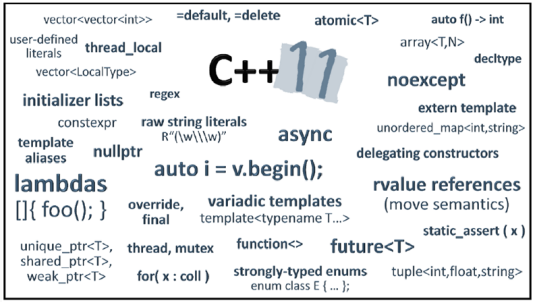
Programmation objet avancée en C++ moderne

Héritage - Classes et fonctions paramétrées - Bibliothèque standard.

Mathias Paulin
IRIT-CNRS Université Paul Sabatier

Année universitaire 2016-2017



1. Opérateurs spécifiques
   - Objets fonction
   - Litterals

2. Héritage et hiérarchie de classes
   - Classes dérivées
   - Héritage multiple
   - Navigation dynamique dans les hiérarchies de classe

Opérateurs spécifiques

Objets fonction

Surcharge de l'opérateur ()

Utile pour définir qu'un objet se comporte comme une fonction.

- Permet d'écrire des opérations prenant des opérations non triviales en paramètre.
- Utilisé par les algorithmes de la bibliothèque standard.

```
class Add {
  complex v;
public :
  Add (complex c): val{c} {}
  Add (float r, float i) val{{r, i}} {}
  void operator()(complex &c) const {c += val;}
}

void h(vector<complex>& vec, list<complex>& lst, complex z) {
  for_each(vec.begin(),vec.end(),Add{2,3});
  for_each(lst.begin(),lst.end(),Add{z});
}
```

Opérateurs spécifiques

Litterals

Litterals

Constantes textuelles

- Définies pour tous les types de base

```
123       // int
1.2       // double
1.2F      // float
1ULL     // unsigned long long
"as"    // C-style string (const char[3])
```

- Définition possibles d'autres litterals

```
"Hi"s               // string (not C-Style)
1.2i                // imaginary
101010111000101b    // binary integer
17.3km              // kilometers (unit)
```

Littérals

Surcharge de l'opérateur littéral

- `X operator"" suffix();` avec suffix le suffixe à donner au littéral.

```
template<typename T> void f(const T&);

constexpr complex operator"" i(long double d) { // imaginary literal
  return {0,d};     // complex is a literal type
}
std::string operator"" s(const char* p, size_t n) {// std::string literal
  return string{p,n}; // requires free-store allocation --> non constexpr
}
...
auto z = 2+1i;      // complex{2,1}
f("Hello");         // char *
f("Hello"s);        // 5 chars string
f("\tHello\n"s);    // 7 chars string
```

Littérals

Surcharge de l'opérateur littéral

- Quatre type de littéral peuvent être suffixés
  - Littéral entier - accepté par opérateur littéral prenant un `unsigned long long` ou un `const char *` en paramètre ou par un opérateur littéral template.
  - Littéral réel - accepté par opérateur littéral prenant un `long double` ou un `const char *` en paramètre ou par un opérateur littéral template.
  - Littéral chaine - accepté par opérateur littéral prenant une paire `(const char *, size_t)` en paramètre.
  - Littéral caractère - accepté par opérateur littéral prenant un paramêtre de type `char`, `wchar_t`, `char16_t` ou `char32_t`.

Littérals

Opérateur littéral template :

- Prend son argument comme un jeu de paramètres de template (variadic) et non pas comme paramètre de fonction.

```
template<char... chars>
constexpr int operator"" _b3(); // base 3 integer, i.e., ternary

201_b3 // means operator"" _b3"""<2,0,1>(); so 9*2+0*3+1 == 19
241_b3 // means operator"" _b3"""<2,4,1>(); so error: 4 'isnt a ternary digit
```

- Un jeu de paramètre de template de longueur variable permet de décomposer, à la compilation, un entier en une suite de chiffre.

Opérateur littéral template

```
constexpr int ipow(int x, int n) { // x to the nth power for n>=0
  return (n>0) ? x*ipow(x, n-1) : 1;
}

template<char c> // handle the single ternary digit case
constexpr int b3_helper() {
  static_assert(c<'3',"not a ternary digit");
  return c;
}

template<char c, char... tail> // peel off one ternary digit
constexpr int b3_helper() {
  static_assert(c<'3',"not a ternary digit");
  return ipow(3,sizeof...(tail))*(c-'0')+b3_helper(tail...);
}
```

Opérateur littéral template

```
template<char... chars>
constexpr int operator"" _b3() { // base 3 integer, i.e., ternary
  return b3_helper(chars...);
}
```

Attention

La bibliothèque standard réserve tous les suffixes ne commençant pas par _ !

Héritage et hiérarchie de classes

Classes dérivées

Relations entre classes

- Partage de concepts abstraits :
  - propriétés générales,
  - fonctionalités
- Partage de concepts concrets :
  - représentation,
  - implantation.

Relations entre classes

- Partage de concepts abstraits :
  - propriétés générales,
  - fonctionalités
- Partage de concepts concrets :
  - représentation,
  - implantation.

⇒ Construction de hiérarchies de classes : relation d'héritage.

- Héritage d'interface : permet l'utilisation de différentes classes dérivées via l'interface d'une classe de base.
- Héritage d'implantation : permet de réduire l'effort d'implantation en réutilisant le travail d'une classe de base.

Relations entre classes

- Héritage d'implantation :
  - fonctions et données,
  - réutilisation,
- Héritage d'interface :
  - fonctions virtuelles, redéfinition,
  - utilisation uniforme,
  - polymorphisme dynamique (à l'éxécution).

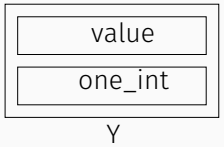
Relations entre classes

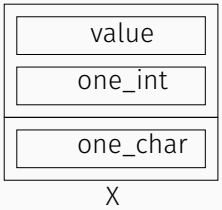
- Héritage d'implantation :
  - fonctions et données,
  - **réutilisation**,
- Héritage d'interface :
  - fonctions virtuelles, redéfinition,
  - utilisation uniforme,
  - **polymorphisme dynamique** (à l'éxécution).
- Implantation de *template*
  - programmation générique,
  - utilisation uniforme,
  - **polymorphisme statique** (à la compilation).

Notation - stockage

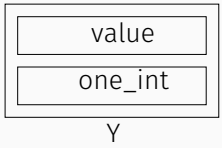
```
class Y {
  float value;
  int one_int;
public :
  Y(float v=0.f, int i=0):
      value{v}, one_int{i} {}
  // ...
};
```

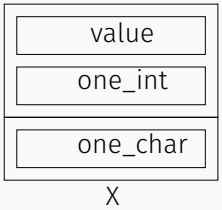
```
class X : public Y {
  char one_char;
public :
  X(...);
};
```





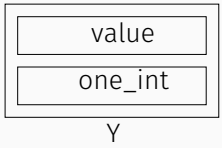
Notation - stockage

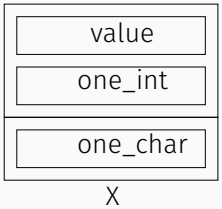




- Pas de sur-coût mémoire lié à l'héritage.
- Adresses croissantes de la classe de base à la plus profonde.
  - Sous certaines conditions.
  - POD : Plain Old Data - Compatible avec langage C.

Notation - stockage





- Pas de sur-coût mémoire lié à l'héritage.
- Adresses croissantes de la classe de base à la plus profonde.
  - Sous certaines conditions.
  - POD : Plain Old Data - Compatible avec langage C.
- Polymorphisme par références ou pointeurs
  - Un objet dérivé peut être manipulé comme un objet de classe de base.
  - La réciproque est fausse

Contrôle d'accès

Un membre de classe peut-être :

- private - nom utilisable uniquement dans la classe de définition ou les classes amies.
- protected - comme private mais accessibles aussi dans les classes dérivées et leurs amies.
- public - nom utilisable depuis n'importe quelle autre fonction/classe.

Correspond aux trois types de méthodes/fonctions accédant à une classe :

- Les méthodes/fonctions implantant la classe (private).
- Les méthodes/fonctions implantant une classe dérivée (protected).
- Les méthodes/fonctions d'utilisateurs généraux (public).

Contrôle d'accès

Le contrôle d'accès est lié au nom du membre, pas à son rôle (méthode, attribut, ...)

Attention - Il est possible de définir plusieurs sections de visibilité dans une classe :

```
class S {
public:
  int m1;
public:
  int m2;
};
```

Pour des raisons de performances, le compilateur peut réordonner les sections de visibilité.

Contrôle d'accès

Le contrôle d'accès est lié au nom du membre, pas à son rôle (méthode, attribut, ...)

Attention - Il est possible de définir plusieurs sections de visibilité dans une classe :

```
class S {
public:
  int m1;
public:
  int m2;
};
```

Pour des raisons de performances, le compilateur peut réordonner les sections de visibilité.

⇒ Éviter de raisonner sur l'ordre des champs, peut poser de gros problèmes ...

Membres *protected*

Utilité : Permet de proposer des services spécifiques aux classes dérivées pour améliorer les performances par rapport aux méthodes membres publiques (ex. contrôle de cohérence)

Attention à l'utilisation des membres protected

- Déconseillé pour les données
  - Difficulté à vérifier des invariants sur toutes les classes dérivées.
  - Difficulté de maintenance si changement dans la représentation de base.
  - Considéré comme une erreur de conception.

Membres *protected*

- Une classe dérivée n'a accès à des membres protected que de sa propre classe.

```
class Buffer {
protected:
  char a[128];
  // ...
};
```

```
class Linked_buffer : public Buffer {
  // ...
};
```

```
class Circular_buffer : public Buffer {
  // ...
  void f(Linked_buffer& p) {
    a[0] = 0;   // OK: access to 'Circular_buffers own protected member
    p.a[0] = 0; // error: access to protected member of different type
  }
};
```

Contrôle d'héritage

```
class X : public B { /* ... */ };
class Y : protected B { /* ... */ };
class Z : private B { /* ... */ };
```

Comme un membre, une classe de base peut être :

- public - Une classe est un sous-type de sa classe de base. Qualificatif le plus utilisé.
- private - Une classe est une restriction de sa classe de base. Permet de définir des contraintes d'utilisation plus forte que dans la classe de base en rendant private toutes les méthodes et attributs hérités.
- protected - Utile dans des hiérarchies de classe, limite l'accès à la base aux seules classes dérivées.

Contrôle d'héritage

```
class X : public B { /* ... */ };
class Y : protected B { /* ... */ };
class Z : private B { /* ... */ };
```

Comme un membre, une classe de base peut être :

- public - Une classe est un sous-type de sa classe de base. Qualificatif le plus utilisé.
- private - Une classe est une restriction de sa classe de base. Permet de définir des contraintes d'utilisation plus forte que dans la classe de base en rendant private toutes les méthodes et attributs hérités.
- protected - Utile dans des hiérarchies de classe, limite l'accès à la base aux seules classes dérivées.

- Qualificatif d'héritage par défaut : private pour les classes, public pour les struct.

Contrôle d'héritage

Le qualificatif d'héritage définit un contrôle d'accès

- aux membres d'une classe de base,
- à la conversion de pointeurs et de références à la classe dérivée vers la classe de base.

Si la classe **D** hérite de **B** :

- Si **B** est une base private, ses membres *publics* et *protected* ne peuvent être utilisés que par les membres de **D** et ses classes amies. Seuls les amis et les membres de **D** peuvent convertir un **D*** vers un **B***.

Contrôle d'héritage

Le qualificatif d'héritage définit un contrôle d'accès

- aux membres d'une classe de base,
- à la conversion de pointeurs et de références à la classe dérivée vers la classe de base.

Si la classe **D** hérite de **B** :

- Si **B** est une base protected, ses membres *publics* et *protected* ne peuvent être utilisés que par les membres et les amis de **D** ainsi que par ses dérivées. Seuls les amis et les membres de **D** et de ses dérivées peuvent convertir un **D*** vers un **B***.

Contrôle d'héritage

Le qualificatif d'héritage définit un contrôle d'accès

- aux membres d'une classe de base,
- à la conversion de pointeurs et de références à la classe dérivée vers la classe de base.

Si la classe **D** hérite de **B** :

- Si **B** est une base public, ses membres *publics* peuvent être utilisés par toute classe ou fonction. Ses membres *protected* peuvent être utilisés par les membres et les amis de **D** ainsi que par ses dérivées. toute fonction peut convertir un **D*** vers un **B***.

Constructeurs et destructeurs

- Les objets sont construits de la base vers les dérivées (Bottom-Up). Une classe de base est construite avant les membres qui sont construit avant une classe dérivée.
- Les objets sont détruits des dérivées vers la bases (top-down). Une classe dérivée est détruite avant les membres qui sont détruits avant la classe de base.
- Chaque classe initialise ses bases et ses membres (mais pas directement les membres des bases).
- Les destructeurs, dans une hiérarchie, doivent être déclarés virtual
- Les constructeurs par copie doivent être implanté attentivement, pour éviter d'oublier des membres.
- L'appel à une fonction virtuelle, le transtypage dynamique ou l'identification dynamique de type dans un constructeur ou destructeur dépend de l'état de construction ou destruction de l'objet.

Polymorphisme dynamique

- Déclaration et définition dans une classe de base.
- Redéfinition dans une classe dérivée.

```
class X {
  int x_field;
public :
  X(){}
  virtual int f(float p);
};
```

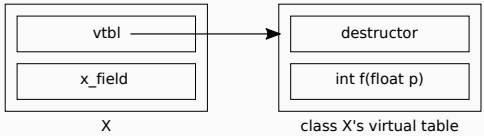
```
class Y : public X {
  int y_field;
public :
  Y():X{} {}
  int f(float p) override;
};
```

- Le mot clé `virtual` indique que la méthode peut être redéfinie.
- La redéfinition d'une fonction virtuelle doit respecter le profil défini dans la classe de base.
- Une fonction redéfinissant une fonction virtuelle est automatiquement virtuelle.

Implantation du polymorphisme dynamique

- Stockage par le compilateur d'informations permettant d'appeler la bonne fonction.
- Chaque classe possède une table de fonction virtuelle : **vtbl**
- Chaque objet d'une classe (de base ou dérivée) possède un pointeur vers la vtbl de sa classe.

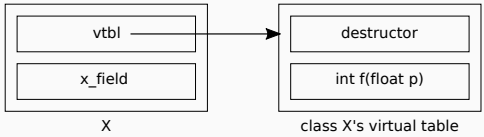
```
class X {
  int x_field;
public :
  X(){}
  virtual ~X(){};
  virtual int f(float p);
};
```



Implantation du polymorphisme dynamique

- Stockage par le compilateur d'informations permettant d'appeler la bonne fonction.
- Chaque classe possède une table de fonction virtuelle : **vtbl**
- Chaque objet d'une classe (de base ou dérivée) possède un pointeur vers la vtbl de sa classe.

```
class X {
  int x_field;
public :
  X(){}
  virtual ~X(){};
  virtual int f(float p);
};
```

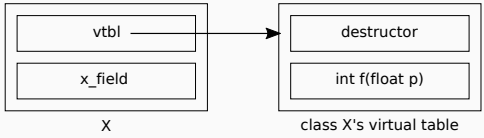


- Surcoût pour l'appel de fonctions virtuelles (ed Négligeable à ∼25% par rapport à une fonction standard)

Implantation du polymorphisme dynamique

```
class X {
  int x_field;
public :
  X(){}
  virtual ~X(){};
  virtual int f(float p);
};
```

```
class Y : public X {
  int y_field;
public :
  Y();
  ~Y(){};
  int f(float p) override;
};
```



Appel explicite d'une fonction de base

- Dans la redéfinition d'une fonction, on peut vouloir appeler la fonction redéfinie, effectuant un traitement de base, et étendre ce traitement à la classe dérivée.
- Utilisation de l'opérateur de visibilité **::**

```
int Y::f(float p) {
  int v = X::f(p);   // call base's f()
  // ...
  return v;
}
```

Appel explicite d'une fonction de base

- Dans la redéfinition d'une fonction, on peut vouloir appeler la fonction redéfinie, effectuant un traitement de base, et étendre ce traitement à la classe dérivée.
- Utilisation de l'opérateur de visibilité **::**

```
int Y::f(float p) {
  int v = X::f(p);   // call base's f()
  // ...
  return v;
}
```

- Les destructeurs des classes de base sont appelés automatiquement
- Les constructeurs doivent être appelés explicitement dans la construction de l'objet

```
Y::Y() : X(), y_field{} { // Attention to the order
}
```

Contrôle de la redéfinition

- Difficulté pour le programmeur de s'assurer de la redéfinition de la bonne fonction dans les hiérarchies de classes

Contrôle de la redéfinition

- Difficulté pour le programmeur de s'assurer de la redéfinition de la bonne fonction dans les hiérarchies de classes

Soit la hiérarchie de classe :

```
struct B0 {
  void f(int) const;
  virtual void g(double);
};

struct B1 : B0 { /* ... */ }
struct B2 : B1 { /* ... */ }
struct B3 : B2 { /* ... */ }
struct B4 : B3 { /* ... */ }
```

Imaginons le code suivant ...

```
struct D : B5 {
  // override f() in base class
  void f(int) const;
  // override g() in base class
  void g(int);
  // override h() in base class
  virtual int h();
}
```

- Ou sont les erreurs ?

Contrôle de la redéfinition

- Erreurs courantes quand la hiérarchie définit de nombreuses fonctions distribuées dans de nombreux fichiers d'entête.

Imaginons le code suivant ...

```
struct D : B5 {
  // override f() in base class
  void f(int) const;
  // override g() in base class
  void g(int);
  // override h() in base class
  virtual int h();
}
```

- `B0::f()` n'est pas virtuelle. On ne peut pas la redéfinir, seulement la surcharger.
- `D::g()` n'a pas le même profil que `B0::g()`. Ce n'est pas une redéfinition, juste une surcharge.
- `D::h()` ne redéfinit rien, il n'y a pas de fonction virtuelle `B0::h()`.

Contrôle de la redéfinition

Pour éviter ces erreurs, il est possible de contrôler explicitement la redéfinition de fonction par l'utilisation des préfixes ou suffixes suivants dans la déclaration :

- virtual : préfixe indiquant qu'une fonction PEUT être redéfinie.
- =0 : suffixe indiquant qu'une fonction virtuelle DOIT être redéfinie.
- override : suffixe indiquant une redéfinition d'une fonction de base.
- final : suffixe indiquant qu'une fonction ne peut pas être redéfinie.

En absence de ces contrôles, une fonction est virtuelle si et seulement si elle redéfinit une fonction virtuelle d'une classe de base.

Contrôle de la redéfinition

Pour éviter ces erreurs, il est possible de contrôler explicitement la redéfinition de fonction par l'utilisation des préfixes ou suffixes suivants dans la déclaration :

- virtual : préfixe indiquant qu'une fonction PEUT être redéfinie.
- =0 : suffixe indiquant qu'une fonction virtuelle DOIT être redéfinie.
- override : suffixe indiquant une redéfinition d'une fonction de base.
- final : suffixe indiquant qu'une fonction ne peut pas être redéfinie.

En absence de ces contrôles, une fonction est virtuelle si et seulement si elle redéfinit une fonction virtuelle d'une classe de base.

- Les compilateurs génèrent des warnings en cas d'utilisation inconsistante des contrôles explicites.

Contrôle de la redéfinition

- virtual : préfixe indiquant qu'une fonction PEUT être redéfinie.
  - à n'utiliser que pour introduire une NOUVELLE fonction virtuelle dans le système.
- override : suffixe indiquant une redéfinition d'une fonction de base.
  - à utiliser pour indiquer une redéfinition.
  - est un mot clé contextuel.
  - ne fait pas partie du type de la fonction.
- final : suffixe indiquant qu'une fonction ne peut pas être redéfinie.
  - à utiliser pour signifier qu'une fonction n'est plus virtuelle.
  - est un mot clé contextuel.
  - peut s'utiliser sur une classe.
  - N'est pas un moyen d'optimisation !

Importation des fonctions héritées

- Les fonctions ne peuvent pas être surchargée au delà de leur espace de visibilité :

```
struct Base {
  void f(int);
};
struct Derived : Base {
  void f(double);
};
```

```
void use(Derived d) {
  d.f(1); // call Derived::f(double)
  Base& br = d;
  br.f(1);// call Base::f(int)
}
```

Importation des fonctions héritées

- Les fonctions ne peuvent pas être surchargée au delà de leur espace de visibilité :

```
struct Base {
  void f(int);
};
struct Derived : Base {
  void f(double);
};
```

```
void use(Derived d) {
  d.f(1); // call Derived::f(double)
  Base& br = d;
  br.f(1);// call Base::f(int)
}
```

- La clause `using` permet d'assurer la surcharge dans les hiérarchies de classes

```
struct D2 : Base {
  // Brings Base::f in this scope
  using Base::f;
  void f(double);
};
```

```
void use2(D2 d) {
  d.f(1); // call D2::f(int)
  Base& br = d;
  br.f(1);// call Base::f(int)
}
```

Les constructeurs ne sont pas hérités

- Normal dans le sens de l'héritage par augmentation.
- Problématique dans le sens de l'héritage par spécialisation.

```
template<class T> struct Vector : std::vector<T> {
  T& operator[](size_type i) {
    check(i);
    return this->elem(i);
  }
  const T& operator[](size_type i) const {
    check(i);
    return this->elem(i);
  }
  void check(size_type i) {
    if (this->size()<i) throw range_error{"Vector::check() failed"};
  }
};
```

Les constructeurs ne sont pas hérités

- Problème lors de l'utilisation de la classe.

```
Vector<int> v { 1, 2, 3 }; // error: no initializer-list constructor
```

- Il faut importer explicitement les constructeurs :

```
template<class T> struct Vector : std::vector<T> {
  using vector<T>::vector; // inherit constructors
  ...
}
Vector<int> v { 1, 2, 3 }; // use initializer-list constructor from std::vector
```

- Attention à l'ajout de données membres
  - Utiliser l'initialisation en ligne

Les constructeurs ne sont pas hérités

- Attention à l'ajout de données membres
  - Utiliser l'initialisation en ligne

```
struct B1 {
  B1(int) {}
};

struct D1 : B1 {
  using B1::B1; // implicitly declares D1(int)
  int x {0};    // note: x is initialized
};
```

- Constructeurs en ligne difficile à maintenir.
- Restreindre l'import de constructeurs à l'héritage par spécialisation.

Pointeur de données membre

- Utile pour référencer indirectement une donnée membre.
- Opérateurs `->*` et `.*` : accès via un pointeur vers une données membres.

Pointeur vers fonction membre

Utile pour des couches ”middleware”.

```
class Std_interface {
  virtual void void method()=0;
}

using PMem = void (Std_interface::*)(); // pointer-to-member type
void f(Std_interface* p) {
  PMem s = &Std_interface::method;  // pointer to method()
  p->method();     // direct call
  p->*s();         // call trough pointer to member
}
```

Héritage et hiérarchie de classes

Héritage multiple

Relations d'héritage

- Héritage d'interface : permet l'utilisation de différentes classes dérivées via l'interface d'une classe de base.
- Héritage d'implantation : permet de réduire l'effort d'implantation en réutilisant le travail d'une classe de base.

Relations d'héritage

- Héritage d'interface : permet l'utilisation de différentes classes dérivées via l'interface d'une classe de base.
- Héritage d'implantation : permet de réduire l'effort d'implantation en réutilisant le travail d'une classe de base.

Une classe peut combiner différents aspects aspects de l'héritage.

- Héritage de plusieurs interfaces.
- Héritage de plusieurs interfaces et d'une implantation.
- Héritage de plusieurs implantations.

Héritage multiples - contraintes

- Héritage de plusieurs interfaces.
  - Interfaces = classes abstraites sans données membres.
  - Risque d'ambiguïté de noms de fonctions
- Héritage de plusieurs interfaces et d'une seule implantation.
  - Une seule classe concrète, pas de duplication de données
  - Risque d'ambiguïté de noms de fonctions

Héritage multiples - contraintes

- Héritage de plusieurs interfaces.
  - Interfaces = classes abstraites sans données membres.
  - Risque d'ambiguïté de noms de fonctions
- Héritage de plusieurs interfaces et d'une seule implantation.
  - Une seule classe concrète, pas de duplication de données
  - Risque d'ambiguïté de noms de fonctions
- Héritage de plusieurs implantations.
  - Duplication possible de données.
  - Besoin de contrôler la construction de la hiérarchie.

Résolution des ambigüité de noms

- Deux classes de base peuvent avoir une fonction membre avec le même profil :

```
class Satellite {
public:
  virtual Pos center() const = 0;
  virtual Debug_info debug();
  // ...
};
```

```
class Displayed {
public:
  virtual void draw() = 0;
  virtual Debug_info debug();
  // ...
};
```

- L'utilisation d'une héritière de ces deux classes peut générer des ambiguïtés

```
class Comm_sat : public Satellite,
                 public Displayed {
  public:
  Pos center() const override;
  void draw() override;
  // ...
};
```

Résolution des ambigüité de noms

- Deux classes de base peuvent avoir une fonction membre avec le même profil :

```
class Satellite {
public:
  virtual Pos center() const = 0;
  virtual Debug_info debug();
  // ...
};
```

```
class Displayed {
public:
  virtual void draw() = 0;
  virtual Debug_info debug();
  // ...
};
```

- L'utilisation d'une héritière de ces deux classes peut générer des ambiguïtés

```
class Comm_sat : public Satellite,
                 public Displayed {
  public:
  Pos center() const override;
  void draw() override;
  // ...
};
```

```
void f(Comm_sat& cs) {
  Debug_info di = cs.debug(); // error
  // explicit disambiguation
  di = cs.Satellite::debug(); // OK
  di = cs.Displayed::debug(); // OK
}
```

Résolution des ambigüité de noms

- Résolution explicite peu lisible
- Solution plus élégante par redéfinition de fonction virtuelle

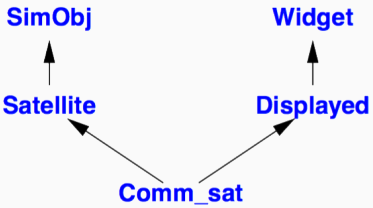
```
class Comm_sat : public Satellite, public Displayed {
public:
  Debug_info debug() override {
    // override Comm_sat::get_debug() and Displayed::get_debug()
    Debug_info di1 = Satellite::get_debug();
    Debug_info di2 = Displayed::get_debug();
    return merge_info(di1,di2);
  }
  // ...
};
```

- Une fonction déclarée dans une classe dérivée redéfinit TOUTES les fonctions virtuelles héritées de même nom et profil

Résolution des ambigüité de noms

- En pratique, la conception séparée de composants (design pattern *Adapter*)

```
class Satellite : public SimObj {
  // map SimObj facilities
public:
  // call SimObj::DBinf() and
  // extract information
  virtual Debug_info debug();
};
class Displayed : public Widget {
  // map Widget facilities
public:
  // read Widget data and
  // compose Debug_info
  virtual Debug_info debug();
};
```



Utilisation répétée d'une classe de base

- Si une seule classe de base possible, la hiérarchie d'héritage est un ARBRE.
- Si plusieurs classes de base possibles, une classe peut alors apparaître plusieurs fois dans la hiérarchie.

```
struct Storable {
  // persistent storage
  virtual string get_file() = 0;
  virtual void read() = 0;
  virtual void write() = 0;
  virtual ~ Storable() { }
};

class Transmitter : public Storable {
public:
  void write() override;
  // ...
};
```

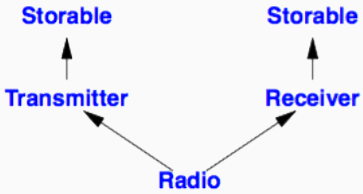
```
class Receiver : public Storable {
public:
  void write() override;
  // ...
};

class Radio : public Transmitter,
              public Receiver {
public:
  string get_file() override;
  void read() override;
  void write() override;
  // ...
};
```

Utilisation répétée d'une classe de base

Deux cas sont envisageables :

- Un objet Radio possède deux sous-objets Storable (de Transmitter et de Receiver)
- Un objet Radio possède un seul sous-objet Storable partagé




```
void Radio::write() {
  Transmitter::write();
  Receiver::write();
  // ... write radio-specific information ...
}
```

- Si non spécifié, un objet possède deux objets de base différents.
  - Pas de problèmes pour héritage d'interface.
  - Redéfinition de fonction virtuelle pour lever l'ambiguïté

Utilisation répétée d'une classe de base

- Que se passe-t-il si un objet Storable gère des données ne pouvant être dupliquées
  - Resources critiques, ...

```
struct Storable {
public:
  Storable(const string &s);  // store in file named s
  virtual void read() = 0;
  virtual void write() = 0;
  virtual ~Storable() { }
protected :
  string filename;

  Storable(const Storable&) = delete;
  Storable& operator=(const Storable&) = delete;
};
```

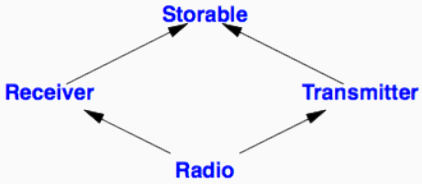
- Nécessite de modifier les classes Radio, Transmitter et Receiver pour qu'il n'y ait pas duplication.

Utilisation répétée d'une classe de base

- Héritage virtuel
  - Chaque base "virtuelle" d'un objet est représentée une seule fois.

```
class Transmitter : public virtual Storable {
public:
  void write() override;
  // ...
};

class Receiver : public virtual Storable {
public:
  void write() override;
  // ...
};
class Radio : public Transmitter, public Receiver {
public:
  void write() override;
  // ...
};
```



Héritage en losange

- Duplication supprimée par héritage virtuel.

Héritage en losange

- Duplication supprimée par héritage virtuel.
- Constructeurs appelés une seule fois et non pas pour chaque branche.
  - Une classe doit connaitre ses bases virtuelles (directe ou non) et les initialiser.
  - Une base virtuelle doit être initialisée avant les autres.

Héritage en losange

- Duplication supprimée par héritage virtuel.
- Constructeurs appelés une seule fois et non pas pour chaque branche.
  - Une classe doit connaitre ses bases virtuelles (directe ou non) et les initialiser.
  - Une base virtuelle doit être initialisée avant les autres.
- Attention à la "contamination" virtuelle ... (en cas de dérivation de la classe la plus profonde)
  - Obligation de construire les bases virtuelles
  - Import de constructeurs (contraignant)
- Problème inexistant avec les destructeurs.

Héritage en losange

- Duplication supprimée par héritage virtuel.
- Constructeurs appelés une seule fois et non pas pour chaque branche.
  - Une classe doit connaitre ses bases virtuelles (directe ou non) et les initialiser.
  - Une base virtuelle doit être initialisée avant les autres.
- Attention à la "contamination" virtuelle ... (en cas de dérivation de la classe la plus profonde)
  - Obligation de construire les bases virtuelles
  - Import de constructeurs (contraignant)
- Problème inexistant avec les destructeurs.

Limiter l'utilisation de l'héritage multiples à la séparation de l'interface et de l'implantation.

Héritage et hiérarchie de classes

Navigation dynamique dans les hiérarchies de classe

Utilité

Le mécanisme d'héritage fournit :

- La construction d'un système comme une hiérarchie de classe
  - Le système complet est une abstraction
  - Les fonctionalités sont des interfaces
  - Les détails sont des implantations
- Un polymorphisme dynamique dans la hiérarchie
  - Fonctions virtuelles re-définissables dans la hiérarchie
  - Fonctions surchargées spécialisant les traitements

Utilité

Le mécanisme d'héritage fournit :

- La construction d'un système comme une hiérarchie de classe
  - Le système complet est une abstraction
  - Les fonctionalités sont des interfaces
  - Les détails sont des implantations
- Un polymorphisme dynamique dans la hiérarchie
  - Fonctions virtuelles re-définissables dans la hiérarchie
  - Fonctions surchargées spécialisant les traitements

Pourquoi vouloir changer le type d'un objet ou connaitre son type réel ?

Utilité

Les systèmes de gestion de données, de raisonnement, de calcul, de commande, d'affichage ... sont programmés de façon autonome. Pour cela, un système

- doit fonctionner pour toutes les données qu'on peut lui fournir,
- ne peut rien connaitre sur le contenu de ces données,
- ne peut que définir ses interfaces de manipulation,
- Est programmé en fonction de ses interfaces et non pas des données réelles.

Utilité

Les systèmes de gestion de données, de raisonnement, de calcul, de commande, d'affichage ... sont programmés de façon autonome. Pour cela, un système

- doit fonctionner pour toutes les données qu'on peut lui fournir,
- ne peut rien connaitre sur le contenu de ces données,
- ne peut que définir ses interfaces de manipulation,
- Est programmé en fonction de ses interfaces et non pas des données réelles.

Un système ne peut communiquer que des objets qu'il connait :

- Les données réelles héritent des classes du système.
- Le système ne renvoie que des types internes, indépendamment du type réel des données.

Utilité

Les contraintes imposées propres et nécessaires mais

- perte de l'information de type des données entre l'entrée et la sortie du système.

Besoin d'un mécanisme pour retrouver le typage perdu :

Utilité

Les contraintes imposées propres et nécessaires mais

- perte de l'information de type des données entre l'entrée et la sortie du système.

Besoin d'un mécanisme pour retrouver le typage perdu :

- Introspection
  - Un objet peut renseigner sur son type.
- Pointeur et références
  - Seuls moyens de manipulation polymorphe d'objets.

Utilité

Les contraintes imposées propres et nécessaires mais

- perte de l'information de type des données entre l'entrée et la sortie du système.

Besoin d'un mécanisme pour retrouver le typage perdu :

- Introspection
  - Un objet peut renseigner sur son type.
- Pointeur et références
  - Seuls moyens de manipulation polymorphe d'objets.

  Outils d'introspection du C++
  - Opérateurs de transtypage : `dynamic_cast<T>(...)`
  - Information de type :`class std::type_info` et `typeid(...)`

L'opérateur `dynamic_cast<type_name>(object)` prend deux paramètres :

- Un nom de type, noté entre chevrons `<type_name>`
- Un objet, par pointeur ou référence noté entre parenthèses `(object)`

dynamic_cast et pointeurs

- Syntaxe
  - `T* dynamic_cast<T*>(p)`
- Typage ascendant
  - Si **p** est de type **T*** ou de type **D***, avec **T** une classe de base de **D** le résultat est le même que l'affectation directe.
- Typage descendant
  - Si **p** est de type **B***, avec **T** une classe dérivée de **B** le résultat est un pointeur vers l'objet réel dérivé de **p**. Si ce n'est pas le cas, l'opérateur renvoie `nullptr`.

dynamic_cast, pointeurs, Typage ascendant

- Typage naturel d'une classe vers une base.
- Pas de sur-coût par rapport à une affectation de pointeurs.

```
class Z : public X, protected Y {
  // ...
}

void f(Z *p) {
  X* pi1 = p;                    // OK
  X* pi2 = dynamic_cast<X*>(p); // OK

  Y* pi3 = p;                    // error, Y is a protected base.
  Y* pi4 = dynamic_cast<Y*>(p); // OK, pi4 is nullptr
}
```

dynamic_cast, pointeurs, Typage descendant

- Impossible de déterminer à la compilation si le typage est correct.
- Nécessite un paramètre pointeur vers une classe polymorphe.
  - Informations de typage stockées dans la vtable.
- Utile pour déterminer l'adresse de base d'un sous-objet.
  - `dynamic_cast<void *>(Sous_type *ptr)`
  - Pour interfacer C++ et fonctions de bas niveau (driver, noyau système ...)

```
class S: public R { // polymorphic base (R has virtual functions)
};
class D : public E { // base not polymorphic (E has no virtual functions)
};
void g(R* pr, E* pe) {
  S* pd1 = dynamic_cast<S*>(pr); // OK
  D* pd2 = dynamic_cast<D*>(pe); // error: E not polymorphic
}
```

dynamic_cast et référence

- Syntaxe
  - `T& dynamic_cast<T&>(r)`
- Fonctionnement
  - Par transtypage ascendant ou descendant.
  - N'est pas la question
    - "r désigne-t-elle un objet de type T ?"
  - Est l'assertion.
    - "r désigne un objet de type T !"
  - Si l'assertion est fausse, lève une exception `bad_cast`
    - Rappel : pas de références nulles.

dynamic_cast, références

- Même fonctionnement et limitations qu'avec les pointeurs.
- Gestion des erreur de typage différentes.
  - Traitement d'exception `bad_cast` et non pas test à `nullptr`

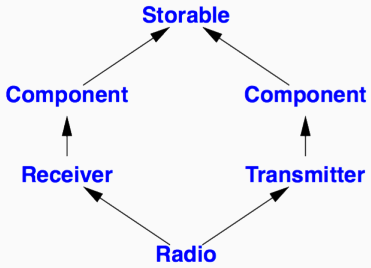
```
class S: public R { // polymorphic base (R has virtual functions)
};
class D : public E { // base not polymorphic (E has no virtual functions)
};
void g(R& pr, E& pe) {
  try {
    S& pd1 = dynamic_cast<S&>(pr); // OK
    R& pd2 = dynamic_cast<R&>(pe); // error: R and E unrelated
  } catch (bad_cast) {
    // ..
  }
}
```

dynamic_cast et héritage multiple

- Réalisable quand la classe T est unique dans la hiérarchie.

```
class Component : public virtual Storable {
};
class Receiver : public Component {
};
class Transmitter : public Component {
};
class Radio : public Receiver, public Transmitter {
};

void h1(Radio& r) {
  Storable* ps = &r; // a Radio has a unique Storable
  // pc = nullptr; a Radio has two Components
  Component* pc = dynamic_cast<Component*>(ps);
}
```



dynamic_cast et static_cast

- static_cast :
  - Transtypage réalisé à la compilation, sans sur-coût de calcul.
  - Pas de vérification de correction
  - Correction gérée par le programmeur.
  - Ne fonctionne pas pour les bases virtuelles ni les bases privées.

```
void g(Radio& r) {
  Receiver* prec = &r; // Receiver is an ordinary base of Radio
  Radio* pr = static_cast<Radio*>(prec);  // OK, unchecked
  pr = dynamic_cast<Radio*>(prec); // OK, run-time checked

  Storable* ps = &r; // Storable is a virtual base of Radio
  pr = static_cast<Radio*>(ps); // error: cannot cast from virtual base
  pr = dynamic_cast<Radio*>(ps); // OK, run-time checked
}
```

dynamic_cast, static_cast, const_cast

- dynamic_cast et static_cast respectent les qualificatif const.

```
class Users : private set<Person> { /* ... */ };

void f2(Users* pu, const Receiver* pcr) {
  static_cast<set<Person>*>(pu); // error: access violation
  dynamic_cast<set<Person>*>(pu); // error: access violation
  static_cast<Receiver*>(pcr); // error: 'cant cast away const
  dynamic_cast<Receiver*>(pcr);// error: 'cant cast away const

  Receiver* pr = const_cast<Receiver*>(pcr); // OK — Unsafe
}
```

type_info, typeid

- L'opérateur dynamic_cast est utile pour la conversion de type à l'exécution
  - préserve la flexibilité et l'extensibilité, comme les fonctions virtuelles.
- Il est parfois essentiel de connaitre le type exact d'un objet
  - Son nom
  - Son organisation mémoire,
  - ...
- L'opérateur typeid implante ce service
- `const std::type_info& typeid(type_name);`
  - A partir d'un nom de type, renvoie une référence à la description du type.
- `const std::type_info& typeid(expression);`
  - A partir d'une expression, renvoie une référence à la description du type du résultat.

type_info, typeid

- Le type référencé par les paramètres `type_name` ou `expression` doit être un rtype complétement défini lors de l'appel à `typeid`.
- Si la valeur du paramètre est `nullptr` , l'opérateur lève l'exception `bad_typeid`
- Si le paramètre n'est pas un type polymorphe ou une lvalue, le résultat est déterminé à la compilation sans évaluer l'opérande.
- Le résultat désigne toujours le type de l'objet lors de sa déclaration.

type_info, typeid

- La classe type_info correspond globalement à :

```
class type_info {
  // data
public:
  virtual ~type_info(); // is polymorphic

  bool operator==(const type_info&) const noexcept; // can be compared
  bool operator!=(const type_info&) const noexcept;

  bool before(const type_info&) const noexcept; // ordering
  size_t hash_code() const noexcept; // for use by unordered_map and the like

  const char* name() const noexcept; // name of type

  type_info(const type_info&) = delete; // prevent copying
  type_info& operator=(const type_info&) = delete; // prevent copying
};
```

Du bon usage du RTTI

- Utiliser le RTTI uniquement lorsque nécessaire.
- La vérification statique (à la compilation) est bien plus sure et efficace
- La vérification statique conduit à du meilleur code, mieux structuré.
- Les interfaces fondées sur les fonctions virtuelles combinent vérification statique et polymorphisme dynamique

Du bon usage du RTTI

- Utiliser le RTTI uniquement lorsque nécessaire.
- La vérification statique (à la compilation) est bien plus sure et efficace
- La vérification statique conduit à du meilleur code, mieux structuré.
- Les interfaces fondées sur les fonctions virtuelles combinent vérification statique et polymorphisme dynamique

Il est important de ne pas sur-utiliser le RTTI

Mauvais usage du RTTI

- Evaluation redondante d'information sur le type.
- Pensée non structurée.

```
void rotate(const Shape& r) {
  if (typeid(r) == typeid(Circle)) {
    // do nothing
  } else if (typeid(r) == typeid(Triangle)) {
    // ... rotate triangle ...
  } else if (typeid(r) == typeid(Square)) {
    // ... rotate square ...
  }
 // ...
}
```

Mauvais usage du RTTI

- Evaluation redondante d'information sur le type.
- Pensée non structurée.

```
void rotate(const Shape& r) {
  if (typeid(r) == typeid(Circle)) {
    // do nothing
  } else if (typeid(r) == typeid(Triangle)) {
    // ... rotate triangle ...
  } else if (typeid(r) == typeid(Square)) {
    // ... rotate square ...
  }
 // ...
}
```

- Ce code a réellement été écrit !
- Résultat de la non compréhension de l'héritage d'interface !

Mauvais usage du RTTI

```
class Object { //polymorphic
};
class Container : public Object {
public:
  void put(Object*);
  Object* get();
};
class Ship : public Object {
};
Ship* f(Ship* ps, Container* c) {
  c->put(ps); // put the Ship into the container
  Object* p = c->get(); // retrieve an Object from the container
  if (Ship* q = dynamic_cast<Ship*>(p)) { // run-time check that the Object is a Ship
    return q;
  } else {
    // ... do something else (typically, error handling) ...
  }
}
```

Mauvais usage du RTTI

```
class Object { //polymorphic
};
class Container : public Object {
public:
  void put(Object*);
  Object* get();
};
class Ship : public Object {
};
Ship* f(Ship* ps, Container* c) {
  c->put(ps); // put the Ship into the container
  Object* p = c->get(); // retrieve an Object from the container
  if (Ship* q = dynamic_cast<Ship*>(p)) { // run-time check that the Object is a Ship
    return q;
  } else {
    // ... do something else (typically, error handling) ...
  }
}
```

- Programmeurs java (pré générique) : utilisation erronnée du RTTI.

Vérification statique vs RTTI

- Utilisation de container génériques
  - template C++

```
Ship* f(Ship* ps, std::vector<Ship*>& c) {
  c.push_back(ps); // put the Ship into the container
  // ...
  return c.pop_back(); // retrieve a Ship from the container
}
```

Vérification statique vs RTTI

- Utilisation de container génériques
  - template C++

```
Ship* f(Ship* ps, std::vector<Ship*>& c) {
  c.push_back(ps); // put the Ship into the container
  // ...
  return c.pop_back(); // retrieve a Ship from the container
}
```

- Code vérifiable statiquement.
- Code plus concis
- Gère quasiment tout les besoins si combinés avec fonctions virtuelles.
- L'argument des template (`<T>`) remplace `Object` en permettant une vérification statique.

Questions ?