

Éléments de C++, partie 2

420-V31-SF – Programmation de Jeux Vidéo III

Au menu

- Forward declaration
- Les énumérations
- SFML – Les Textures vs `std::vector`
- Tests unitaires: une précision
- Pour votre culture personnelle: `Inline`

420-V31-SF – Programmation de Jeux Vidéo III

Forward declarations

Forward declaration

- Il est tout à fait possible qu'une classe ai besoin d'utiliser une autre classe comme attribut. Et que cette seconde classe ai aussi besoin d'utiliser la première comme attribut.
- Si on réfléchit à comment le code est compilé en C++ que se produit-il?
- Exemple vu dans le développement du TP1
 - Dans Zombie.h on inclue Game.h
 - Dans Game.h, on inclue Zombie.h
 - Ordre de compilation: il se passe quoi?
 - Solution 1: évitez cela (mais on pourrait vraiment en avoir besoin)
 - Solution 2: ...


Forward declaration

- On appelle cette situation : problème de dépendance cyclique
- Il se produit une boucle infinie d'inclusions
- Solution: les Forward declarations
- (déclarations prospectives en français, vous comprenez pourquoi on va continuer avec le terme anglais)

Problème de dépendance cyclique

```
/* parent.h */  
  
#include "enfant.h"  
  
class Parent  
{  
    Enfant* unEnfant;  
};
```

```
/* enfant.h */  
  
#include "parent.h"  
  
class Enfant  
{  
    Parent* unParent;  
};
```



Solution « forward declaration »:

```
/* Parent.h */  
  
class Enfant; /* Forward declaration  
  
class Parent  
{  
    Enfant* unEnfant;  
};
```

```
/* enfant.h */  
  
#include "parent.h"  
  
class Enfant  
{  
    Parent* unParent;  
};
```

Juste besoin de la faire d'un côté, mais on peut très bien la faire des deux cotés si pointeurs des deux côtés. Si pointeur juste d'un côté alors forward declaration juste d'un côté

Forward declaration

```

//***** a.h *****
#pragma once

#include "abase.h"
#include "b.h"

// Forward Declarations
class C;
class D;

class A : public ABase
{
    B m_b;
    C *m_c;
    D *m_d;

public:
    void SetC(C *c);
    C *GetC() const;

    void ModifyD(D *d)

};
#endif

```

```

//***** a.cpp *****
#include "a.h"
#include "d.h"

void A::SetC(C* c)
{
    m_c = c;
}

C* A::GetC() const
{
    return m_c;
}

void A::ModifyD(D* d)
{
    d->SetX(0);
    d->SetY(0);
    m_d = d;
}

```

• L'inclusion de « abase.h » dans « a.h » est nécessaire pour compléter la déclaration de la classe A qui hérite de ABase.

• L'inclusion de « b.h » dans « a.h » est nécessaire puisque la classe B est utilisée de manière directe dans la classe A (le compilateur doit connaître la taille de B pour déterminer la taille de A).

• Dans « a.h » et « a.cpp » la classe C est seulement utilisée comme pointeur. Dans ce cas il faut favoriser une « forward declaration » dans « a.h » plutôt que l'inclusion de « b.h ».

• La classe D est utilisée comme référence à un pointeur dans « a.h », donc on favorise l'utilisation d'une « forward declaration ». Par contre, dans « a.cpp » la classe D est utilisée concrètement, c'est pourquoi il y a inclusion de « d.h » dans « a.cpp »

Quand c'est possible, vaut mieux utiliser les « forward declarations » plutôt que les inclusions de .h

Forward declaration

La notion de déclaration prospective fonctionne non seulement aux pointeurs (*) mais aussi avec les références (&).

Permet de régler les problèmes d'inclusions "circulaires"

Mais permet aussi de diminuer le temps de compilation.

Sans inclusion circulaire, si vous avez seulement un pointeur ou une référence sur une classe dans le .h, la forward declaration est aussi envisageable, voir recommandable.

Forward declaration

En résumé voici la recette:

- 1- Remplacer l'inclusion du fichier .h par class [nom de la classe]
- 2- Utiliser seulement des références ou des pointeurs pour cette dite-classe.
- 3- Inclure le .h de la classe utilisée dans le .cpp

Juste besoin de la faire d'un côté, mais on peut très bien la faire des deux cotés si pointeurs des deux côtés. Si pointeur juste d'un côté, alors forward declaration juste d'un côté aussi.

420-V31-SF – Programmation de Jeux Vidéo III

Les énumérations

Énumération (enum)

Le mot clé **enum** permet de définir un ensemble de constantes de type entier(int).

En l'absence d'une <valeur>, la première constante prend la valeur zéro.

Toute constante sans <valeur> sera augmenté de 1 par rapport à la constante précédente.

Énumération (enum)

Voici un exemple de définition:

```
enum Cours
{
    INFO, // Pas de valeur définie, alors = 0
    MATH, // Pas de valeur définie, alors = "INFO + 1"
    CHIMIE,
    FRANCAIS
};
```

Pour définir une valeur, faire comme suit:

```
enum Cours
{
    INFO = 420,
    MATH = 213,
};
```

Énumération (enum)

Un enum peut aussi être utilisé comme index d'un tableau (normal puisque ce sont des int en fait):

```
const string nomCours [4] =  
    { "INFORMATIQUE", "MATHEMATIQUE",  
      "CHIMIE", "FRANCAIS" };  
  
cout << "Cours: "  
      << nomCours[FRANCAIS]  
      << endl;
```

Énumération (enum)

On peut aussi l'utiliser comme type énuméré (comme en C# ou Java)

```
Cours monCours = Cours::INFO;
```

Puis on peut l'utiliser comme marqueur de condition:

```
if (monCours == Cours::INFO)
{
    ...
}
Else if (monCours == Cours::MATH)
{
    ...
}
```

Énumération (enum)

Dans une classe C++, l'enum déclaré est automatiquement statique

Pourquoi de toute façon il faudrait le faire porter à chaque fois par chaque instance de la classe?

On atteint un élément de l'Enum par
`nomDeLaClasse::nomEnum::Element`

On peut parfaitement laisser un enum public, sauf si justement on veut restreindre sa portée.

420-V31-SF – Programmation de Jeux Vidéo III

SFML – Les Textures vs `std::vector`

Textures – Problèmes inattendus

Problème: Texture vs std::vector

Texture n'est pas fait pour pouvoir être copié, si vous le faites, vous perdez votre texture.

Or, le push_back de vector crée une copie de votre classe dans le vector. Si votre classe avait un attribut texture, celui-ci vient de perdre son contenu.

Solution 1: initialiser la texture APRÈS avoir poussé votre classe dans le vecteur

Solution 2 (La meilleure): utiliser un vector de pointeurs (attention à la gestion de la mémoire)

CONSEIL : De manière générale, en C++, avec des conteneurs dynamiques, utilisez des pointeurs comme contenu, puisque charger une classe dans un conteneur le copie.

420-V31-SF – Programmation de Jeux Vidéo III

Tests Unitaires - précision

Tests unitaires

- Le test unitaire teste une méthode et une seule à la fois
- Il est possible que le test vous oblige à utiliser une autre méthode de la classe...
- ... et c'est bien correct comme ça!

Tests unitaires

- Dans un test unitaire, on suppose que seule la méthode testée est "suspecte"
- Toutes les autres méthodes sont considérées "valides"
- Ce n'est pas parfait, mais dans les systèmes éprouvés et massifs où surtout on ajoute ou modifie des méthodes une à la fois, faire un test de régression à chaque fois sur la méthode ajoutée ou modifiée (donc la méthode "suspecte", ça fait sens).

420-V31-SF – Programmation de Jeux Vidéo III

Pour votre culture de programmeur - Inline

Inline

Ce mot-clé demande au compilateur de remplacer l'appel de la fonction par le corps intégral de la fonction. C'est la même mécanique que la pré-compilation avec les `#define`

Avantage principale: L'appel à une fonction demande un peu de temps machine. De copier-coller le contenu d'une méthode partout où celle-ci est appelée optimise le temps-machine.

Désavantage principale: Le code, "modifié" par le compilateur, sera un peu plus gros et demandera un peu plus de mémoire.

Ce qui faudrait mettre en inline: une méthode sur une ligne, peut-être 2, n'utilisant que des variables de base.

Inline -syntaxe

Simplement mettre inline avant la fonction que l'on veut "inliner"

Se prête davantage à de petites fonctions hors-classe, style C.

https://www.tutorialspoint.com/cplusplus/cpp_inline_functions.htm

ex:

```
inline int factorielle(int n)
{
    return (n < 2) ? 1 : n * fac(n-1);
}
```

Dans une classe C++ standard ce qui est inline doit être déclaré ET défini dans le .h.

Inline - bénéfices

Outre la diminution d'appels au cpu, le compilateur va s'attarder sur les méthodes inline et peut les optimiser davantage

- Une ligne comme "temp+=0" ne fait concrètement rien et pourrait être éliminée.
- Une comparaison qui retourne toujours vraie ou faux dans un remplacement inline précis pourrait être éliminée ou remplacé directement par true ou false.
- Une comparaison comme $x+1 == 0$ pourrait être remplacé par $x == -1$ (on passe de deux opérations à une)
- Une opération comme $(y-1) + 1$ pourrait être remplacée par y . (deux opérations éliminées).

Dans tous ces cas, c'est que dans un remplacement inline donné, les paramètres données sont souvent prévisibles

Inline - désavantages

- Dans les applications où la petite taille du code est plus importante que la vitesse, comme dans de nombreux systèmes embarqués et microcontrôleurs, inline est généralement désavantageux sauf pour de très petites fonctions.
- L'augmentation de la taille du code peut faire en sorte qu'un petit bloc de code critique ne puisse plus tenir dans le cache, ce qui fait en sorte que ce code ne sera plus chargé dans le cache et provoquera concrètement un ralentissement.
- Si la taille du code augmente trop, les ressources matérielles telles que la taille de la RAM pourraient devenir insuffisantes, ce qui pourrait conduire à des programmes qui soit ne pourraient plus être exécutés ou qui pourraient crasher en cours d'exécution. Avec les ordinateurs actuelles, c'est devenu quelque chose d'excessivement peu probable, mais ça peut encore être un problème pour les systèmes embarqués.

Inline – au final

- Dans les compilateurs modernes:
 - *Le compilateur est libre d'ignorer vos inline s'il constate que ce n'est pas avantageux*
 - *Le compilateur va parfois inliner automatiquement certaines méthodes (vos get et set simples entre autre)*
 - *Inline: toujours nécessaire? Peut-être pas!*