Plan

- Introduction, rappels, généralités
- Approche impérative
- Approche fonctionnelle
- Approche objet
- Approche générique
- Compléments

Points abordés

- Concepts objets
- Définition d'une classe, d'une méthode
- Constructeur, destructeur
- Constructeur de copie, affectation par copie
- Méthodes et attributs de classe
- Héritage
- Identification dynamique de type

Approche objet

- Simula au Norwegian Computing Centre d'Oslo (1967)
- Smalltalk au PARC (Palo Alto Research Center) de Xerox (1976)
- Approche procédurale : basée sur une décomposition de fonctions, chaque fonction offre un service (en utilisant les services fournis par d'autres fonctions), les données sont partagées
- Approche objet : basée sur une décomposition d'entités collaborantes, chaque entité offre des services (en utilisant les services fournis par d'autres entités) et mémorise des données
  - Avantages : évolutivité, modularité

Objet

- Objet = état + comportement + identité
  - état : données mémorisées par l'objet, résulte de l'histoire de l'objet
  - comportement : services rendus
  - identité : deux objets ayant le même état et le même comportement restent deux objets distincts
- État = attributs + liens vers d'autres objets
  - l'identité peut être vu comme un attribut implicite et non modifiable

Comportement

- Un service particulier est
  - identifié par une opération
  - réalisé par une méthode
- Un objet (émetteur) demande l'exécution d'un service à un autre objet (récepteur) en envoyant un message précisant l'opération demandée et les éventuels arguments nécessaires
  - sur réception du message, l'objet récepteur exécute la méthode associée à l'opération
  - le message peut être synchrone ou asynchrone, avec ou sans réponse
- Autres termes courants : client, serveur

Encapsulation

- Deux sens
  - Le fait de grouper l'état et le comportement d'un objet dans une même entité
  - Le fait de masquer la façon dont un objet mémorise son état (attributs et liens) et rend des services (méthodes) ; ses clients ne doivent pas en dépendre
    - Protection
- L'abstraction (masquer les détails non significatifs) impose l'encapsulation

Classe

- Beaucoup d'objets ont le même comportement et la même façon de représenter leur état
- La classe est un mécanisme permettant la définition d'une structure (liste des attributs) et d'un comportement (définition des méthodes) utilisés par les objets
- Un objet est dit instance de sa classe
- La classe est aussi chargée de la création (instanciation) d'objets
- Vision constructive ("modèle ou moule d'objets") ou ensembliste ("ensemble d'objets")

Méta-classe

- La classe est-elle un objet, est-elle instance d'une méta-classe ?
- Et la méta-classe (instance d'une méta-méta-classe) ?
- Les langages qui respectent cette logique (Smalltalk par exemple) doivent résoudre cette difficulté
- D'autres langages (Java par exemple) proposent des objets représentant les classes, ce qui permet l'introspection
- Il est aussi possible de définir des propriétés « de classe » :
  - valeur de l'attribut unique pour toutes les instances
  - il n'y a pas d'objet récepteur dans la méthode

Héritage

- Certaines classes peuvent avoir des caractéristiques communes
  - attribut identique (nom et type)
  - opération (méthode identique ou pas)
- Il est possible de décrire ces caractéristiques communes dans une classe, puis de définir de nouvelles classes qui réutilisent les caractéristiques définies par cette première classe
  - une sous-classe hérite d'une super-classe
  - une sous-classe peut ajouter des attributs, des méthodes et redéfinir des méthodes de la super-classe
- Héritage multiple : une classe peut hériter de plusieurs classes
  Complexité : conflits sur propriétés, héritage répété. . .

Polymorphisme

- Le mot polymorphisme est formé à partir du grec ancien πολλοί (polloí) qui signifie « plusieurs » et μορφος (morphos) qui signifie « forme » (Wikipedia)
- En informatique, le polymorphisme est la possibilité d'utiliser le même code avec des données de types différents
- Un même message peut être envoyé à 2 objets différents (instances de classes différentes) : c'est le polymorphisme sur l'objet récepteur (nécessite souvent une super-classe commune)
- Définition simple : c'est la possibilité d'avoir 2 méthodes différentes associées à une même opération
- Autre terme : liaison dynamique

Objet : synthèse

- Une identité (unique, non modifiable)
- Un état (modifiable lors de l'exécution des méthodes)
- Une offre de services (non modifiable, mais la réponse dépend de l'état)
- Un type (une classe) (non modifiable)

Classes et objets en C++

- Extension du concept d'agrégat (`struct`)
- Encapsulation / protection
- Nouveau mot-clef : `class`
- Données membres, fonctions membres, autres membres
- Allocation statique, automatique ou dynamique
- Les classes n'existent que dans les fichiers sources

Une classe en C++

```
class  Pile {      // struct Pile {
public:
  void      empile( double v );
  double    depile();
  double & sommet();
  bool      est_vide();

private:
  unsigned int taille_{ 10 };
  double *     tab_   { new double[taille_] };
  unsigned int index_ { 0 };
};
```

En général dans le fichier d'entête

Définition d'une fonction membre

```
void Pile::empile( double v )
{

  // objet récepteur : this
  // Pile * const this;

  if( this->index_ < taille_ ) {
    tab_[index_++] = v;
  }
  else {
    throw std::out_of_range{ "pile pleine" };
  }
}
```

Dans le fichier d'implémentation

Fonctions « inline » (C99 et C++)

- Mot clef `inline`

```
inline bool Pile::est_vide()
{
  return index_ == 0;
}
```

En général dans le fichier d'entête

- Ou définition à la place de la déclaration

```
class Pile {
  //...
  bool est_vide() {
    return index_ == 0;
  }
  //...
};
```

Bonne pratique

- Les attributs sont privés (ou protégés : voir plus loin). Les accesseurs inline permettent de ne pas perdre en performance.

Démo

GitLab

cours/objet/Pile_v1.cpp

Constructeur et destructeur

- Constructeur

```
Pile::Pile( unsigned int taille )
{
  taille_ = taille;
  tab_    = new double[taille_];
  index_  = 0;
}

Pile p1{ 10 };
Pile * p2 = new Pile{ 20 };
```

- Destructeur (~ : opérateur de complément à 1)

```
Pile::~Pile()
{
  delete[] tab_;
}
```

Destruction d'un tableau

Resource Acquisition Is Initialisation

- Le constructeur acquiert les ressources nécessaires à l'objet, le destructeur les libère

```
{
  std::ifstream data{ "input.txt" };
  // ...
}

{
  std::lock_guard lock{ mutex };
  // ...
}
```

Démo

GitLab

cours/objet/Pile_v2.cpp

Bonnes pratiques

- Un constructeur doit initialiser l'état de l'objet de façon à ce qu'il soit prêt à rendre les services qu'il offre
- Le destructeur doit libérer toutes les ressources acquises par l'objet au cours de sa vie (mais ce n'est pas un suicide !)

~~`delete this;`~~

Constructeur de copie

```
// En C :
// struct Personne p1 = { 'F', 20 };
// struct Personne p2 = p1;

Pile p1{ 10 };
Pile p2{ p1 };

Pile::Pile( const Pile & p )
{
  taille_ = p.taille_;
  index_  = p.index_;
  tab_    = new double[taille_];
  for( unsigned i{ 0 }; i < p.index_; ++i ) {
    tab_[i] = p.tab_[i];
  }
}
```

Démo

GitLab

cours/objet/Pile_v3.cpp

Affectation par copie

```
// En C :
// struct Personne p1, p2; p1 = p2;

Pile p1{ 10 }, p2{ 20 }; p1 = p2;

Pile & Pile::operator=( const Pile & p )
{
  if( this != &p ) {
    delete[] tab_;
    taille_ = p.taille_;
    index_  = p.index_;
    tab_    = new double[taille_];
    for( unsigned i{ 0 }; i < index_; ++i ) {
      tab_[i] = p.tab_[i];
    }
  }
  return *this;
}
```

Risque d'échec

Affectation par copie : meilleure version

```
Pile & Pile::operator=( const Pile & p )
{
  if( this != &p ) {
    Pile tmp{ p };
    swap( tmp );
  }
  return *this;
}

void Pile::swap( Pile & p )
{
  using std::swap;
  swap( taille_, p.taille_ );
  swap( tab_,    p.tab_    );
  swap( index_,  p.index_  );
}
```

Affectation par copie : autres versions

```
Pile & Pile::operator=( const Pile & p )
{
  Pile tmp{ p };
  swap( tmp );
  return *this;
}

Pile & Pile::operator=( Pile p )
{
  swap( tmp );
  return *this;
}
```

Démo

GitLab

cours/objet/Pile_v4.cpp

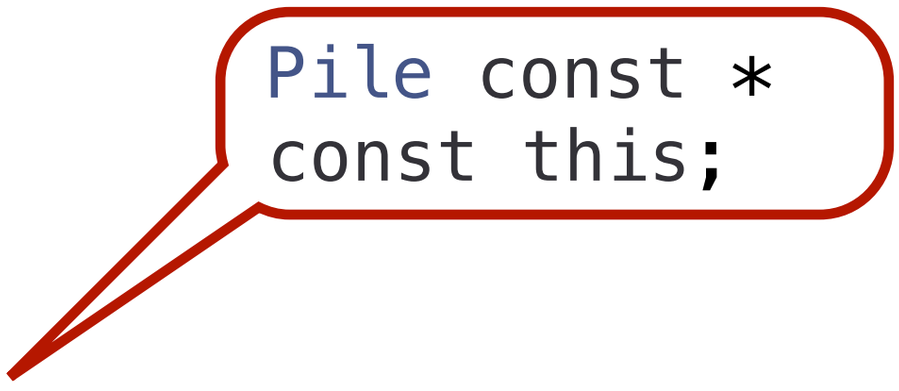
Bonnes pratiques

- Respecter le protocole pour l'affectation par copie
- Utiliser l'idiome du swap pour l'affectation par copie

Membres constants

```
class Pile {
public:
  // ...
  double   depile();
  double & sommet();
  bool     est_vide()      ;

private:
  unsigned int        taille_;
  double *            tab_;
  unsigned int        index_;
};
```



Initialisation par le constructeur

```
class Pile {
public:
  Pile( unsigned int taille )
    : taille_{ taille }
    , tab_   { new double[taille_] }
    , index_ { 0 }
  {}
  // ...

private:
  unsigned int const taille_;
  double *     const tab_;
  unsigned int       index_;
};
```

Bonne pratique

- Pour les attributs, utiliser la syntaxe d'initialisation dans le constructeur

Constructeur par défaut

```
Pile tab[10];

class Pile {
public:
   Pile( unsigned int taille = 100 );
   // ...
};

/* ATTENTION : il y a un intrus */

Pile p1( 10 );
Pile p2{ 10 };
Pile p3();
Pile p4{};
Pile p5;
```

Démo

GitLab

cours/objet/Pile_v5.cpp

Bonne pratique

- Proposer un constructeur par défaut uniquement quand cela a du sens

Classe concrète (forme canonique)

```
class Pile {
public:
  // Constructeur par défaut
  Pile( unsigned int taille = 100 );

  // Constructeur de copie
  Pile( const Pile & p );

  // Affectation par copie
  Pile & operator=( const Pile & p );

  // Destructeur
  ~Pile();

 // ...
};
```

default, delete

```
class X {
  X               ( const X & p ) = delete;
  X & operator=( const X & p ) = delete;
  // ...
};

class Y {
  Y               ( const Y & p ) = default;
  Y & operator=( const Y & p ) = default;
  // ...
};
```

Bonne pratique

- Rendre explicite le choix pour les 3 ou 4 opérations utilitaires d'une classe concrète (version spécifique, `default`, `delete`)

Constructeur comme opérateur de conversion

```
// mot-clef explicit : le constructeur n'est
// pas un opérateur de conversion implicite

class Nombre {
public:
  explicit Nombre( int n );
  // ...
};

extern void foo( Nombre );

void foo() {
  foo( 3 ); // foo( Nombre( 3 )); reste possible
}
```

Bonne pratique

- Utiliser `explicit` sauf si la conversion implicite via le constructeur a du sens

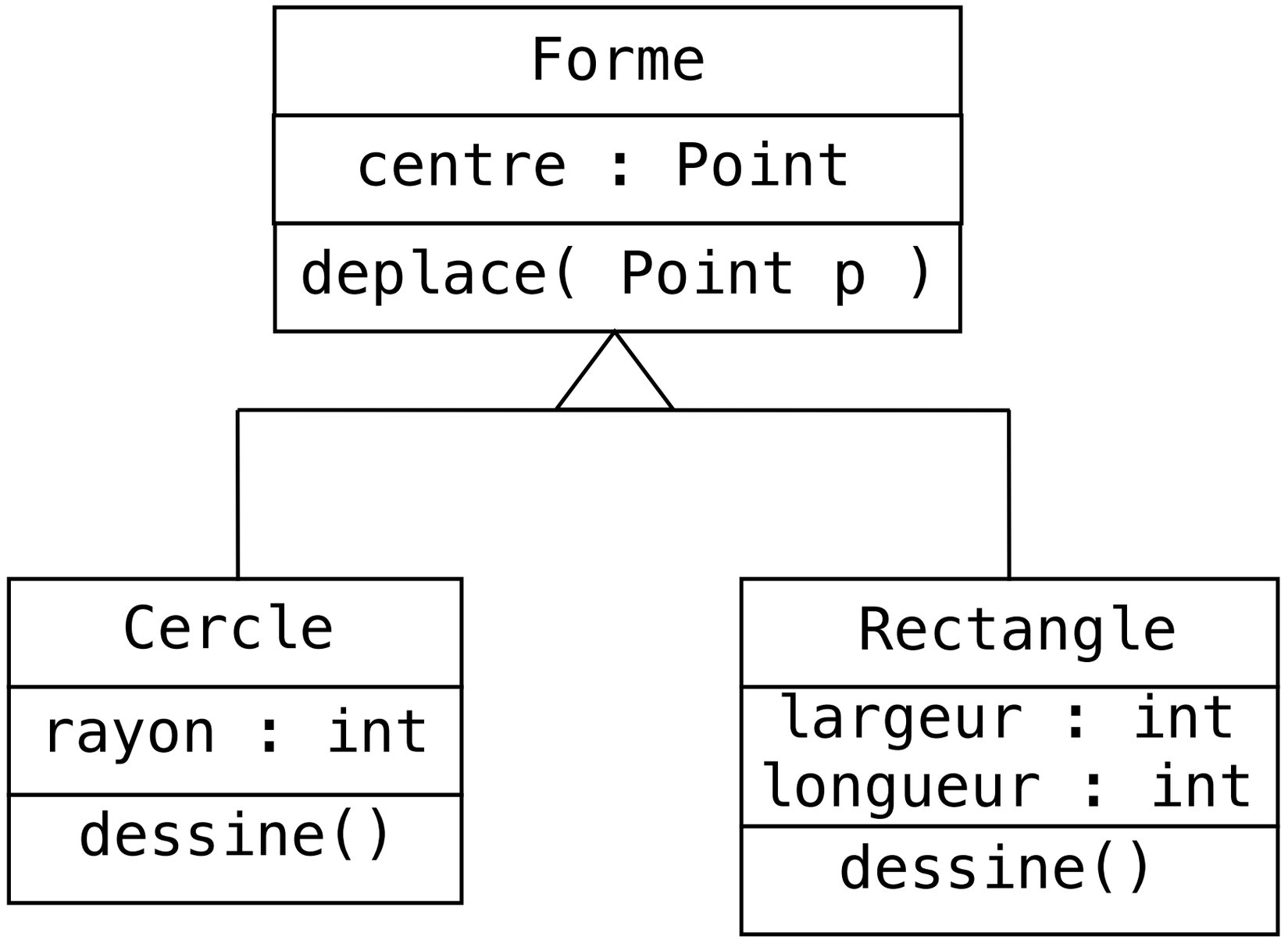
Méthodes et attributs de classe

```
class A {
public:
    A() { ++nb_instances_; }
   ~A() { --nb_instances_; }
   static int nb_instances() {
      return nb_instances_;
   }

private:
   inline static int nb_instances_{ 0 };
};
```

Quelle est l'erreur ?

L'héritage en C++



Classe de base

```
// Classe de base des objets graphiques
class Forme {
public:
  // Une forme a un centre
  explicit Forme( Point centre );
  Forme( const Forme & f );

  // Une forme sait se déplacer
  void deplace( Point nouveau_centre );

protected:  // Accessible aux sous-classes
  Point centre_;
};
```

Pas de racine de l'arbre d'héritage

Classe dérivée

```
// Un Cercle est une Forme
class Cercle : public Forme {
public:
   // On peut construire un cercle avec
   // un centre et un rayon
   Cercle( Point centre, int rayon );
   Cercle( const Cercle & c );

   // Un cercle sait se dessiner
   void dessine();

private:
   int rayon_;
};
```

Manipulation des instances

```
Point p1, p2;
Forme f{ p1 }, * pf{ &f };
Cercle c{ p2, 3 }, * pc{ &c };
f = c;
// c = f;
pf = pc;
// Attention
*pf = *pc;
// pc = pf;
pc = static_cast<  Cercle * >( pf );
Forme & rf{ c };
```

Bonne pratique

- Utiliser l'héritage public

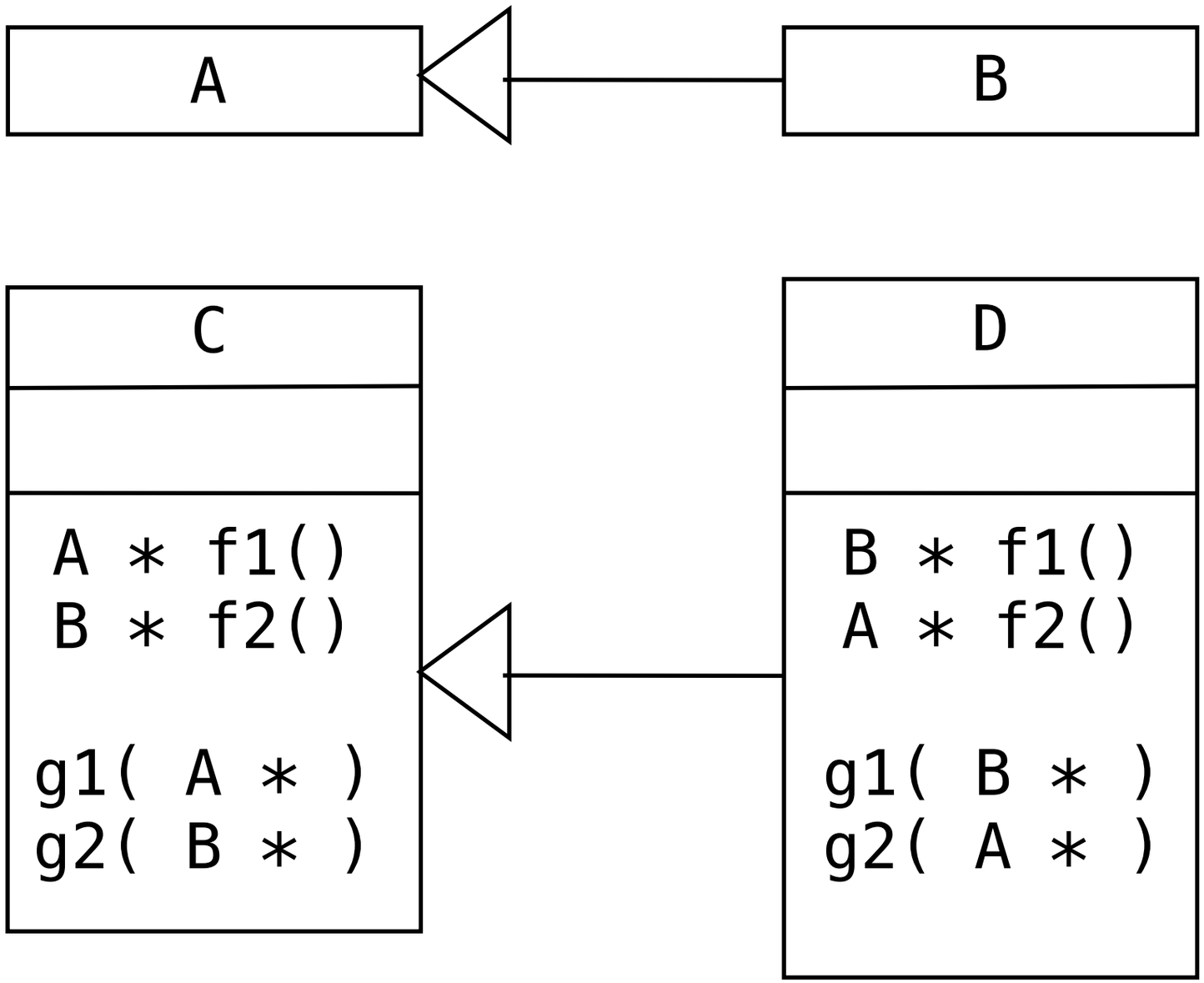
Redéfinition d'une méthode

```
class Cercle : public Forme {
public:
  // ...

  void deplace( Point nouveau_centre ) {
    // ...
    Forme::deplace( nouveau_centre );
    // ...
  }

  // ...
};
```

Covariance et contravariance



Appel d'un constructeur de la super-classe

```
class Cercle : public Forme {
public:
  // On peut construire un cercle avec
  // un centre et un rayon
  Cercle( Point centre, int rayon )
    : Forme{ centre }, rayon_{ rayon }
  {}

  // Constructeur de copie
  Cercle( const Cercle & c )
    : Forme{ c }, rayon_{ c.rayon_ }
  {}

  ~Cercle();
  // ...
};
```

Quelle différence ?

```
class A {
public:
  A()                 { /* .. */ }
  A( const A & ) { /* .. */ }
};
```

```
class B1 : public A {
public:
  B1( const B1 & ) { /* .. */ }
};

class B2 : public A {
public:
  B2( const B2 & ) = default;
};
```

Bonne pratique

- Le constructeur de copie de la sous-classe appelle celui de la super-classe

Démo

GitLab

cours/objet/Forme_v1.cpp

Dessiner une Forme

```
// pf->dessine();

class Forme {
public:
  explicit Forme( Point centre );
  Forme( const Forme & f );

  void deplace( Point nouveau_centre );
  void dessine();

protected:
  Point centre_;
};
```

Liaison dynamique

```
class Forme {
public:
  explicit Forme( Point centre );
  Forme( const Forme & f );
  virtual ~Forme();

  void deplace( Point nouveau_centre );
  virtual void dessine();

protected:
  Point centre_;
};
```

Destructeur virtuel !

Bonnes pratiques

- Si votre classe est destinée à être utilisée comme classe de base, son destructeur doit être virtuel.
- N'utiliser une classe comme classe de base que si son destructeur est virtuel.

Redéfinition dans la classe dérivée

```
class Cercle : public Forme {
public:
   // ...

   // Un cercle sait se dessiner
            void dessine() override;

private:
   int rayon_;
};
```

Classe abstraite

```
class Forme {
public:
   explicit Forme( Point centre );
   Forme( const Forme & f );
   virtual ~Forme();

   void deplace( Point nouveau_centre );
   virtual void dessine() = 0;

protected:
   Point centre_;
};
```

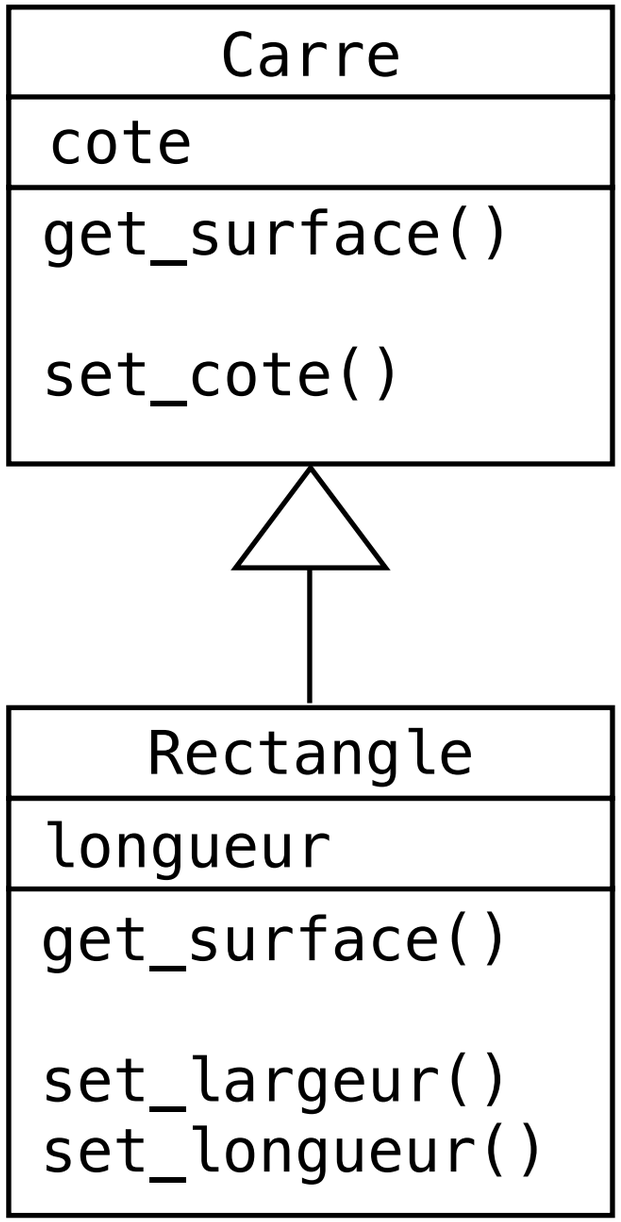
Principe de substitution de Liskov

- Barbara Liskov (ACM SIGPLAN Notices - mai 1988)
  - “If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.”

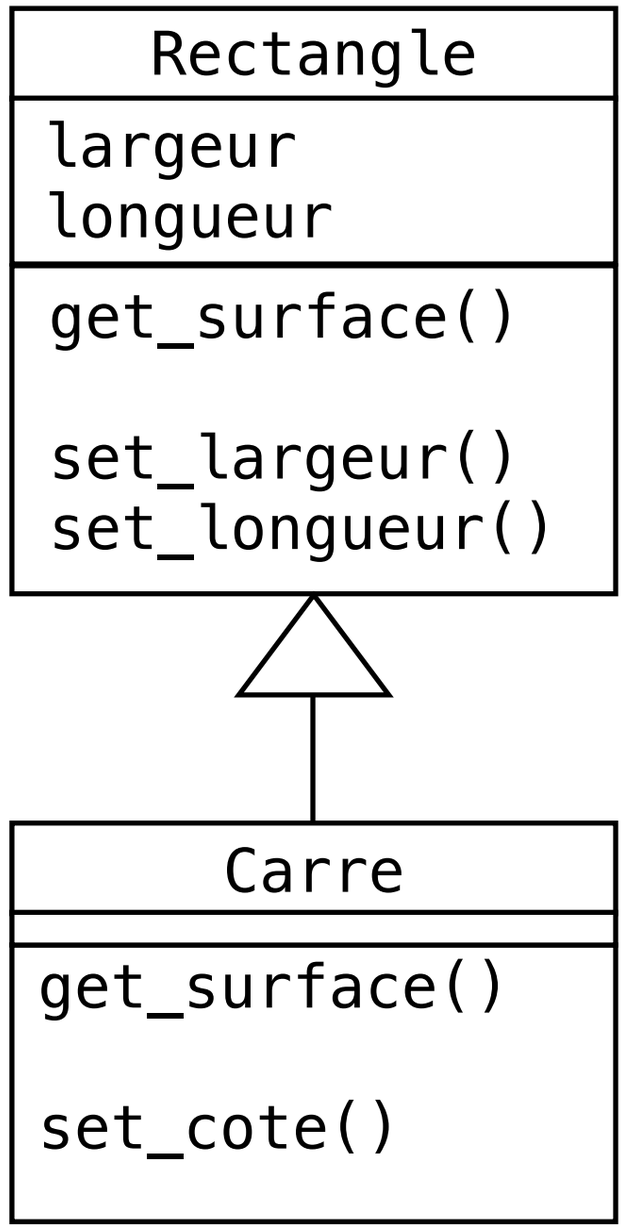
- Robert C. Martin (C++ Report - mars 1996)
  - Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Sémantique de l'héritage

{ côté × 2 ⇒ surface × 4 }



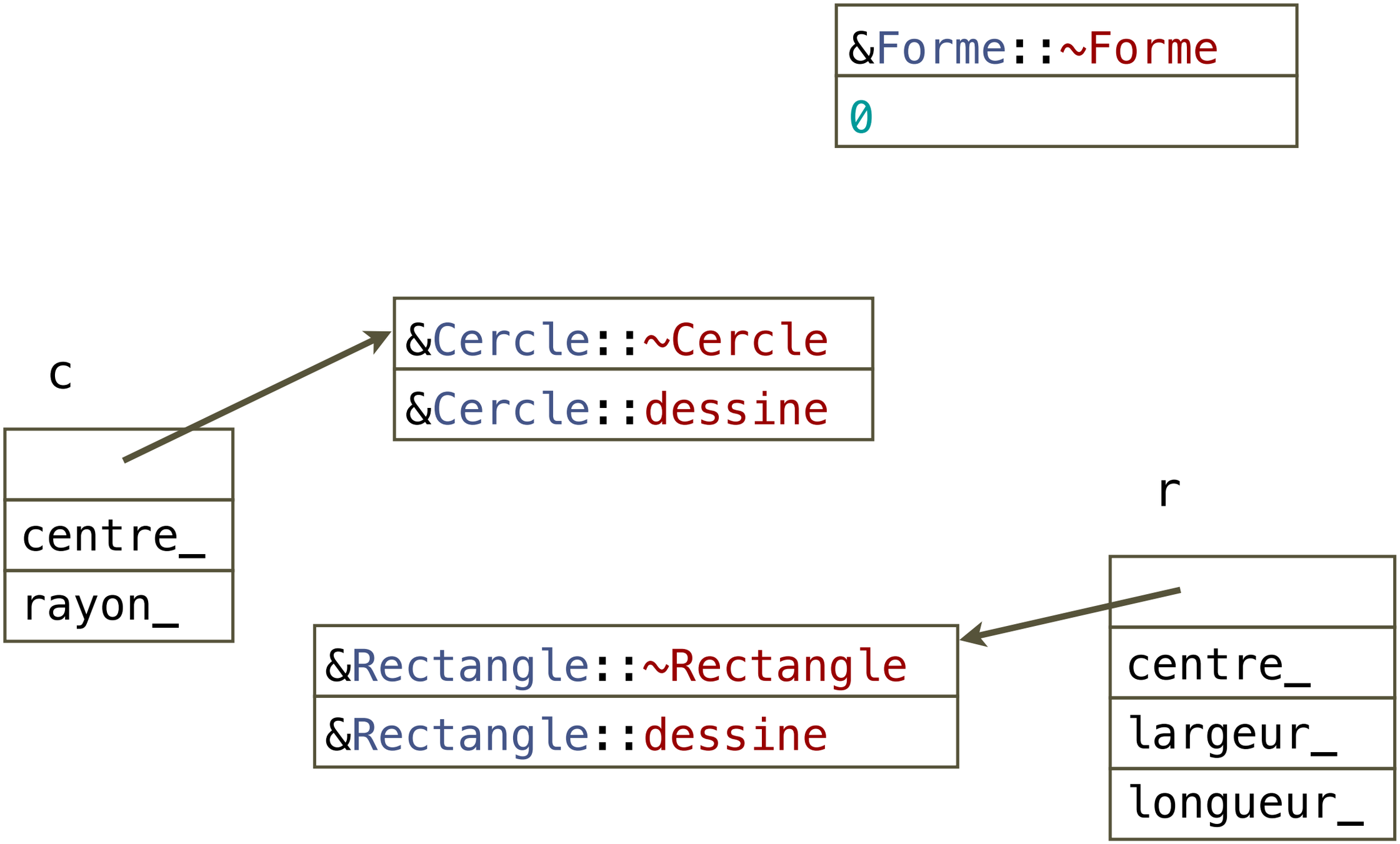
{ largeur × 2 ⇒ surface × 2 }



Bonnes pratiques

- L'héritage par restriction n'est pas une bonne idée.
- L'héritage de structure n'est pas suffisant comme justification de l'héritage.
- Une super-classe sert à factoriser des comportements.
- Les classes intermédiaires (qui ne sont pas des feuilles dans le graphe d'héritage) doivent être si possible abstraites.

Liaison dynamique en C++ : principe de réalisation



Bonne pratique

- Le polymorphisme se met en œuvre via des fonctions virtuelles (et pas via un attribut qui mémorise le type de l'objet).

Démo

GitLab

cours/objet/Forme_v2.cpp

Identification dynamique de type en C++ (1)

```
void foo( Forme * f ) {
  // Sans vérification
  Cercle * c1{ static_cast< Cercle * >( f )};

  // Avec vérification (il faut au moins
  // une fonction virtuelle)
  Cercle * c2{ dynamic_cast< Cercle * >( f )};
  if( c2 != nullptr ) {
    // ...
  }
  // Écriture courante :
  if( c2 ) {
    // ...
  }
}
```

Identification dynamique de type en C++ (2)

```
void foo( Forme & f ) {
  try {
    Cercle & c{ dynamic_cast< Cercle & >( f )};
    // ...
  }
  catch( std::bad_cast e ) {
    // ...
  }
}
```