Rappel sur :
librairies statiques et dynamiques

librairies statiques et dynamiques

- Utiliser des librairies statiques, revient à inclure la définition des fonctions de la librairie dans votre fichier exécutable, pendant l'étape de l'édition de liens (donc pendant la compilation et *avant* le lancement du programme).
  - Avantage : l'exécutable qui en résulte contient, avant l'exécution, tout ce qui lui est nécessaire pour fonctionner
  - Inconvénient :
    - Fichier trop lourd,
    - S'il y a une mise à jour de la librairie, il faut recompiler le programme
- Utiliser des librairies dynamiques, revient à indiquer à votre programme l'emplacement d'où il pourra charger en mémoire ces définitions *après* le lancement du programme.
  - Avantage :
    - Programme léger
    - Pas besoin de recompiler le programme s'il y a une mise à jour de la bibliothèque
  - Inconvénient
    - Les bibliothèques doivent être disponibles sur la machine hôte qui va exécuter votre programme
    - Attention au chemin sur la machine hôte (variables d'environnement), sinon il faut recompiler votre programme

Linux : librairies statiques et dynamiques

- Les répertoires
  - /usr/include/ : il contient les fichiers d'en tête des librairies installées sur votre système.
  - /usr/lib/ : il contient les librairies binaires installées sur votre système
- Statique
  - libjpeg.a (-ljpeg), libGL.a (-lGL), libm.a (-lm), libsocket.so(-lsocket)
  - g++ prog.cpp -Lchemin -lXXX -o prog (Chemin optionnel si le chemin est inclus dans la variable d'environnement. Option liée au nom de la librairie)
  - Crétaion de librairie : ar -rv libname.a prog1.o prog2.o... progn.o (la commande, inserer d'autres fichiers, verbose)
  - Utilisation : #include<jpeglib.h>

- Dynamique
  - libname.so (libjpeg.so)
  - Variable d'environnement : export LD_LIBRARY_PATH=/usr/local/lib
  - g++ prog.cpp -Lchemin -lXXX -o prog (Chemin optionnel si le chemin est inclus dans la variable d'environnement. Option liée au nom de la librairie)
  - Création de librairie : gcc -o libname.so -shared prog1.o prog2.o ... progn.o
  - Utilisation #include<jpeglib.h>, puis appeler la fonction à utiliser

Remarque : Si la librairie est dans le même repertoir qeu le programme le fichier qui l'utilise alors **#include ''libname.h''**

Windows : librairies statiques et dynamiques

- Le répertoire dans chaque application

- Statique
  - libjpeg.lib (-ljpeg)
  - g++ prog.cpp -Lchemin -lXXX -o prog (Chemin optionnel si le chemin est inclus dans la variable d'environnement. Option liée au nom de la librairie)
  - Crétaion de librairie : ar -rv libname.a prog1.o prog2.o... progn.o (la commande, inserer d'autres fichiers, verbose)
  - Utilisation : #include<jpeglib.h>

- Dynamique
  - libname.dll (dllMatrcie.dll)
  - Variable d'environnement : export LD_LIBRARY_PATH=/usr/local/lib
  - Création de librairie : gcc -o libname.so -shared prog1.o prog2.o ... progn.o
  - Utilisation #include<jpeglib.h>

Remarque : Si la librairie est dans le même repertoir qeu le programme le fichier qui l'utilise alors **#include ''libname.h''**

Séance 4

Héritage & Polymorphisme

Plan

- Héritage
- Polymorphisme
- Fonction virtuelle
- Classe abstraite

Héritage: Introduction

- L'héritage permet d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise => c'est la relation « est un » plus quelque chose.
- **Exemple:**

  Un étudiant est une personne, il a un nom et un prénom (personne). De plus, il a un numéro d'étudiant.

  => Etudiant est dérivée de Personne
- **Principe:**
  - Déclarer une classe de base, dite aussi classe parente (Personne) pour déclarer une classe dérivée (Etudiant).
  - La classe dérivée hérite tous les membres (données et méthodes) de la classe de base.

Héritage: Introduction

```
class Personne {
    private:
        string nom;
    public:
        string getNom() const { return nom; }
};

class Etudiant : public Personne {
    private:
        string id;
    public:
        string getId() const { return id; }
};
```

Héritage: Principe

- L'héritage permet :
  - la réutilisation du code déjà écrit
  - l'ajout de nouvelles fonctionnalités
  - la modification d'un comportement existant (redéfinition)

Attention => Redéfinition vs Surcharge

Héritage : Principe

```
class Personne {
    private:
        string nom;
    public:
        string getNom() const;
        void afficher() const { cout << nom << endl; }
};

class Etudiant : public Personne {
    private:
        string id;
    public:
        string getId() const;              // ajout de nouvelles fonctionnalités
        void afficher() const {            // modification d'un comportement existant
                Personne::afficher();      // réutilisation du code déjà écrit
                cout << id << endl;
        }
};
```

Héritage : Constructeur & Destructeur

- Le constructeur de la classe de base est appelé **avant** le constructeur de la classe dérivée.
  - Ou on appelle explicitement un constructeur de la classe mère dans le constructeur de la classe fille
  - Sinon, le constructeur par défaut de la classe mère est appelé automatiquement (le compilateur envoie un message d'erreur si celui n'existe pas)

- *Les destructeurs sont appelés dans l'ordre inverse.*

Héritage : Constructeur

```
class Personne {
    private:
        string nom;
    public:
        Personne(string n) : nom {n} { }
};
class Etudiant : public Personne {
    private:
        string id;
    public:
        Etudiant(string nom, string id) : Personne(nom), id {id} { }
};
```

Héritage publique et privé

```
class ClasseBase {
    /* Contenu de la classe de base */
};

class ClasseDerivee: public|protected|private ClasseBase {
    /* Définition de la classe dérivée */
};
```

Héritage: droits d'accès

mode de dérivation	Statut dans la classe de base	Statut dans la classe dérivée
public	public	public
	protected	protected
	private	inaccessible
protected	public	protected
	protected	protected
	private	inaccessible
private	public	private
	protected	private
	private	inaccessible

exemple

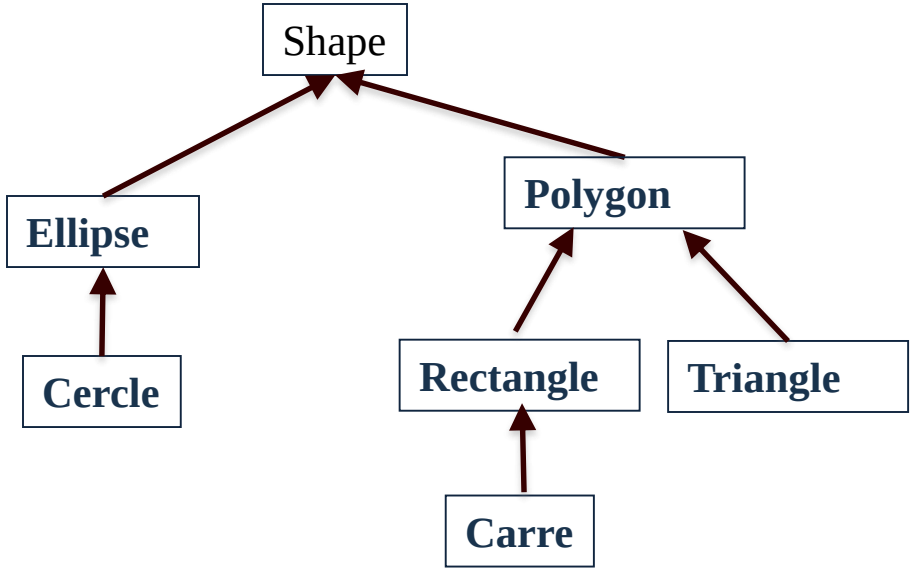
```
class Personne {
    private:
         string nom;
    public:
        Personne (string s) : nom(s){}
       string getNom() const {
                return nom;
         }
};
class Etudiant : protected Personne {
   private:
         int id;
    public:
  Etudiant (string s, int i) : Personne(s),  id(i){}
  int getId() const { return id;  }
};
```

```
#include <iostream>
using namespace std;
int main()  {
  Etudiant e("Joe",3);
  cout << e.getNom() << endl;
  cout<< e.getId() << endl;
   return 0;
}
```

Héritage simple

L'héritage permet de spécialiser une classe en définissant une relation de type « est une sorte de ».

Un cercle est un spécialisation d'une ellipse, il en possède les propriétés plus d'autres qui lui sont spécifiques.
On dérive donc la classe Cercle de la classe Ellipse.



Héritage multiple

```
class A {
        ...
};
class B {
        ...
};
class C: public A, public B {
        ...
};
```

- La classe C hérite de A et B : une instance de C possède à la fois les données et les fonctions membres de A et B.
- Quand un objet C est créé, le constructeur de A est appelé en premier, en suite celui de B (dans l'ordre).
- Quand un objet C est détruit, le destructeur de B est appelé en premier, après celui de A (sens inverse).

Polymorphisme

- Un objet polymorphe est un objet susceptible de prendre plusieurs formes pendant l'exécution.

- Le polymorphisme représente la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation.

- Le polymorphisme est implémenté en C++ avec les fonctions virtuelles (**virtual**) et l'héritage.

Héritage

L'héritage permet de spécialiser une classe en définissant une relation de type « est une sorte de ».

Un cercle est un spécialisation d'une ellipse, il en possède les propriétés plus d'autres qui lui sont spécifiques.
On dérive donc la classe Cercle de la classe Ellipse.

ellipse.h

```
class Ellipse
{
public :
      Ellipse();
      Ellipse (float x, float y, float a, float b);
      Ellipse(const Ellipse & e);
     ~Ellipse();

protected :
     float m_cX, m_cY;
     float m_a, m_b;

public :
     void deplace(float dx, float dy);
     void zoom(float z);
     float surface();
     virtual void affiche();
};
```

Autorise la redéfinition de la fonction dans les classes dérivées

ellipse.cpp

```
#include <iostream.h>

void Ellipse::affiche()
{
 std::cout << "Ellipse de grand axe " << m_a;
std ::cout << " et de petitt axe " << m_b << "\n";
}
```

cercle.h

```
#include "ellipse.h"

class Cercle : public Ellipse
{
public :
     Cercle();
      Cercle (float x, float y, float  diametre);
     ~ Cercle();
public :
     void affiche() override;
};
```

prog.cpp

```
#include  "cercle.h"

int main()
{
     Cercle c(5, 5, 15);
     c. affiche();
     return 0;
}
```

cercle.cpp

```
#include <iostream.h>
#include "cercle.h"

Cercle::Cercle() : public Ellipse()
{
}
Cercle::Cercle(float x, float y, float  diametre)  : public Ellipse( x, y, diametre, diametre)
{
}
void Cercle::affiche()
{
     std::cout << "Cercle de rayon " <<  m_a / 2 << "\n";
}
```

Le constructeur de la classe dérivée appelle généralement un des constructeurs de la classe de base.

Polymorphisme

polynom.h

```
class Polygon {
    protected :
        int width, height;
    public:
        virtual int getArea() {return 0;}
};
```

polynom.cpp

```
class Rectangle : public Polygon {
    public:
        int getArea() override { return width*height; }
};
class Triangle : public Polygon {
    public:
        int getArea() override { return width*height/2; }
};
```

Polymorphisme

main.cpp

```
int main(){
vector<const Polygon *> vp;
Polygon * p1 = new Rectangle(4,5);
Polygon * p2 = new Triangle(4,5);
vp.push_back(p1);
vp.push_back(p2);

for (auto p : vp) {
        cout << p->getArea() << endl;
}

// n'oubliez pas delete !
return 0;
}
```

main.cpp

```
int main(){
vector<const Rectangle *> vr;
Polygon * r1 = new Rectangle(4,5);
Polygon * t2 = new Triangle(4,5);
Polygon * r3 = new Rectangle(4,5);
vr.push_back(r1); //OK
vr.push_back(t2); //NOK
vr.push_back(r3); //OK
for (auto r : vr) {
        cout << r->getArea() << endl;
}
// n'oubliez pas delete !
return 0;
}
```

Polymorphisme

Un objet héritant une méthode d'une classe parente peut réagir de façon différente à l'appel de cette méthode.

ellipse.h

```
class Ellipse
{
public :
      Ellipse();
      Ellipse (float x, float y, float a, float b);
      Ellipse(const Ellipse & e);
     ~Ellipse();

protected :
     float m_cX, m_cY;
     float m_a, m_b;

public :
     void deplace(float dx, float dy);
     void zoom(float z);
     float surface();
     virtual void affiche();
};
```

cercle.h

```
#include “ellipse.h”

class Cercle : public Ellipse
{
public :
     Cercle();
      Cercle (float x, float y, float  d);
     ~ Cercle();
public :
     void affiche() override;
};
```

cercle.cpp

```
#include “cercle.h”

Cercle::Cercle() : public Ellipse()
{
}

Cercle::Cercle(float x, float y, float  d)  :
    public Ellipse( x, y, d, d)
{
}

void Cercle::affiche()
{
   std::cout << "Cercle de rayon ";
   std::cout << m_a / 2 << "\n";
}
```

ellipse.cpp

```
#include <iostream.h>

void Ellipse::affiche()
{
 std::cout << "Ellipse de grand axe " << m_a;
std ::cout << " et de petit axe " << m_b << "\n";
}
```

main.cpp

```
#include  " cercle.h"

int main()
{
    Ellipse  e(0, 0, 8.5, 10.2);
    e.deplace(-1, 1);
    e.affiche();

    Cercle  c(-2.5, 2.5, 7,4);
    c.deplace(0.5, 1.5);
    c. affiche();

     Ellipse *p1;
     p1 = new Ellipse;
     p1 ->affiche();

     Ellipse *p2;
     p2 = new Cercle;
     p2->affiche();

    return 0;
}
```

La fonction deplace() n’est pas redéfinie dans la classe Cercle, appelle celle de Ellipse.

La fonction affiche() est redéfinie dans la classe Cercle, appelle celle de Cercle.

Appelle la fonction affiche() de la classe Ellipse.

Si la fonction affiche() est virtuelle et est redéfinie dans la classe Cercle, appelle celle de Cercle bien que le pointeur soit de type Ellipse. C'est le mécanisme de polymorphisme d'héritage.

Classe abstraite

- Une fonction membre est dite virtuelle pure lorsqu'elle est déclarée de la façon suivante :

  virtual type nomMethode(paramètres) =0;

- Une classe est dite abstraite si elle contient au moins une fonction virtuelle pure.

- Une classe abstraite ne peut pas être instanciée : on ne peut pas créer d'objet à partir d'une classe abstraite.
- Il est obligatoire d'avoir une définition pour les fonctions virtuelles pures au niveau des classes dérivées

Classe abstraite

```
class Polygon {
    protected :
        int width, height;
    public:
        virtual int getArea() = 0;    //methode virtuelle pure
};
```

- La classe Polygon ne peut pas être instanciée : elle est dite abstraite
- La méthode getArea() est virtuelle pure et ne possède aucune définition
  => toutes les classes dérivées doivent contenir une méthode getArea()
- Une classe dans laquelle il n'y a plus une seule fonction virtuelle pure est dite concrète et devient instanciable.

Pour résumer

```
class Shape {
    public :
        virtual void draw() const = 0;
        virtual void error(const string & msg);
        int objectID() const;
        ...
};

class Rectangle : public Shape { ... };

class Ellipse : public Shape {...};
```

Lorsque vous utilisez **le remplacement**, le compilateur génère des erreurs au lieu de créer silencieusement de nouvelles fonctions membres.

```
class BaseClass
{
   virtual void funcA();
   virtual void funcB() = 0 const;
   virtual void funcC(int = 0);
   void funcD();
};
class DerivedClass: public BaseClass
{
   virtual void funcA() ; //
   virtual void funcB() ; //
   virtual void funcC( double = 0.0 ) ; //
   void funcD() ;  //
}
```

Utilisez la substitution pour empêcher le comportement d'héritage par inadvertance dans votre code.

```
class BaseClass // (classe abstraite)
{
   virtual void funcA() ;
   virtual void funcB() = 0 const;
   virtual void funcC(int = 0) ;
   void funcD() {};
};

class DerivedClass: public BaseClass
{
   virtual void funcA() ; // ok, works as intended

   virtual void funcB()   ; // DerivedClass::funcB() is non-const, so it does not
                            // override BaseClass::funcB() const and it is a new member function

   virtual void funcC(double = 0.0); // DerivedClass::funcC(double) has a different
                                     // parameter type than BaseClass::funcC(int), so
                                     // DerivedClass::funcC(double) is a new member function
  void funcD() ;  // NOK, doublon. Le même nom dans la classe mere

};
```

Lorsque vous utilisez **le remplacement**, le compilateur génère des erreurs au lieu de créer silencieusement de nouvelles fonctions membres.

```
class BaseClass
{
   virtual void funcA();
   virtual void funcB() = 0 const;
   virtual void funcC(int = 0);
   void funcD();
};
class DerivedClass: public BaseClass
{
   virtual void funcA() override; //
   virtual void funcB() override; //
   virtual void funcC( double = 0.0 ) override; //
                                                //
   void funcD() override;  //
};
```

Lorsque vous utilisez **le remplacement**, le compilateur génère des erreurs au lieu de créer silencieusement de nouvelles fonctions membres.

```
class BaseClass
{
   virtual void funcA();
   virtual void funcB() = 0 const;
   virtual void funcC(int = 0);
   void funcD();
};
class DerivedClass: public BaseClass
{
   virtual void funcA() override; // ok
   virtual void funcB() override; // compiler error: DerivedClass::funcB() does not
                                  // override BaseClass::funcB() const
   virtual void funcC( double = 0.0 ) override; // compiler error:
                                                // DerivedClass::funcC(double) does not
                                                // override BaseClass::funcC(int)
   void funcD() override;  // compiler error: DerivedClass::funcD() does not
                           // override the non-virtual BaseClass::funcD()
};
```

Spécificateur final

- Pour spécifier que les méthodes ne peuvent pas être substituées et que les classes ne peuvent pas être héritées, utilisez la mot clé finale.

- final respecte le contexte et a une signification particulière uniquement lorsqu'il est utilisé après une déclaration de méthode ou un nom de classe ; sinon, il ne s'agit pas d'une mot clé réservée.

Spécificateur final (suite)

- L'exemple suivant utilise la mot clé **finale** pour spécifier qu'une fonction virtuelle ne peut pas être substituée.

```
#include <iostream>
class BaseClass
{
    virtual void afficher()  final {
        std::cout << "je suis la classe de base";
   };
};

class DerivedClass: public BaseClass
{
   // compiler error: attempting to
   // override a final function
   virtual void afficher(){
      std::cout << "je suis la classe derivee";
   };
};
```

```
#include <iostream>
int main()
{
     Deriveclass dc;
     dc.afficher();
      return 0;
};
```

Spécificateur final (suite)

L'exemple suivant utilise la **dernière** mot clé pour spécifier qu'une classe ne peut pas être héritée.

```
class BaseClass final
{
};

class DerivedClass: public BaseClass // compiler error: BaseClass is
                                     // marked as non-inheritable
{
};
```

Pour résumer

```
class Shape {
    public :
        virtual void draw() const = 0;                  // fonction virtuelle pure
        virtual void error(const string & msg); // fonction virtuelle
        int objectID() const;                           // fonction non-virtuelle
        ...
};

class Rectangle : public Shape { ... };

class Ellipse : public Shape {...};
```

Pour résumer

- Fonction virtuelle pure : héritage d'interface
  - classes dérivées doivent hériter l'interface de la fonction
  - classes dérivées concrètes doivent fournir l'implémentation de la fonction
- Fonction virtuelle : héritage d'interface + héritage d'implémentation par défaut
  - classes dérivées doivent hériter l'interface de la fonction
  - elles peuvent hériter l'implémentation par défaut si elles veulent
- Fonction non-virtuelle : héritage d'interface + héritage d'implémentation obligatoire
  - classes dérivées doivent hériter l'interface et l'implémentation de la fonction (invariance par rapport aux spécialisations)

Vocabulaire

- **Variable** : associe un nom (un symbole) à une valeur qui peut éventuellement varier au cours du temps ; une variable est d'un type donné, défini une fois pour toute (type prédéfini dans le langage ou créé par le développeur).
- **Encapsulation** : regroupement des variables et des fonctions au sein d'une même entité appelée « classe ».
- **Classe** : prototype qui définit des attributs et des méthodes communes à tous les objets d'une certaine nature.
- **Interface de classe** : description de la structure interne de la classe incluant une liste des données membres et le prototype des fonctions membres ; dans un « .h ».
- **Implantation de classe** : définition (code) des fonctions déclarées dans l'interface de classe ; dans un « .cpp ».
- **Instanciation d'un objet** : permet de créer un objet d'un type donné ; analogue à une déclaration de variable.
- **Héritage** : permet de définir une hiérarchie de classe, chaque classe fille héritant des méthodes et des données de son/ses antécédent(s).
- **Polymorphisme** : deux objets héritant une même méthode d'une classe parente, peuvent réagir de façon différente à l'appel de cette méthode et, à cette fin, redéfinir la méthode. Il est ensuite possible d'appeler la méthode d'un objet sans se soucier de son type intrinsèque.

Héritage

L'héritage permet de spécialiser une classe en définissant une relation de type « est une sorte de ».

comptebancaire.h

```
Class CompteBancaire

{
public :
    CompteBancaire();
    CompteBancaire(std::string p, int num);
    ~ CompteBancaire();
protected :
    int m_numéro;
    int m_solde;
    std::string m_propriétaire;
public :
    void Créditer(float);
    void Débiter(float);
    void Fermer();
    int Numéro();
    int Solde();
    std::string Propriétaire();
};
```

compteepargne.h

```
#include "comptebancaire.h"

class CompteEpargne : public CompteBancaire
{
public :
     CompteEpargne (std::string p, int num);
    ~ CompteEpargne ();
protected :
    float m_tauxIntérêt;
public :
    void FixerTauxIntérêt();
};
```

prog.cpp

```
#include <iostream.h>
#include  " compteepargne.h"

int main()
{
    CompteEpargne ce;
    ce. FixerTauxIntérêt(2.25);
    ce.Crediter(1000);
    std::cout << ce. Solde() << std::endl;
    return 0;
}
```

compteepargne.cpp

```
#include "compteepargne.h"

CompteEpargne::CompteEpargne() : public
    CompteBancaire()
{
}
CompteEpargne::CompteEpargne(std::string p,
    int num) : public CompteBancaire( p, num)
{
}
```

Le constructeur de la classe dérivée appelle généralement un des constructeurs de la classe de base.

Polymorphisme

Un objet héritant une méthode d'une classe parente peut réagir de façon différente à l'appel de cette méthode.

comptebancaire.h

```
Class CompteBancaire

{
public :
    CompteBancaire();
    CompteBancaire(std::string p, int num);
    ~ CompteBancaire();
protected :
    int m_numéro;
    int m_solde;
    std::string m_propriétaire;
public :
    void Créditer(float);
    virtual void Débiter(float);
    void Fermer();
    int Numéro();
    int Solde();
    std::string Propriétaire();
};
```

Autorise la redéfinition de la fonction dans les classes dérivées

compteepargne.h

```
#include "comptebancaire.h"

class CompteEpargne : public CompteBancaire
{
public :
    CompteEpargne (std::string p, int num);
   ~ CompteEpargne ();
protected :
    float m_tauxIntérêt;
public :
    void FixerTauxIntérêt();
    virtual void Débiter(float);
};
```

prog.cpp

```
#include  " compteepargne.h"

int main()
{
    CompteBancaire cb;
    cb. Débiter(1200);

    CompteEpargne ce;
    ce. Débiter(500);

    return 0;
}
```

Si la fonction Debiter() n'est pas redéfinie dans la classe CompteEpargne, appelle celle de CompteBancaire.

comptebancaire.h

```
#include "comptebancaire.h"

void CompteEpargne:: Débiter(float v)
{
    m_solde -= v;
}
```

compteepargne.cpp

```
#include "compteepargne.h"

void CompteEpargne:: Débiter(float v)
{
    if (v <m_solde)
            m_solde -= v;
}
```