Modélisation et programmation

Cours 3

ENSIMAG 2A – MMIS

Rappels

Règle de 3 et de 5

- Si une classe définie l'une des 3 méthodes suivantes, alors les 3 doivent être définies
  1. Destructeur : `~POINT()`.
  2. Constructeur par recopie : `POINT(const POINT &)`.
  3. Opérateur de recopie : `operator=(const POINT &)`.

- Il est également important de re-définir un constructeur.

- Avec le `C++ 2011`, il est recommandé de définir deux méthodes complémentaires
  1. Constructeur par déplacement mémoire : `POINT(const POINT &&)`.
  2. Recopie par déplacement mémoire : `operator=(const POINT &&)`.

- En `C++ 2011`, il est possible de déléguer la création d'une de ces méthodes ou d'interdire son utilisation en utilisant des qualificateurs : `default` ou `delete`.

Utilisation de const

- **const** permet de protéger une variable en écriture
- En C++, **const** a plusieurs supplémentaires
  - pointeur sur un double constant **const double** * p
  - pointeur constant sur un double **double** * **const** p
  - référence et objet pointé sont constants **const double** & p
- La position de **const**

```
1   const char *function1()
2
3   void function2(Type const & param);
4
5   class Class2
6   { void Method1() const;
7     int MemberVariable1;
8   }
9
10  const int*const Method3(const int*const&)const;
```

Float et double

- Les **`float`** ne sont pas significativement plus rapides que les **`double`**.
- La vitesse dépend de la plateforme, du compilateur, des bibliothèques appelées.
- La précision est très différente entre les **`float`** et les **`double`**.

Pointeurs et références

Pointeurs : retour sur la définition

- Un pointeur est une variable contenant l'adresse d'une autre variable d'un type donné.
- Exemple : `double *y`
- `y` est un pointeur vers un **`double`**: `y` contient l'adresse en mémoire où est stocké un **`double`**.
- Par défaut, le pointeur n'est pas initialisé : son contenu est celui de l'adresse mémoire avant que celui ne soit créé. Un pointeur non initialisé peut accéder à une zone mémoire du système actuellement attribué à l'OS ou à un autre programme.
- Un pointeur doit nécessairement être typé.
- Le compilateur sait que l'objet créé est un pointeur : il ne lui alloue pas de mémoire, et il connaît le nombre de blocs mémoire qui suivent le bloc pointé.
- Pour accéder à l'adresse d'une variable on utilise l'opérateur &.

```
1    double  x =  4;
2    double *y = &x;
```

A quoi ça sert?

- Permettent de manipuler de façon simple des données pouvant être volumineuses et complexes (passage de paramètres pour les fonctions).
- Permettent de créer des tableaux dont chacun des éléments est de taille différente.
- Permettent d'ajouter des éléments à un tableau en cours d'utilisation.
- Permettent de créer des chaînes.

Passage d'argument : par pointeur

```
1   #include<iostream>
2   using namespace std;
3
4   void xpy(int *x, int *y)
5   { (*x) += (*y); }
6   int xpy2(int *x, int *y)
7   { return (*x) + (*y); }
8   int xpy3(int x, int y)
9   { return x+y; }
10
11  int main()
12  {
13    int a = 5, b = 6;
14    cout<<xpy3(a,b)<<endl;
15    cout<<xpy2(&a,&b)<<endl;
16    xpy(&a,&b);
17    cout<<a<<endl;
18    return 0;
19  }
```

Modifier les valeurs sans faire de copie.

La référence

- Le pointeur peut-être complexe à manipuler : plus flou que le passage d'argument par copie.
- Notion inexistante en C.
- Permet de manipuler des variables existantes ayant des portées différentes.
- Déclaration

```
1    type & variable = valeur;
```

- Une référence doit obligatoirement être initialisée.
- L'intérêt d'une référence réside essentiellement dans le passage de paramètres.

Une référence n'est pas une adresse.

Une référence n'est pas un pointeur.

Passage d'argument : par référence

```
1   #include<iostream>
2   using namespace std;
3
4   void xpy(int &x, int &y)
5   { x += y; }
6   int xpy2(int &x, int &y)
7   { return x+y; }
8   int xpy3(int x, int y)
9   { return x+y; }
10
11  int main()
12  {
13    int a = 5, b = 6;
14    cout<<xpy3(a,b)<<endl;
15    cout<<xpy2(a,b)<<endl;
16    xpy(a,b);
17    cout<<a<<endl;
18    return 0;
19  }
```

Ecriture plus "élégante" d'un point de vue utilisateur qu'un pointeur.

Différence entre pointeur et référence

- On ne peut pas référencer une référence après sa définition.
- Une fois la référence créée, elle ne peut pas être réaffectée.
- Une référence ne peut pas être définie comme `null`: une référence se rapporte toujours à un objet, valide ou invalide.
- Une référence ne peut pas être non-initialisée : elle ne peut pas être réinitialisée, elle doit être initialisée dès sa création.
- Pour le passage d'arguments d'une fonction, on peut ajouter le mot clé `const` pour spécifier le caractère lecture seule de l'argument.
- Pour les classes, une référence s'initialise dans la liste d'initialisation.
- Les références peuvent également être utilisées pour le type de retour: cela permet d'éviter la copie d'objet.
- Tant que la référence existe, l'objet retourné existe.

Pointeur *this

- Stocke l'adresse mémoire de la classe instanciée.
- Permet l'accès par pointeur aux variables pour les fonctions pointeurs de la classe.
- N'est pas compté dans la taille de l'objet (`sizeof`).
- N'est pas accessible par les fonctions membres statiques.
- N'est pas modifiable.
- Permet d'identifier plus facilement les données membres des données externes.
- Permet d'enchaîner les appels de fonctions.
- Très utile dans le cas de surcharge d'opérateurs.

Héritage

Héritage décomplexé

Dans le cadre de ce cours nous allons aborder les thèmes suivants

- polymorphisme
- héritage multiple
- classe abstraite

Rappel sur l'héritage simple

- A partir d'une classe existante, `POINT`, on souhaite définir une nouvelle classe `POINT_COLOR`.
- La classe `POINT` est appelée classe mère ou classe de base.
- La classe `POINT_COLOR` est appelée classe file ou classe dérivée.
- `POINT_COLOR` dérive/hérite de la classe `POINT`.
- La classe `POINT_COLOR` hérite automatiquement de toutes les données et méthodes de la classe `POINT`, sans avoir besoin de les réécrire
- Une classe peut avoir plusieurs classes dérivées.
- Une classe dérivée peut être utilisée comme classe de base pour une autre classe dérivée.
- Une classe dérivée peut hériter
  - d'une classe de base : c'est l'héritage simple.
  - de plusieurs classes de bases simultanément: c'est l'héritage multiple.

Et ça sert à quoi?

- Réutilisation de classes existantes.
- Adaptation de classes existantes à des besoins propres.
- Evolution des classes sans avoir à tout réécrire.
- Modularisation des classes afin de les spécialiser progressivement.
- Limitation de la taille des classes : le nombre plutôt que la taille.

On peut écrire une classe dérivée sans disposer du code source complet de la classe de base : il suffit de disposer du fichier de déclaration et des fichiers compilés (sur la bonne plateforme) correspondant.

Création d'une classe dérivée

- Reprenons la classe `POINT` des cours précédents.
- Supposons que l'on souhaite manipuler des `POINT` ayant une couleur.
- La première possibilité consiste à modifier la classe `POINT`
  - il faut modifier le fichier source contenant l'implémentation de la classe
  - recompiler les anciens programmes avec la nouvelle version de la classe en risquant de briser la compatibilité.
- La seconde possibilité consiste à construire une classe `POINT_COLOR` à partir de la classe base `POINT`
  - Il n'est pas nécessaire de modifier le fichier implémentation de la classe `POINT`
  - On peut continuer d'utiliser les anciens programmes sans risque.
  - Il est faut impérativement un accès à la définition de `POINT` et à une version compilée de la classe (éventuellement).

La seconde possibilité est la meilleure.

Une classe haute en couleur

- Fichier `POINT_COLOR.h`

```
1   #ifndef POINT_COLOR_H
2   #define POINT_COLOR_H
3   #include "POINT.h"
4
5   class POINT_COLOR :
6     public POINT
7   {
8     protected :
9       int couleur;
10    public :
11      void colorer (int c);
12  };
13  #endif
```

Fichier `POINT_COLOR.c`

```
1   #include "POINT_COLOR.h"
2   void POINT_COLOR::colorer(int c)
3   {
4     couleur = c;
5   }
```

La déclaration d'une classe dérivée se compose de cinq éléments

- le mot **`class`** pour désigner que nous créons une nouvelle classe.
- **un nom** pour désigner la classe.
- le séparateur **:** pour désigner la dérivation
- un **accès** pour signaler la nature de la dérivation
- un nom de classe

Manipulation de fonctions membres

Comment afficher la couleur d'un `POINT_COLOR`?

1. Créer une fonction spécifique

```
1  void POINT_COLOR::afficherCouleur( ) const
2  {
3    std::cout<<"(x,y) : ("<<x<<" , "<<y<<" )"<<std::endl;
4    std::cout<<"(c) : ("<<c<<" )"<<std::endl;
5  }
```

   mais cette fonction recopie l'implémentation de l'affichage de la classe `POINT`.

2. Utiliser en interne, la fonction `afficher` de la classe `POINT`

```
1  void POINT_COLOR::afficherCouleur( ) const
2  {
3    afficher();
4    std::cout<<"(c) : ("<<c<<" )"<<std::endl;
5  }
```

   mais la fonction `afficher()` et `afficherCouleur()` réalise la même opération pour des classes différentes.

3. Surcharger la fonction `afficher()`

```
1  void POINT_COLOR::afficher( ) const
2  {
3    POINT::afficher();
4    std::cout<<"(c) : ("<<c<<" )"<<std::endl;
5  }
```

Les applications de l'héritage

Comme illustré dans l'exemple précédent, l'héritage permet

- de factoriser les fonctionnalités communes
- d'obtenir des fonctionnalités communes avec des implémentations différentes
- d'obtenir des fonctionnalités spécifiques

Ce que l'on a vu sur les fonctions est également vrai pour

- les attributs
- les opérateurs
- les patrons

La factorisation du code permet une meilleure lisibilité, robustesse et sécurité du code.

Cas du constructeur

- La création d'un objet de la classe d'un `POINT_COLOR` se fait par l'intermédiaire de la création d'un objet de la classe `POINT`.
- Par défaut, le constructeur de la classe `POINT_COLOR` est celui de la classe `POINT`.
- On peut surcharger le constructeur de la class `POINT` avec un constructeur spécifique.
- Il est préférable, lorsque c'est possible, de réutiliser un constructeur de la classe de base.

```
1  POINT_COLOR::POINT_COLOR(int x, int y, int c):POINT(x,y)
2  {
3    couleur = c;
4  }
```

- Lors de l'appel du constructeur, le constructeur par défaut, i.e. sans argument, de la classe de base est toujours appelé avant celui de la classe dérivée sauf dans le cas d'un appel explicite à un autre constructeur de la classe de base.
- Lors de l'appel du destructeur, celui de la classe dérivée est toujours appelé avant celui de la classe de base.

Opérations de copie

- Définition des classes

```
1   class POINT
2   {
3     protected :
4       int dim; double *pCor;
5     public :
6       POINT(int d, double v=0);
7       POINT(const POINT &P);
8       POINT & operator =
9         (const POINT &);
10  };
```

```
1   class POINT_COLOR:public POINT
2   {
3     protected :
4       int colour;
5     public :
6       POINT_COLOR
7         (const POINT_COLOR &P);
8       POINT_COLOR & operator =
9         (const POINT_COLOR &);
10  };
```

- Programme d'appel

```
1   int main()
2   {
3     POINT_COLOR A(2);
4     POINT B=A; // Appel au constructeur par recopie de la classe POINT
5     POINT C;
6     C =A; // Appel a l'operateur = de la classe POINT
7   }
```

Dérivation et accès

- La syntaxe de définition d'une classe dérivée s'écrit de la manière suivante

```
class derivedName : type_derivation baseName1
                    type_derivation baseName2
```

- Le type de dérivation influence la visibilité des membres de la classe de base et l'accès à ces membres pour la classe dérivée.

- Tableau récapitulatif

type_derivation	visibilité des membres	accès aux membres
public	$\frac{\text{public} \rightarrow \text{public}}{\text{protected} \rightarrow \text{protected}}$	public, protected
protected	$\frac{\text{public} \rightarrow \text{protected}}{\text{protected} \rightarrow \text{protected}}$	public, protected
private	$\frac{\text{public} \rightarrow \text{private}}{\text{protected} \rightarrow \text{private}}$	public, protected

Dans le cas général, on utilise la dérivation **public**.

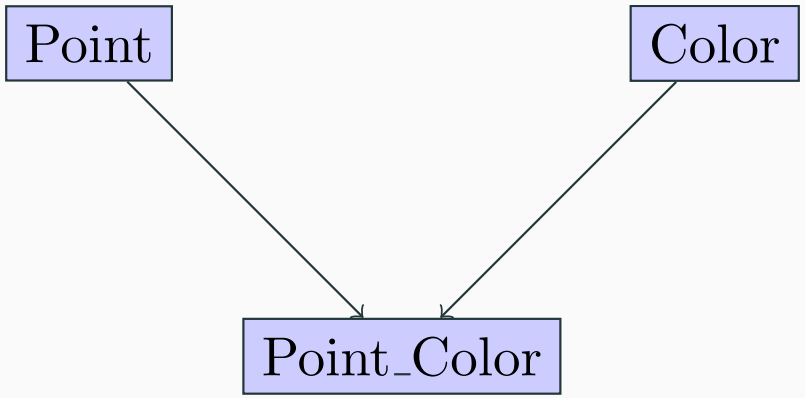
Transtypage

- Il est possible de transtyper un pointeur sur un objet de la classe dérivée en un pointeur sur la classe de base
  ```
  POINT *pPoint = (POINT_COLOR *) pPointColor;
  ```
- Le contraire n'a pas de sens.
- On peut déterminer le type d'un objet dérivé à partir de son pointeur sur la classe de base
  ```
  pPOINT_COLOR = dynamic_cast<POINT *> (pPoint)
  ```
- On peut également appliquer un transtypage dynamique sur une référence.

Héritage multiple

Définition

- L'héritage multiple permet de faire hériter une classe de plusieurs classes de base.



- La syntaxe est la suivante

```
class POINT_COLOR: public POINT, public COULEUR
```

- La classe `POINT_COLOR` hérite de tous les traits des classes `POINT` et `COULEUR`.

Fonction virtuelle

Une fonction membre d'une classe peut-être déclarée virtuelle

- les classes dérivées peuvent implémenter la même fonction.
- l'appel de la fonction au travers d'un pointeur sur un objet de la classe mère appellera la fonction de la classe dérivée adéquate.

```
1   class POINT
2   { virtual void afficher()
3     {std::cout << "(x,y) = ( " << x << " ," << y << " )"<<std::endl;}};
4   class POINT_COLOR: public POINT
5   { virtual void afficher()
6     {std::cout << "(x,y) = ( " << x << " ," << y << " )"<<std::endl;
7       std::cout<< "c= " << c <<std::endl}};
8   class POINT_RGB: public POINT
9   { virtual void afficher()
10    {std::cout << "(x,y) = ( " << x << " ," << y << " )"<<std::endl;
11      std::cout<< "RGB = (" << r << " , "<< g <<" , "<<b<< " )
12        << std::endl}};
```

```
1   int main()
2   {
3     POINT *A = (POINT *) (POINT_COLOR(2));
4     POINT *B = (POINT *) (POINT_RGB(2));
5     A->afficher(); // appel de POINT_COLOR::afficher();
6     B->afficher(); // appel de POINT_RGB::afficher();
7   }
```

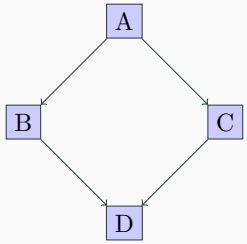
Classe abstraite

- L'implémentation de `POINT::afficher()` est inutile car les classes dérivées proposent la fonction `print()`.
- On introduit donc la notion de fonction virtuelle pure: c'est une fonction virtuelle dont on ne spécifie pas l'implémentation dans la classe de base.
  `virtual void print()=0;`
- La classe `POINT` est appelée classe abstraite: elle joue le rôle d'une interface.
- Il est impossible d'instancier un objet d'une classe abstraite.
- Les classes dérivées doivent implémenter les méthodes virtuelles pures de la classe de base.

```
1   class POINT
2   { virtual void afficher() = 0;};
3   class POINT_COLOR: public POINT
4   { virtual void afficher()
5     {std::cout << "(x,y) = ( " << x << " ," << y << " )"<<std::endl;
6       std::cout<< "c= " << c <<std::endl}};
7   class POINT_RGB: public POINT
8   { virtual void afficher()
9     {std::cout << "(x,y) = ( " << x << " ," << y << " )"<<std::endl;
10      std::cout<< "RGB = (" << r << " , "<< g <<" , "<<b<< " )
11        << std::endl}};
```

Classe virtuelle

- Dans la configuration suivante



  la classe D hérite (indirectement) deux fois de la classe A.

- Les membres de A apparaissent deux fois dans la classe D :
  - Les fonctions membres ne sont pas dupliquées
  - Les données membres sont dupliquées

- Si on souhaite utiliser cette duplication, on spécifie dans l'utilisation des membres la filiation
  A::B::x, A::C::x, B::x ou C::x.

- Si on ne souhaite pas utiliser cette duplication, on déclare la classe A comme virtuelle.
  `class B:public virtual A{};`,
  `class C:public virtual A{};` et
  `class D:public B, public C{};`

Constructeur

- Si A n'est pas virtuelle dans les classes B et C, les appels aux constructeurs seront: A, B, A, C et D.
- Si A est virtuelle dans les classes B et C, on ne construit qu'un seul objet de type A.
- Les arguments qui permettent de créer la classe A ne proviennent pas de B ou C: ils proviennent de la classe D.
- Dans la cas de classe virtuelle, on est autorisé à spécifier dans le constructeur de D des informations destinées à A

```
D(double a,double b):B(a,b),A(a,b)
```

Le constructeur d'une classe virtuelle est toujours appelé avant les autres.

Destructeur

- Le cas du destructeur virtuel est particulier : il doit toujours être défini même si il est vide.
- Un destructeur virtuel n'est jamais surchargé : les effets des destructeurs sont cumulatifs.
- Une classe virtuelle doit toujours contenir un destructeur virtuel

```
1  class InterfacePOINT
2  {
3    public:
4      virtual ~InterfacePOINT();
5  }
6  InterfacePOINT::~InterfacePOINT(){}
```

Le côté obscur de l'héritage virtuel

L'héritage virtuel complexifie la gestion de la table des méthodes virtuelles et des attributs.

L'héritage virtuel est trop cher pour être utilisé en permanence.

L'héritage virtuel ne doit s'utiliser que lorsque les conditions suivantes sont réunies :

1. une classe ancêtre commune à toute une bibliothèque.
2. les classes seront dérivées par un code utilisant la bibliothèque.
3. les classes pourront être associées par un code utilisant la bibliothèque.

Les amis

Les amis

- Un ami est un outil qui permet à une classe d'offrir des accès privilégiés à une autre classe ou fonction.
- Un ami renforce l'encapsulation s'il est utilisé correctement.
- Il existe plusieurs types d'amitiés
  1. fonction, amie d'une classe,
  2. fonction membre d'une classe, amie d'une autre classe,
  3. fonction amie de plusieurs classes,
  4. toutes les fonctions membres d'une classe, amies d'une autre classe.

Fonction amie d'une classe

- On souhaite vérifier que 2 **POINT** coincident

```
1   class POINT
2   {
3     ...
4       friend int coincide(const POINT &, const POINT &);
5   }
```

- La fonction **coincide** a besoin d'accéder à des données privées de la classe **POINT**. Elle peut soit utiliser des accesseurs, soit utiliser le mot clé **friend** qui permet de contourner la notion de propriété.

- Le mot clef **friend** peut se placer indifféremment dans la déclaration de la fonction.

- En général, une fonction amie d'une classe possède au moins un argument d'entrée ou de retour du type de la classe.

- Si ce n'est pas le cas, la fonction n'a pas besoin d'être déclarée amie.

Fonction d'une classe amie

```
1  class COLOR;
2  class POINT
3  {
4    ...
5    public:
6      friend int COLOR::colore(POINT &);
7  };
```

- La fonction membre **`colore`** permet de colorier en fonction des coordonnées de `POINT`.
- La fonction membre **`colore`** peut accéder à n'importe quel membre (public ou privé) de la classe `POINT`.
- Pour compiler sans erreur, il faut définir `COLOR` avant `POINT` et déclarer `POINT` avant `COLOR`.

Fonction amie de plusieurs classes

- Tout fonction membre ou indépendante peut-être déclarée amie de plusieurs classes

```
1   class POINT
2   {
3     ...
4     public :
5       friend void colore(POINT &, COLOR &)
6   };
7   class COLOR
8   {
9     ...
10    public :
11      friend void colore(POINT &, COLOR &)
12  };
13
14  void colore( POINT & p, COLOR & c)
15  {
16    ...
17  }
```

Toutes fonctions amies d'une autre classe

- Dans le cas où toutes les fonctions d'une classe sont déclarées amies d'une autre classe, on ne répète pas le mot clé `friend` pour chaque fonction.
- La solution est de déclarée la classe comme amie
  `friend class COLOR;`
- Afin de compiler la classe `POINT`, il faut déclarer la classe `COLOR` avant.
- Ce mécanisme (généralisable à d'autres classes) permet d'éviter de déclarer les entêtes.

Evaluer les avantages et inconvénients des amis

Le plus

- Elles offrent plus de possibilités de conception : une fonction amie et une fonction membre ont les mêmes privilèges. Seule la syntaxe d'appel est modifiée.
  La syntaxe pour une fonction amie est `f(objet &)`.
  La syntaxe pour une fonction membre est `objet.f()`.

Les moins

- L'amitié n'est pas héritée : une fonction amie d'une classe dérivée n'est pas amie d'une classe de base.
- L'amitié n'est pas transitive : une classe amie d'une classe amie n'est pas une classe amie (par défaut).
- L'amitié n'est pas réciproque.
- Elles ne comprennent pas le polymorphisme.
- Elles augmentent le risque de collisions.

Bilan sur les amis

Utilisez une fonction membre quand vous le pouvez,
utilisez une fonction amie quand vous le devez.

Les exceptions

Gestion des exceptions

- La détection d'un incident et son traitement dans les programmes importants doivent se faire dans des parties différentes du code.
- Plusieurs stratégies sont possibles
  - Les méthodes/fonctions renvoie un statut : oblige l'utilisateur à être vigilant.
  - Les expressions conditionnelles : gestion au cas par cas.
  - Les assertions : essentiellement utilisées en mode debug.
  - Les exceptions : flot de contrôle non local.

L'avantage des exceptions est le découplage total entre la détection d'une anomalie (exception) de son traitement en s'affranchissant de la hiérarchie des appels.

Gestion de l'exception

- Deux étapes de gestion
  - détection d'une erreur qui lève une exception avec la méthode : **`throw`**.
  - récupération de l'exception par un gestionnaire pour traitement : **`catch`**.
- une instruction susceptible de lever une exception se met dans un bloc particulier : **`try`**.
- lorsqu'une instruction est levée, la reprise de l'exécution se fait après le bloc **`try`**/**`catch`**.
- une exception levée par **`throw`** non traitée entraîne l'arrêt du programme.

Définition

- Les exceptions sont définies par un système de classes. La définition dans l'espace de nom `std` s'écrit

```
1    std
```

- `what` permet d'envoyer un message.
- `throw` lance une exception, c'est-à-dire un objet.
- Dans une bibliothèque, il est indispensable de dériver la classe `exception`

Exemple d’exception

```
1    exception
```

Lever une exception

```
1    exception
```

- Le constructeur est invoqué à la volée.
- On peut limiter les exceptions qu'une méthode est en mesure de lever
  `POINT(unsigned int d, double val) throw(ErreurAllocation)`

Gestionnaire d'exception

```
1    POINT(unsigned int d, double val) throw(ErreurAllocation)
```

- On ajoute deux autres parties dans le traitement
  - `Erreur` : Permet de traiter les erreurs futurs sans modifier le code client.
  - `std::exception` : Affiche simplement un message d'erreur.
- Si une exception n'est pas attrapée, elle est renvoyée dans la pile d'appel jusqu'à l'invocation de la terminaison.