

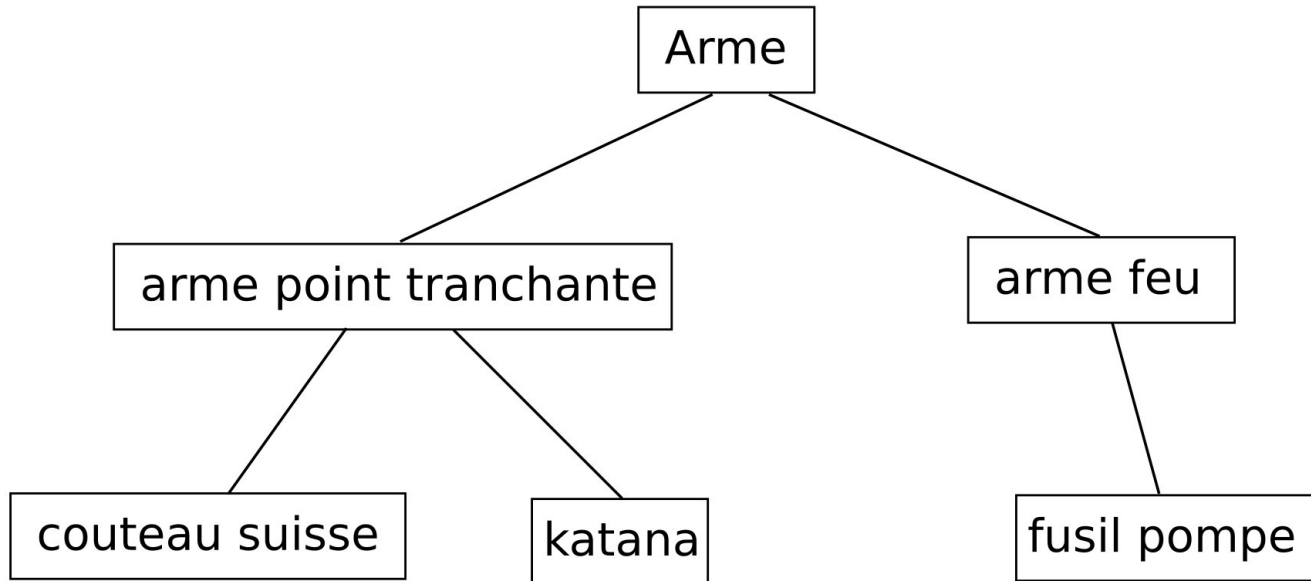
C++

Programmation Objet:
Héritage, polymorphisme

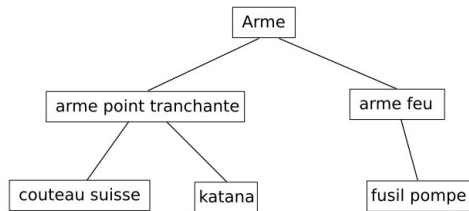
virtual void function() = 0;

`dynamic_cast<T>();`

Héritage

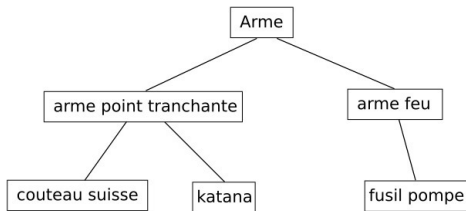


Héritage



```
class arme
{
public:
    void set_nom(std::string const& n) {nom=n;}
    std::string const& get_nom() const {return nom;}
private:
    std::string nom;
};
```

Héritage



```
class arme
{
public:
    void set_nom(std::string const& n) {nom=n;}
    std::string const& get_nom() const {return nom;}
private:
    std::string nom;
};
```

```
class arme_point_tranchante : public arme
{
public:
private:
    float finesse_lame;
    float longueur_lame;
};

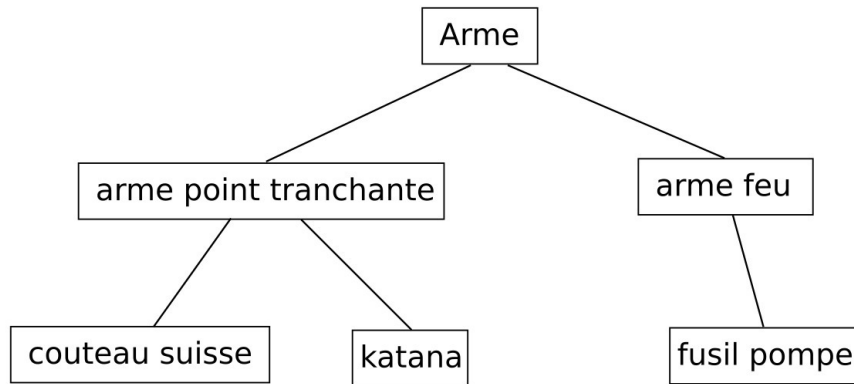
class katana : public arme_point_tranchante
{
public:
private:
    int age;
};

class couteau_suisse : public arme_point_tranchante
{
public:
private:
    int nombre_ustensil;
};
```

```
class arme_feu : public arme
{
public:
    void tire() {munition--;}
private:
    int munition;
    int calibre;
};

class fusil_pompe : public arme_feu
{
public:
private:
    int nombre_coup_minute;
};
```

Intérêt Héritage

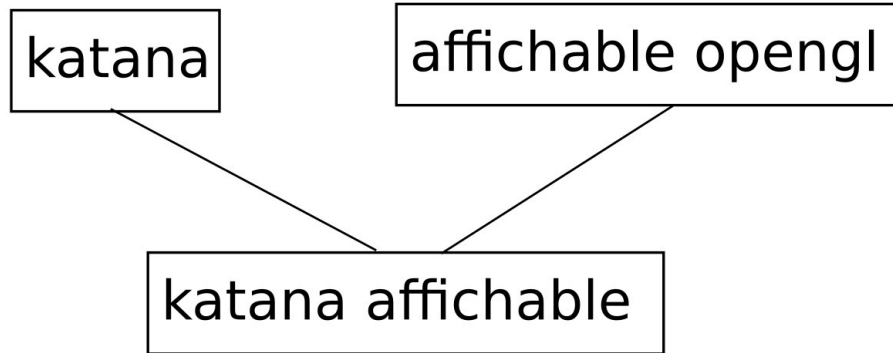


- Evite la réécriture de code

Toutes armes peuvent gérer un nom
Toutes les armes à feu peuvent tirer
...

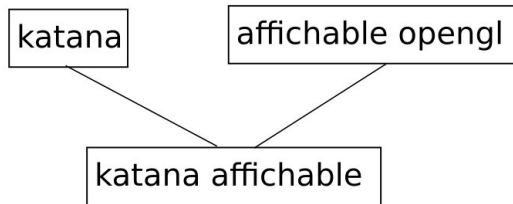
- Structure les objets entre eux

Héritage multiple



Héritage multiple non limité en C++

Héritage multiple



```
class katana : public arme_point_tranchante
{
public:
private:
    int age;
};
```

```
class affichable_opengl
{
    GLuint vbo;
    GLuint vboi;
    int nbr_triangle;

public:
    void affiche() const
    {
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glVertexPointer(3, GL_FLOAT, 0, 0);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboi);
        glDrawElements(GL_TRIANGLES, 3*nbr_triangle, GL_UNSIGNED_INT, 0);
    }
};
```

```
class katana_affichable : public katana, affichable_opengl
{
};
```

Limite de l'héritage

- Ne pas abuser de l'héritage

Bonnes pratiques

- L'héritage doit toujours répondre à la question

<classe fille> est un type de <classe parente>

Sinon ce n'est pas une relation d'héritage

↳ aggrégation

(A contient B)

Limite de l'héritage

Exemple d'heritage incorrect

```
class guerrier : public katana
{
public:
    float points_attaque() const
    {
        return 2.0*longueur_lame;
    }
};
```

A ne pas faire

Un guerrier n'est pas un type de katana
(même si le code semble convenir)

Ici: guerrier.nom retourne le nom
du katana

Version correcte

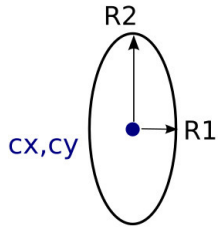
```
class guerrier
{
    katana arme; Relation d'aggrégation

public:
    float points_attaque() const
    {
        return 2.0f*arme.longueur_lame;
    }
};
```

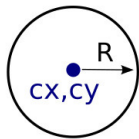
un guerrier **possède** un katana

Limite de l'héritage

Exemple relation cercle/ellipse



```
struct ellipse
{
    ellipse(float R1_arg, float R2_arg, float cx_arg, float cy_arg)
        : R1(R1_arg), R2(R2_arg), cx(cx_arg), cy(cy_arg)
    {}
    float R1;
    float R2;
    float cx, cy;
};
```



```
struct cercle
{
    cercle(float R_arg, float cx_arg, float cy_arg)
        : R(R_arg), cx(cx_arg), cy(cy_arg)
    {}

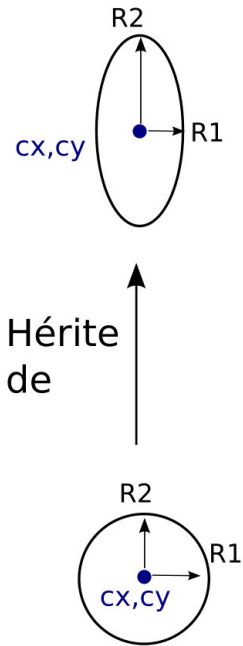
    float R;
    float cx, cy;
};
```

2 Classes indépendantes

Beaucoup de choses communes => réécriture de code

Limite de l'héritage

Exemple relation cercle/ellipse



```
struct ellipse
{
    ellipse(float R1_arg, float R2_arg, float cx_arg, float cy_arg)
        : R1(R1_arg), R2(R2_arg), cx(cx_arg), cy(cy_arg)
    {}
    float R1;
    float R2;
    float cx, cy;
};
```

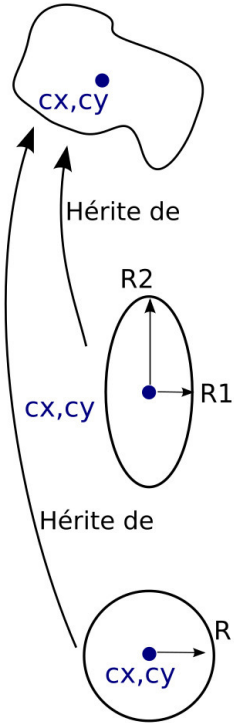
```
struct cercle : public ellipse
{
    cercle(float R_arg, float cx_arg, float cy_arg)
        : ellipse(R_arg, R_arg, cx_arg, cy_arg) {}
    //R1==R2 constamment => R2 est inutile
};
```

Danger d'intégrité
cercle avec $R1 \neq R2$ possible

=> La structure d'un cercle n'est pas forcément la même structure qu'une ellipse

Limite de l'héritage

Exemple relation cercle/ellipse



```
struct courbe_fermee
{
    courbe_fermee(float cx_arg, float cy_arg)
        : cx(cx_arg), cy(cy_arg) {}
    float cx, cy;
};
```

```
struct ellipse : public courbe_fermee
{
    ellipse(float R1_arg, float R2_arg, float cx_arg, float cy_arg)
        : courbe_fermee(cx_arg, cy_arg), R1(R1_arg), R2(R2_arg) {}
    float R1, R2;
};
```

```
struct cercle : public courbe_fermee
{
    cercle(float R, float cx_arg, float cy_arg)
        : courbe_fermee(cx_arg, cy_arg), R(R_arg) {}
    float R;
};
```

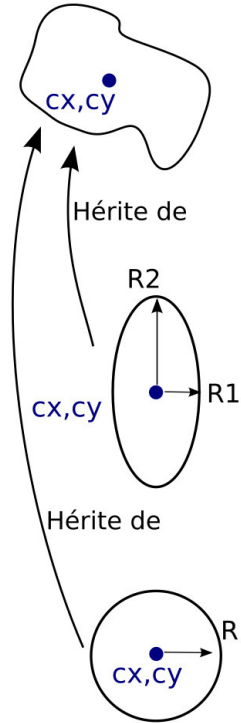
Cercle et Ellipse sont ici cousins

+ Concaténation du code pour le centre.

+ Respect de la structure interne de l'ellipse et du cercle

Limite de l'héritage

Exemple relation cercle/ellipse



Même si

- Cercle est un "type" d'ellipse
 - ↳ Pas de partage de la structure interne
 - ↳ Pas d'héritage direct (relation cousin)

Polymorphisme

Les classes parent et fille sont des types différents

Héritage simple = gain uniquement pour l'écriture du code
(pas de duplication)

On peut convertir une classe fille vers le type parent
(info classe fille perdues)

Conversion classes dérivées

```
struct arme_feu
{
    int munition;

    arme_feu(int munition_arg) :munition(munition_arg){}
    void tire() {munition--;}
};

struct fusil_pompe : public arme_feu
{
    fusil_pompe(int munition_arg) :arme_feu(munition_arg){}
    void tire() {munition-=2;}
};
```

fusil à pompe = arme à feu qui tire 2 coups à la fois

Conversion classes dérivées

```
int main()
{
    arme_feu a0(6);
    fusil_pompe a1(6);

    test_arme(a0);
    test_arme(a1);
}
```

```
struct arme_feu
{
    int munition;

    arme_feu(int munition_arg) :munition(munition_arg){}
    void tire() {munition--;}
};

struct fusil_pompe : public arme_feu
{
    fusil_pompe(int munition_arg) :arme_feu(munition_arg){}
    void tire() {munition-=2;}
};
```

*fusil_pompe
converti en arme_feu*

```
void test_arme(arme_feu& a)
{
    std::cout<<"J'ai "<<a.munition<<" munitions"<<std::endl;
    a.tire();
    a.tire();
    std::cout<<"Il me reste "<<a.munition<<" munitions"<<std::endl;
}
```

*J'ai 6 munitions
Il me reste 4 munitions
J'ai 6 munitions
Il me reste 4 munitions*

On ne tire qu'une munition
à la fois
Information fusil à pompe perdue

Conversion classes dérivées

```
struct arme_feu
{
    int munition;

    arme_feu(int munition_arg) :munition(munition_arg){}
    void tire() {munition--;}
};

struct fusil_pompe : public arme_feu
{
    fusil_pompe(int munition_arg) :arme_feu(munition_arg){}
    void tire() {munition-=2;}
};
```

```
int main()
{
    arme_feu a0(6);
    fusil_pompe a1(6);

    std::vector<arme_feu> v;
    v.push_back(a0);v.push_back(a1);

    for(int k=0;k<2;++k)
        v[k].tire();

    std::cout<<v[0].munition<<" , "<<v[1].munition<<std::endl;
}
```

converti en arme_feu →

une seule munition →

Mot clé virtual

```
struct arme_feu
{
    int munition;

    arme_feu(int munition_arg) :munition(munition_arg){}
    virtual void tire() {munition--;}
};
```

 *fonction polymorphe*

```
struct fusil_pompe : public arme_feu
{
    fusil_pompe(int munition_arg) :arme_feu(munition_arg){}
    void tire() {munition-=2;}
};
```

Mot clé virtual

```
struct arme_feu
{
    int munition;

    arme_feu(int munition_arg) :munition(munition_arg){}
    virtual void tire() {munition--;}
};

struct fusil_pompe : public arme_feu
{
    fusil_pompe(int munition_arg) :arme_feu(munition_arg){}
    void tire() {munition-=2;}
};
```

on peut appeler
la fonction de
fusil_pompe

```
int main()
{
    arme_feu a0(6);
    fusil_pompe a1(6);

    test_arme(a0);
    test_arme(a1);
}
```

```
void test_arme(arme_feu& a)
{
    std::cout<<"J'ai "<<a.munition<<" munitions"<<std::endl;
    a.tire();
    a.tire();
    std::cout<<"Il me reste "<<a.munition<<" munitions"<<std::endl;
}
```

← tire est une fonction polymorphe

*J'ai 6 munitions
Il me reste 4 munitions
J'ai 6 munitions
Il me reste **2** munitions*

On n'a pas perdu
l'information que
a1 était un fusil_pompe

Polymorphisme

Attention: Polymorphisme n'est préservé que pour des **références** et des **pointeurs**.

Information polymorphique perdue dans le cas d'une copie

```
void test_arme(arme_feu a)  copie de la classe
{
    std::cout<<"J'ai " <<a.munition<<" munitions"<<std::endl;
    a.tire();
    a.tire();
    std::cout<<"Il me reste " <<a.munition<<" munitions"<<std::endl;
}

int main()
{
    arme_feu a0(6);
    fusil_pompe a1(6);

    test_arme(a0);
    test_arme(a1);
}
```

Polymorphisme perdu

J'ai 6 munitions
Il me reste 4 munitions
J'ai 6 munitions
Il me reste 4 munitions

Polymorphisme

Pointeurs préservent le polymorphisme

```
void test_arme(arme_feu* a)
{
    std::cout<<"J'ai "<<a->munition<<" munitions"<<std::endl;
    a->tire();
    a->tire();
    std::cout<<"Il me reste "<<a->munition<<" munitions"<<std::endl;
}

int main()
{
    arme_feu a0(6);
    fusil_pompe a1(6);

    test_arme(&a0);
    test_arme(&a1);
}
```

J'ai 6 munitions
Il me reste 4 munitions
J'ai 6 munitions
Il me reste 2 munitions

Polymorphisme

Conteneur avec éléments de comportement variable

```
int main()
{
    arme_feu a0(6);
    fusil_pompe a1(6);

    std::vector<arme_feu*> v;
    v.push_back(&a0); v.push_back(&a1);

    for(int k=0; k<2; ++k)
        v[k]->tire();

    std::cout<<v[0]->munition<<" , "<<v[1]->munition<<std::endl;
}
```

*doit être un vecteur
d'adresse, sinon
copie de la classe*



5,4

Intérêt du polymorphisme

- Conteneur d'élément ayant des comportements différents

Nécessite un vecteur de pointeurs

(rare cas où les pointeurs sont indispensables en C++)

```
std::vector<Parent*>  
std::list<Parent const*>  
std::map<int, Parent*>
```

- Fonction générique
(comportement différent en fonction de la classe)

Nécessite un argument de type reference ou pointeur

```
void function(Parent const& x);  
void function(Parent& x);
```

Conditions de fonctionnement

```
class Parent{  
    virtual ...  
};  
class Fille : public Parent {};  
  
int main()  
{  
    Fille a;  
  
    Parent b=a; polymorphisme perdu  
  
    Parent& c=a;           polymorphe  
    Parent const & d=a;    polymorphe  
    Parent* e=&a;          polymorphe  
    Parent const * f=&a;   polymorphe  
}
```

Polymorphisme, limitation

Polymorphisme = modification dynamique pendant l'exécution (changement de type non prévue à la compilation)

- Inconvénient: Temps de calcul

Une fonction polymorphe est plus lente qu'une fonction non polymorphe + optimisation difficile

- Ne pas utiliser le polymorphisme pour exécuter des tâches critiques en temps de calcul.
- Ne pas s'interdire le polymorphisme pour des tâches non critiques !

Polymorphisme vs template

Polymorphisme et template permettent la généricité

Polymorphisme	Template
+ Dynamique (modification en run-time) +/- Lié à l'héritage + Structure de le code (règle d'appels) - Force relations d'héritage (limite l'utilisation de classes externes) - Ralenti l'exécution Fonctions génériques Conteneurs adaptatifs (adaptation dynamique)	+ Statique (réalisé à la compilation) + Ne ralenti pas l'exécution +/- S'applique à toute classes + Vraie généricité - Pas de structuration - Ralenti la compilation Fonctions génériques Classes génériques (paramétrable à la création)

Le dynamic_cast

Permet de (re)convertir une classe Parent en classe Fille

```
struct arme_feu
{
    int munition;

    arme_feu(int munition_arg) :munition(munition_arg){}
    virtual void tire() {munition--;}
};

struct fusil_pompe : public arme_feu
{
    int grenade;

    fusil_pompe(int munition_arg) :arme_feu(munition_arg),grenade(3){}
    void tire() {munition-=2;}
    void lance_grenade() {grenade--;}
};
```

n'existe que dans classe
fusil_pompe

Le dynamic_cast

Permet de (re)convertir une classe Parent en classe Fille

```
struct arme_feu
{
    int munition;

    arme_feu(int munition_arg) :munition(munition_arg){}
    virtual void tire() {munition--;}
};

struct fusil_pompe : public arme_feu
{
    int grenade;

    fusil_pompe(int munition_arg) :arme_feu(munition_arg),grenade(3){}
    void tire() {munition--;}
    void lance_grenade() {grenade--;}
};
```

Convertis arme* en fusil_pompe*
si la classe d'origine est un
fusil_pompe.

Renvoi nullptr sinon.

```
void lanceur_grenade(arme_feu* arme)
{
    fusil_pompe* fusil=dynamic_cast<fusil_pompe*>(arme);
    if(fusil!=nullptr)           pointeur null en C++
    {
        fusil->lance_grenade();
        std::cout<<"Je lance une grenade"<<std::endl;
    }
    else
    {
        std::cout<<"Je ne suis pas un fusil a pompe"<<std::endl;
    }
}

int main()
{
    arme_feu a0(6);
    fusil_pompe a1(6);

    lanceur_grenade(&a0);
    lanceur_grenade(&a1);
}
```

*Je ne suis pas un fusil a pompe
Je lance une grenade*

Le dynamic_cast

Fonctionne également avec des références

```
struct arme_feu
{
    int munition;

    arme_feu(int munition_arg) :munition(munition_arg){}
    virtual void tire() {munition--;}
};

struct fusil_pompe : public arme_feu
{
    int grenade;

    fusil_pompe(int munition_arg) :arme_feu(munition_arg),grenade(3){}
    void tire() {munition--;}
    void lance_grenade() {grenade--;}
};
```

Utilisation des
exceptions

```
void lanceur_grenade(arme_feu& arme)
{
    try
    {
        fusil_pompe& fusil=dynamic_cast<fusil_pompe&>(arme);
        fusil.lance_grenade();
        std::cout<<"Je lance une grenade"<<std::endl;
    }
    catch(const std::bad_cast& e)
    {
        std::cout << "Je ne suis pas un fusil a pompe"<<std::endl;
    }
}

int main()
{
    arme_feu a0(6);
    fusil_pompe a1(6);

    lanceur_grenade(a0);
    lanceur_grenade(a1);
}
```

*Je ne suis pas un fusil a pompe
Je lance une grenade*

Limites du dynamic_cast

Bonne programmation:

- Ne pas abuser du dynamic_cast

└─> Révèle une programmation inappropriée

```
x0*=dynamic_cast<X0*>(x);
```

```
if(x0!=nullptr){...}
```

```
x1*=dynamic_cast<X1*>(x);
```

```
if(x1!=nullptr){...}
```

<=>
C

```
switch(type)
```

```
{
```

```
case 1:
```

```
...
```

```
case 2:
```

```
...
```

```
};
```

Préférez: polymorphisme + généricité

VTable

- Le polymorphisme nécessite une table de correspondance
pointeur <-> type d'origine

└─> **vtable** (virtual table)

- Erreurs compilation liés vtable
= Erreurs liés à l'utilisation de virtual
- Rem. Pour utiliser le `dynamic_cast`, il faut au moins
une fonction virtuelle

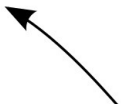
Classe virtuelle pure

Il est courant de ne pas pouvoir compléter les fonctions des classes parents

```
class courbe_fermee
{
public:
    virtual float aire() const {return 0.0f;} //??? pourquoi
};

class cercle : public courbe_fermee
{
    float R;
public:
    cercle(float R_arg):R(R_arg){}
    float aire() const {return M_PI*R*R;}
};

class carre : public courbe_fermee
{
    float L;
public:
    carre(float L_arg):L(L_arg){}
    float aire() const {return L*L;}
};
```



*courbe_fermee
est une "vue d'esprit"
on ne va jamais l'utiliser
en tant que telle.*

Classe virtuelle pure

```
class courbe_fermee
{
public:
    virtual float aire() const=0;
};
```

On ne complète pas aire()
Sera uniquement appelée à
partir des fils

Existence temporaire
sous forme de pointeur

```
int main()
{
    cercle p0(2.0f);
    carre p1(1.5f);

    std::list<courbe_fermee*> elements={&p0,&p1};
    for(const courbe_fermee* p : elements)
        std::cout<<p->aire()<<std::endl;
}
```

- Obligation de compléter
aire() dans la hiérarchie
pour instancier un fils

Classe virtuelle pure

Il est courant de ne pas pouvoir compléter les fonctions des classes parents

```
class courbe_fermee
{
public:
    virtual float aire() const {return 0.0f;} //??? pourquoi
};

class cercle : public courbe_fermee
{
    float R;
public:
    cercle(float R_arg):R(R_arg){}
    float aire() const {return M_PI*R*R;}
};

class carre : public courbe_fermee
{
    float L;
public:
    carre(float L_arg):L(L_arg){}
    float aire() const {return L*L;}
};
```

Existence temporaire
sous forme de pointeur

```
int main()
{
    cercle p0(2.0f);
    carre p1(1.5f);

    std::list<courbe_fermee*> elements={&p0,&p1};
    for(const courbe_fermee* p : elements)
        std::cout<<p->aire()<<std::endl;
}
```

Classe virtuelle pure

- Il est impossible d'instancier une classe virtuelle pure
 - ↳ Existence uniquement sous forme de pointeurs/références

```
cercle c(2.0f);
```

```
courbe_fermee c0; erreur compilation
```

```
courbe_fermee* c1=&c; OK
```

```
courbe_fermee const* c2=&c; OK
```

```
courbe_fermee& c3=c; OK
```

```
courbe_fermee const& c4=c; OK
```

Polymorphisme et destructeurs

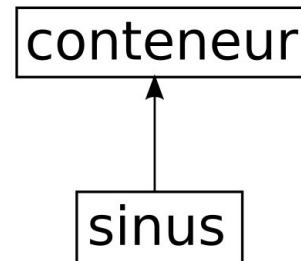
- Si la classe parent contient de la mémoire dynamique:

Il faut appeler le destructeur de la classe parent
lors de la destruction de la classe fille

↳ classe virtuelle => destructeur virtuel

```
int main()
{
    sinus* s=new sinus;

    conteneur *c=s;
    delete c;
}
```



suppression doit prendre
en compte les spécificités de sinus

Polymorphisme et destructeurs

```
class conteneur
{
protected:
    float *data;
public:

    conteneur():data(nullptr){}
    ~conteneur()
    {
        std::cout<<"destructeur conteneur"<<std::endl;
    }

    virtual void genere(int N) {}
};
```

```
class sinus : public conteneur
{
public:
    sinus():conteneur(){}
    ~sinus()
    {
        if(data!=nullptr)
        {
            delete []data;
            data=nullptr;
        }
        std::cout<<"destructeur sinus"<<std::endl;
    }
    void genere(int N)
    {
        data=new float[N];
        for(int k=0;k<N;++k)
            data[k]=std::sin(k/N);
    }
};
```

```
int main()
{
    sinus* s=new sinus;

    conteneur *c=s;
    delete c;
}
```

destructeur conteneur

↳ Fuite mémoire
ne détruit pas les
données de sinus

Polymorphisme et destructeurs

```
class conteneur
{
protected:
    float *data;
public:

    conteneur():data(nullptr){}
    virtual ~conteneur()
    {
        std::cout<<"destructeur conteneur"<<std::endl;
    }

    virtual void genere(int N) {}
};
```

```
int main()
{
    sinus* s=new sinus;

    conteneur *c=s;
    delete c;
}
```

destructeur sinus
destructeur conteneur

```
class sinus : public conteneur
{
public:
    sinus():conteneur(){}
    ~sinus()
    {
        if(data!=nullptr)
        {
            delete []data;
            data=nullptr;
        }
        std::cout<<"destructeur sinus"<<std::endl;
    }
    void genere(int N)
    {
        data=new float[N];
        for(int k=0;k<N;++k)
            data[k]=std::sin(k/N);
    }
};
```

→ Toute la mémoire
est bien libérée