

Atelier C++ Partie 2

Henri Louvin

Notions abordées durant ces cours

- Pointeurs
- Pointeurs intelligents
- **Vecteurs et itérateurs**
- Surcharge de fonction
- Surcharge d'opérateurs
- **Polymorphisme**
- (Templates)

Tout ça à à voir avec
la POO

Projet 'Flatland'

- But : explorer la programmation orientée objet en C++
- Trois fichiers (pour l'instant):
 - **figure.hh** → header de la classe "Figure"
 - **figure.cc** → code source de la classe "Figure"
 - **test.cc** → code source de l'exécutable (contient la fonction main())
- Fichier bonus pour la compilation à fichiers multiples: **Makefile**

Exemple de header de classe (**figure.hh**)

```
class classname
{
    private:
        type1 attrib;
        // ...

    public:
        classname(...);
        ~classname(...);
        type2 method(...);
        // ...
}
```

Exemple de définition de méthode dans le code source de la classe (**figure.cc**)

```
#include "figure.hh"

type2 classname::method(...)
{
    type2 val;
    // ...
    return val;
}
```

Header : **figure.hh**

```
class Figure
{
    private:
        double area;

    public:
        Figure();
        ~Figure();

        double get_area();
        void description();

};
```

Source : **figure.cc**

```
#include <iostream>
#include "figure.hh"

Figure::Figure()
{
    area = 1;
}

Figure::~~Figure()
{
}

double Figure::get_area()
{
    return this->area;
}

void Figure::description()
{
    cout<<"I am a figure of area "<<this->area<<endl;
}
```

Welcome to Flatland!

Contenu initial du fichier **test.cc**

```
#include <iostream>
#include "figure.hh"

using namespace std;

int main()
{
    Figure A;
    Figure B;

    cout<<"A: ";
    A.description();
    cout<<"B: ";
    B.description();

    return 0;
}
```

Compilation avec **make**

Créez un fichier **Makefile** dans le dossier du code avec le contenu

```
CXX=g++  
  
CXXFLAGS=-Wall  
  
LINK.o=$(LINK.cc)  
  
test: test.o figure.o  
  
clean:  
    $(RM) *.o test
```

Compilation/exécution depuis le terminal

```
$ make  
$ ./test
```


DE LA RECHERCHE À L'INDUSTRIE



Partie 3/5

Surcharge de fonction

Qu'est-ce que c'est ?

- Est-il possible de définir dans une même classe deux fonctions ayant le même nom? **OUI** À condition qu'elles ne prennent pas le même type et nombre de paramètres.
- Une fonction est définie par sa **SIGNATURE** :
 - Son nom
 - Ses paramètres
- Par exemple, pour le constructeur de la classe Figure:

```
Figure();
```

Constructeur
par défaut

```
Figure(double a);
```

On surcharge le constructeur
en permettant de lui passer
l'aire en paramètre

Dans le header (**figure.hh**)

```
class Figure
{
    // ...
    public:
        Figure();
        Figure(double);
    // ...
} ;
```

Dans le code source (**figure.cc**)

```
// ...
Figure::Figure()
{
    area = 1;
}
Figure::Figure(double _area)
{
    area = _area;
}
// ...
```

Mise en pratique

- Surchargez le constructeur de la classe **Figure** de façon à pouvoir lui passer l'aire de la figure en paramètre :

```
Figure A(5);  
Figure A = Figure(5);
```

- Le constructeur "sans paramètres" doit initialiser l'aire à **1**
- Adapter **test.cc** pour que la figure **a** utilise l'ancien constructeur, et la figure **b** le constructeur à deux paramètres
- L'exécution du code devrait renvoyer quelque chose du type:

```
$ ./test  
A: "I am a figure of area 1"  
B: "I am a figure of area 5"
```

Version pro (**figure.hh**)

```
class Figure
{
    // ...
    public:
        Figure(double area=1);
    // ...
} ;
```

Version pro (**figure.cc**)

```
// ...
Figure::Figure(double _area) : area(_area)
{
}
// ...
```

DE LA RECHERCHE À L'INDUSTRIE



Partie 4/5

Surcharge d'opérateur

Qu'est-ce que c'est ?

- Est-il possible de définir **pour n'importe quelle classe** des méthodes de type opérations ('+', '-', '==', ...) ? **OUI** À condition de connaître **l'opérateur associé**
- Même si l'on parle d'opérateur pour une classe, les opérateurs ne sont **pas des méthodes**. Ce sont des fonctions hors classes
- Par exemple, dans Flatland, on pourrait considérer qu'une figure est **supérieure** à une autre si son aire est plus grande ou que l'**addition** de deux figures revient à créer un « bébé » figure

Mise en pratique

- Surcharge des opérateurs '>' et '+' pour la classe **Figure**
- Quelles sont les entrées et sorties de chaque opération?
- Les définitions des opérateurs sont à ajouter dans le header de la classe mais **hors de la déclaration** de la classe:

```
bool operator>(Figure const&, Figure const&);
Figure operator+(Figure const&, Figure const&);
```

- Adapter **test.cc** pour que l'exécution du code renseigne sur la figure la plus grande parmi les figures **a** et **b** et affiche le résultat de la somme des deux figures.

```
$ ./test
A: "I am a figure of area 1"
B: "I am a figure of area 5"
--
B is greater than A
A+B : "I am a figure of area 6"
```


Mise en pratique: solution

- Surcharge des opérateurs '>' et '+' pour la classe **Figure**
- Dans **figure.hh** (obligatoire pour include!)

```
bool operator>(Figure const&, Figure const&);
Figure operator+(Figure const&, Figure const&);
```

- Dans **figure.cc**

```
bool operator>(Figure const&, Figure const&);
{
    return a.get_area()>b.get_area();
}

Figure operator+(Figure const&, Figure const&);
{
    return Figure(1);
}
```

DE LA RECHERCHE À L'INDUSTRIE



Partie 5/5 **Polymorphisme**

Rappels(?) : héritage

- L'héritage permet de créer des classes dérivant d'autres classes
- Une classe "filles" contiendra **de base** tous les attributs et toutes les méthodes de sa "mère"
- Nous allons créer deux classes **Circle** et **Square** héritant de la classe **Figure**

"Puisque les cercles et les carrés sont des figures"

- Créer **circle.cc**, **circle.hh**, **square.cc**, **square.hh** (ou pas)
- Faire gaffe aux includes !!
- Updater le **Makefile**

Exemple de fichier **circle.hh**

```
#include "figure.hh"

class Circle : public Figure
{
    private:
        double radius;

    public:
        Circle();
        Circle(double); // initialize with area
}
```

Exemple de fichier **square.hh**

```
#include "figure.hh"

class Square: public Figure
{
    private:
        double side;

    public:
        Square();
        Square(double); // initialize with area
}
```

Héritage : exécution

- Remplacer dans **test.cc** les deux **Figures** par un **Circle** et un **Square**

Exemple de contenu de **test.cc**

```
// ...
Square A(2);
Circle B(5);

cout<<"A: \n";
A.description();
cout<<"B: \n";
B.description();
// ...
```

- Quelques problèmes de compilation ?
 - **Attention** à ajouter `#include "circle.hh"` dans **test.cc**
 - **Attention** aux doubles initialisations de classe (compilateur-dépendant) :

`#ifndef _CIRCLE_H`

`#define _CIRCLE_H`

`#endif`
 - **Attention** à modifier les attributs **private** en **protected**

Héritage : explications du résultat de l'exécution

```
Square A(2);  
Circle B(5);  
  
cout<<"A: \";  
A.description();  
cout<<"B: \";  
B.description();
```

```
$ ./test  
A: "I am a figure of area 1"  
B: "I am a figure of area 1"
```

Facile : la méthode
description() est celle
de la classe **Figure**

Plus sioux : Quel constructeur
a été appelé ?!

Constructeurs et héritage :

À l'appel du constructeur d'une classe fille,
le constructeur **par défaut** de la classe
mère est appelé **automatiquement**

Héritage : premiers fix

Exemple fichier **circle.cc** corrigé

```
#include <iostream>
#include "circle.hh"
#include <math.h>

Circle::Circle(double a): Figure(a)
{
    radius = sqrt(a/M_PI);
}

void Circle::description()
{
    cout<<"\nI am a circle of area "<< area <<" and radius "<<radius<<"\n"<<endl;
}
```

Appel du constructeur **non-défaut** de la classe mère !

Masquage de la méthode `description()`

← Quid de la **signature** de la fonction ?
Pas de souci, les noms sont différents :
"Circle::description" != "Figure::description"

Héritage : explications du résultat de l'exécution 2

```
Square A(2);  
Circle B(5);  
  
cout<<"A: \";  
A.description();  
cout<<"B: \";  
B.description();
```

```
$ ./test  
A: "I am a square of area 2"  
B: "I am a circle of area 5"
```

Victoire !

Que se passe-t-il si...

- Disons qu'on décide d'homogénéiser les descriptions :
 - En passant l'appel à la méthode **description()** dans une fonction hors du **main()**

Utilisation d'une fonction **describe()** dans **test.cc**

```
// ...  
Square A(2);  
Circle B(5);  
  
describe('A',A);  
describe('B',B);  
// ...
```

Solution

Exemple de contenu pour le fichier **test.cc**

```
#include <iostream>
#include "circle.hh"

using namespace std;

void describe(char name, Figure F)
{
    cout<<name<<": \n";
    F.description();
}

int main()
{
    Figure A(2);
    Circle B(5);

    describe('A',A);
    describe('B',B);

    return 0;
}
```

La résolution de liens : statique ou dynamique ?

➤ Test d'exécution du code

```
// ...
Square A(2);
Circle B(5);

describe('A',A);
describe('B',B);
// ...
```

```
$ ./test
A: "I am a figure of area 2"
B: "I am a figure of area 5"
```

On a de nouveau perdu
le type "circle" !!!

Par contre, l'aire est la
bonne ?!

Ce qui a changé :
La méthode **description()** est appelée
depuis la fonction **describe()**

Résolution de liens : explications

Isolons la fonction **describe()** du reste du code

```
// ...
void describe(char name, Figure F)
{
    cout<<name<<": \n";
    F.description();
}
// ...
```

Pour le compilateur, quel
est le type de **F** ici ?

Header de la classe **Figure**

```
class Figure
{
    // ...
    void description();
    // ...
}
```

Lien entre l'appel et la fonction :

Le compilateur appelle la "version **Figure**" de la méthode parce que la variable est de type **Figure**

→ **Résolution statique de liens**

Résolution dynamique de liens et polymorphisme

- On souhaite appeler à l'exécution la "bonne version" de la méthode
- Il est possible que le type **dynamique** d'une variable ne puisse pas être connu à la compilation
- Le compilateur doit permettre au système d'effectuer l'appel de la fonction à la volée

Résolution dynamique de liens :

La même instruction aura deux comportements différents selon le type de variable → "**POLYMORPHISME**"

Plus simple qu'il n'y paraît !

Les méthodes virtuelles

```
// ...
void describe(char name, Figure F)
{
    cout<<name<<": \n";
    F.description();
}
// ...
```

La fonction trouvée par le compilateur est déclarée "virtuelle"

Header de la classe **Figure**

```
class Figure
{
    // ...
    virtual void description();
    // ...
}
```

À l'exécution, le type de **F** est résolu et le lien dynamique est suivi

Header de la classe **Circle**

```
class Circle : public Figure
{
    // ...
    void description();
    // ...
}
```

POLYMORPHISME ! \o/

Les deux ingrédients du polymorphisme

- Le mot clé **virtual** pour définir les méthodes virtuelles
 - Seulement dans le fichier header (**.hh**)
 - Les méthodes virtuelles héritées par les classes filles sont virtuelles par héritages
- Il est nécessaire d'utiliser des **pointeurs** ou des **références** (types dynamiques)

Références et résolutions de lien

Deux versions de la fonction `describe()`

Passage par copie

```
// ...
void describe(char name, Figure F)
{
    cout<<name<<": \n";
    F.description();
}
// ...
```

La figure passée en argument est **COPIÉE** dans un type **Figure**.

L'objet résultant est du type **Figure** et **n'est pas l'objet** passé en argument puisque c'est une copie

La résolution de lien est obligatoirement statique

Passage par référence

```
// ...
void describe(char name, Figure const& F)
{
    cout<<name<<": \n";
    F.description();
}
// ...
```

L'objet passé en argument est une **RÉFÉRENCE** vers la Figure de base.

L'objet **F** dans le corps de la fonction **est réellement l'objet** passé en argument

La résolution de lien peut être dynamique

Un autre exemple de la nécessité de polymorphisme

```
#include <iostream>
#include "circle.hh"

using namespace std;

int main()
{
    Figure *FF, *FC;
    Circle *CC;

    FF = new Figure(2);
    FC = new Circle(5);
    CC = new Circle(5);

    cout<<"FF: ";
    FF->description();
    cout<<"FC: ";
    FC->description();
    cout<<"CC: ";
    CC->description();
    cout<<"--"<<endl;

    return 0;
}
```

Que se passe-t-il là, là et là si la méthode **description()** n'est pas déclarée virtuelle ?

DE LA RECHERCHE À L'INDUSTRIE



Partie Bonus **Templates**

Familles de figures

- Suppositions :
 - Dans Flatland, les bébés figures naissent avec une aire de **1** et ont deux parents
 - Deux parents de même type ont des enfants de leur type
 - L'enfant de deux parents de types différents a le type d'un de ses parents (probabilité 0.5)
- Créez une fonction **kid** prenant deux **Figures** en argument et retournant une **Figure** d'aire 1

Utilisation de la fonction dans **test.cc**

```
// ...  
Figure C(2);  
Figure D(5);  
  
Figure A = kid(C,D);  
// ...
```

Familles de figures

- Si l'on souhaite définir la fonction **kid()** pour chaque type de figure et chaque combinaison de parents, il faut masquer et/ou surcharger la fonction
- Ou alors, définir un **template**
- Les templates sont des "patrons" de fonctions, permettant de définir des fonctions s'appliquant sur des **types non définis**
- Pour les reproductions mono-type, le template serait :

```
template<typename Type> Type kid(Type const &parent1, Type const &parent2)
{
    return Type(1);
}
```

- À placer **dans un header** (figure.hh c'est bien)

Avant-dernier exercice

- Remplacez la fonction **kid** par un template permettant de générer des enfants de deux **Figures**, de deux **Circles**, ou de deux **Squares**

```
template<typename Type> Type kid(Type const&, Type const&)
{
    return Type(1);
}
```

- L'utilisation du template est la même que celle de l'ancienne fonction :

Utilisation du template dans **test.cc**

```
// ...
Figure C(2);
Figure D(5);

Figure A = kid(C,D);
// ...
```

Dernier exercice

- L'enfant de deux parents de types différents a le type d'un de ses parents (probabilité 0.5)
- Créez un template pour la fonction **kid** pour deux parents de type différents. La syntaxe est du type :

```
template<typename Type1, typename Type2> Type kid(Type1 parent1, Type2 parent2)
```

- Pour pouvoir utiliser la fonction **rand()** renvoyant un entier aléatoire entre **0** et **RAND_MAX**

```
#include <random>
```

- Pour initialiser la seed du générateur de nombre aléatoires (dans **test.cc**) :

```
srand(time(NULL))
```

Solution

Définition du template (**figure.hh**)

```
#include <random>
#include "circle.hh"

// ...

template<typename Type1, typename Type2>
Figure* kid(Type1&, Type2&)
{
    Figure* baby;

    double ksi = ((double) rand())/RAND_MAX;
    if(ksi>0.5)
        baby = new Type1(1);
    else
        baby = new Type2(1);

    return baby;
}
```

Utilisation (**test.cc**)

```
// ...

Circle A(2);
Square B(5);

Figure *C = kid(A,B);

cout<<"--"<<endl;
cout<<"dad: "<<A.description()<<endl;
cout<<"mom: "<<B.description()<<endl;
cout<<"baby: "<<C->description()<<endl;
cout<<"--"<<endl;

// ...

delete C; C=0;
```

Solution 2 : avec des unique_ptr

Définition du template (**figure.hh**)

```
template<typename Type1, typename Type2> std::unique_ptr<Figure> kid (Type1& papa1,
Type2& papa2)
{
    unique_ptr<Figure> baby;
    Type1 t1;
    if(papa1.get_name()==t1.get_name())
        baby = std::unique_ptr<Type1>(new Type1(1));
    else
        baby = std::unique_ptr<Type2>(new Type2(1));
    return baby;
}
```

Utilisation (**test.cc**)

```
unique_ptr<Circle> A(new Circle(2));
unique_ptr<Square> B(new Square(2));

unique_ptr<Figure> C = kid(A,B);
```


Récupération des slides & stuff :

- Via **git** :

```
$ git clone https://github.com/henrilouvin/cpp.git
```

- Via browser :

<https://github.com/henrilouvin/cpp>