AP6
Polymorphisme

Petit Complément utile
pour l'interpréteur…

IUT 2 Grenoble
Université Pierre-Mendès-France
Informatique

1. Identification de type à l'exécution

- La norme ANSI stipule que C++ doit offrir un mécanisme permettant d'identifier, lors de l'exécution, le type d'une variable, d'une expression ou d'un objet
- En particulier, il est possible de connaître le véritable type d'un objet désigné par un pointeur ou une référence
- L'opérateur qui permet cela est :
  `typeid`
- Cet opérateur renvoie un résultat de classe :
  `type_info`
- La classe **`type_info`** fournit une méthode **`name()`** qui renvoie le type sous la forme d'une chaîne de caractère (**`char*`**)
- Les opérateurs **`==`** et **`!=`** permettent de comparer deux objets de la classe **`type_info`**

1. Identification de type à l'exécution

```
#include <iostream>
#include <typeinfo>      // pour typeid
using namespace std ;

class point
{ public :
     virtual void affiche ()
      { } // indispensable pour qu'il y ait polymorphisme !!
} ;

class pointCol : public point
{ public :
     void affiche ()
      { }           // ici vide
} ;

main()
{ point p ; pointCol pc ;
  point * adp ;
  adp = &p ;
  cout << "type de  adp : " << typeid (adp).name()  << "\n" ;
  cout << "type de *adp : " << typeid (*adp).name() << "\n"  ;
  adp = &pc ;
  cout << "type de adp  : " << typeid (adp).name()  << "\n" ;
  cout << "type de *adp : " << typeid (*adp).name() << "\n"  ;
}
```

trace

```
type de  adp : point *
type de *adp : point

type de  adp : point *
type de *adp : pointCol
```

1. Comparaison de types à l'exécution

```
// mêmes déclarations que dans l'exemple précédent

main()
{ point p1, p2 ;
  pointCol pc ;
  point * adp1, * adp2 ;
  adp1 = &p1 ; adp2 = &p2 ;
  cout << "En A : les objets pointes par adp1 et adp2 sont de " ;
  if (typeid(*adp1) == typeid (*adp2)) cout << "meme type\n" ;
                                       else cout << "types differents\n" ;
  adp1 = &p1 ; adp2 = &pc ;
  cout << "En B : les objets pointes par adp1 et adp2 sont de " ;
  if (typeid(*adp1) == typeid (*adp2)) cout << "meme type\n" ;
                                       else cout << "types differents\n" ;
}
```

trace

```
En A : les objets pointes par adp1 et adp2 sont de meme type
En B : les objets pointes par adp1 et adp2 sont de types differents
```

1. Comparaison de types à l'exécution

```
// mêmes déclarations que dans l'exemple précédent

void fct (point & a, point & b)
{ if (typeid(a) == typeid(b))
              cout << "reference a des objets de meme type \n" ;
        else cout << "reference a des objets de types differents \n" ;
}

main()
{ point p ;
  pointCol pc1, pc2 ;
  cout << "Appel A : " ; fct (pc1, pc2) ;
  cout << "Appel B : " ; fct (p, pc1) ;
}
```

trace

```
Appel A : reference a des objets de meme type
Appel B : reference a des objets de types differents
```

2. Les Cast dynamiques

- Dans une situation de poymorphisme, étant donné un objet désigné par un pointeur :
  - On sait agir sur cet objet en fonction de son type,
  - On peut connaître son type exact au moment de l'exécution…
- … mais le type du pointeur utilisé reste toujours celui défini lors de la compilation !

```
…
main()
{ point p ; pointCol pc ;
  point * adp ;
  adp = &pc ;
  cout << "type de adp  : " << typeid (adp).name()  << "\n" ;
  cout << "type de *adp : " << typeid (*adp).name() << "\n"  ;
}
```

trace

```
type de  adp : point *
type de *adp : pointCol
```

2. Les Cast dynamiques

- Dans l'exemple précédent, si l'on sait que adp, qui est de type **`point *`**, pointe sur un objet de type **`pointCol`**, on peut souhaiter convertir sa valeur en un pointeur de type **`pointCol*`**
- Ceci peut-être réalisé à l'aide de l'opérateur de cast dynamique :

```
pointCol * adpc = dynamic_cast <pointCol *> (adp);
```

- La conversion aboutit si l'objet pointé est d'un type identique ou d'un type descendant du type d'arrivée demandé
- Lorsque l'opérateur n'aboutit pas :
  - il renvoie NULL s'il s'agit d'une conversion de pointeur
  - il déclence une exception bad_cast s'il s'agit d'une conversion de référence.

2. Les Cast dynamiques

```
// Situation de polymorphisme : C dérive de B qui dérive de A
int main()
{ A a ; B b ; C c ;
  A * ada, * ada1 ;
  B * adb, * adb1 ;
  C * adc ;
  ada = &a ;     // ada de type A* pointe sur un A ;
                 //  sa conversion dynamique en B* ne marche pas
  adb =  dynamic_cast <B *> (ada) ; cout << "dc <B*>(ada) " << adb << "\n" ;
  ada = &b ;     // ada de type A* pointe sur un B ;
                 //  sa conversion dynamique en B* marche
  adb =  dynamic_cast <B *> (ada) ; cout << "dc <B*> ada " << adb << "\n" ;
                 // sa conversion dynamique en A* marche
  ada1 = dynamic_cast <A*> (ada) ; cout << "dc <A*> ada " << ada1 << "\n" ;
                 // mais sa conversion dynamique en C* ne marche pas
  adc =  dynamic_cast <C *> (ada) ; cout << "dc <C*> ada " << adc << "\n" ;
  adb = &b ;     // adb de type B* pointe sur un B
                 // sa conversion dynamique en A* marche
  ada1 = dynamic_cast <A *> (adb) ; cout << "dc <A*> adb " << ada1 << "\n" ;
                 // sa conversion dynamique en B* marche
  adb1 = dynamic_cast <B *> (adb) ; cout << "dc <A*> adb1 " << adb1 << "\n" ;
                 // mais sa conversion dynamique en C* ne marche pas
  adc =  dynamic_cast <C *> (adb) ; cout << "dc <C*> adb1 " << adc << "\n" ;
  return 0;
}
```

Pour l'interpréteur (1)

- On représenterait les différents types de valeurs ainsi :

```
#include <typeinfo>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

class Valeur {
public:
  virtual ~Valeur() {};
};

class ValeurEntiere : public Valeur {
public:
 ValeurEntiere(int val=0) {
  this->val=val;
 }
 int getValeur() { return val; }
private:
 int val;
};
```

```
class ValeurReelle : public Valeur {
public:
 ValeurReelle(float val=0.0) {this-
>val=val;}
 float getValeur() { return val; }
private:
 float val;
};

class ValeurChaine : public Valeur {
public:
 ValeurChaine(string val="") {
   this->val=val;
 }
 string getValeur() { return val; }
private:
 string val;
};
```

Pour l'interpréteur (2)

- Et on gèrerait le polymorphisme ainsi :

```
int main() {

  Valeur *v;

  v=new ValeurEntiere(0);
  if (typeid(*v)==typeid(ValeurEntiere)) {
    cout << ((ValeurEntiere*)v)->getValeur() << endl;
  }

  v=new ValeurReelle(100.0);
  if (typeid(*v)==typeid(ValeurReelle)) {
    cout << ((ValeurReelle*)v)->getValeur() << endl;
  }

  v=new ValeurChaine("salut");
  if (typeid(*v)==typeid(ValeurChaine)) {
    cout << ((ValeurChaine*)v)->getValeur() << endl;
  }

  return 0;
}
```