La couche objet du C++

- Déclaration de classe
- Pointeurs sur les membres d'une classe
- Surcharge des opérateurs
- Allocation dynamique II
- Héritage
- Polymorphisme
- Typage dynamique et opérateurs de transtypage
- Exceptions II

C++ : Déclaration de classe

Une classe est introduite soit par `class` soit par `struct`.
Elle contient des déclarations de composants appartenant à une des catégories suivantes:

- déclarations de type;
- déclarations de constructeurs/destructeurs.
- déclarations d'attributs;
- déclarations de méthodes;
- déclarations d'amitié

C++ : Déclaration de classe

```
struct paire{
   // Types associés
   typedef int first_t; typedef float second_t;

   // Attributs
   int first;           float second;

   // Getters
   int get_first();     float get_second();

   // Setters
   void set_first(int); void set_second(float);

   // Constructeur
   paire(int, float);
};
```


C++ : Déclaration anticipée

On peut déclarer une classe avant de la définir:

```
struct ma_classe;
```

Ceci permet d'utiliser le type `ma_classe` et permet d'écrire des classes qui se font mutuellement référence.

C++ : Définition de méthodes

Normalement, les méthode déclarées dans une classes sont définies en dehors de la classe. Pour cela, leur nom est précédé de celui de la classe et de l'opérateur de résolution de portée `::`

```
   // Getters de la classe paire:
   int paire::get_first(){
       return first;
   }

   float paire::get_second(){
      return second;
   }
```


C++ : Définition de méthodes

En cas de compilation séparée (recommandée, surtout en programmation objet!) la déclaration de la classe, se trouve dans un fichier entête (paire.h) et les définitions dans un fichier de code (paire.cc)

Si on définit une méthode dans une classe (comme en Java), le compilateur la considère `inline`, mais on peut en prendre l'adresse.

C++ : Méthodes statiques

Comme en Java, une méthode peut être une méthode d'objet, ou une méthode statique. Dans ce dernier cas, elle doit être précédée du modificateur `static`.

```
struct paire{
   //Constructeur
   paire(int,double);

   //...

   // Une "method factory" (!!) de paire
   static paire make_paire(int,double);
};

paire paire::make_paire(int x, double y){
return paire(x,y);
}
```


C++ : Appels de méthodes

Si `p` est un objet de type `paire`, et `int get_first()` une méthode d'objet de la classe `paire`, `get_first()` peut être appelée à partir de `p` en écrivant `p.get_first()`.
Si `pt` est un pointeur sur objet de type `paire*`, `get_first()` peut être appelée à partir de `pt` en écrivant `pt->get_first()`.

Si `paire make_paire(int,float)` est une méthode statique, elle peut être appelée soit à partir d'un objet (`p.make_paire(4,3.2)`) soit à l'aide de l'opérateur de résolution de portée: `paire::make_paire(4,3.2)`.

C++ : Méthodes `const`

Lors de la déclaration, on peut spécifier après la liste des arguments d'une méthode d'objet qu'elle est constante, c'est-à-dire qu'elle ne modifie pas l'objet, tous les attributs de l'objets sont considérés constants dans cette méthode (sauf s'ils sont déclarés `mutable`).

C++ : Méthodes `const`

```
struct paire{
   int first;
   float second;

   int get_first() const;
   float get_second() const;

   void set_first(int);
   void set_second(float);

   paire(int, float);
};
```


C++ : Méthodes `const`

Si une variable objet est déclarée constante, elle ne peut qu'appeler des méthodes constantes.

```
const paire x(4,5);
x.set_first(3);
```

Donne à la compilation :

```
error:  passing ‘const paire’ as ‘this’
argument of ‘void paire::set_first(int)’
discards qualifiers
```


C++ : Constructeur

Comme en Java, si aucun constructeur n'est spécifié, on utilise lors de la création de l'objet un constructeur par défaut sans argument, qui ne fait rien. Dès lors qu'un constructeur est spécifié, ce constructeur par défaut n'existe plus.

Comme une méthode, le constructeur est normalement seulement déclaré dans la classe et défini en dehors.

```
paire::paire(int x, float y){
  first=x; second=y;
}
```

Comme pour une fonction ou une méthode, le constructeur peut avoir des valeurs par défaut pour certains de ces éléments.

C++ : Constructeur

Si un constructeur n'a qu'un seul argument, il induit une possibilité de conversion implicite du type de cet argument vers le type de la classe.

```
struct complex{
//...
    complex(double i);
};

int foo(complex t){/*...*/}

int main{
  return foo(3.4); // ok
}
```

Pour éviter ce comportement, il faut faire précéder le constructeur de `explicit`.

C++ : Créer un objet

- Lors de la déclaration d'une variable de type objet, on doit crée un objet correspondant. Si il y a un constructeur sans argument, il sera utilisé par défaut, sinon, on indique après le nom de la variable, entre parenthèses, les valeurs des arguments du constructeur.

  `paire x(3,5.2f);` déclare et crée l'objet `x`.

  On peut aussi affecter une valeur à la variable, il y a alors appel à un constructeur dont le type de l'argument correspond à la valeur et qui ne doit pas être `explicit`.

C++ : Créer un objet

- On peut créer un objet anonyme en utilisant le nom de la classe suivi entre parenthèses des arguments permettant de créer l'objet.

  ```
  paire paire::get_paire(int x, double y){
    return paire(3,5.2f);
  }
  ```

- On peut allouer explicitement un objet en utilisant l'opérateur `new` qui retourne un pointeur sur l'objet.

  `paire* p=new paire(3,5.2f);`

  Cet objet devra être explicitement désalloué (`delete p;`).

C++ : Constructeur par copie

À chaque fois qu'un objet est dupliqué (passage par valeur dans une fonction par exemple), un constructeur par copie est appelé. Si celui-ci n'est pas explicitement défini, l'action du constructeur par défaut consiste à copier la valeur des attributs.

On peut redéfinir ce constructeur; il s'agit d'un constructeur qui prend en paramètre une référence constante d'un objet du type courant:

`paire(const paire&);` par exemple

Attention, si ce constructeur est explicitement défini, il faut explicitement définir la copie des valeurs des attributs qui ne se fait plus toute seule.

C++ : Destructeur

À chaque fois qu'un objet est détruit (fin de vie d'une variable automatique, désallocation explicite), un destructeur de l'objet est appelé. Par défaut, ce destructeur détruit seulement l'objet et non les objets associés. (Penser aux listes chaînées par exemple).

On peut redéfinir ce destructeur. Il n'a aucun argument, aucun type de retour; il porte le nom de la classe précédé d'un tilde: exemple : `~paire();`

C++ : Attributs

Comme les méthodes, les attributs peuvent être des attributs d'objet ou de classe (s'ils sont précédés de `static`).

Hormi les attributs statiques constants d'un type "primitif", aucune initialisation n'est autorisée lors de la déclaration des attributs. Les attributs statiques sont initialisés en dehors de la classe.

```
struct toto_t{
  static int compteur;

  toto_t(){
    compteur++;
  }
};

int toto_t::compteur = 0;
```

C++ : Attributs

L'initialisation des attributs non statiques est spécifiée lors de la définition des constructeurs, entre la parenthèse qui ferme la liste des argument et l'accolade qui ouvre le corps du constructeur. La liste des initialisations est introduite par `:`.

```
paire::paire(int x,float second): first(x), second(second){
   //Empty
}
```

L'accès aux attributs se fait comme pour les méthodes, grâce à un objet suivi d'un point, ou, pour les attributs statiques grâce au nom de la classe suivi de `::`.

C++ : Types associés

Un type peut être défini dans une classe, par exemple:

`typedef int first_t;` dans la classe `paire`

On accède à un tel type grâce à l'opérateur de résolution de portée:

```
paire x(3,5.2);
paire::first_t n=x.get_first();
```

Ce mécanisme est très important en programmation générique.

Par exemple, chaque collection de la biliothèque standard définit un type `iterator`.

C++ : Accès aux composants de la classe et `this`

L'accès aux composants de la classe courante se fait directement par le nom du composant (attribut ou méthode).

Si un attribut est masqué par un argument ou une variable locale, on peut y accéder
– soit à l'aide du nom de la classe suivi de `::` s'il s'agit d'un attribut statique,
– soit à l'aide de `this` qui est un pointeur sur l'objet courant.

```
void paire::set_first(int first){ this->first = first; }
```

Si le composant est hérité d'une classe mère, il est conseillé d'utiliser l'opérateur de résolution de portée ou `this`. Ce sera même obligatoire avec les classes *template*.

C++ : Visibilité des composants

En C++, il y a trois niveaux de visiblité:

- `public` les composants publics sont accessibles par toutes les fonctions ou classes;
- `private` les composants privés ne sont accessibles que par des méthodes de la classe;
- `protected` les composants protégés ne sont accessibles que par des méthodes de la classe ou des classes héritant de la classe.

C++ : Visibilité des composants

On fixe le niveau de visibilité en l'écrivant sur une ligne suivi de `:`.

Tous les composants déclarés ensuite auront cette visibilité jusqu'à la prochaine spécification.

Si la classe est déclarée par `struct`, par défaut les composants sont publics, si elle est déclarée par `class`, ils sont privés.

C++ : Visibilité des composants

```
struct paire{
 private:
   int first;
   float second;

 public:
   int get_first() const;
   float get_second() const;

   void set_first(int);
   void set_second(float);

   paire(int, float);
};
```

C++ : Fonctions ou classes amies

Dans une classe, on peut spécifier des fonctions, méthodes ou classes extérieures qui pourront accéder aux composants privés de la classe. Pour cela, on écrit une ligne similaire à la déclaration de la fonction, classe ou méthode précédée de `friend`.

```
friend void affiche(const paire& p);
friend struct triple;
friend int triple::get_first();
```

La déclaration d'amitié ne déclare pas l'entité.
La relation d'amitié n'est pas transitive.

Pointeurs sur des membres de classe

Pour les membres statiques, on accède aux adresses des membres comme aux membres eux-même grâce à l'opérateur de résolution de portée:

```
struct A{
  static int n;
  static void affiche(int x);
};
//...

   int * p=&A::n;
   void (*f)(int)= &A::affiche;
```


Pointeurs sur des membres de classe

Par contre, les pointeurs sur les membres objets ont un type particulier :

```
struct A{
  int n;
  void affiche(int x);
};
//...

   int A::* p=&A::n;
   void (A::*f)(int)= &A::affiche;
```

L'opérateur de déréférencement est obligatoire pour prendre l'adresse d'une méthode non statique.
Si `x` est un objet de type `A`, on peut alors écrire `x.*p` ou `(x.*f)(4)`.

C++ : Surcharge d'opérateurs

On peut définir le comportement des opérateurs pour des types sur lesquels il n'était pas défini. Pour cela, deux solutions

- On utilise une fonction de nom `operator` *Op*:

```
paire operator+(const paire& p, const paire& q){
  return paire(p.first+q.first, p.second+q.second);
}
```

  Si `x` et `y` sont deux paires, on peut alors écrire `operator+(x,y)` ou `x+y` indifféremment.

C++ : Surcharge d'opérateurs

- On utilise une méthode de nom `operator` *Op* dans la classe correspondant au premier argument.

```
paire operator+(const paire& q) const{
  return paire(first+q.first, second+q.second);
}
```

  On peut alors écrire `x.operator+(y)` ou `x+y` indifféremment.

C++ : Surcharge d'opérateurs

Attention, les deux écritures ne sont pas totalement équivalentes.

Un opérateur défini comme une fonction permet le transtypage sur ces deux opérandes, alors que s'il est défini comme une méthode, le transtypage n'est pas possible sur le premier argument, car l'identification de la méthode (grâce au type du premier argument) est préalable à tout transtypage.

C++ : Surcharge d'opérateurs

Imaginons que le type `paire` dispose d'un constructeur `paire(int n);` qui permet un transtypage de `int` à `paire`

```
paire p(3,4.5);
int x(3);
p=x+p;
```

Le code précédent est correct si la surchage de l'opérateur `+` est faite par une fonction, pas par une méthode de la classe `paire`.

C++ : Opérateurs non surchargeables

```
::
.
.*
?:
sizeof
typeid
static_cast
dynamic_cast
const_cast
reinterpret_cast
```


C++ : Surcharge de ++

```
struct Integer{
  int val;
  //...

  Integer& operator++(){ // ++ infixe
    ++val;
    return *this;
  }

  Integer operator++(int){ // ++ postfixe
    Integer i(*this)
    ++val;
    return i;
  }
};
```

L'argument `int` est fictif et ne doit pas être utilisé.

C++ : Surcharge du transtypage (cast)

`operator` **type** `() const;`

Nécessairement dans une classe:

```
struct test{
  //...

  operator int() const;
};
```

permet la conversion d'un objet `test` en `int`.

La syntaxe est très particulière : pas de type de retour.

La conversion dans l'autre sens peut être rendue possible grâce à un constructeur à un argument et non "`explicit`" de la classe `test`.

C++ : Surcharge de (): Foncteur

On peut surcharger l'opérateur de passage d'arguments.

On obtient une classe dont les objets peuvent être utilisés comme des fonctions. On parle de foncteurs.

L'arité de cet opérateur n'est pas fixé.
Ce mécanisme est beaucoup utilisé dans la bibliothèque standard.

C++ : Surcharge de (): Foncteur

```
struct polynomial{
  unsigned degree;
  double * coeff;
  //...

  //evaluation of the polynomial in x
  double operator()(double x) const;
};

double polynomial::operator()(double x){
  double r=0;
  for(unsigned i=degree; i>=0; --i){
    r=r*x+coeff[i]
  }
  return r;
};
```

C++ : Surcharge de l'opérateur []

Cette opérateur n'est surchargeable que dans une classe.

Si on veut pouvoir utiliser le résultat comme "lvalue", il faut que le type de retour soit une référence:

```
struct int_paire{  //...
   int& operator[](int i){
     switch(i){
        case 1: return first;
        case 2: return second;
        default : throw out_of_bounds_exception();
     }
   }
};

//...
int_paire p;    p[1]=4;
```


C++ : Surcharge de l'opérateur []

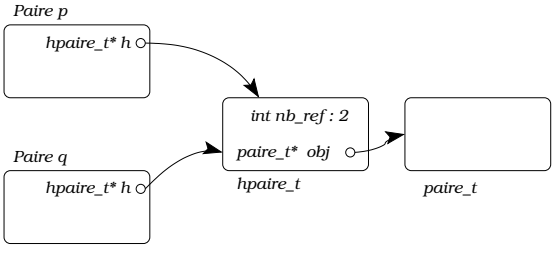
Mais il faut aussi prévoir le cas des objets constants:

```
struct int_paire{

 //...

   int& operator[](int i);

   const int& operator[](int i) const;

};
```


C++ : Surcharge des opérateurs * (unaire) et ->

Ces opérateurs sont surtout utilisés dans des classes qui définissent des itérateurs, des pointeurs "intelligents", etc...

C++ : Surcharge des opérateurs * (unaire) et ->



C++ : Surcharge des opérateurs * (unaire) et ->

```
struct Paire{
 private :
 hpaire_t* h;

 public :
 //...
 /// @return Une reference sur la paire encapsulee
  paire_t& operator*(){ return *(h->obj); }

  const paire_t& operator*()const{return *(h->obj);}
};
```


C++ : Surcharge des opérateurs * (unaire) et ->

Dans ce type de classe, il est naturel de surcharger aussi `->`, ce qui évite de devoir écrire `(*p).mamethode()`.

Noter que l'opérateur `->` ne peut être surchargé que dans une classe.

C++ : Surcharge des opérateurs * (unaire) et ->

```
struct Paire{
 private :
 hpaire_t* h;

 public :
 //...
 /// @return Une reference sur la paire encapsulee
   paire_t& operator*(){ return *(h->obj); }
   const paire_t& operator*()const{return *(h->obj);}

 /// @return Un pointeur sur la paire encapsulée
   paire_t* operator->(){ return h->obj; }
   const paire_t* operator->() const{return h->obj;}
};
```


C++ : Surcharge de =

Surcharger l'opérateur d'affectation n'a un effet que pour l'affectation elle-même. Toute copie d'objet est prise en charge par le constructeur de copie.

Généralement, redéfinir l'opérateur d'affectation va de paire avec une redéfinition du constructeur de copie et leur action est souvent similaire.

Dans ces classes, le destructeur est souvent lui aussi redéfini.

Par défaut, l'affectation recopie les valeurs des attributs; si on surcharge =, il faut explicitement préciser ce que l'on veut.

Exemple

```
struct hpaire_t{
  private :
  paire_t* obj;
  int nb_ref;

  hpaire_t(paire_t* p) obj(p), nb_ref(1) {}

  ~hpaire_t(){
    delete obj;
  }

  friend struct Paire;
};
```


Exemple

```
struct Paire{
  private :
  hpaire_t* h;

  void release(){ --h->nb_ref;
    if(h->nb_ref<=0) delete h; handler = NULL;
  }

  public:

  Paire() : h(NULL) {}
  Paire(paire_t* p) : h(new hpaire_t(p)) {}

  ~Paire(){ release(); }
  //...
```


Exemple

```
  //...

  Paire(Paire & r) : h(r.h){
    if (h)  ++h->nb_ref;
  }

  Paire& operator=(Object& r){
    if(&r != this){ release(); h = r.h;
                    if (h) ++h->nb_ref; }
    return *this;
  }
  //...
};
```


Exemple

```
  Paire p(new paire_t(3,7));
  Paire q(p);

  q = Paire(new paire_t(7,6));

  std::cout << q->get_first() << std::endl;
```

C++ : Surcharge de << et >>

Pour gérer l'écriture et la lecture d'un objet sur un flot standard (`#inclide<iostream>`) , il faut surcharger les opérateurs `<<` et `>>`.

Dans ce cas, le premier argument de l'opérateur est le flot; il faut donc les surcharger à l'aide de fonctions.

C++ : Surcharge de << et >>

```
std::ostream operator<<(std::ostream& o, const paire& p){
   o << '(' << p.get_first() << ','
            << p.get_second() << ')' ;
   return o;
}
std::istream operator>>(std::istream& i, paire& p){
//Declarée friend dans paire.  Pourquoi ?
   char c;
   i >> c;  if(c!='(')  throw syntaxe_exception();
   i >> p.first >> c;
   if(c!=',')  throw syntaxe_exception();
   i >> p.second >> c;
   if(c!=')')  throw syntaxe_exception();
   return i;
}
```


C++ : Surcharge de new et delete

On peut surcharger les opérateurs d'allocation dynamique.

Si la surcharge est faite en tant que fonction, elle est appelable pour n'importe quel type.

Si la surcharge est faite en tant que méthode (statique), elle n'est appelable que pour le type objet contenant la définition de cette méthode.

La syntaxe de la redéfinition est:

```
void* operator new(size_t size, ...);
void* operator new[](size_t size, ...);

void operator delete(void* p, size_t size);
void operator delete[](void* p, size_t size);
```


C++ : Surcharge de new et delete

L'opérateur `new` peut avoir plusieurs arguments, mais le premier doit être de type `size_t` et n'a pas besoin d'être renseigné lors de l'utilisation; sa valeur est la taille du type.

L'opérateur `delete` a deux paramètres qui n'ont pas besoin d'être renseignés, le premier est un pointeur sur l'objet à libérer, le second est sa taille.

Noter que pour un tableau, cette taille est supérieure au produit de la taille de chaque objet par le nombre d'objets.

Après redéfinition, les opérateurs par défaut restent accessibles, en écrivant `::new` ou `::delete`.

C++ : Surcharge de new et delete

On peut ajouter des paramètres à `new`:

`void* operator new(size_t s, void* p);` par exemple.

Cet opérateur s'utilise alors sous la forme `new(p) toto();`

La syntaxe de cet exemple existe par défaut, elle signifie d'utiliser l'espace pointé par `p` pour stocker le nouvel objet; cet espace doit avoir été alloué par ailleurs.

En appelant le destructeur d'un objet comme une méthode, on peut détruire un objet sans désallouer son espace.

C++ : Surcharge de new et delete

```
//allocation memoire + creation objet
toto *t=new toto(4,5);

//...

//destruction objet sans desallocation
t->~toto();

//...

//creation d'objet a l'emplacement t (sans allocation)
toto *u=new(t) toto(3,7);

//...

//destruction objet + desallocation
delete u;
```

Échec d'allocation dynamique

(Identifiants déclarés dans `#include<new>`)

Selon la norme, en cas d'échec de `new`, un gestionnaire d'erreur est appelé, qui lance une exception de type `std::bad_alloc`.

On peut modifier le comportement de ce gestionnaire grâce à la fonction `std::set_new_handler(void (*f)() );` .

Ce sera alors la fonction pointée par `f` qui sera exécutée en cas d'échec.

Échec d'allocation dynamique

Si l'on désire qu'en cas d'échec `new` renvoie un pointeur nul, on peut utiliser l'opérateur `new` avec l'argument `std::nothrow` (de type `std::nothrow_t`):

```
int* p=new(std::nothrow) int[1000000];
if(p==NULL) //gestion de l'erreur
```

Dans certaines implémentations, ceci est le comportement par défaut.

Héritage

En C++, une classe peut hériter de plusieurs classes:

```
struct A{ /* ... */ };

struct B{ /* ... */ };

struct C : A, B { /* ... */ };
```

Dans cet exemple, la classe `C` hérite de `A` et `B`.

Héritage : droits

Pour préciser comment une classe `C` étend une classe `A`, on fait précéder le nom de `A` d'un qualificatif de visibilité: `public`, `protected` ou `private`.

Ce qualificatif réduit la visibilité des composants de `A` à travers la classe `C` ou un de ses objets.

Par exemple: `struct C : private A { /* ... */ }`.

Si `void foo();` est une méthode publique de `A` et `x` un objet de type `C` déclaré en dehors de la classe `C`, on ne peut pas écrire `x.foo();`.

Héritage : droits

L'héritage public est le plus fréquent et correspond à l'héritage classique.

L'héritage privé peut être une façon d'implémenter la composition:

```
struct Cercle :  private Point {//...};
```

Par défaut, une classe déclarée `struct` hérite publiquement et une classe déclarée `class` hérite de manière privée.

Héritage : constructeur

Lors de la construction d'un objet, on peut appeler le constructeur de classes dont on hérite directement, de la même façon que l'on initialise les attributs d'objets.

```
struct Cercle : private Point {
  double rayon;

  Cercle(int x, int y, double rayon)
            : Point(x,y), rayon(rayon) {}

};
```

Si on ne le fait pas, la classe mère doit posséder un constructeur sans argument.

Héritage : redéfinition de méthode

On peut dans une classe fille C redéfinir une méthode définie dans une de ces classes mères A.

```
struct A{
  void foo();
};
```

```
struct C : public A{
  void foo();
};
```

Dans ce cas, si x est un objet de type C et que l'on veut accéder à la méthode la classe A, on devra utiliser l'opérateur de résolution de portée et écrire:

```
x.A::foo();
```


Héritage multiple d'une classe

```
struct A{
  static int nA;     const int n;          A : n(nA++) {}
};
int A::nA = 0;
struct B : public A{};            struct C : public A{};
struct D : public B, public C{
  void aff(){ cout << B::n << "," << C::n << endl; }
};
```

Dans ce cas, la classe D hérite deux fois de la classe A.

On ne peut pas appeler n directement dans D: ambiguïté.

La méthode aff affiche par exemple 0,1.

Les "sous-objets" sont créés dans l'ordre dans lesquels les classes mères sont citées.

Héritage multiple d'une classe

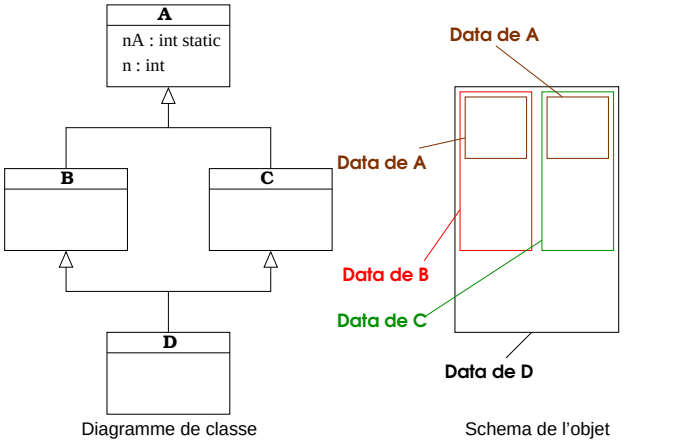


Diagramme de classe

Schema de l'objet

Héritage multiple d'une classe

```
struct A{ /* ... */ };
struct B : public A{ /*...*/ };
struct C : public A, public B{ /*...*/ };
```

La déclaration de C n'est pas licite car il y aura ambiguïté lors de l'accès aux composants de A. En particulier, on ne peut pas spécifier qu'on veut accéder à un composant de la classe A directement héritée.

Héritage multiple d'une classe

```
struct A{ /* ... */ };
struct B : public A { /*...*/ };
struct C : public A { /*...*/ };
struct D : public B, public A { /*...*/ };

//...
D d;   A* p = &d;
```

La conversion d'un pointeur de type D* en A* est ambiguë. À la compilation, on obtient:

```
error:  'A' is an ambiguous base of 'D'
```


Héritage virtuel

On peut utiliser le mot virtual pour faire en sorte qu'une classe ne puisse apparaître qu'une fois parmi les ancêtres d'une autre classe.

Si B et C héritent virtuellement de A, pour toute classe D qui aura à la fois B et C parmi ses ancêtres les composants de A vus à travers B ou C seront identiques.

On peut dire que A n'héritera alors qu'une fois de D.

Héritage virtuel

```
struct A{
  static int nA;
  const int n;
  A : n(nA++) {}
};
int A::nA = 0;
struct B : virtual public A{};
struct C : virtual public A{};
struct D : public B, public C{
  void aff(){ cout << B::n << "," << C::n << endl; }
};
```

La méthode `aff` affiche deux fois le même nombre.

Héritage virtuel

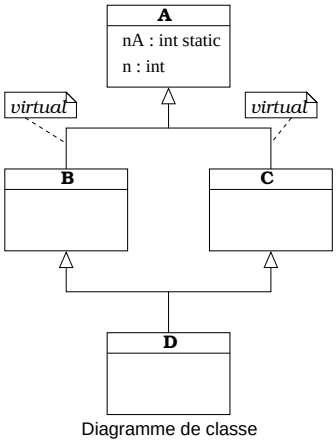
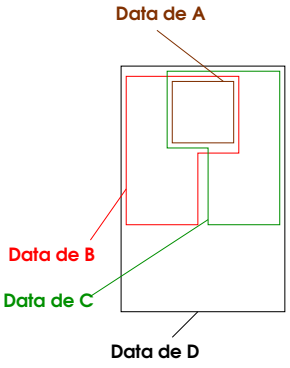


Diagramme de classe



Schema de l'objet

Héritage virtuel

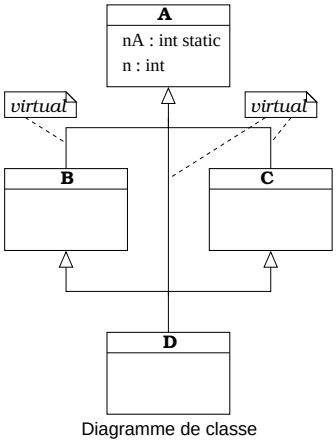
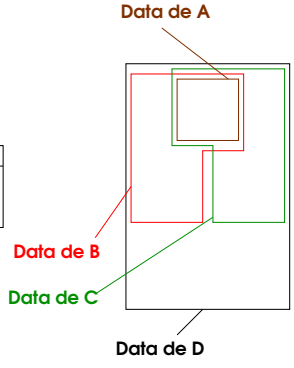


Diagramme de classe



Schema de l'objet

Héritage virtuel

Les "sous-objets" de la hiérarchie virtuelle sont créés au moment de la création de l'objet principal, sans intermédiaire.

```
struct A{
  const int n;
  A(int k) : n(k) {}
};

struct B : virtual public A{ B(): A(1) {} };
struct C : virtual public A{ C(): A(2) {} };
};

struct D : public B, public C{
  D(): A(3) {}
  void aff(){ std::cout << n  << std::endl; }
};
```

`aff()` affiche 3.

Non polymorphisme

Par défaut, en C++, les types sont résolus à la compilation, ce qui empêche le polymorphisme.

Non polymorphisme

```
struct A{
  int norm() { /***/ }
  int bi_norm() { return 2*norm(); }
};
struct C : public A{
  int norm() { /***/ }
};
```

Lors de la compilation de la méthode `bi_norm()`, le compilateur examine le type de l'objet courant dans cette méthode (`A`) et fait en sorte d'appeler la méthode `norm()` correspondante.

Donc si `x` est un objet de type `C`, `x.bi_norm()` est un appel de la méthode `bi_norm()` de `A` à l'intérieur de laquelle sera appelée la méthode `norm()` de ... `A`.

Polymorphisme : méthodes virtuelles

```
struct A{
  virtual int norm() { /***/ }
  int bi_norm() { return 2*norm(); }
};
```

Si on déclare une méthode `virtual`, son appel n'est pas résolu à la compilation, mais à l'exécution, le programme regarde quel est le vrai type de l'objet appelant et appelle la méthode correspondant à cet objet.

Ce mécanisme ralentit l'exécution.

Classe abstraite : méthode virtuelle pure

On peut dans une classe déclarer une méthode virtuelle sans préciser de code. Pour cela, on fait suivre la déclaration de la méthode de `= 0`.

Exemple: `virtual int get_coord() const = 0;`

Une telle méthode est dite virtuelle pure.

Une telle classe ne peut être instanciée, et cette méthode devra être définie dans toute classe qui l'étend et que l'on souhaite instancier.

Interface et classe abstraite

En C++, il n'y a pas de mot clé pour caractériser une classe abstraite ou une interface.

Si une classe contient au moins une méthode virtuelle pure, ou si son constructeur est protégé, elle est abstraite.

Si une classe contient uniquement des déclarations de méthodes virtuelles pures publiques, c'est une interface.

Destructeur virtuel

Si des objets alloués dynamiquement doivent êtres pointés par des pointeurs d'un sur-type, il est sain que les destructeurs soient déclarés virtuels.

```
struct A{};
struct C : virtual public A{};

//..
   A* p=new C;
   C x;
//...
   delete p;
```


Hiérarchie polymorphe

Une classe qui contient des déclarations virtuelles définit un type polymorphe.

Toute type qui étend un type polymorphe est lui aussi polymorphe.

Les objets des types polymorphes sont plus gros ($\simeq$ 4 octets), pour stocker des informations sur leur type.

Le programme utilise ces informations pour répondre au polymorphisme, mais on peut y avoir accès dans une certaine mesure.

Typage dynamique: typeid

L'opérateur `typeid` prend en argument un type ou une référence d'objet, et retourne `std::type_info` (inclure `typeinfo`).

Si l'argument est une référence de type non polymorphe, `type_info` correspond au type de la référence, mais si le type est polymorphe, `type_info` indique le véritable type de l'objet.

```
struct A { virtual void test(){} };
struct C : public A{};

//..
   A* p=new C;
   if(  typeid(*p)==typeid(C) ) /// true
      //...
```

Typage dynamique : type_info

```
class type_info{
public:
  virtual ~type_info();
  bool operator==(const type_info &rhs) const;
  bool operator!=(const type_info &rhs) const;
  bool before(const type_info &rhs) const;
  const char* name() const;
private:
  type_info(const type_info &rhs);
  type_info &operator=(const type_info &rhs);
};
```

L'ordre sous-jacent à `before` n'est pas spécifié...

Transtypage dynamique : dynamic_cast

Si `C` étend `A`, on peut écrire `C c; A* p=&c;`

Pour récupérer un pointeur de type `C*` à partir de `p`, on peut écrire `C* q = (C*)p`, mais c'est dangereux.

Entre types polymorphes, grâce au typage dynamique, on peut vérifier que l'objet pointé par `p` est bien de type `C` ou d'un sous-type:

```
C* q = dynamic_cast<C*>(p);
```

Si l'opération n'est pas licite, cela déclenche une exception `bad_cast`.

Si les types ne sont pas polymorphes, on a une erreur de compilation.

Transtypage dynamique : dynamic_cast

On peut aussi utiliser le transtypage pour obtenir une référence sur un objet.

```
C& q = dynamic_cast<C&>(*p);
```


Transtypage dynamique : dynamic_cast

Le transtypage dynamique permet aussi de faire du transtypage horizontal:

```
struct A{virtual void fa(){} };
struct B{virtual void fb(){} };
struct C: public A, public B {};

//...

C c;
A& a = c;
B& b = dynamic_cast<B&>(a);
```


Transtypage dynamique : dynamic_cast

Le transtypage vertical n'est pas prioritaire sur le transtypage horizontal.

```
struct A{virtual void fa(){} };
struct B: public A {};
struct C: public A {};
struct D: public B, public C {};

//...

D d;
B& b = d;
A& a = dynamic_cast<A&>(b);
```

donne

```
error:  'A' is an ambiguous base of 'D'
```


Transtypage static : static_cast

Cet opérateur a une syntaxe similaire au précédent.

Il effectue un transtypage statique (à la compilation) mais, contrairement au transtypage usuel, il ne peut pas transformer un pointeur sur objet constant en pointeur sur objet non constant ou une référence constante en référence non constante.
On peut l'utiliser sur des types non polymorphes.

Transtypage de constance : const_cast

Cet opérateur a une syntaxe similaire aux précédents.

Il permet de supprimer le caractère constant d'un pointeur ou d'une référence mais ne permet pas d'autre modification de type.

Ainsi, un `const int*` peut uniquement être transformé en `int*`.

Transtypage de réinterprétation : reinterpret_cast

Cet opérateur a une syntaxe similaire aux précédents.

Il réinterprète les données en mémoire dans le type spécifié, mais ne viole pas le caractère constant.

Ainsi, si `x` est un `int`,

```
reinterpret_cast<double&>(x);
```

est équivalent à `*((double*) &x)`.

Les exceptions : throw

Pour lever une exception, on utilise l'opérateur `throw` suivi d'une valeur de n'importe quel type.

Cette exception, si elle n'est pas attrapée se propage en provoquant l'arrêt successif des méthodes et fonctions appelantes jusqu'à la fonction `main`, puis provoque l'arrêt du programme.

Les exceptions : try catch

Pour qu'une exception soit interceptée, elle doit survenir dans un bloc introduit par `try` suivi d'au moins un bloc introduit par `catch(`type_exc e`)`.

Si une exception est survient dans le bloc `try`, les types d'exceptions gérés par les blocs `catch` sont examinés successivement. Le premier bloc ayant un argument du type de l'exception ou d'une de ses classes mères est exécuté.

Les exceptions : try catch

On peut utiliser `catch(...)` pour le dernier bloc `catch` qui sera alors exécuté si aucun autre catch ne correspond à l'exception survenue.

Pour les autres `catch`, le type spécifié sera généralement une référence afin d'éviter la recopie de l'objet exception.

Pour relancer dans un bloc `catch` l'exception interceptée, il suffit d'écrire `throw ;`

Les exceptions : try catch

Si aucun bloc `catch` convenable n'est trouvé, la fonction `std::terminate()` est appelée, qui affiche un message d'erreur puis appelle la fonction `abort()`.

On peut modifier la fonction appelée en utilisant `std::set_terminate(void (*f)())` (inclure `exception`), ce sera alors la fonction `*f` qui sera exécutée au lieu de `std::terminate()`.

Les exceptions : exceptions autorisées

On n'est pas obligé de spécifier qu'une fonction ou méthode est susceptible de propager une exception, mais c'est possible:

```
int ma_fonction(int x) throw (int, double, my_exception);
```

Si une exception d'un autre type survient dans la fonction/méthode, la fonction `std::unexpected()` est appelée, qui affiche un message d'erreur puis appelle la fonction `abort()`.

La encore, on peut modifer la fonction appelée avec `std::set_unexpected(void (*f)())`.

Les exceptions : dans un constructeur

Un constructeur peut lever une exception; la construction de l'objet échoue alors.

Pour effectuer le nettoyage des composantes de l'objet mort-né, le C++ permet que le corps d'un tel constructeur soit un bloc `try` suivi de blocs `catch` qui permettent de libérer la mémoire éventuellement réservée avant la levée de l'exception.

Même si on n'utilise pas `throw ;` dans les blocs `catch`, l'exception sera propagée.

Les exceptions : dans un constructeur

```
struct A{
  A() throw(int)
  try{
   ///..
  }
  catch(int x){
    ///...
  }

  //...
};
```


Les exceptions standards

```
class exception
{
public:
    exception() throw();
    exception(const exception &) throw();
    exception &operator=(const exception &) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

`what()` est appelé pour compléter l'éventuel message d'erreur.

Les exceptions standards

Les exceptions standards suivantes ne sont pas supposées être lancées par l'utilisateur. Les deux premières sont définies dans `exception`, la dernière dans `new`

- `exception` le type de base des exceptions standards
- `bad_exception` qui peut être utilisé dans le gestionnaire "unexpected"
- `bad_alloc` utilisable par `new`

Les exceptions standards

Les autres exceptions prédéfinies (`stdexcept`) appartiennent à deux grandes catégories:

- `logic_error` qui indique généralement un bug dans le programme
  sous-classes: `domain_error`, `invalid_argument`, `length_error`, `out_of_range`
- `runtime_error` qui sont des exceptions qui peuvent survenir dans la vie normale d'un programme
  sous-classes: `range_error`, `overflow_error`, `underflow_error`

Les exceptions standards

Ces types possèdent un constructeur prenant en argument un `const char*` permettant de préciser le message d'erreur attaché à l'exception.

Pour définir ses propres exceptions, on étendra généralement `runtime_error`.