalisée en temps d'exécution, elle vérifie la compatibilité avant la conversion, offrant plus de sécurité.
 - 5 `bit_cast()` (C++20) : conversion au niveau binaire sans interprétation du type



- ↳ Les opérateurs de conversion en C++ moderne sont volontairement verbeux
- ↳ Leur longueur agit comme un signal d'alerte, incitant les développeurs à réfléchir à l'opération qu'ils réalisent



Les conversions explicites

Conversion explicite : `const_cast`

- Il permet d'ajouter ou de retirer le qualificateur `const` d'une variable
- C'est le seul cast parmi les cinq autorisés à retirer le `const`
- En théorie, un objet `const` doit le rester
- En pratique, on peut vouloir passer un objet `const` à une fonction qui attend un objet non-`const`
 - Si on est certain que cette fonction ne modifiera pas l'objet, on peut utiliser `const_cast()`

```
1 void bibliothequeTierce(char* str);
2
3 void f(const char* str) {
4     bibliothequeTierce(const_cast<char*>(str)); // Potentiellement dangereux !
5 }
```



N'utilisez `const_cast()` que si vous êtes certain que la fonction ne modifiera pas l'objet



Les conversions explicites

Conversion explicite : `static_cast`

- Effectue la conversion entre les types convertibles
- Elle s'effectue au moment de la compilation
- Syntaxe

```
static_cast<type_to_convert_to>(value_to_convert_from)
```

- Exemple

```
int myinteger {static_cast<int>(123.456)};
```



Conversion dynamique : `dynamic_cast`

- Effectuée au moment de l'exécution
- Ne peut être appliquée que sur des **pointeurs** et des **références** aux types de classe **polymorphes**
 - Une classe est polymorphe si elle contient au moins une **fonction virtuelle**

```
dynamic_cast<target_type>(objBase);
```

`target_type` peut être un type de pointeur ou de référence

- Comportement de `dynamic_cast`

- Avec les pointeurs

- Si la conversion réussit, retourne un pointeur valide vers `target_type`
- Si la conversion échoue, retourne `nullptr`

```
1 Base* basePtr {new Derived};
2 if (auto derivedPtr {dynamic_cast<Derived*>(basePtr)} ; derivedPtr!=nullptr) {
3     // Conversion réussie
4 } else {
5     // Conversion échouée
6 }
```



Conversion dynamique : `dynamic_cast`

- Comportement de `dynamic_cast`

- Avec les références

- Si la conversion réussit, retourne une référence valide vers `target_type`
- Si la conversion échoue, lève une exception de type `std::bad_cast`

```
1 try {
2     Base& baseRef {*basePtr};
3     Derived& derivedRef {dynamic_cast<Derived&>(baseRef)};
4 } catch (const std::bad_cast& e) {
5     std::cerr << "Erreur de conversion : " << e.what() << std::endl;
6 }
```



- Utilisez `dynamic_cast` avec des pointeurs pour éviter les exceptions
- Capturez toujours les exceptions lors de l'utilisation de références



Les conversions

Conversion explicite : reinterpret_cast

- ↳ Reinterpréter les données d'un type en un autre type
 - ⚠ Aucune vérification de la validité de cette opération n'est faite
 - ⚠ L'opérateur de transtypage le plus dangereux
- ↳ À utiliser avec précaution pour réinterpréter la représentation binaire d'un objet en un autre type d'objet
 - ⚠ Problème de portabilité : la représentation binaire d'un type peut dépendre de l'architecture matérielle

```
1 int i{7};
2 char* p {reinterpret_cast<char*>(&i)}; // static_cast ne permet pas cette
   conversion
3 std::cout << (p[0] == '\x7' ? "This system is little-endian\n" : "This system is
   big-endian\n");
```



Plan

- 1 Les nouveautés au niveau des classes en C++11/C++17
- 2 Les conversions
 - Conversions implicites
 - Conversions explicites
 - Conversion via constructeurs ou l'opérateur de conversion
- 3 Programmation fonctionnelle en C++11/C++17
 - Pointeurs de fonctions
 - Foncteurs
 - Fonctions lambdas
 - La classe modèle std::function



Les conversions

Conversion explicite : bit_cast

- ↳ std::bit_cast() est définie dans <bit> (dans la bibliothèque standard)
 - ⚠ Création d'un nouvel objet du type cible spécifié en effectuant une copie intégrale des bits de l'objet source vers cet objet cible
- ↳ std::bit_cast() réalise les vérifications des conditions suivantes :
 - ⚠ Taille identique des objets source et cible
 - ⚠ Types trivialement copiables (sans constructeur, destructeur ou opérations spéciales)

```
1 float f {1.2f};
2 int &i {reinterpret_cast<int &>(f)}; // OK
3 int &j {std::bit_cast<int &>(f)}; // Erreur de compilation : non copiable
4
5 int i{7};
6 char* p {std::bit_cast<char*>(&i)};
7 std::cout << (p[0] == '\x7' ? "This system is little-endian\n" : "This system is
   big-endian\n");
```



Les conversions

Conversion via constructeurs

- ↳ Tout constructeur (sauf constructeurs de copie et de déplacement) avec un seul argument peut être utilisé par le compilateur pour effectuer une conversion implicite
 - ⚠ Les conversions implicites peuvent parfois mener à des comportements inattendus ou des erreurs si elles ne sont pas bien comprises
- ↳ La déclaration explicit permet d'empêcher qu'un constructeur soit utilisé pour une conversion implicite

```
1 struct Widget {
2     explicit Widget(int); // explicit constructor
3     ...
4 };
```



Les conversions

Exemple de conversion implicite via un constructeur

```
1 struct IntArray { // Classe modélisant un tableau d'entiers
2     // Le constructeur autorise la conversion implicite
3     IntArray(std::size_t size) { /* ... */ }; // Créer un tableau avec la taille size
4     // ...
5 };
6
7 auto processArray(const IntArray& x) -> void {
8     // ...
9 }
10 auto main() -> int {
11     // Les lignes de code suivantes sont probablement incorrectes,
12     // mais valides grâce à la conversion de type implicite fournie par
13     // le constructeur IntArray::IntArray(std::size_t)
14
15     // Probablement incorrect
16     IntArray a = 42; // IntArray a = IntArray(42);
17
18     // Probablement incorrect
19     processArray(42); // processArray(IntArray(42));
20 }
```



Les conversions

Écriture des opérateurs de conversion

- ↳ Les opérateurs de conversion sont des méthodes spéciales qui permettent à un type de se comporter comme s'il était d'un autre type.
- ↳ Ils sont définis au sein d'une classe et peuvent être implicites ou explicites.
 - Ils sont par défaut implicites
 - Depuis C++11, il est possible de définir un opérateur de conversion comme explicite avec `explicit` pour interdire les conversions implicites



Les conversions

Exemple d'interdiction de conversion implicite via un constructeur

```
1 class IntArray {
2     // Le constructeur exige une conversion explicite
3     explicit IntArray(std::size_t size) { /* ... */ };
4     // ...
5 };
6
7 auto processArray(const IntArray& x) -> void {
8     // ...
9 }
10
11 auto main() -> int {
12     IntArray a(42), b{42}; // Ok
13     IntArray c = 42; // Erreur de compilation
14     processArray(42); // Erreur de compilation
15     return 0;
16 }
```



Les conversions

Écriture d'un opérateur de conversion

```
1 struct Cellule {
2     Cellule(double valeur) : _valeur(valeur) {}
3
4     // Opérateur de conversion implicite vers double
5     operator double() const {
6         return _valeur;
7     }
8
9     // Opérateur de conversion explicite vers std::string
10    explicit operator std::string() const {
11        return std::to_string(_valeur);
12    }
13
14 private:
15    double _valeur;
16 };
17
18 Cellule cell(1.23);
19 double num1 {cell}; // Initialisation directe : elle fonctionne
20 double num2 = cell; // Conversion implicite
21 double num3 = static_cast<double>(cell); // Conversion explicite
22 std::string str1 {cell}; // Ok : initialisation directe
23 std::string str2 = cell; // Erreur de compilation : pas de conversion implicite
```



Plan

- 1 Les nouveautés au niveau des classes en C++11/C++17
- 2 Les conversions
- 3 Programmation fonctionnelle en C++11/C++17
 - Pointeurs de fonctions
 - Foncteurs
 - Fonctions lambdas
 - La classe modèle `std::function`

Plan

- 1 Les nouveautés au niveau des classes en C++11/C++17
- 2 Les conversions
 - Conversions implicites
 - Conversions explicites
 - Conversion via constructeurs ou l'opérateur de conversion
- 3 Programmation fonctionnelle en C++11/C++17
 - Pointeurs de fonctions
 - Foncteurs
 - Fonctions lambdas
 - La classe modèle `std::function`

Chiheb Ameur ABID

Partie 2 Chiheb Ameur ABID
└ Programmation fonctionnelle en C++11/C++17
└ Pointeurs de fonctions

Présentation

Pointeur de fonction ?

- En plus d'avoir des pointeurs faisant référence à des variables de divers types, le langage C permet la définition de pointeurs de fonction.
- Ces pointeurs servent à identifier l'adresse d'entrée de la première instruction d'une fonction (point d'entrée).
- On utilise souvent les pointeurs de fonction pour passer une fonction en argument à une autre fonction.
- La déclaration d'un pointeur de fonction ressemble à la déclaration d'un pointeur de type et à la déclaration d'une fonction à la fois.

Chiheb Ameur ABID

Partie 2 Chiheb Ameur ABID
└ Programmation fonctionnelle en C++11/C++17
└ Pointeurs de fonctions

Déclaration

Déclaration

➤ La déclaration d'un pointeur de fonction se fait par les parenthèses qui permettent de déterminer la variable pointeur

```
type-retour (*identificateur)( [ type1 param1, ... ] );
```

```
1 void (*pFonction1)(void);
2 int (*pFonction2)(int, int);
3 void* (*pFonction3)(void*)
4 (void (*)(void)) (*pFonction4)((int (*)(int, int)), int);
```

- 1 **pFonction1** fait référence à une fonction ne retournant rien et ayant aucun paramètre d'entrée
- 2 **pFonction2** fait référence à une fonction retournant un entier et ayant deux entiers comme signature
- 3 **pFonction3** fait référence à une fonction retournant un pointeur indéterminé et un pointeur indéterminé comme signature
- 4 **pFonction4** fait référence à une fonction retournant un pointeur de fonction (ne retournant rien et n'ayant aucun paramètre d'entrée) et ayant un pointeur de fonction (de même nature que pFonction2) et un entier comme signature.

Déclaration



Il faut faire attention à l'ambiguïté des déclarations d'un pointeur de fonction et d'une fonction retournant un pointeur.

```
1 int (*pFonction)(int);  
2 int *Fonction(int);
```

On vient de déclarer deux types complètement différents :

- **pFonction** fait référence à un pointeur de fonction retournant un entier et ayant un entier comme paramètre d'entrée
- **Fonction** fait référence à une fonction retournant un pointeur d'entier et ayant un entier comme paramètre d'entrée

Utilisation

Affectation et appel

On assigne la valeur d'un pointeur de fonction en spécifiant l'adresse d'une fonction

```
1 int Fonction(int N);  
2 ...  
3 int (*pFonction)(int) = &Fonction; /* L'opérateur & est facultatif */
```

On l'appelle simplement comme une fonction standard par le pointeur

```
int A = pFonction(N);
```

Utilisation

Exemple : utiliser une fonction comme paramètre

```
1 int Resultat(int V1, int V2, int (*Compare)(int, int)) {  
2     return Compare(V1, V2);  
3 }  
4  
5 int Max(int V1, int V2) {  
6     return (V1 >= V2) ? (V1) : (V2);  
7 }  
8  
9 int Min(int V1, int V2) {  
10    return (V1 <= V2) ? (V1) : (V2);  
11 }  
12  
13 void main(void) {  
14     printf("Min de 1 et 2 = %d\n", Resultat(1, 2, &Min));  
15     printf("Max de 1 et 2 = %d\n", Resultat(1, 2, &Max));  
16 }
```

Plan

- 1 Les nouveautés au niveau des classes en C++11/C++17
- 2 Les conversions
 - Conversions implicites
 - Conversions explicites
 - Conversion via constructeurs ou l'opérateur de conversion
- 3 Programmation fonctionnelle en C++11/C++17
 - Pointeurs de fonctions
 - Foncteurs
 - Fonctions lambdas
 - La classe modèle `std::function`

Présentation

Foncteurs ?

- Un foncteur ou un objet de fonction est n'importe quel type qui surcharge l'opérateur d'appel `operator()`
 - ☞ Des objets que l'on utilise comme fonctions
 - ☞ Par rapport à un pointeur de fonction, le foncteur permet de sauvegarder l'état

Chiheb Ameur ABID

```
1 #include <iostream>
2
3 class MyClass {
4 public:
5     void operator()()
6     {
7         std::cout << "Function object called." << '\n';
8     }
9 };
10
11 int main() {
12     MyClass myobject;
13     myobject(); // invoke the function object
14 }
```



Plan

- 1 Les nouveautés au niveau des classes en C++11/C++17
- 2 Les conversions
 - Conversions implicites
 - Conversions explicites
 - Conversion via constructeurs ou l'opérateur de conversion
- 3 Programmation fonctionnelle en C++11/C++17
 - Pointeurs de fonctions
 - Foncteurs
 - Fonctions lambdas
 - La classe modèle `std::function`

Chiheb Ameur ABID



Utilisation des foncteurs

Intérêt des foncteurs

- En tant qu'objets, les foncteurs peuvent posséder des attributs
- Souvent utilisés avec les algorithmes de la bibliothèque standard STL

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4 class IsGreaterThan {
5 public:
6     IsGreaterThan(const int &v):val(v) {}
7     bool operator()(const int &v) const {return v>val;}
8 private:
9     int val;
10 };
11 void printMatch(const vector<int>& vec, IsGreaterThan fcb) {
12     for (const auto &elt : vec)
13         if (fcb(elt)) cout<<elt<<" ";
14     cout<<endl;
15 }
16
17 int main() {
18     vector<int> vec {150,20,80,120,200};
19     printMatch(vec, IsGreaterThan {100});
20     return 0;
21 }
```



Fonctions lambdas

Définition

- Une fonction lambda est une fonction anonyme
 - ☞ Peut être utilisée sans qu'on lui donne de nom
- Une lambda peut être directement écrite dans une fonction ou un algorithme, au plus près de l'endroit où elle est utilisée
- Les lambdas offrent les avantages suivants :
 - ☞ Permet au compilateur de mieux optimiser le code
 - ☞ Permet d'éviter de déclarer une fonction visible et utilisable par tout le monde



Fonctions lambdas

Déclaration d'une fonction lambda

↳ Syntaxe

```
[zone de capture](arguments de la lambda) -> type de retour { instructions }
```

- ☞ Zone de capture : par défaut, une lambda est en totale isolation et ne peut manipuler aucune variable de l'extérieur. Grâce à cette zone, la lambda va pouvoir modifier des variables extérieures
- ☞ Arguments de la lambda : exactement comme pour les fonctions, les paramètres de la lambda peuvent être présents ou non
- ☞ Type de retour : peut être omis dans certains cas
- ☞ Instructions

```

1 #include <iostream>
2 #include <string>
3 int main() {
4     [](std::string const & message) -> void {
5         std::cout << "Received message : " << message << std::endl;
6     };
7     return 0;
8 }

```



Fonctions lambdas

Fonctions lambdas

Capturer une variable

↳ Deux modes de capture

- 1 Par valeur
- 2 Par référence

Capture par valeur

↳ Il suffit d'inscrire le nom de la variable dans la zone de capture

- ☞ Par défaut, une variable capturée par valeur ne peut pas être modifiée

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     int un_entier {42};
5     auto lambda=[un_entier]() ->void {
6         // un_entier=0; : Interdit!!!
7         cout<<"Un entier : "<<un_entier<<endl;
8         return;
9     };
10    lambda();
11    return 0;
12 }

```



Fonctions lambdas

Capturer une variable

↳ Syntaxe

```

1 [capture](parametre)->type_de_retour {
2     traitement
3 }

```

- ☞ Les [] marquent le début de la lambda expression et comporte des spécifications concernant le passage des paramètres
 - [&objet] : une référence sur la variable objet
 - [objet] : passage par copie
 - [] : pas d'accès aux variables externes
 - [&] : accès a toutes les variables par référence
 - [=] : accès a toutes les variables par copie
- ☞ Les () spécifient les paramètres de la fonction lambda



Capture par valeur

↳ Pour pouvoir modifier les variables capturée par copie, il faut déclarer la fonction lambda comme mutable

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     int un_entier {42};
5     auto lambda=[un_entier]() mutable ->void {
6         un_entier=0; // Ok!
7         cout<<"Un entier : "<<un_entier<<endl;
8         return;
9     };
10    lambda();
11    return 0;
12 }

```



Fonctions lambdas

Capture par référence

- Il suffit d'ajouter l'esperluette & devant la variable à capturer par référence
- On travaille sur la variable elle-même et non pas une copie

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int un_entier {42};
6     auto lambda=[&un_entier]() ->void {
7         un_entier=0;
8         cout<<"Un entier : "<<un_entier<<endl;
9         return;
10    };
11    lambda();
12    cout<<"Entier : "<<un_entier<<endl;
13    return 0;
14 }
```



std::function

Présentation

- Une classe générique permettant d'envelopper une fonction appelable

```
template<class R, class... Args> class function<R(Args...)>;
```

- R définit le type de retour
- Args... définit la liste des arguments de la fonction

- Un objet de type `std::function` peut encapsuler une fonction, un foncteur ou une expression lambda

```
1 #include <functional>
2
3 auto somme(int x,int y) -> int {
4     return x+y;
5 }
6 auto main() -> int {
7     std::function<int(int,int)> f {somme};
8     f(10,5);
9     return 0;
10 }
```



Plan

- Les nouveautés au niveau des classes en C++11/C++17
- Les conversions
 - Conversions implicites
 - Conversions explicites
 - Conversion via constructeurs ou l'opérateur de conversion
- Programmation fonctionnelle en C++11/C++17
 - Pointeurs de fonctions
 - Foncteurs
 - Fonctions lambdas
 - La classe modèle `std::function`



Merci pour votre attention



Questions ?

