Héritage

1. Description de l'héritage

- L'héritage est le troisième des paradigmes de la programmation orientée objet (le 1er étant l'encapsulation, le 2e la structure de classe).

- L'emploi de l'héritage conduit à un style de programmation par raffinements successifs et permet une programmation incrémentielle effective.

- L'héritage peut être :

    - Simple: la classe dérivée n'a qu'une seule classe de base.
    - Multiple: la classe dérivée en a plus d'une classe de base.

- Il représente la relation: EST-UN

    - Un chat est un animal
    - Un avion est un véhicule
    - Un bateau est un véhicule

  alors que l'objet membre représente la relation: A-UN

    - Une voiture a un moteur

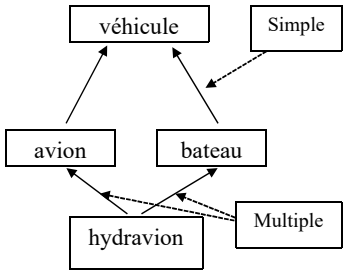
- L'héritage est mis en œuvre par la construction de classes dérivées.

Classes de base :

- La classe « vehicule » est la classe supérieure par rapport à « avion » et « bateau ».
- Les classes « avion » et « bateau » sont les classes supérieures pour « hydravion ».

Classes dérivées :

- Les classes « avion » et « bateau » sont des classes dérivées (sous-classes) de la classe « vehicule ».
- La classe « hydravion » est une classe dérivée de la classe « avion » et « bateau ».
- La classe « hydravion » est considérée aussi comme une classe dérivée de la classe « vehicule ».



Héritage simple :

- La classe dérivée n'a qu'une seule classe de base.
- Les classes « avion » et « bateau » n'ont qu'une seule classe de base : la classe « vehicule ».
  => C'est un héritage simple.

Héritage multiple :

- Une classe dérivée a plus d'une classe de base.
- La classe « hydravion » hérite des classes « avion » et « bateau ».
- La classe « hydravion » a deux classes de base.
  => C'est un héritage multiple.

2. Classe dérivée

- Une classe dérivée modélise un cas particulier de la classe de base et de ce fait, se trouve ainsi enrichie d'informations supplémentaires.
- La classe dérivée possède les propriétés suivantes:
  - Elle contient les données membres de la classe de base,
  - Elle peut en posséder de nouvelles,
  - Elle possède (à priori) les méthodes de sa classe de base,
  - Elle peut redéfinir (masquer) certaines méthodes,
  - Elle peut posséder de nouvelles méthodes.

3. Syntaxe de l'héritage

- Héritage simple :

```
class classe_dérivée:protection classe_de_base {/* etc. */}
```

- Héritage multiple :

```
class  classe_dérivée:protection  classe_de_base_1,  protection
classe_de_base_2 {/* etc. */}
```

4. Accès aux membres hérités

- Les droits d'accès protègent les données et les méthodes et réalisent aussi l'encapsulation.
- Les droits d'accès sont accordés aux fonctions membres ou aux fonctions globales.
- L'unité de protection est au niveau de la classe : ainsi tous les objets de la classe bénéficient de la même protection.

- Il y a 3 catégories de protection :
  - un membre **public** est accessible à toute fonction,
  - un membre **protected** n'est accessible qu'aux fonctions membres de la classe de base ou des classes dérivées ou aux fonctions amies.
  - un membre **private** n'est accessible qu'aux fonctions membres de la classe ou aux fonctions amies.

- Il y a 3 formes de dérivation :
  - **« public » :** le statut des membres hérités ne change pas,
  - **« protected » :** le statut des membres « public » hérités de classe de base change de catégorie et devient « protected »,
  - **« private » :** le statut des membres « public » et « protected » hérités de la classe de base change de catégorie et devient « private ». L'héritage est dit verrouillé à ce niveau, car tous les membres de la classe dérivée sont « private » et donc ne peuvent plus être hérités !

- Si la classe dérivée hérite publiquement de la classe de base,
  - les membres de la classe dérivée auront accès aux membres publics (champs et méthodes) de la classe de base,

  - par contre ils n'auront pas accès aux membres privés de la classe de base (à cause du 1er paradigme de la programmation orientée objet : l'encapsulation).

5. Constructeurs et destructeurs

- Pour construire une classe dérivée, il faut construire d'abord sa classe de base;
- Le constructeur de la classe de base est donc appelé **avant** le constructeur de la classe dérivée.
- De façon symétrique, le destructeur de la classe de base est appelé **après** le destructeur de la classe dérivée.
- Dans le cas de l'héritage multiple, l'ordre d'appel des constructeurs des classes de base doit être le même que lors de la définition de l'héritage.

```
/* définition du constructeur de la classe vehicule */
vehicule::vehicule(double v, int n){
     // etc.
}
/* définition du constructeur de la classe avion */
avion::avion(int m,double v,int p):vehicule(v,p){
     // etc.
}
/* définition du constructeur de la classe bateau */
bateau::bateau(double t,double v,int p):vehicule(v,p){
     // etc.
}
/* définition du constructeur de la classe hydravion */
hydravion::hydravion(int m,double v,int p,
                double t):avion(m,v,p),bateau(v,p,t){// etc.}
```

- Si la classe de base a un constructeur autre que celui par défaut, la classe dérivée doit nécessairement avoir un constructeur, sinon il est impossible de créer un objet.
- Si dans l'appel du constructeur de la classe dérivée, le nom du constructeur de la classe de base n'est pas mentionné explicitement, le constructeur par défaut de la classe de base sera pris en considération.
- Si la classe de base ne possède pas ce constructeur, il va y avoir une erreur de compilation.

6. Constructeur de recopie

- On doit recopier les champs de la classe dérivée et ceux de la classe de base.

- Deux cas peuvent se présenter:

1. La classe dérivée n'a pas de constructeur de recopie:
   - Les appels des constructeurs se feront comme suit:
     - le constructeur de recopie par défaut de la classe dérivée,
     - le constructeur de recopie explicite ou par défaut de la classe base.

2. La classe dérivée a un constructeur de recopie:
   - Les appels des constructeurs se feront comme suit:
     - le compilateur appelle ce constructeur de recopie. C'est à ce constructeur de recopie d'appeler celui de la classe de base (s'il veut, et habituellement on veut qu'il le fasse). Si l'on ne fait pas cet appel et si la classe de base n'a pas de constructeur de recopie explicite, alors c'est le constructeur par défaut qui est appelé. Si la classe de base n'en possède pas, c'est le constructeur avec arguments par défaut qui est appelé et, s'il n'en existe pas un, il y aura erreur de compilation.

7. Opérateur d'affectation (« = »)

- Deux cas peuvent se présenter:

1. La classe dérivée n'a pas surdéfini l'opérateur d'affectation.

   - Dans ce cas, le compilateur appelle:
     - L'opérateur d'affectation de la classe de base (par défaut ou surdéfini).
   - Sinon:
     - L'opérateur d'affectation par défaut de la classe dérivée.

2. La classe dérivée a surdéfini l'opérateur d'affectation.

   - Dans ce cas, le compilateur appelle seulement cet opérateur. Celui-ci va appeler l'opérateur d'affectation de la classe de base s'il le veut (habituellement, oui!).

8. Redéfinition des fonctions membres

- La **surdéfinition** consiste à définir dans un même programme (ou classe) deux ou plusieurs fonctions portant le même nom, même si le nombre et le type de paramètres (arguments) sont différents. Le type de retour n'est pas pris en considération.

- L'opération de **redéfinition** (surcharge) d'une fonction consiste à définir une seconde fonction, copie conforme de la première, réalisant de ce fait des tâches localisées.
- La redéfinition n'est pas permise au niveau d'un programme ou d'une classe. En effet, on ne peut pas définir deux fois la même chose dans un segment de programme ! La redéfinition est possible uniquement dans un style de programmation incrémentielle comme l'héritage.
- L'opération de redéfinition (ou surcharge) d'une fonction de la classe de base consiste à définir dans la classe dérivée une fonction portant le même nom que la fonction de la classe de base et qui aura un rôle similaire (de préférence), mais il sera limité uniquement au niveau de la classe dérivée.

```
/* définition de la méthode affiche -1- dans vehicule */
void vehicule::affiche();

/* surdéfinition de la méthode affiche -1- dans vehicule */
void vehicule::affiche(int);

/* redéfinition de la méthode affiche -1-
    (hyp. déclarée « public ») de vehicule
    au niveau de la classe avion */
void avion::affiche();
```


- La nouvelle définition (au niveau de la classe dérivée) va cacher l'ancienne définition (au niveau de la classe de base).
- On peut toute fois faire appel à la fonction de base dans la fonction dérivée en utilisant l'opérateur de résolution de portée (::) comme suit :

```
/* définition de la méthode affiche -1- dans vehicule */
void vehicule::affiche();

/* redéfinition de la méthode affiche -1- (hyp. publique)
        de vehicule au niveau de la classe avion */
void avion::affiche(){
    vehicule::affiche(); // appel de la méthode affiche -1-
}
```

9. Compatibilité entre objets d'une classe de base et objets d'une classe dérivée

- Est-ce qu'il est permis de convertir le type d'une instance de la classe dérivée vers le type d'une instance de la classe de base ? Et l'inverse la conversion de la classe de base vers une classe dérivée est-elle permise ?

- Est-ce qu'il est permis de convertir un pointeur sur une instance de la classe dérivée vers pointeur sur une instance de la classe de base ? Et l'inverse la conversion de la classe de base vers une classe dérivée est-elle permise ?

- Prenons l'exemple qui suit, et examinons les 4 cas de figure :

```
vehicule v(……); // constructeur de la classe de base
avion a(……); // constructeur de la classe dérivée

vehicule* ptrv; // pointeur du type classe de base
avion* ptra; // pointeur du type classe dérivée

ptrv = &v;
ptra = &a;
```

- Conversion d'une classe dérivée vers sa classe de base.
  - Elle est permise.
  - On part du principe qu'une classe dérivée est forcément « une classe de base ». Un « avion » est forcément un « vehicule ».

  - Le compilateur laisse tomber les membres excédentaires (provenant de la classe dérivée).
  - Le compilateur provoque une erreur si l'on tente d'accéder à ces membres excédentaires (par exemple « altitude ») à partir de la classe de base.

```
v = a; // Ok !
cout << v.vitesse << end;   // ok.
cout << v.altitude << endl; // erreur, car véhicule n'a pas
                               d'informations sur le nombre
                               de moteurs.
```

- Conversion d'une classe de base vers une classe dérivée.
  - Elle est interdite.
  - Une classe de base peut-être un tout-venant !
  - Un « vehicule » n'est pas forcément une « voiture ». Il peut-être un « bateau » ou une « moto » ou tout autre véhicule.
  - Un « vehicule » n'a pas de connaissances à priori des membres de « voiture », « bateau » ou « moto ». Que faire alors des membres manquants ?

```
a = v; // Erreur !
```

- Conversion de pointeurs de (classe dérivée)* à (classe de base)*.
  - La conversion d'un pointeur du type classe dérivée vers un pointeur du type classe de base est permise.

- Cependant, idem au cas -1-, l'accès aux membres excédentaires provoque une erreur à la compilation.

```
ptrv = ptra // ok,
cout << ptrv->vitesse << endl; // ok
cout << ptrv->altitude << endl; // erreur
```

- Conversion de pointeurs de (classe de base)* à (classe dérivée)*
  - La conversion d'un pointeur du type classe de base vers un pointeur du type classe dérivée est permise sous la condition d'utiliser un « cast » explicite.
  - Même, si l'accès aux membres excédentaires est permis, il ne faut pas oublier que ses membres n'ont pas été initialisés et donc, ne devraient contenir aucune valeur ou bien des valeurs erronées (les variables sont associées à des cases mémoires qui n'ont pas été réinitialisées).

```
ptra = ptrv // erreur
ptra = (avion *) ptrv // ok

cout << ptra.altitude << endl; // Va afficher n'importe quelle valeur !
```


10. Fonctions et classes virtuelles

- Ligature statique => le choix de la fonction membre dépend du type statique par exemple de l'objet receveur.
  - Le typage statique est le type par défaut en C++.

- Ligature dynamique => le choix de la fonction dépend du type dynamique.
  - Polymorphisme : le même nom de fonctions, mais plusieurs implémentations.
  - De ce fait, pour un même appel, on peut avoir des résultats différents.
  - La fonction sélectionnée correspond à l'objet qui a fait réellement l'appel.
  - L'association de l'appel à la partie du code à exécuter est différée au moment de l'exécution du programme.
  - Elle attend donc l'exécution pour définir le type.
  - L'opération est plus coûteuse.
  - Le mot clé « **`virtual`** » permet de masquer le typage statique et forcer la ligature dynamique.

- Peuvent être `virtuels`:
  - Fonctions membres non statiques,
  - Destructeurs.

- Ne peuvent pas être `virtuels`:
  - Champs membres,
  - Constructeurs.

```
class X {
public:
     void f() { cout << "x:f\n";}
};
class Y:public X {
public:
     void f() {cout << "y:f\n";}
};
X a;
Y b;
X* ptr = &a;
ptr->f();
ptr = &b;
ptr->f(); // comment faire appel à la bonne fonction « f » ?
```

- Nous allons déclarer la fonction « f » comme étant une fonction virtuelle.
- Ainsi le choix de la bonne fonction se fera au moment de l'exécution (ligature dynamique) et non pas au moment de la compilation (ligature statique).

```
class X {
public:
     virtual void f() { cout << "x:f\n";}
};
```


11. Ambiguïtés liées à l'utilisation de l'héritage multiple

- L'héritage multiple est peu utilisé en pratique à cause des problèmes d'ambiguïtés.
- Ces ambiguïtés sont de deux natures :
    - Dans le premier cas de figure, le champ en double « vitesse » provient par héritage de deux classes différentes « avion » et « bateau ».
    - Dans le deuxième cas de figure, le champ en double « vitesse » provient par héritage de la même classe « vehicule ».

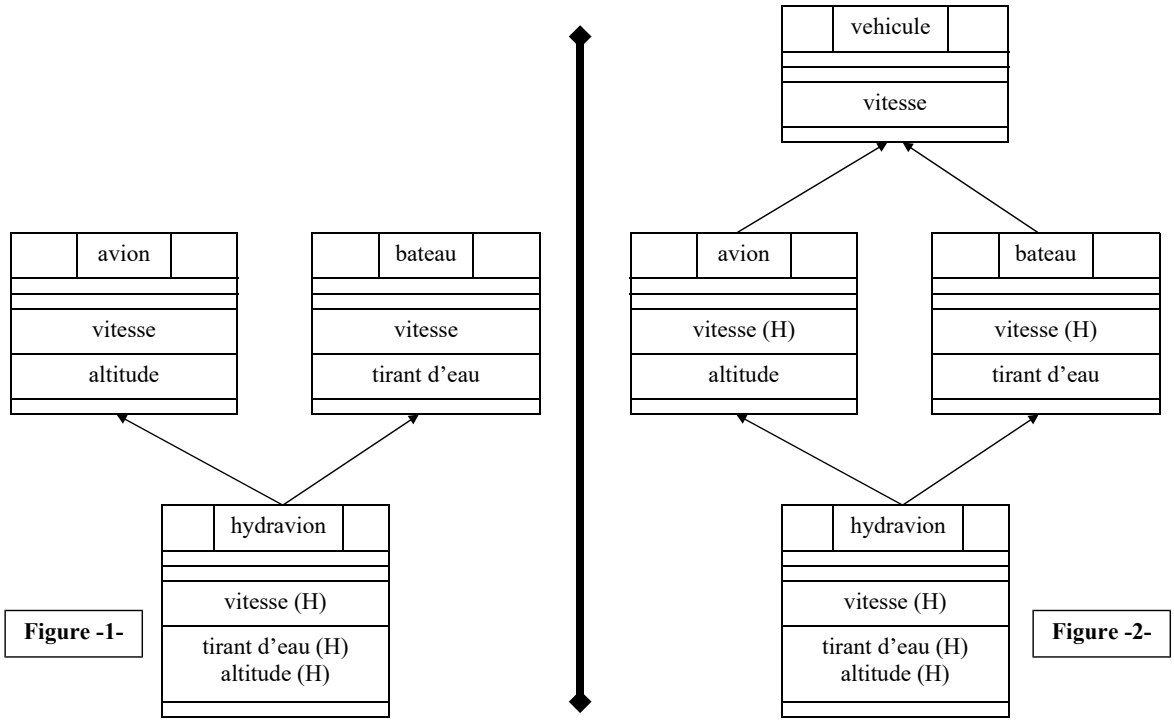


Figure -1-

Figure -2-

- Dans les deux cas de figure, lequel des membres « vitesse », la classe « hydravion » va-t-elle récupérer ?

12. Éliminer les ambiguïtés

- Dans le premier cas de figure, l'ambiguïté est levée grâce à l'utilisation de l'opérateur de résolution de portée (**::**) :

```
/* définition d'une instance de la classe hydravion */
hydravion h(……);

// la variable vitesse est déclarée « protected »
// pour qu'elle soit accessible par héritage.
cout << h.vitesse << endl ; // génère une erreur, ambiguïté !

// on précise ici que « vitesse » vient de « avion ».
cout << h.avion::vitesse << endl ;

// on précise ici que « vitesse » vient de « bateau ».
cout << h.bateau::vitesse << endl ;
```

- L'écriture est un peu lourde, mais il n'y a pas d'autres solutions !

- Dans le deuxième cas de figure, l'ambiguïté est levée en déclarant la classe de base qui contient l'élément générant l'ambiguïté, comme étant une classe virtuelle.

```
/* définition de la classe « vehicule » */
class vehicule {……};

/* définition de la classe « avion » */
class avion:public virtual vehicule {……};

/* définition de la classe « bateau » */
class bateau:public virtual vehicule {……};

/* définition de la classe « hydravion » */
class hydravion:public avion,public bateau {……};
```

  - Dans les descendants des classes dérivées (ici dans les descendants des classes « avion » et « bateau »), les éléments ne seront créés qu'une seule fois.
  - Avant donc que le compilateur ne définisse les champs membres d'une classe, il regarde d'abord si ces champs ont été déjà définis. Si c'est le cas, il ne fera rien.
  - Par opposition à une fonction virtuelle dont l'objectif permet de fixer la fonction à utiliser au moment de l'exécution du programme, la dérivation virtuelle n'entre en considération que dans le cadre d'une compilation statique. Elle ne sert donc qu'à lever les ambiguïtés au moment de la compilation.
  - Ne pas confondre donc « fonction virtuelle » et « classe virtuelle ».

13. Classes virtuelles et appel des constructeurs

- Dans un cadre général, l'appel des constructeurs se fait dans l'ordre suivant:

  - Le constructeur de la classe de base dans l'ordre d'appel.
  - Le constructeur de la classe dérivée.

```
class hydravion:public avion, public bateau {};

      1- Constructeur avion
      2- Constructeur bateau
      3- Constructeur hydravion
```

- Dans le cas où nous sommes en présence d'une classe virtuelle, l'appel des constructeurs se fait dans l'ordre suivant:

  - Constructeur de la classe virtuelle avant tout le monde y compris ses propres ascendants.
  - Ordre précédent (cadre général).

14. Identification dynamique du type

14.1. Généralités

- L'identification dynamique du type est très utile si le type de l'objet est inconnu, suite à un héritage. Par exemple, le cas d'un pointeur déclaré comme ayant le type de la classe de base alors qu'il désigne réellement une instance de la classe dérivée.

- L'identification de type n'est possible qu'en programmation polymorphique.

- Une classe polymorphique est une classe disposant de méthodes virtuelles.

- Les objets polymorphiques gardent des informations sur leur type dynamique.

- Ces informations permettent d'appeler au moment de l'exécution du programme la méthode (virtuelle) de la véritable classe de l'objet.

- Ces informations autrement utilisées permettent à un programmeur de connaître le type effectif (utilisé) de l'objet à l'exécution et de tester, si une conversion de type est permise.

- Cette identification porte le nom de « Run Time Type Information » ou « RTTI ».

- Le mécanisme d'identification se compose de 3 composantes : « dynamic_cast », « typeid » et « type_info ».

14.2. L'opérateur « dynamic_cast »

- La syntaxe de l'opérateur est :

```
dynamic_cast<Type *>(ptr)
```

- De manière générale, l'opérateur « dynamic_cast » convertit le pointeur « ptr » vers un pointeur « Type * », si l'objet pointé (i.e. *ptr) est du type « Type » ou dérivé de manière directe ou indirecte du type « Type ». Dans le cas contraire, l'expression retourne un pointeur et la valeur de ce dernier est nulle.
- Cette manière de procéder permet de vérifier la validité de la conversion (de s'assurer qu'elle ait eu lieu de façon sûre).

- Prenons l'exemple suivant :

  - Il y a au moins une méthode virtuelle : le destructeur de la classe A.
  - On peut donc se servir des différentes composantes de la « RTTI ».

```
class A {
public:
    virtual ~A(){}
} ;
class B : public A {} ;
class C : public B {} ;
```

- Les instructions (1 à 3) ci-dessous sont-elles correctes ? Seraient-elles sans danger ?

```
A* pa = new A;
A* pb = new B;
A* pc = new C;

C* p1 = (C *) pc;        // #1
C* p2 = (C *) pa;        // #2
B* p3 = (C *) pc;        // #3
```

- Dans les 3 instructions, « pc » et « pa » sont du type « A* » alors que « p1 » et « p2 » sont du type « C* » et « p3 » du type « B* ».
- Pour qu'elles soient correctes, il faudra que l'objet pointé soit du même type, ou un type dérivé de manière directe ou indirecte.
- Dans les 3 instructions, il s'agit d'une conversion de (classe de base)* à (classe dérivée)*. Pour que ces conversions soient possibles, c.-à-d. correctes, côté compilateur, il faut utiliser obligatoirement un « cast » explicite (voir paragraphe -9- ci-dessus). C'est ainsi que nous avons utilisé un « cast » vers « C* ».
- Le « cast » de l'instruction « #1 » est sans danger, car il a permis à un pointeur du type « C » de pointer une instance de « C ». Le « cast » de l'instruction « #2 » quant à lui, est dangereux parce qu'il a permis d'affecter l'adresse d'une instance ayant le type de base (ici « A ») à un pointeur sur une classe dérivée (ici « C »). La classe « C » peut avoir des membres en plus que la classe « A » qui n'en détient pas ! Pour l'instruction « #3 », son « cast » est sans danger. En effet, il est permis d'affecter l'adresse d'un objet d'une classe dérivée (ici « C ») à un pointeur sur une classe de base (ici « B »).

- Utiliser les fonctions virtuelles quand c'est possible et l'opérateur « dynamic_cast » seulement quand c'est nécessaire :

```
class A {
    int val;
public:
    A(int a=0);
    virtual void affiche() const;
    int getValeur() const;
};
class B : public A {
public:
    B(int a=0);
    void affiche() const ;
    virtual void getData() const ;
};
class C : public B {
    char c;
public:
    C(char x='a', int y=0) ;
    void affiche() const ;
    void getData() const ;
};
```

```
A* tab[3] =
   {new A(55),new B(77),new C('D',99)};

B* ptrb;

for (int i = 0; i < 3; i++){

   tab[i]->affiche();

   if( ptrb=dynamic_cast<B *>(tab[i]))
      ptrb->getData();
}
```

sortie :

```
Affiche -A-
Affiche -B-
La valeur est: 77
Affiche -C-
Le caractere est: D ; et la valeur est:
99
```

- On remarque bien que la méthode « getData » n'a été invoquée que deux fois avec le bon appel (la fonction étant virtuelle), pour tab[1] et tab[2] qui contiennent des instances de « B » et « C ».
- Quant à la méthode « affiche », elle n'a été appelée que trois fois et, à chaque fois avec le bon appel (là aussi la méthode est virtuelle).

14.3. L'opérateur « typeid »

- Cet opérateur permet de comparer le type de deux objets et dire, s'il est identique ou différent.
- Il renvoie une instance constante de la classe « type_info ». Cette instance ne peut pas être copiée (l'opérateur d'affectation et le constructeur de recopie ont été déclarés « private » dans la classe « type_info »).

- La méthode « before » permet de déterminer si une classe est avant une autre dans la hiérarchie.
- La classe « type_info » a redéfini les opérateurs logiques égalité (« == ») et différence (« != »).
- La méthode « name » retourne le nom de la classe.

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

- L'implémentation de « type_info » est dépendante du compilateur.


```
for (int i = 0; i < 3; i++){
    cout << "On traite maintenant le type "
         << typeid(*tab[i]).name() << ".\n";

    tab[i]->affiche();

    if( ptrb = dynamic_cast<B *>(tab[i])) ptrb->getData();

    if (typeid(C) == typeid(*tab[i]))
         cout << "C'est bien la classe C.\n";

}
```

- L'affichage en sortie, dépendant du compilateur utilisé (ici gcc), est comme suit :

```
On traite maintenant le type 1A.
Affiche -A-
On traite maintenant le type 1B.
Affiche -B-
La valeur est: 77
On traite maintenant le type 1C.
Affiche -C-
Le caractere est: D ; et la valeur est: 99
C'est bien la classe C.
```

14.4. « RTTI » oui, mais …

- Nous allons reprendre le programme du paragraphe (14-1) et le réécrire en n'utilisant que l'opérateur « typeid » :

```
A* tab[3] = {new A(55),new B(77),new C('D',99)};
B* ptrb;
C* ptrc;
for (int i = 0; i < 3; i++){
    if (typeid(C) == typeid(*tab[i])){
       ptrc = (C *) tab[i];
       ptrc->affiche();
       ptrc->getData();
    }else if (typeid(B) == typeid(*tab[i])){
       ptrb = (B *) tab[i];
       ptrb->affiche();
       ptrb->getData();
    }else
       tab[i]->affiche(); // ça ne peut-être que A
}
```

- C'est clair : nous venons de compliquer le programme pour rien !
- En outre, le fait de réfléchir ainsi c'est mal concevoir son programme pour une raison toute simple. Que faire si vous aviez affaire à de nombreux types ? Modifier le segment de programme pour inclure une « infinité » de « if/else » ? Impensable !
- Conclusion : Utiliser l'identification dynamique de type « RTTI » à bon escient !

15. Classes abstraites

15.1. Intérêt

- Placer dans une classe abstraite toutes les fonctionnalités que nous souhaitons disposer lors de la création des descendants.
- Une classe abstraite sert comme classe de base pour une dérivation.
- Une classe abstraite n'existe que pour être héritée.

15.2. Définition

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

```
virtual void affiche() = 0;
```

- C'est donc une méthode virtuelle qui est égale à zéro.
- Sa signature et le type de la valeur de retour sont fournis dans la classe et rien d'autre (pas le corps de la méthode c.-à-d. la définition).

- Une classe abstraite peut contenir des méthodes et des champs (ils peuvent être hérités) et une ou plusieurs fonctions virtuelles pures.

```
class X {

    // affiche est une fonction virtuelle pure, car = 0.

    virtual void affiche() = 0;

};
```

- Une classe abstraite ne permet pas l'instanciation des objets.

```
int main () {
    X a; // Erreur
    return 0;
}
```


- Les classes qui héritent d'une classe abstraite doivent obligatoirement définir la ou les fonctions virtuelles pures.

```
class Y:public X {

    // même si le mot clé virtual ne précède pas
    // le nom de la fonction affiche, elle reste
    // quand même virtuelle, car dans la classe
    // de base, elle a été déclarée ainsi.
    // donc nous n'avons pas besoin de le préciser
    // une seconde une fois.

    void affiche() {
       cout << "Y:f\n";
    }
};
```