Module C++

FIP 1A CNAM

Séance 2

Programmer en C++

POINTEURS VS RÉFÉRENCES

Les pointeurs en C++

- Les pointeurs existent en C++, de par son « héritage » du C.
- Ils sont cependant plus fortement typés en C++:

- Valide en C

```
bird* b;
rock* r;
void* v;
v = r;
b = v;
```

- Invalide en C++,

=> Il faudra faire un cast.

Les références en C++

- Les références, comme les pointeurs, sont des variables du type qu'elles permettent de manipuler. Il y a toutefois quelques règles à respecter:

  - Elles ne peuvent jamais être nulles ou pointer vers une valeur nulle.

  - Elles pointent vers le même objet tout au long de leur cycle de vie, on ne peut pas les réassigner.

  - Elles doivent être initialisées dès leur déclaration.

Quand utiliser une référence?

- On les voit en majorité dans deux cas:

  - Comme un argument ou paramètre d’une fonction ou d’une méthode,

  - Comme type de retour d’une fonction.

Utiliser une référence comme argument

- Revient à passer une adresse comme avec un pointeur, mais avec une syntaxe « plus propre ».

- Utiliser une référence comme argument signifie que les modifications apportées à l'objet dans la fonction sont conservées en dehors de la fonction.

- Il faut faire très attention en cas de destruction de l'objet dans la méthode.

Utiliser une référence comme type de retour d'une fonction

- Du point de vue de la syntaxe, revient à renvoyer un objet

- D'un point de vue des performances, revient à renvoyer une adresse.

- Ne doit jamais pointer vers un objet qui ne vit que dans le scope de la fonction. Sinon on enverra une référence vers un emplacement mémoire indéfini.

Utiliser une référence comme type de retour d'une fonction

```
Stock& getStock();
```

Ici on renvoie une référence vers un objet de type Stock.

Modifier l'objet retourné le modifie également dans la classe qui contient cette méthode

Programmer en C++

CONST-CORRECTNESS

Qu’est ce que le mot clé « const »?

Il est utilisé pour les variables qui n’ont pas vocation à être modifiées

1. Compris par le compilateur (contrairement au préprocesseur)
2. Permet au compilateur d’optimiser le code
3. Permet d’avoir des messages d’erreur clairs

Pourquoi utiliser « const »?

- Le code est plus lisible,
- Le code est plus rapide,
- Évite les « nombres magiques »

ATTENTION

- Une valeur constante ne peut être utilisée au moment de la compilation
- On ne peut donc pas l'utiliser pour une déclaration de taille de tableau.
- Une constante en dehors d'une classe n'a que le scope du fichier.

  => Attention au nom utilisé pour éviter un conflit de nom.

Le pointeur « const »

```
const int * u;
```

Lire de droite à gauche permet de lire: « u est un pointeur vers un int constant. »

```
int * const  v = *u;
```

Lire de droite à gauche permet de lire: « v est un pointeur constant vers un int quelconque. »

=> le pointeur est constant on doit donc l'initialiser.

Le pointeur « const »

```
const int constante1 = 42;          // valeur constante initialisée à 42.
const int * ConstPointer1 = &int1; // pointeur vers valeur rendue constante int1
int const * ConstPointer2 = &int2; // pointeur vers valeur rendue constante int2
int * const Constant3 = &int1;     // pointeur constant vers la valeur int1 non constante
int const * const Constant4 = &int1; // pointeur constant vers la constante constant1
const int * const Constant5 = &int1; // pointeur constant vers la constante constant1
```

Le mot clé « const » s’applique au terme à sa gauche ou bien au terme directement à sa droite.

Les arguments « const »

Lors d’un passage par valeur, la définition « const » n’a de sens que pour le créateur de la méthode, car une copie est de toute façon créée immédiatement, et la valeur passée n’est donc en aucun cas affectée.

Si on travaille par valeur et qu’on veut garantir qu’on ne modifie pas la valeur de l’argument, on travaille plutôt en initialisant une référence constante vers l’argument.

Les fonctions et « const »

Quand on renvoie par valeur, l'opérateur const n'a d'importance que dans le cas d'un type défini par l'utilisateur. Cela ne change rien sur les types de base.

```
int f3() { return 1; }
const int f4() { return 1; }

int main() {
const int j = f3(); // Pas de problème
int k = f4();       // Pas de problème non plus.
}
```

Les fonctions et « const »

En C++, lorsqu'on travaille sur une classe, il est presque toujours plus efficace de passer une adresse qu'une valeur. Pour garantir qu'on ne modifie pas la classe, on spécifie « const » l'argument de la fonction.

On utilisera donc une référence constante plutôt qu'un pointeur, et ainsi la syntaxe sera exactement la même que si on passait par valeur, mais l'appel sera transparent pour le programmeur qui utilise notre fonction, et bien plus optimisé.

Les fonctions et « const »

Quand on renvoie un type « custom » par valeur, on ne peut en aucun cas utiliser le retour comme une lvalue. (On ne peut donc pas lui assigner de valeur ou la modifier ensuite.)

Quand on renvoie une adresse, on autorise la modification du contenu situé à l'adresse. On préfère donc généralement renvoyer une référence constante si ce n'est pas un comportement souhaité.

Programmer en C++

CONVERSIONS EXPLICITES

Les conversions spécifiques à C++

- Les casts hérités du C:

```
float f = (float)variableDouble;
```

- Ne sont pas vérifiés par le compilateur,
- Sont plus difficiles à trouver dans le code,
- Expriment mal l’intention du programmeur.

Les conversions entre états des données

- Les casts à la compilation:

  static_cast<T>()

- Les casts à l'exécution:

  dynamic_cast<T>()

Les conversions entre états des données

- Les casts de constance:

const_cast<T>()

- Permet de supprimer le caractère « const » d’un objet.

Les conversions entre types de données

reinterpret_cast<T>()

- Le plus dangereux des casts, il dit au compilateur de traiter la variable comme un objet du type « T »

Programmer en C++

LES BASES DE LA PROGRAMMATION ORIENTÉE OBJET

Programmer via une conception « orientée objet »

QU'EST CE QU'UNE CLASSE? UN OBJET?

Les Classes

- Les classes sont un concept élargi de structures de données.
- Elles peuvent contenir des données, mais également des comportements.
- Elles représentent un nouveau type de données (« user defined type »)

Les Objets

- Les objets sont les instanciations des classes.
- Ils contiennent donc des données, et peuvent avoir des comportements.
- En termes de variables, une classe représente le type, et l'objet est la variable elle-même.

Les Objets

- Les objets sont les instanciations des classes.
- En termes de variables, une classe représente le type, et l'objet est la variable elle-même.
- On peut avoir une infinité d'objets liés à une classe dans un programme.
- L'identité d'un objet est caractérisé par la valeur de ses attributs.

En résumé

- La classe est un concept théorique, qui modélise la réalité mais n’est pas la réalité.
- A l’exécution, la classe est instanciée et donne lieu à des objets, qui eux sont spécifiques de la réalité. Un objet est un état d’une classe.
- La classe peut être instanciée en autant d’objets que l’on veut, tous reprenant (on dit qu’ils les exposent) les méthodes et propriétés de leur classe "génitrice".
- Le processus d’instanciation revient donc à « donner vie » à une classe et à spécifier ses propriétés.
- Une classe comporte des propriétés de classe(communes à tous les objets), d’instances (spécifiques à chaque objet) et des méthodes (traitements ou comportements)

Programmer via une conception « orientée objet »

CLASSES VS STRUCTS

Les classes en C++

- Elles sont définies par le mot clé « Class »
- Elles représentent un type à part entière
- Elles peuvent contenir des données, des fonctions et des méthodes
- Elles peuvent hériter d’une autre classe

Les structures en C++

- Elles sont définies par le mot clé « Struct »
- Elles représentent un type à part entière
- Elles peuvent contenir des données, des fonctions et des méthodes
- Elles peuvent hériter d’une autre structure

Struct VS Class

- Une seule différence **technique**: la visibilité par défaut.

  - Dans une Struct, les membres sont publics par défaut.

  - Dans une Class, les membres sont privés par défaut.

Struct VS Class

- Dans les conventions on n’utilise pas les structures et les classes pour la même chose.
- - Une Struct représente un ensemble de données qui ont besoin d’être regroupées.
- - Une Class représente un objet qui a un comportement.
- On utilisera une classe si l’objet souhaité a le moindre membre non public.

Définir ses classes en C++

- Pour des raisons de performances, on sépare la déclaration et la définition
- On inclut la déclaration de la classe qui inclut celle de ses membres dans le fichier header
- On définit les fonctions membres de la classe dans le fichier .cpp

Le fichier header

- Est régulièrement inclus dans d’autres fichiers dans la phase de compilation
- Contient la déclaration et certaines définitions (« fonctions légères »)
- Contient en général les #include_guards

Les include_guards

- Servent à protéger contre l’inclusion multiple
- Sont appliqués par le pré-processeur

Les include_guards

```
#ifndef REFERENCESVSPOINTERS_H
#define REFERENCESVSPOINTERS_H

class FibonacciByReference
{
public:
    FibonacciByReference();
    static int Fibonacci(int numero);

private:
    static void ParcourirSuiteFibonacci(int& nombre1, int& nombre2);
};

#endif // REFERENCESVSPOINTERS_H
```

Le mot clé « new »

- Il sert à allouer un espace mémoire à une instance d’un objet du type qu’on lui passe en paramètre.
- Il fait appel à un des constructeurs de la classe/structure en question
- Il renvoie une adresse

Programmer en C++

MÉTHODES NÉCESSAIRES AU BON FONCTIONNEMENT D'UNE CLASSE

L’UTILISATION DES CONSTRUCTEURS

Les constructeurs

- Les constructeurs sont les moyens d’instancier une classe. Chaque classe en a au moins un.
- On peut les surcharger pour changer les arguments.
- Ce sont eux qui sont chargés de l’allocation mémoire de notre objet. (Pas de malloc en C++). Ils garantissent que les objets sont bien initialisés.

Les constructeurs

- Lors de l’instanciation d’une classe fille, les constructeurs des classes mères sont d’abord appelés.
- L’appel à un constructeur peut donc en réalité passer par l’appel de nombreux constructeurs.
- L’instanciation de membres constants ne peut se faire dans le corps du constructeur, il faut faire appel à un mécanisme propre au C++: la liste d’initialisation.

Les listes d’initialisation

- Les listes d’initialisation sont un moyen d’instancier les membres d’une classe.

```
Mammifere::Mammifere(std::string _nom, int _poids): Animal(_nom), poids(_poids), age(10)
{
    std::cout << "Mammifère :" << this->Nom << " est né." << std::endl;
};
```

- Ici, le constructeur fait appel à un autre constructeur, et instancie des membres à la volée.
- La liste d'initialisation est le seul moyen d'appeler les constructeurs parents avec des paramètres.

Les listes d’initialisation

- Les utiliser évite de d’abord allouer l’espace mémoire lié à un membre, puis à lui assigner la bonne valeur. L’espace est directement alloué en lui assignant la valeur.

  => Gain de performances.

- Les listes d’initialisation sont absolument nécessaires pour instancier les membres constants d’une classe. Ceci est valable pour les membres qui sont des références vers d’autres variables.

L’UTILISATION DES DESTRUCTEURS

Les destructeurs

- Les destructeurs sont les moyens de supprimer l’instance d’une classe.
- Chaque classe en a un et un seul.
- Ce sont eux qui sont chargés de la désallocation mémoire de notre objet.
- C’est tout ce dont dispose C++ en termes de « Garbage Collector »

Les destructeurs

Il est particulièrement nécessaire d'implémenter dans un destructeur les actions suivantes:

- Libération de l'espace mémoire contenu à l'endroit vers lequel les pointeurs membres de la classent pointent.
- Libération de l'espace mémoire contenu par des tableaux membres de la classe.

Les destructeurs

Attention:

- Quand la classe définie a pour but d'être héritée par d'autres classes, il est très important de toujours déclarer le destructeur comme **virtuel**.
- Sinon, l'objet pourrait ne pas être correctement détruit à l'appel du destructeur et certains espaces mémoires pourraient être perdus.

LES FONCTIONS « INLINE »

Les étapes d’un appel de fonction

1. Les arguments sont stockés sur la pile.
2. L’adresse de la valeur de retour est allouée sur le tas
3. L’adresse de retour de la fonction est aussi stockée sur le tas.
4. L’adresse de la fonction est appelée à travers un appel CPU.
5. La fonction lit les arguments et exécute le code de la fonction.
6. La valeur de retour de la fonction est stockée à l’adresse spécifiée en 2.
7. L’exécution retourne au niveau de l’appelant et le tas est vidé.

L’optimisation des appels avec « inline »

Une fonction peut être déclarée « inline », c’est-à-dire que le code est recopié par le compilateur pour chaque objet.

Avantage:

- Le code est plus rapide à exécuter.

Inconvénient:

- Le code est plus volumineux et plus lourd.

L’optimisation des appels avec « inline »

Le choix de déclarer une fonction « inline » n’est pas anodin, il faut donc peser le pour et le contre à chaque fois.

Déclaration inline recommandée

- La fonction ne fait pas un traitement lourd.
- Accesseurs des membres d’une classe

Inline n’est pas recommandé quand :

- Le code est plus volumineux et plus lourd.

Déclaration inline non recommandée

- La fonction prend en paramètre un objet externe à la classe
- La fonction exécute un traitement lourd