Programmation Objet : Concepts Avancés
Cours 2 : C++

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

10 octobre 2008

C + Simula 67 = C++ ?

« I designed and implemented C++ because I had some problems for which it was the right solution : I needed C-style access to hardware and Simula-style program organization »

Bjarne Stroustrup
Créateur du C++
(1982)

C + Simula 67 = C++ ?

« C makes it easy to shoot yourself in the foot ; C++ makes it harder, but when you do, it blows away your whole leg. »

Bjarne Stroustrup
Créateur du C++
(1982)

C + Simula 67 = C++ ?

Bjarne Stroustrup
Créateur du C++
(1982)

Domaines d'utilisation de C++

- C++ fournit les mécanismes d'abstraction de la programmation orientée objet tout en permettant une programmation de bas-niveau (héritée du C).
- C++ est très utilisé dans les domaines suivants :
  - Multimédia (jeux vidéos, le noyau de YouTube, ...).
  - Calcul intensif (Google, traitement d'image, ...).
  - Embarqué (NASA pour les robots Spirit et Opportunity, ...).
- C++ a 26 ans ! Il existait bien avant Java et de gros efforts ont été faits pour construire des bibliothèques (libres ou propriétaires) couvrant les besoins de l'industrie informatique.
- Aujourd'hui, l'industrie du logiciel migre vers des langages un peu plus modernes (C# ou Java) mais vous serez de toute évidence confrontés à C++.

Ce qui est donc important de savoir sur C++

Comment éviter cela ?

Retour sur la méthode

- Nous allons découvrir le langage C++ de la façon suivante :
  - En partant de nos connaissances en JAVA qui est aussi un langage de programmation objet, statiquement typé à base de classes.
  - En exhibant les différences avec JAVA.
  - Avec un esprit critique : en nous souvenant du transparent précédent.

Structure d'un programme

- Héritage de C :
  - Un programme est formé par la liaison de fichiers objets (extension .o).
  - Le programme `ld` est donc mis à contribution (`man ld`).
  - Un fichier objet contient des données et du code machine.
  - Chaque fonction ou donnée est référencée par un nom symbolique.
  - Chaque référence externe a aussi un nom.
  - Le programme `nm` permet d'inspecter ces composants (`man nm`).
  - L'étape de liaison sert à connecter les références externes à des objets définis.
  - C'est ce procédé qui implémente la compilation séparée en C et C++.
  - Le mécanisme de bibliothèques fonctionne aussi de façon analogue.
- Une avancée :
  - Les conflits de noms externes apparaissent rapidement dans de gros projets C.
  - On préfixe souvent les noms externes par le nom du module.
  - C++ propose un système d'espace de noms (*namespace*).

Espaces de noms

```
namespace my_namespace_1 {
    namespace my_subnamespace_1 {
       class A {} ;
       enum C { A, B } ;
       namespace my_subsubnamespace_3 { ... }
    }
    namespace my_subnamespace_2 { ... }
    namespace my_subnamespace_1 { /* Reprise de la définition du premier namespace */
    }
}
```

- Les espaces de nom sont hiérarchisés et définis modulairement.
- Les identificateurs peuvent être totalement qualifiés en donnant leur chemin dans cette hiérarchie. Un chemin est une succession d'identificateurs d'espace de nom séparés par des ::.
- On peut rajouter des espaces de noms implicitement pris en compte dans la recherche d'un identificateur : using namespace my_namespace ;

Espaces de noms

```
typedef t void;
namespace N1 {
   typedef bool t;
   namespace N2 { typedef int t; }
   namespace N3 { typedef char t; }
   using namespace N2;
   t x;
   }
}
```

Quel est le type de la variable 'x' ?

Espaces de noms

```
typedef t void;
namespace N1 {
   typedef bool t;
   namespace N2 { typedef int t; }
   namespace N3 { typedef char t; }
   using namespace N2;
   N3::t x;
   }
}
```

Quel est le type de la variable 'x' ?

Espaces de noms

```
typedef t void;
namespace N1 {
   typedef bool t;
   namespace N2 { typedef int t; }
   namespace N3 { typedef char t; }
   using namespace N2;
   N1::N3::t x;
   }
}
```

Quel est le type de la variable 'x' ?

Espaces de noms

```
typedef t void;
namespace N1 {
   typedef bool t;
   namespace N2 { typedef int t; }
   namespace N3 { typedef char t; }
   using namespace N2;
   ::t x;
   }
}
```

Quel est le type de la variable 'x' ?

Espaces de noms

Le mot-clé using namespace est à utiliser avec précaution . . .

De l'importance des dépendances

Lorsqu'une unité de compilation a été modifée, il faut recompiler les unités de compilation qui en dépendent.

- En C++, les solutions à ce problème sont :
  - MAKEFILEs (écrit à la main ou produit automatiquement) ;
  - Les environnements de développement intégrant le calcul des dépendances.

Définition d'une classe : syntaxe

```
class my_class {
   // Membres
};
```

Attention à ne pas oublier le point virgule " ;".

Définition d'une classe : encapsulation des membres

```
class my_class {
   // Liste des membres (par bloc)
   // Membres publics :
   public:
      my_class () { ... } // Constructeur (par défaut).
      my_class (const my_class& c) { ... } // Constructeur (par copie).
      my_class (int i) :
         _i(i),
         j(i + 1)
      { _k = 0; } // Constructeur (a partir d'un entier i, on initialise les attributs _i et _j).
      int j; // Attribut (modifiable).
      virtual int get_i () { return _i; } // Méthode publique (accesseur).
   // Membres privés :
   private:
      int _k; // Attribut privé (on utilisera la convention : préfixé par un '_')
   // Membres "protégés" :
   protected:
      int _i; // Attribut accessible par les classes filles.
};
```

Membres d'une classe : synthèse

- Une classe peut contenir :
  - des constructeurs ;
  - des destructeurs ;
  - des attributs ;
  - des méthodes ;
  - des définitions de types (énumération, classe, union, typedef, . . .).
- À chaque membre est associé un droit d'accès :
  - private : seulement pour les membres de la classe ;
  - protected : seulement pour les membres de la classe et de ses sous-classes ;
  - public : accessible à tous.
- Certaines méthodes peuvent être accessibles seulement en mode const.
- Les attributs et les méthodes peuvent être déclarées static, c'est-à-dire accessibles à toutes les instances de la classe.
- Une classe définit un espace de noms.

struct et class : deux constructions unifiées ?

- Une structure est une classe dont tous les attributs sont publics.
- Une structure peut donc contenir des méthodes.
- Certaines contorsions d'implémentation sont nécessaires pour assurer cette compatibilité arrière.

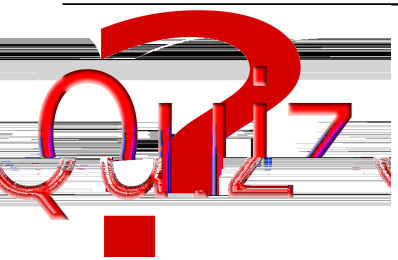
Classes mutuellement récursives

- Il est parfois utile de définir deux classes de façon mutuellement récursive.
- Par exemple, lorsqu'on a une relation contenant/contenu entre deux objets.
- On peut prédéclarer des classes en utilisant la syntaxe :

```
class A;
```

- L'identificateur A peut alors s'utiliser *tant que la définition de A n'est pas nécessaire au compilateur*.
- Ainsi :

```
class B { void foo (A* a) {} }; // Autorisé.
class B { void foo (A a) {} }; // Interdit.
```



Quelle information sur A est nécessaire dans cet exemple ?

Retour sur les attributs

- Un attribut est déclaré ainsi : T identificateur ;
- On fait référence à un attribut :
  1. à l'aide d'une notation pointée : my_instance.identificateur
  2. directement (à l'intérieur de la classe) : identificateur
  3. en précisant un des types de l'instance considérée : my_instance.T::x
- Un attribut de classe est déclaré de cette façon : static T identificateur ;
- On fait référence à un attribut de classe ainsi : T::identificateur

Allocation (sur la pile et sur le tas)

- Lorsqu'une variable est définie dans un scope local, sa durée de vie est bornée par ce scope. Elle est allouée sur la pile.
- On peut allouer une donnée sur le tas pour qu'elle persiste.

```
{
  A a;
  A* b = new A();
}
// a n'existe plus en ce point.
// b n'existe plus en ce point mais les données pointées par b existe encore.
```

- On déclare des variables ainsi : T identificateur(arg_1, ..., arg_n)
- En fonction des arguments, un constructeur particulier est utilisé.

```
// Objets sur la pile.
my_class o1; // Le constructeur par défaut est utilisé.
my_class o2 (o1); // Ici, c'est le constructeur par copie.
my_class o3 (1); // Et cette fois-ci, le constructeur à partir d'un entier.
// Objets alloués dans le tas.
my_class* o4 = new my_class (42); // Cet objet pourrait vivre en dehors du scope courant...
delete o4; // ... mais il est désalloué immédiatement.
```

Gare au *segmentation fault*

- Comme en C, le compilateur ne vous préviendra pas si vous véhiculez l'adresse d'un objet alloué sur la pile

- Ce type d'erreur est très difficile à détecter.
- On peut utiliser les outils `gdb` et `valgrind` pour assister la recherche de la fonction à incriminer.

Initialisation d'une instance de classe

Soit un attribut de type T.

- Si le type T est un type *builtin* (int, bool, char, ...) ou un pointeur, alors l'attribut n'est pas initialisé.
- Si le type T est une classe alors le constructeur par défaut est utilisé.
- Les attributs statiques doivent être initialisés.

Définition des constructeurs (première version)

- Un constructeur pour classe A se définit ainsi :

```
A (type_1 arg_1, ..., type_n arg_n) :
      attribute_1 (exp_1, ..., exp_m),
      ...
      attribute_k (exp_1, ..., exp_p)
{
   ...
}
```

- Lorsqu'on écrit « A a ; », c'est le constructeur sans argument (dit constructeur par défaut) qui est appelé.
- Lorsqu'on écrit « A a ; A b (a) ; A c = a ; », les instances b et c sont initialisées par le constructeur par copie dont la signature est :

  A (const A& other) ;

- Nous verrons ce que signifie "&" un peu plus tard.

Absence de constructeur

Que se passe-t-il si on ne définit pas de constructeurs ?

Absence de constructeur

Que se passe-t-il si on ne définit pas de constructeurs ?

Réponse en TD !

Égalité entre deux instances

- Que signifie « a $==$ b » ?
- Sur les types de C, la réponse est claire
  (à peu près, passons sous silence le type float)
- Sur les objets, à quoi s'attendre ?

Égalité entre deux instances : essayons !

```
#include <stdio.h>

class A {
public:
   int x;
   A (int y) : x (y) {}
};

int main (int argc, char** argv) {
   A x(10), y(10), z(42);
   printf ("%d, %d\n", (x == y), (x == z));
}
```

Réponse du compilateur :

```
no match for ‘operator==’ in ‘x == y’
```

Égalité entre deux instances : une solution . . .

```
#include <stdio.h>

class A {
public:
   int x;
   A (int y) : x (y) {}
   virtual bool operator== (const A& other) {
      return (other.x == x);
   }
};

int main (int argc, char** argv) {
   A x(10), y(10), z(42);
   printf ("%d, %d\n", (x == y), (x == z));
}
```

- Nous allons voir que cette solution n'est pas tout-à-fait satisfaisante. (souvenez-vous du problème des messages multiples . . .)
- Une explication en profondeur de ce problème :

  `http://artis.imag.fr/~Xavier.Decoret/resources/C++/operator==.html`

Retour sur les méthodes

- Pour définir une méthode :
  virtual T_ret my_meth (T_1 arg_1, ..., T_n arg_n) { ... }
- Pour définir une méthode pure :
  virtual T_ret my_meth (T_1 arg_1, ..., T_n arg_n) = 0 ;
- Pour définir une méthode à liaison statique :
  T_ret my_meth (T_1 arg_1, ..., T_n arg_n) { ... }
- Pour définir une méthode statique :
  static T_ret my_meth (T_1 arg_1, ..., T_n arg_n) { ... }
- On peut donner omettre un argument si une valeur par défaut à été précisée :

```
struct A { int my_meth (int x = 0) {} } ;
...
instance.my meth () ; // valide !
```

Méthode

- Pour définir une méthode :
  virtual T_ret my_meth (T_1 arg_1, ..., T_n arg_n) { ... }
- On appelle une méthode :
  1. d'une instance : my_instance.my_meth (exp_1, ..., exp_n)
  2. à l'intérieur de la classe : my_meth (exp_1, ..., exp_n)
     ou encore : this→my_meth (exp_1, ..., exp_n)
- La liaison est tardive : si la méthode est redéfinie dans une sous-classe, c'est le code présent dans la sous-classe qui est appelé.
- On appelle aussi cette propriété la récursion ouverte.
- Dans le corps d'une méthode, on a accès :
  - aux membres publics, privées ou protégés ;
  - aux membres hérités publics ou protégés.

Méthode pure

- Pour définir une méthode pure :
  virtual T_ret my_meth (T_1 arg_1, ..., T_n arg_n) = 0 ;
- On appelle une méthode :
  1. d'une instance : my_instance.my_meth (exp_1, ..., exp_n)
  2. à l'intérieur de la classe : my_meth (exp_1, ..., exp_n)
     ou encore : this→my_meth (exp_1, ..., exp_n)
- L'existence d'une méthode pure implique que la classe est abstraite
- On ne peut pas instancier une classe abstraite : le compilateur vous l'interdit !
  (et il a bien raison de le faire, non ?)

Méthode à liaison statique

- Pour définir une méthode à liaison statique :
  T_ret my_meth (T_1 arg_1, ..., T_n arg_n) { ... }
- On appelle une méthode :
  1. d'une instance : my_instance.my_meth (exp_1, ..., exp_n)
  2. à l'intérieur de la classe : my_meth (exp_1, ..., exp_n)
     ou encore : this→my_meth (exp_1, ..., exp_n)
  3. en spécifiant exactement la classe considérée :
     my_instance.T::my_meth (exp_1, ..., exp_n)
- La liaison statique s'apparente à l'appel de fonction classique : il n'y a pas de liaison tardive même si la méthode est redéfinie dans une sous-classe.
- Dans le corps d'une méthode, on a accès :
  - aux membres publics, privées ou protégés ;
  - aux membres hérités publics ou protégés.

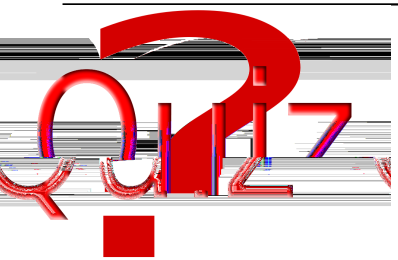
Méthode de classe

- Pour définir une méthode de classe :
  static T_ret my_meth (T_1 arg_1, ..., T_n arg_n) { ... }
- On appelle une méthode :
  1. d'une classe T : T::my_meth (exp_1, ..., exp_n)
  2. à l'intérieur de la classe : my_meth (exp_1, ..., exp_n)
- La liaison statique s'apparente à l'appel de fonction classique : il n'y a pas de liaison tardive même si la méthode est redéfinie dans une sous-classe.
- Dans le corps d'une méthode, on a accès :
  - aux membres de classe publics, privées ou protégés ;
  - aux membres de classe hérités publics ou protégés.
  - mais pas aux attributs d'instance !

Résolution des appels de méthodes

```
struct A {
   virtual void m1 () {};
   virtual void m2 () = 0;
   void m3 () {}
   virtual void m4 () { this→m3 (); }
   virtual void m5 () { this→m1 (); }
   virtual void m6 () { this→m3 (); }
};
struct B : public A {
   virtual void m1 () {};
   virtual void m2 () {};
   void m3 () {}
};
```

```
B* b = new B (); A* a = b;
b→m1 ();
b→m2 ();
b→m3 ();
b→m4 ();
b→m5 ();
b→m6 ();
b→A::m1 ();
b→A::m3 ();
b→A::m5 ();
a→m1 ();
a→m2 ();
a→m3 ();
a→m4 ();
a→m5 ();
a→m6 ();
a→A::m1 ();
a→A::m3 ();
a→A::m5 ();
```

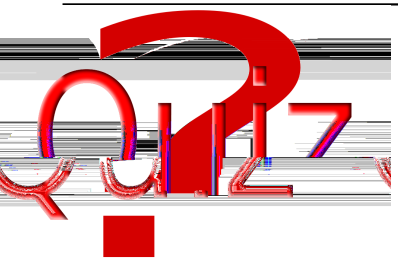


En chaque point d'appel, quelle méthode est exécutée ?

Résolution des appels de méthodes

```
struct A {
   virtual void m1 () {};
   virtual void m2 () = 0;
   void m3 () {}
   virtual void m4 () { this→m3 (); }
   virtual void m5 () { this→m1 (); }
   virtual void m6 () { this→m3 (); }
};
struct B : public A {
   virtual void m1 () {};
   virtual void m2 () {};
   void m3 () {}
};
```

```
B* b = new B (); A* a = b;
b→m1 (); // B : :m1
b→m2 (); // B : :m2
b→m3 (); // B : :m3
b→m4 (); // A : :m3
b→m5 (); // B : :m1
b→m6 (); // A : :m3
b→A : :m1 (); // A : :m1
b→A : :m3 (); // A : :m3
b→A : :m5 (); // B : :m1
a→m1 (); // B : :m1
a→m2 (); // B : :m2
a→m3 (); // A : :m3
a→m4 (); // A : :m3
a→m5 (); // B : :m1
a→m6 (); // A : :m3
a→A : :m1 (); // A : :m1
a→A : :m3 (); // A : :m3
a→A : :m5 (); // B : :m1
```



En chaque point d'appel, quelle méthode est exécutée ?

Surcharge des noms des méthodes et des fonctions

- Il est possible d'utiliser un même nom de méthode pour plusieurs méthodes ayant des signatures différentes.
- Ce mécanisme s'appelle la surcharge.

Surcharge des noms des méthodes et des fonctions

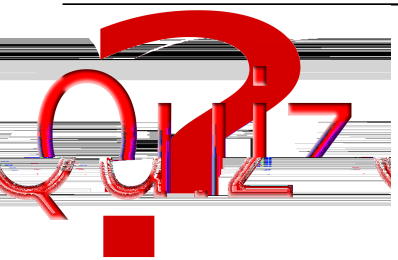
- Il est possible d'utiliser un même nom de méthode pour plusieurs méthodes ayant des signatures différentes.
- Ce mécanisme s'appelle la surcharge.

- Ce mécanisme est pratique mais il ne faut pas en abuser.
- La résolution des méthodes est déjà très complexe !
- Écrire des noms non ambigus, c'est du temps gagné en maintenance.

Résolution des appels de méthode (avec surcharge)

```
struct A {
   virtual void m1 (int x = 0) {};
   virtual void m2 () = 0;
   void m3 () {}
   virtual void m4 () { this→m3 (); }
   virtual void m5 () { this→m1 (); }
   virtual void m6 () { this→m3 (); }
};
struct B : public A {
   virtual void m1 () {};
   virtual void m2 () {};
   void m3 () {}
};
```

```
B* b = new B (); A* a = b;
b→m1 ();
b→m2 ();
b→m3 ();
b→m4 ();
b→m5 ();
b→m6 ();
b→A::m1 ();
b→A::m3 ();
b→A::m5 ();
a→m1 ();
a→m2 ();
a→m3 ();
a→m4 ();
a→m5 ();
a→m6 ();
a→A::m1 ();
a→A::m3 ();
a→A::m5 ();
```

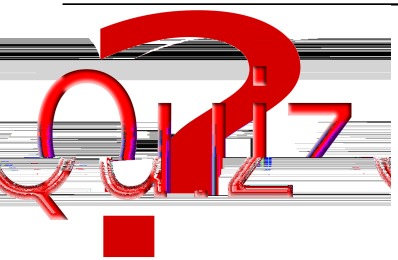


En chaque point d'appel, quelle méthode est exécutée ?

Résolution des appels de méthode (avec surcharge)

```
struct A {
   virtual void m1 (int x = 0) {};
   virtual void m2 () = 0;
   void m3 () {}
   virtual void m4 () { this→m3 (); }
   virtual void m5 () { this→m1 (); }
   virtual void m6 () { this→m3 (); }
};
struct B : public A {
   virtual void m1 () {};
   virtual void m2 () {};
   void m3 () {}
};
```

```
B* b = new B (); A* a = b;
b→m1 (); // B : :m1
b→m2 (); // B : :m2
b→m3 (); // B : :m3
b→m4 (); // A : :m3
b→m5 (); // A : :m1
b→m6 (); // A : :m3
b→A : :m1 (); // A : :m1
b→A : :m3 (); // A : :m3
b→A : :m5 (); // A : :m1
a→m1 (); // A : :m1
a→m2 (); // B : :m2
a→m3 (); // A : :m3
a→m4 (); // A : :m3
a→m5 (); // A : :m1
a→m6 (); // A : :m3
a→A : :m1 (); // A : :m1
a→A : :m3 (); // A : :m3
a→A : :m5 (); // A : :m1
```



En chaque point d'appel, quelle méthode est exécutée ?

Retour sur l'opérateur de comparaison

```
#include <stdio.h>
struct A {
   int x;
   virtual bool operator== (const A& a) { return a.x == x; }
};
struct B : A {
   int y;
   virtual bool operator== (const B& b) { return b.x == x ∧ b.y == y; }
};

bool will_explode (A* ab) {
   A* a = new A ();
   return (*ab == *a);
}

int main (int argc, char** argv) {
   B* b = new B ();
   printf ("%d\n", will_explode (b));
}
```

Souvenez-vous du cours dernier . . . Que se passe-t-il ?

Retour sur l'opérateur de comparaison

```
#include <stdio.h>
struct A {
   int x;
   A (int y) : x(y) {}
   virtual bool operator== (const A& a) { return a.x == x; }
};
struct B : A {
   int y;
   B (int z, int k) : A(z), y(k) {}
   virtual bool operator== (const B& b) { return b.x == x ∧ b.y == y; }
};

bool work_badly (A* a, A* ab) {
   return (*ab == *a);
}

int main (int argc, char** argv) {
   B* b1 = new B (1, 2);
   B* b2 = new B (1, 3);
   printf ("%d\n", work_badly (b1, b2));
}
```

Et maintenant ?

Retour sur l'opérateur de comparaison : solution

```
#include <stdio.h>
struct A {
  int x;
  A (int y) : x(y) {}
};
struct B : A {
  int y;
  B (int z, int k) : A(z), y(k) {}
};

bool operator== (const A& a1, const A& a2) {
  return (a1.x == a2.x);
}

bool operator== (const B& b1, const B& b2) {
  return (b1.x == b2.x) ∧ (b1.y == b2.y);
}

int main (int argc, char** argv) {
  B* b1 = new B (1, 2);
  B* b2 = new B (1, 3);
  printf ("%d\n", *b1 == *b2);
}
```

Retour sur les autres types de membres

- Une classe peut contenir :
  - ✓ des constructeurs ;
  - des destructeurs ;
  - ✓ des attributs ;
  - ✓ des méthodes ;
  - des définitions de types (énumération, classe, union, typedef, . . .).

Une classe dans une classe ?

```
class A {
public:
   class B {};
private:
   class C {};
};

int main (int argc, char** argv) {
   A::B b; // Valide
   A::C c; // Erreur !
}
```

- On peut contraindre certaines classes à n'apparaître que dans un contexte local donné.

Petite digression

- Dans le langage SCALA, on peut définir des types dépendants d'une instance particulière

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node) {
      if (connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    var res = new Node
    nodes = res :: nodes
    res
  }
}
```

```
object IllegalGraphTest extends
Application {
  val g: Graph = new Graph
  val n1: g.Node = g.newNode
  val n2: g.Node = g.newNode
  n1.connectTo(n2) // legal
  val h: Graph = new Graph
  val n3: h.Node = h.newNode
  n1.connectTo(n3) // illegal!
}
```

(source : `http://www.scala-lang.org/node/115`)

Modes const et non-const

- Une méthode de type const ne peut être utilisée que dans sur un objet dont le type est préfixé par const.

```
#include <cstdio>
class A {
public:
   void foo () const { printf ("I am in mode: const\n"); }
   void foo () { printf ("I am in mode: non-const\n"); }
   void bar () const { printf ("a non-const can do what const can do.\n"); }
   void baz () { printf ("but the converse is false!\n"); }
};
int main (int argc, char **argv) {
   const A a_rd = A ();
   A a;
   a_rd.foo ();
   a.foo ();
   a_rd.bar ();
   a.bar ();
   a_rd.baz (); // Ne compile pas !
   a.baz ();
}
```

Le mode const

- En mode const, le type des attributs est préfixé implicitement par const.
- On empêche ainsi certains partage de données.
- On renforce ainsi l'encapsulation.
- Malheuresement, nous avons vu dans le premier cours que cette protection peut être un leurre dans le cas d'un partage de pointeur.

Héritage simple

- La syntaxe de l'héritage simple est :

  class A : (public | private | protected) B {
  ...
  };

- Chaque mot-clé indique la façon dont on hérite de la classe mère :
  - public :
    - accès à tous les membres publics et protégés,
    - ces membres deviennent publics dans la classe fille.
  - protected :
    - accès à tous les membres publics et protégés,
    - les membres publics de la classe mère deviennent protégés dans la classe fille et les membres protégés deviennent privés.
  - private :
    - accès à tous les membres publics et protégés,
    - les membres publics et protégés de la classe mère deviennent privés dans la classe fille.

Des amis pour lesquels la porte est ouverte

- Il y a une autre façon d'avoir accès aux attributs privés d'une classe (et donc de passer outre l'encapsulation) : les classes et fonctions amies introduits par le mot-clé friend :

```
class my_class {
public:
   friend class my_friend_class;
   friend void foo (my_class x);
private:
   int _y;
};
```

- Dans la définition de la classe my_friend_class et le corps de la fonction foo, on peut accéder à l'attribut _y.

```
class my_friend_class {
   void meth (my_class a) { a._y = 43; }
};
void foo (my_class a) {
   a._y = 41;
}
```

Hériter de tout sauf . . .

Certains membres ne sont jamais hérités :

- les constructeurs et les destructeurs (que nous allons voir dans dans quelques transparents)
- l'opérateur d'affectation (que nous allons voir bientôt aussi)
- ses membres amis

Retour sur les constructeurs

- Il faut appeller explicitement le constructeur de la classe mère dans la classe fille à l'aide de cette syntaxe :

```
A (type_1 arg_1, ..., type_n arg_n) :
      mother_class (exp_1, ..., exp_l),
      attribute_1 (exp_1, ..., exp_m),
      ...
      attribute_k (exp_1, ..., exp_p)
{
   ...
}
```

- L'ordre d'initialisation est important.

Sous-typage

- La hiérarchie de classes induit une relation de sous-typage.
- Une instance de type B peut être vue comme une instance de type A si B hérite de A.
  - L'exécution de ce programme conduit à un échec :

```
#include <stdlib.h>
struct A { int x; }
struct B : A {};
void reset (A a) { a.x = 0; }
int main (int argc, char** argv) {
   B b;
   b.x = 42;
   reset (b);
   if (b.x == 0)
      exit (EXIT_SUCCESS);
   else
      exit (EXIT_FAILURE);
}
```

Pourquoi ?

Sous-typage sur des objets pointés

- Pour pouvoir utiliser la relation de sous-typage, il faut manipuler des pointeurs sur objet.

```
#include <stdlib.h>
struct A { int x; }
struct B : A {};
void reset (A* a) { a.x = 0; }
int main (int argc, char** argv) {
   B b;
   b.x = 42;
   reset (&b);
   if (b.x == 0)
      exit (EXIT_SUCCESS);
   else
      exit (EXIT_FAILURE);
}
```

- Sans passage par pointeur, le passage se fait par copie (héritage du C).

Coercion implicite

- On peut définir des opérations de coercion implicite pour pouvoir construire automatiquement un objet de T à partir d'un objet de type U même si U n'est pas un sous-type de T.

```
struct BoundedInteger {
   int bound;
   int value;
   operator int() { return value; }
};

int main (int argc, char** argv) {
   BoundedInteger x;
   int y = x;
}
```

Recouvrement du type

- À toute instance de classe est associée une information dynamique de type (RTTI, *Run Time Type Information*).
- Cette information sert à implémenter la liaison tardive.
- On peut donc aussi “retrouver le type perdu” d'un objet.

```
struct A { int x; }
struct B : A {};
B* indy (A* a) {
   B* potential_b = dynamic_cast<B*>(a);
   if (potential_b != NULL) return potential_b;
   else die_in_horrible_pain ();
}
```

- Ce mécanisme est analogue à l'opérateur instanceof de JAVA.

Héritage multiple

- C++ offre la possibilité d'hériter de plusieurs classes en même temps.
- La syntaxe générale de l'héritage est :

```
class A :
(public | private | protected | public virtual) B_1,
...
(public | private | protected | public virtual) B_n, {
  ...
};
```

- Quel est donc ce nouveau mot-clé public virtual ?

Le cas facile : deux classes mères disjointes

```
struct A { int x; };
struct B { int y; };
class C : public A, public B {};
```

- C est l'union des membres de A et de B.

Un cas un peu moins facile : des méthodes communes

```
struct A {
   int x;
   void foo () { x = 42; }
};
struct B {
   int y;
   void foo () { y = 42; }
};
class C : public A, public B {
   void bar () { this→foo (); }
};
```

La méthode C::bar () :

- ❒ appelle A::foo ()
- ❒ appelle B::foo ()
- ❒ ne fait rien du tout car ce programme ne compile pas !

Un cas un peu moins facile : des méthodes communes

```
struct A {
  int x;
  void foo () { x = 42; }
};
struct B {
  int y;
  void foo () { y = 42; }
};
class C : public A, public B {
  void bar () { this→foo (); }
};
```

- Ouf ! Le compilateur rejette ce programme !

```
multiple-inheritance.cc: In member function 'void C::bar()':
multiple-inheritance.cc:10: error: request for member 'foo' is ambiguous
multiple-inheritance.cc:7: error: candidates are: void B::foo()
multiple-inheritance.cc:3: error: void A::foo()
```

- Il faut choisir en écrivant par exemple :

```
void bar () { this→A::foo (); }
```

Un cas cauchemardesque : les héritages en losange

```
struct Alpha {
   int z;
   virtual void foo () = 0;
   virtual void bar () = 0;
};

struct A : public Alpha {
   virtual void foo () { this→bar (); }
};
struct B : public Alpha {
   virtual void bar () { z = 42; }
};
class C : public A, public B {
   void baz () { z = 43; }
};
```

Ce programme :

- ❒ est valide !
- ❒ est rejeté par le compilateur !

Un cas cauchemardesque : les héritages en losange

```
struct Alpha {
  int z;
  virtual void foo () = 0;
  virtual void bar () = 0;
};

struct A : public Alpha {
  virtual void foo () { this→bar (); }
};
struct B : public Alpha {
  virtual void bar () { z = 42; }
};
class C : public A, public B {
  void baz () { z = 43; }
};
```

- Ouf ! Le compilateur rejette ce programme !

```
multiple-inheritance-3.cc: In member function 'void C::baz()':
multiple-inheritance-3.cc:15: error: reference to 'z' is ambiguous
multiple-inheritance-3.cc:3: error: candidates are: int Alpha::z
multiple-inheritance-3.cc:3: error: int Alpha::z
multiple-inheritance-3.cc:15: error: reference to 'z' is ambiguous
multiple-inheritance-3.cc:3: error: candidates are: int Alpha::z
multiple-inheritance-3.cc:3: error: int Alpha::z
```

- C'est ici qu'intervient le mot-clé public virtual.

Un cas cauchemardesque : les héritages en losange

```
struct Alpha {
   int z;
   virtual void foo () = 0;
   virtual void bar () = 0;
};

struct A : public virtual Alpha {
   virtual void foo () { this→bar (); }
};
struct B : public virtual Alpha {
   virtual void bar () { z = 42; }
};
class C : public A, public B {
   void baz () { this→foo(); }
};
```

La méthode C::baz :

- appelle B::bar.
- rien, ce programme est rejeté par le compilateur !
- Luke Skywalker

Un cas cauchemardesque : les héritages en losange

```
struct Alpha {
   int z;
   virtual void foo () = 0;
   virtual void bar () = 0;
};

struct A : public virtual Alpha {
   virtual void foo () { this→bar (); }
};
struct B : public virtual Alpha {
   virtual void bar () { z = 42; }
};
class C : public A, public B {
   void baz () { this→foo(); }
};
```

- Ce programme est accepté et appelle B::bar
- La sémantique de public virtual est assez complexe ...

Comportement exceptionnel des objets

- Parfois, le type de retour d'une méthode ne suffit pas à capturer tous les comportements possibles de l'objet.
- En particulier, lorsqu'une erreur se produit, il n'est pas toujours évident de suivre la convention C/UNIX (un entier négatif ou un pointeur nul).
- C'est pourquoi C++ introduit le mécanismie des exceptions.
- Pour lancer une exception : throw valeur
- Par exemple : throw -1 ou encore throw (new A())
- Pour capturer une exception :

```
try {
} catch (int a) {
   // ...
} catch (A a) {
   // ...
} catch (...) { // une exception inconnue
   // ...
}
```

Déclarer les exceptions lancées par une méthode

```
struct A {
   // Cette méthode lance des exceptions
   void meth_1 () throw (...) {}
   // Cette méthode aussi.
   void meth_2 () {}
   // Cette méthode lance une exception qui est un entier ou un pointeur sur A.
   void meth_3 () throw (int, A*) {}
   // Cette méthode ne lance aucune exception.
   void meth_4 () throw () {}
};
```

Cas à prendre en compte

Que se passe-t-il lorsqu'on lance une exception dans un constructeur ?

L'opérateur delete

- En C comme en C++, il n'y a pas de ramasse-miette (*garbage collector*).
- Pour éviter les fuites de mémoire, il faut donc explicitement libérer les données allouées sur le tas qui ne servent plus.

Pourquoi ne pas utiliser `free` comme en C ?

Destructeur

- Pour respecter le principe d'encapsulation, on veut internaliser la destruction d'un objet et pouvoir la définir d'une façon différente dans chaque classe.
- C'est ici qu'on introduit les destructeurs dont la syntaxe est :

```
struct A {
   ~A () {
      // ...
   }
};
A* a = new A ();
delete a; // Appelle le destructeur de A.
```

Détruire un tableau

- Pour allouer un tableau de N éléments de type T : new T[N]
- Pour libérer un tableau a : delete[] a
- Le destructeur du type T est appelé pour chaque élément du tableau.

```
#include <stdio.h>
struct A {
  ~A () { printf ("I am dying !\n") ; }
} ;

int main (int argc, char** argv) {
  A * array = new A[3] ;
  delete [] array ;
}
```

```
/enseignement/poca/exemples/c++ % g++ -o foo delete-array.cc
/enseignement/poca/exemples/c++ % ./foo
I am dying !
I am dying !
I am dying !
```

Destructeur virtuel

```
struct A { ~A () {} };
struct B : A {
   int* very_big_data;
   // ...
   ~B () { delete[] very_big_data; }
};
int main (int argc, char** argv) {
   A* a = new B (); // Something very big is allocated
   delete a; // Hopefully, it is released now ... No ? No !
}
```

Comment résoudre ce problème ?

Destructeur virtuel

```
struct A { virtual ~A () {} };
struct B : A {
   int* very_big_data;
   // ...
   virtual ~B () { delete[] very_big_data; }
};
int main (int argc, char** argv) {
   A* a = new B (); // Something very big is allocated
   delete a; // Hopefully, it is released now ... No ? No !
}
```

Comment résoudre ce problème ?

Référence

- Une référence sur une valeur est l'adresse de cette valeur.
- Si la valeur est de type T alors une référence sur cette valeur a le type T& (ou U& si U est un sous-type de T).
- On manipule une référence comme la valeur elle-même (pas de déréférencement).
- L'objectif des références est de fournir des pointeurs non nuls.

Référence non nulle : une convention

- Encore une fois en C++, il s'agit d'une convention !
- En effet, on peut très bien écrire :

```
int& x = *((int*)NULL);
```

- ...sans être grondé par le compilateur !

L'opérateur d'affectation

- On peut surcharger l'opérateur d'affectation dans une classe A :

```
class A {
   public: A& operator= (const A& a) {
      // Par convention, on renvoie toujours une référence sur this.
      return *this;
   }
};
```

- Le type de retour sert à chainer les affectations :

```
x1 = x2 = x3;
```

À quoi peut bien servir cette énième surcharge ?

Le problème de l'auto-affectation

- Voici un cas d'utilisation de l'opérateur d'affectation :

```
struct A {
   int* my_big_data;
   A& operator= (const A& other) {
      // Copy of other.my_big_data into my_big_data
      // Release of my_big_data
      delete[] my_big_data;
   }
};
```

Et si on affecte une instance à elle-même ?

Le problème de l'auto-affectation

- Voici un cas d'utilisation de l'opérateur d'affectation :

```
struct A {
   int* my_big_data;
   A& operator= (const A& other) {
      if (this != &other) {
         // Copy of other.my_big_data into my_big_data
         // Release of my_big_data
         delete[] my_big_data;
      }
   }
};
```

Et si on affecte une instance à elle-même ?

Qui a le droit de vie ou de mort ?

- L'introduction des références permet de se donner la convention suivante :

  Détenir un pointeur sur un objet signifie qu'on est son propriétaire et qu'on doit donc le désallouer.

- En pratique, cette convention couvre une majorité de cas.
- La plupart des erreurs de fonctionnement des programmes écrits en C++ sont liées à ce problème.
- Nous étudierons donc cette question en profondeur en TP.

Qu'est-ce que la programmation générique ?

- Il s'agit d'écrire des programmes paramétrés par des types.
- De tels programmes sont dits polymorphes.
- Un programme polymorphe est par définition réutilisable.
- En O'Caml, le programme suivant est polymorphe :

  let rec map f = function [] → [] | x : : xs → f x : : map f xs

  puisque son type est : $(\alpha \rightarrow$ 'b) $\rightarrow \alpha$ list $\rightarrow$ 'b list
  (α et 'b sont des paramètres de types)
- En C++, ce polymorphisme est implémenté par les patrons de classes et de fonctions (*templates*).

Patrons de classe

```
template <class T, class U>
class pair {
   private:
      T _first;
      U _second;

   public:
   pair (const T& first, const U& second):
      _first (first),
      _second (second)
   {}
   const T& first () const { return _first; }
   const U& second () const { return _second; }
};

pair<int, int> my_pair;
pair<int, bool> my_pair_2;
pair<A, pair<B, C> > my_pair_3;
```

- Un patron de classes définit une famille de classe.
- On peut le voir littéralement comme du code à trous.
- Spécifier ces trous par des types, l'instanciation, revient à dupliquer la définition de ce patron.

Patrons de fonction

- De même, on peut définir des patrons de fonctions :

```
template <class T>
pair<T, T> transpose (const pair<T, T>& p) {
   return (pair (p.second (), p.first ()));
}
```

- On peut appliquer cette fonction sur les valeurs my_pair, my_pair_2 et my_pair_3.
- Encore une fois, le code de la fonction est dupliquée pour chacune des instanciations nécessaires du type T.
- En O'CAML, le code d'une fonction polymorphe est unique.

Standard Template Library

- C++ est accompagnée d'une bibliothèque standard formée :
  - des fichiers en-tête reprenant la bibliothèque C.
  - un module fournissant des chaînes de caractères (`string`).
  - un module fournissant des entrées/sorties (`iostream`).
  - la STL (Standard Template Library) comprenant des structures de données génériques (`map`, `set`, `hash_map`, `list` ...).

Spécialisation de patrons

- On peut spécialiser un patron de classe (ou une fonction) pour qu'il s'instancie d'une manière particulière pour un jeu de types données :

```
template <>
class pair<char, char> {
   private: int _data;
   public:
      pair (char first, char second):
         _data (((int)first) << 8 + second)
      { ... }
      char first() const { return _data >> 8; }
      char second() const { return _data & 0xff; }
};
```

- pair<char, char> est implémentée par cette classe tandis que les paires d'autres couples de types sont implémentées par la classe définie précédemment.

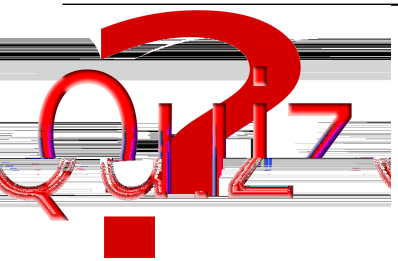
Spécialisation de patrons : vers la méta-programmation

- La spécialisation permet de construire des fonctions des types dans les types, évaluées par le compilateur.

```
template <class T>
struct traits {
  typedef std::list<T> set_type;
};

template <>
struct traits<char> {
  typedef std::vector<bool> set_type;
};

template <>
struct traits<int> {
  typedef std::set<int> set_type;
};
```



Mais à quoi cela peut-il servir ?

g++

- Nous utiliserons le compilateur libre de GNU, g++.
- Les options intéressantes pour vous sont :
  - `-g` : compiler avec les informations de déboguage.
  - `-Wall` : activé tous les avertissements possibles.
  - `-O2` : compilation optimisante.
- Il n'est jamais inutile de lire la documentation de votre compilateur.

Make

```
CXX=g++
CXX_FLAGS=-Wall -g
CPP_FLAGS=
LD_FLAGS=

OBJECTS=$(SOURCES:.cc=.o)

.PHONY: all clean

all: $(TARGET)

$(TARGET): $(OBJECTS)
	$(CXX) -o $(TARGET) $(CXX_FLAGS) $(CPP_FLAGS) $(LD_FLAGS) $(OBJECTS)

%.o: %.cc
	$(CXX) $(CPP_FLAGS) $(CXX_FLAGS) -c $<

clean:
	rm --force $(OBJECTS) $(TARGET) *~
```

Subversion

- SUBVERSION est un gestionnaire de version.
- Un tel outil sert à :
  - Sauvegarder l'historique d'un projet ;
  - Permettre le développement (concurrent) en équipe ;
- SUBVERSION est centralisé.
- Un répositoire vous sera fourni.
- Les deux commandes essentielles :
  - `svn checkout` : obtenir la version courante du répositoire
  - `svn commit` : envoyer ses modifications vers le répositoire.
  - `svn update` : mettre à jour sa version de travail.
- Lire : `http://svnbook.red-bean.com/`

Bibliothèques utiles

- C++ est très fourni en bibliothèque.
- Mieux vaut chercher si il existe une bibliothèque qui fait le travail plutôt que réinventer la roue !
- Voici quelques bibliothèques utiles :
  - La bibliothèque standard (!)
  - Boost (`http://www.boost.org/`)
  - Qt (`http://trolltech.com/`)
  - Xerces (`http://xerces.apache.org/xerces-c/`)