2. Héritage avancé

2.1 Compléments sur l'héritage

La syntaxe étendue de l'héritage en C++ est la suivante :
class B : [public|protected|private] class A1 [,[public|protected|private] class Ai]$^{2..n}$

R Dans cette syntaxe, on peut remarquer :
- Une classe C++ peut hériter de plusieurs classes (héritage multiple),
- trois modes d'héritage de chaque classe : publique, protégé et privé. Ces modificateurs permettent de définir la façon par laquelle la classe dérivée accède aux membres hérités de ses classes mères (super-classes).

Le tableau suivant résume les droits d'accès de la classe héritière selon mode d'héritage :

		Visibilité des Membres de la classe Mère		
		public	protected	private
Mode d'héritage	public	public	protected	private
	protected	protected	protected	private
	private	interdit	interdit	interdit

Par défaut c'est le mode d'héritage private qui est utilisé pour les classes et public pour les structures (struct).

Definition 2.1.1 Une méthode est dite redéfinie lorsqu'elle est reprise dans une classe dérivée avec le même nom et les mêmes paramètres formels et une nouvelle implémentation. Une telle méthode est dite aussi méthode polymorphe (avec plusieurs implémentations).

Lorsqu'on invoque une méthode polymorphe, en Java c'est l'implémentation du type courant (dynamique) de l'objet qui est invoquée et non celle du type déclaré (statique). En C++, par défaut, c'est l'implémentation de son type déclaré (statique) qui est invoquée, si on veut invoquer l'implémentation du type dynamique on doit précéder à sa déclaration dans la classe mère par le mot clé « virtual ».

Dans l'exemple des classes Gcomplex et Complex, on a la méthode void print() qui est une méthode polymorphe (redéfinie entre Complex et Gcomplex).

Si on a par exemple le code suivant :

```
1 Complex *c=new GComplex(10,10);
2 c->print();
```

avec l'implémentation ancienne on aura appel à la méthode print() de la classe Complex (type statique de l'objet c). Si on veut que cet appel invoque l'implémentation du type dynamique de c (celle de la classe GComplex), il faut ajouter à la déclaration de la méthode print dans la classe Complex le mot clé virtual.

Dans Complex.h on aura la déclaration :

```
1 class Complex {
2 public:
3     ...
4     virtual void print();
5     ...
6 };
```

Dans le premier cas on dit que la méthode print() est à liaison statique (liaison au moment de l'édition de lient). Dans le deuxième la liaison de la méthode print() est retardée jusqu'au moment de l'appel effectif de cette dernière (au moment de l'exécution) afin d'avoir la bonne méthode à invoquer. Dans le deuxième cas, la méthode est appelée en C++ une méthode virtuelle.

2.1.1 Appels des constructeurs

Le constructeur de la classe dérivée invoque automatiquement les constructeurs des classes mères successivement l'un après l'autre (dans l'ordre d'apparition dans la clause d'héritage). Dans le cas ou les super-constructeurs ne nécessitent pas de paramètres l'appel se fait automatiquement, le cas échéant, il faut indiquer dans le code du constructeur de la classe fille les paramètres à transmettre à chaque constructeur.

Dans notre exemple, si on veut inclure un constructeur de la classe Gcomplex avec 2 paramètres réels qui les transmis automatiquement au constructeur de la classe Complex :

```
1 Complex::Complex(float a,float b) {
2     rel=a;
3     img=b;
4     p=new Point(a,b);
5 }
6 GComplex::GComplex(float a,float b):Complex(a,b){
7     // les instructions propres au constructeur de GComplex.
8 }
```

Dans le cas de l'héritage multiple, les appels aux constructeurs des super-classes doivent être séparés par une virgule.

2.1.2 Héritage Multiple

On parle d'héritage multiple lorsque une classe hérite de plusieurs autres classes.

```
1 class A {...};
2 class B: public A {...};
3 class C: public A {...};
4 class D: public B, public C {...};
```

La classe D hérite des classes B et C, tout objet de la classe D est un objet de la classe B et aussi un objet de la classe C.

Inconvénients de l'héritage multiple

1. **L'ambiguité :** supposont qu'on a des membres (données ou méthodes) qui ont le même nom dans les classes B et C, dans ce cas, il faut utiliser l'opérateur de résolution de portée (: :).
2. **La redondance :** on suppose que la classe A contient la définition :

```
1 class A { public: int x;
2 void m();
```

```
3  };
```

x et m vont figurer dans les deux classes B et C par héritage (simple). La classe D héritera deux(2) copies de x et de m à partir des classes B et C (A : :B : :x et A : :C : :x), idem pour la méthode m().

Pour contourner cette redondance, on utilise l'héritage virtuel décrit dans la section suivante.

2.1.3 Héritage virtuel

Étant donné les classes précédentes A,B,C et D tel que :

```
1  class A {
2      public:
3          int x;
4          A();
5  };
6  A::A(){
7      cout<<" Constructeur de A";
8  }
9  class B:public A {
10     public:
11         int x;
12         B();
13 };
14 B::B(){
15     cout<<" Constructeur de B";
16 }
17 class C:public A {
18     public:
19         int x;
20         C();
21 };
22 C::C(){
23     cout<<" Constructeur de C";
24 }
25 class D:public B,public C  {
26 public:
27     int x;
28     D();
29 };
30 D::D(){
31     cout<<" Constructeur de D";
32 }
```

Dans le code de test, si on instancie un objet de D :

```
1  D d;
```

A l'exécution on aura :

```
1 Constructeur de A
2 Constructeur de B
3 Constructeur de A
4 Constructeur de C
5 Constructeur de D
```

Afin d'éviter ce problème de redondance(répétition de l'utilisation d'une classe héritée par plusieurs chemins), on doit utiliser l'héritage virtuel qui consiste à n'utiliser une classe dans un schéma d'héritage qu'une seule fois : Dans l'exemple précédent, c'est la classe A qui est utilisée à répétition, dans la déclaration des classes B et C (les dérivées directes de A) on ajoute :

```
1 Class B: virtual public A {
2 ...
3 }
```

et

```
1 Class C: virtual public A {
2 ...
3 }
```

à l'exécution, On aura la résultat voulu :

```
1 Constructeur de A
2 Constructeur de B
3 Constructeur de C
4 Constructeur de D
```

2.2 Classes Abstraites et interfaces

Une méthode abstraite en C++ est appelée méthode virtuelle pure (méthode sans implémentation ou sans corps). Pour définir une telle méthode on doit lui affecter un 0 dans sa déclaration.

Exemple : dans le fichier Complex.h on peut définir la méthode abstraite dessiner comme suit :

```
1 class Complex {
2 public:
3     ...
4     virtual void dessiner ()=0;
5     ...
6 }
```

Definition 2.2.1 Une classe qui contient au moins une méthode virtuelle pure est une classe pure (abstraite). On ne peut pas instancier des objets à partir de cette classe.

Les classes abstraites sont très importantes dans l'AOO, elles servent de classes de base pour l'héritage. Pour qu'une classe dérivée d'une classe abstraite soit instanciable, elle doit donner une implémentation (corps) à toutes les méthodes pures de la classe abstraite.

Les méthodes pures peuvent être vues comme des clauses de contrat à réaliser par les classes héritières. Une classe qui hérite d'une classe abstraite, et ne donne pas d'implémentation à toutes les méthodes pures, demeure elle-même abstraite.

Une Interface en Java est une collection nommée de méthodes abstraites, les classes et les interfaces sont liés par la relation d'implémentation, une classe qui implémente une interface Java doit donner une implémentation à toutes ses méthodes abstraites.

En C++, Le concept d'interface proprement dit (comme en Java) n'existe pas, néanmoins elle peut être substituée par une classe abstraite ne contenant que des méthodes pures (abstraites) et éventuellement des constantes statiques. La relation d'implémentation est remplacée en C++ par l'héritage du fait que C++ autorise l'héritage multiple.

2.3 Le mécanisme RTTI (RunTime Type Information)

La norme ANSI C++ a introduit un mécanisme permettant d'obtenir des information sur le type d'une variable, d'une expression ou d'un objet. L'utilisation utile de ce mécanisme porte sur les objets polymorphes pointés ou référencés.

Les informations sur le type permettent d'identifier et/ou de comparer ce dernier afin de procéder à des optimisations adéquates. C++ introduit l'opérateur **typeid** qui retourne comme résultat un objet de la classe **type_info**. Cette dernière comporte la méthode **name()** qui renvoi une chaine de caractères représentant le type correspondant à l'objet. De plus, cette classe redéfini les opérateurs == et **!=** afin de comparer les types (objets de la classe type_info), l'opérateur == renvoi "false" si les objets type_info correspondent à deux objets de types différents.

2.3.1 Exemple illustratif

```
#include <iostream>
#include <typeinfo> // pour typeid
using namespace std ;
class Complex {
    ...
    virtual void print(){...}
};
class Gcomplex :public Complex {
    ...
    virtual void print(){...}
};
main() {
Complex c(10,10) ;
GComplex gc(20,20) ;
Complex * cp ;
cp = &c ;
cout << "type de cp : " << typeid (cp).name() << "\n" ;
cout << "type de *cp : " << typeid (*cp).name() << "\n" ;
```

```
19 cp = &gc ;
20 cout << "type de cp : " << typeid (cp).name() << "\n" ;
21 cout << "type de *cp : " << typeid (*cp).name() << "\n" ;
22 }
```

Résultat de l'exécution de l'exemple précédent :

```
1 type de cp : Complex *
2 type de *cp : Complex
3 type de cp : Complex *
4 type de *cp : Gcomplex
```

— Le type de l'objet pointé peut varier par contre le type du pointeur sur l'objet reste le même (statique).
— Les noms des types peuvent différer d'une version de C++ à une autre.
— Les objets utilisés doivent être polymorphes (comportant des méthodes virtuelles) sinon la méthode **name()** renvoi toujours le type statique du pointeur.
— On peut directement comparer des objets renvoyés par typeid :

```
1  if(typeid(objet1)==typeid(objet2)) {...}
2
```

En connaissant le type pointé par cp, il est utile parfois de convertir le pointeur lui-même vers le type réellement pointé. Dans le cas précédent "cp" pointe un objet de type "GCOmplex" hors cp il est de type "Complex *", on peut le convertir vers le type "Gcomplex *" en utilisant l'opérateur **dynamic_cast** :

```
1 Gcomplex * ncp=dynamic_cast <Gcomplex *> (cp);
```

En phase de compilation, le compilateur vérifie simplement que les type Complex et GComplex sont dérivés l'un de l'autre, la faisabilité de conversion sera confirmée au moment de l'exécution.

— Pour les pointeurs la conversion renvoi 0 (pointeur NULL) si la conversion est impossible.
— Pour les références (&) , l'exécution provoque l'exception **bad_cast**.

2.4 Gestion des Exceptions

Comme pour les cas des langages modernes, Le C++ est doté d'un mécanisme de gestion des exceptions semblable plus au moins à celui de Java et de C#.

Cette gestion d'exceptions est introduite dans le cadre de réaliser des traitements spécifiques aux événements qui en sont la cause. Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal. Il se peut aussi que le programme se termine, si aucun traitement possible n'est approprié.

Lorsqu'une erreur survienne, le programme en cours suspend son exécution normale et passe le contrôle à un gestionnaire d'exception. Si ce dernier est en mesure de traiter le type d'exception lancée on dit qu'il a attrapé l'exception. Une bonne gestion d'exceptions permet aux applications professionnelles d'une part de signaler les erreurs et d'autre part de continuer leur exécution et ne pas se contenter de s'arrêter brutalement sans même une indication sur la nature de l'erreur.

Trois aspects à prendre en compte lors du traitement des exceptions :

1. **Portion du code à protéger :** Malgré que toute instruction peut pratiquement lever une exception, La mise en place d'une gestion d'exception globale est couteuse en terme de ressources, donc, il faut limiter les zones de programmes susceptibles de provoquer des exceptions importantes.
2. **Levée d'exception :** Contrairement aux programmes Java s'exécutant sous le contrôle de la JVM qui détecte automatiquement les erreurs à l'exécution et crée l'objet exception adéquat puis le passe à l'application, les programmes C++ s'exécutent en stand-alone et doivent lever manuellement les exceptions adéquates par l'instruction :

```
throw objet;
```

3. **interception et traitement :** Pour cela on utilise l'instruction try ... catch. Une hiérarchie de classes d'exceptions est définie en C++ afin de pouvoir bien identifier l'exception provoquée par un programme. On peut utiliser plusieurs clause catch pour les différentes exceptions

La figure 2.1 illustre la hiérarchie des différents types d'exceptions qu'un programme peut provoquer. On utilise le squelette syntaxique suivant pour gérer les exceptions :

```
catch
try {
    // ici le code protégé
} catch( NomException e1 ) {
    // gérer l'exception de type (classe) e1
} catch( NomException e2 ) {
    // gérer l'exception de type (classe) e2
} catch( NomException eN ) {
    // gérer l'exception de type (classe) e3
}
}
```

R La fonction *terminate*() : Si aucun bloc catch ne peut attraper l'exception lancée, la fonction *terminate*() est alors appelée. Par défaut elle met fin au programme par un appel à la fonction abort(). On peut définir sa propre fonction *terminate*() par un appel à la fonction *set*$_t$*erminate*() définie dans <exception.h> :

```
#include <exception.h>
#include <iostream.h>
void my_terminate()
{
 cout << "my_terminate" << endl;
```

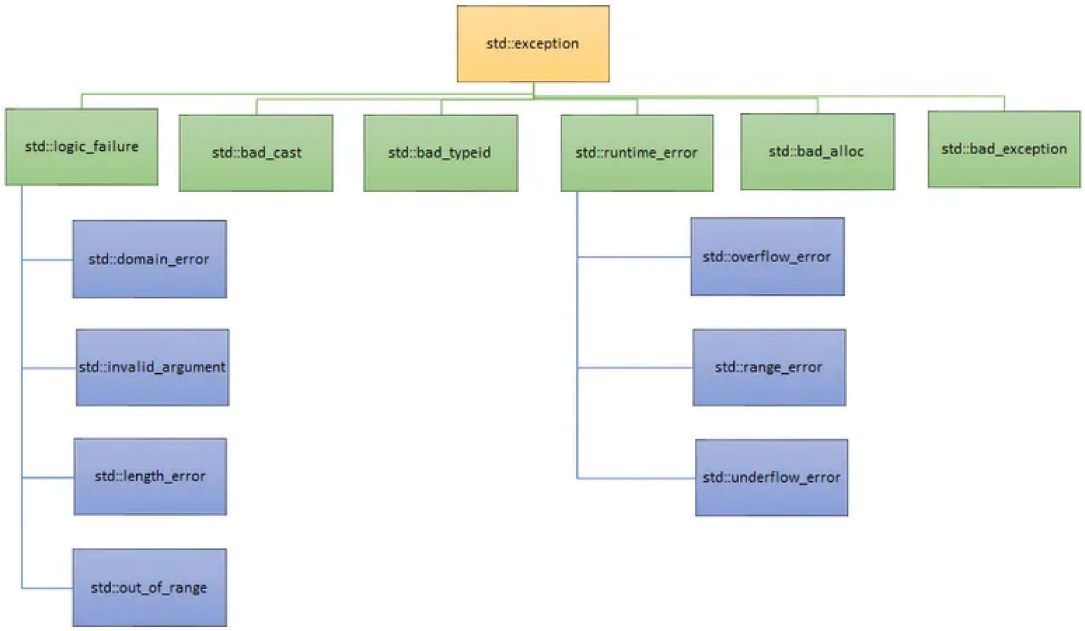

FIGURE 2.1 – Hiérarchie des classes d'exceptions

```
6   exit(1); // on ne retourne pas à la fonction appelante
7  }
8  void main()
9  {
10  try {
11  set_terminate((terminate_function) my_terminate);
12  }
13  catch ( Erreur ) { // ... }
14 }
```

R La fonction *unexpected*() : Si une fonction (ou une méthode) lance une exception qui n'est pas déclarée par un throw dans son fichier d'entête, la fonction *unexpected*() est alors appelée. Par défaut, elle met fin au programme par un appel à la fonction abort(). Cependant, on peut définir sa propre fonction *unexpected*() par un appel à la fonction *set_unexpected*() définie dans <exception.h> comme suit :

```
1 typedef void ( *unexpected_function ) ();
2 unexpected_function set_unexpected (unexpected_function
      t_func);
```

Exemple :

```
1 #include <exception.h>
2 #include <iostream.h>
3 void my_unexpected()
```

```
{
 cout << "my_unexpected" << endl;
 exit(1); // on ne doit pas retourner à la fonction appelante
}
class Erreur; // la déclaration de la classe Erreur
class Alpha; // la déclaration de la classe Alpha
class Essai
{
 public:
 void f1() throw (Erreur);
};
void Essai::f1() throw (Erreur) // la méthode de la classe
    Essai
{ Toto obj1;
 throw obj1; // ? cette fonction est supposée lancer un objet
     de Erreur
}
void main()
{
 try
 {
 set_unexpected ( (unexpected_function) my_unexpected);
 Essai e1;
 e1.f1();
 }
 catch ( Erreur )
 { // ...
 }
}
```

L'exemple suivant montre comment intercepter et traiter une exception :

```
#include <iostream.h>
class Erreur // Première exception possible, représentée par la classe
    Erreur
{
public:
 int cause; // Entier spécifiant la cause de l'exception.
 // Le constructeur. Il appelle le constructeur de la classe cause.
 Erreur (int c) : cause(c) { }
 // Le constructeur de copie. Il est utilisé par le mécanisme des
    exceptions
 Erreur (const Erreur &source) : cause(source.cause) { }
};
class Other {}; // Objet correspondant à toutes les autres exceptions.
int main(void)
{ int i; // Type de l'exception à générer
 cout << "Tapez 0 pour générer une exception de type Erreur, 1 de type
    entier:"
 cin >> i; // On va générer une des trois exceptions possibles
 try // Bloc où les exceptions sont prises en charge
 {
 switch (i) // Selon le type d'exception désirée
 {
```

```
20   case 0:
21   {
22      Erreur a(0);
23      throw (a); // On lance l'objet correspondant (ici, de la classe
        Erreur).
24   // Cela interrompt le code. Un break est donc inutile ici.
25   }
26  case 1:
27   {
28   int a = 1;
29   throw (a); // Exception de type entier
30   }
31   default: // Si l'utilisateur n'a pas tapé 0 ou 1
32   {
33   Other c; // On crée l'objet c (type d'exception Other)
34   throw (c); // et on le lance
35   }
36   }
37   } // Fin du bloc try. Les blocs catch suivent :
38   catch (Erreur &tmp) // Traitement de l'exception Erreur ...
39   { // (avec récupération de la cause)
40   cout << "Erreur ! (cause " << tmp.cause << ")" << endl;
41   }
42   catch (int tmp) // Traitement de l'exception int...
43   {
44   cout << "Erreur int ! (cause " << tmp << ")" << endl;
45   }
46   catch (...) // Traitement de toutes les autres exceptions
47   { // On ne peut pas récupérer l'objet ici.
48   cout << "Exception inattendue !" << endl;
49   }
50   return 0;
51  }
```