

## Programmation Avancée Les différents mécanismes des langages (dont C++) pour la généricité

Norme ISO

Raphaëlle Chaîne  
raphaelle.chaine@iris.cnrs.fr  
2024-2025

1

1

## Dérivation et constructeurs

```
class B { contenant les données membres m1, .. mn
};
class D : typedérivation B1, typedérivation B2, ...,
        typedérivation Bn { ...
};
```

Les objets de la classe dérivée sont initialisés dans l'ordre suivant :

- initialisation\* de chacune des bases directes
- initialisation\*\* des membres spécifiques à la classe dérivée
- puis exécution des instructions du corps du constructeur appelé

\*Dans l'ordre des dérivations

\*\*Dans l'ordre de leurs définitions dans la définition de la classe

65

65

- Lors de la définition d'un **constructeur** pour une classe dérivée, la **liste d'initialisation** doit mentionner les constructeurs choisis \*:

- pour l'initialisation de chacune des bases,
- pour l'initialisation de chacun des objets membres spécifiques à la classe dérivée

```
D::D (liste_arg) : B1(liste_arg1),...,Bn(liste_argn) ,
                m1(liste_arg11) , ... , mn(liste_argnn)
                {corps du constructeur}
```

\*Appel aux constructeurs par défaut sinon (s'ils existent!) 66

66

- Exemple :  

```
class Employe
{public :
    Employe(int n=0) : num(n)
    {std::cout << "initialisation
    Employe\n";}
protected :
    int num;
    //int num(0); C++11
    //int num(0); C++11
};

class Cadre : public Employe
{public :
    Cadre(int n, int e, int nb) :
        Employe(n), echelon(e),
        resp(new Employe[nb])
    {std::cout << "initialisation
    Cadre\n"; }
private :
    int echelon;
    Employe * resp;
};

Cadre dd(27,3,5);
```
- Attention : La version suivante n'est pas acceptée!  

```
Cadre::Cadre(int n, int e, int nb) : num(n),
    echelon(e),resp(new Employe[nb]){}
}
```

67

67

- **En l'absence de tout constructeur** pour la classe dérivée :

- Génération par C++ d'un **constructeur par défaut**, qui initialise

- les parties héritées
- et les membres spécifiques à la classe dérivée

selon leurs propres lois d'initialisation par défaut\*

```
class B          class C          class D : public B
{public :        {public :        { private :
    B()          C()          C c;
    {std::cout << "B\n";}    {std::cout << "C\n";} };
};
D d;
```

\*Si elles existent!

68

68

- **En l'absence de constructeur par copie** pour la classe dérivée :

- Génération par C++ d'un constructeur par copie qui initialise :

- les parties héritées
- et les membres spécifiques à la classe dérivée

selon leurs propres règles de copie \*

```
class B          class C          class D : public B
{public :        {public :        { private :
    B(const B&)   C(const C&)   C c;
    {std::cout << "B\n";}    {std::cout << "C\n";} };
};
D d,dd(d); // Attention, que faut-il ajouter pour cet exemple?
```

69

69

Que penser de :

```
class B
{public :
  B()
  {std::cout <<"def B\n";}
  B(const B& b)
  {std::cout <<"cop B\n";}
};
class D : public B
{ public :
  D()
  {std::cout <<"def D\n";}
  D(const D& d)
  {std::cout <<"cop D\n";}
private :
  C c;
};

class C
{public :
  C()
  {std::cout <<"def C\n";}
  C(const C& c)
  {std::cout <<"cop C\n";}
};

D d1,d2(d1);
```

70

70

- Intérêt des possibilités d'*upcast* implicite dans l'écriture d'un constructeur par copie

```
class Employe
{public :
  Employe(const Employe& e);
protected :
  int num;
};

class Cadre : public Employe
{public :
  Cadre();
  Cadre(const Cadre& c);
private :
  int echelon;
};

Cadre::Cadre(const Cadre& c) : Employe(c),
                             echelon(c.echelon) {...}
```

On a utilisé la conversion implicite standard de *c* d'un Cadre & en Employe &

71

71

- En l'absence d'opération d'affectation pour la classe dérivée :

- Génération par C++ d'une surcharge de l'opérateur = qui affecte
  - les parties héritées
  - et les membres spécifiques à la classe dérivée

en utilisant leurs propres règles d'affectation\*, et retourne une référence sur l'objet affecté

```
class B
{public :
  B& operator=(const B& b)
  {...
   std::cout<<"=B\n";
   return *this;}
};
class D : public B
{ private : C c; };

class C
{public :
  C& operator=(const C& c)
  {...
   std::cout<<"=C\n"; return *this;}
};
```

72

72

- Intérêt des possibilités d'*upcast* implicite dans l'écriture d'un opérateur d'affectation

```
class Employe
{blabla};

class Cadre : public Employe
{public :
  Cadre& operator=(const Cadre&);
private :
  int echelon;
};

Cadre& Cadre::operator = (const Cadre& c)
{ if ( this != &c )
  { Employe* thise=this; Employe* ce=&c;
    *thise = *ce; //Affectation entre Employe
    echelon= c.echelon;
  }
  return *this;
}
```

73

73

## Destruction

- En l'absence de destructeur, le langage génère un destructeur avec un corps vide
- Lors de la destruction d'un objet dérivé, les destructeurs des données membres et des classes de base sont appelés
- Les parties héritées et les données membres spécifiques à la classe dérivée sont détruites dans l'ordre inverse de leur initialisation

74

74

## Constructeurs, destructeur et C++11

- Mots clés default et delete

Implantation par défaut par le compilateur

```
struct LaClasse {
  LaClasse()=default; //C++11
  ~LaClasse()=default; //C++11
  ... blabla ...
};
```

Suppression de l'implantation par défaut par le compilateur

```
struct SansCopie{
  SansCopie & operator =(
    const SansCopie & ) = delete;
  SansCopie ( const SansCopie & ) = delete;
};
```

75

75

## Constructeurs plus souples avec C++11

### – Délégation de constructeurs

```
struct LaClasse {
    LaClasse( int a1, int a2): _a1(a1), _a2(a2){}
    LaClasse( int a1): _a1(a1) {}
    LaClasse() : LaClasse(55,2) {}
    int _a1;
    int _a2=5; // initialization C++11
                // on peut même écrire int _a2{5};
};
```

Appel d'un constructeur dans un autre constructeur :  
Moins de duplication de code qu'avant C++11

76

76

## Constructeurs plus souples avec C++11

### – Héritage de constructeur

```
struct AutreClasse: public LaClasse {
    using LaClasse::LaClasse;
    ...
};
```

- Héritage des constructeurs de la classe de base, les données membres supplémentaires dans AutreClasse sont initialisées par défaut

77

77

Comment bien munir une classe de toutes les fonctions membres indispensables?

- Utiliser les constructeurs / affectation / destructeurs du langage
- Sauf pour les capsules RAIL (**Resource Acquisition Is Initialization**)
  - Gestion de ressources supplémentaires (tas, fichier) via des données-membres
  - **Allouées dans le constructeur/affectation** (levée d'une exception sinon!)
  - **Libérées dans le destructeur/affectation**
  - Nécessité de **copies profondes**

78

78

## Rule of 3 (avant C++11) : triplet constructeur par copie / affectation /destructeur

- Constructeur par copie

```
LaClasse::LaClasse(const LaClasse &)
```

- Opérateur d'affectation

```
const LaClasse &
LaClasse::operator=(const LaClasse &)
```

- Destructeur

```
LaClasse::~~LaClasse()
```

- Lorsqu'on ne prend pas ceux générés par le langage, il faut redéfinir les 3!

79

79

## Le langage était-il efficace avant C++11?

- On se place dans le cas où LaClasse correspond à une capsule RAIL.
- Que font les instructions suivantes?

```
LaClasse a;
a=F();
```

Sachant que :

```
LaClasse F()
{return LaClasse(8);}
```

80

80

## Le langage était-il efficace avant C++11?

```
LaClasse F()
{return LaClasse(8);}
```

```
LaClasse a; a=F();
```

- Création d'une variable a et initialisation par défaut (potentiellement avec des demandes d'allocation dans le tas)

- Appel à F

- Initialisation de la valeur de retour de F par le constructeur par conversion d'un entier (potentiellement avec des demandes d'allocation dans le tas)

- Affectation à a de la valeur de retour de F

- Destruction de ressources potentiellement réservées dans le tas par a pour accueillir la création de copie (demande d'allocations potentielles)
- Destructeur appelé sur la valeur de retour

81

– a sera détruit en fin de bloc

81

## Le langage était-il efficace avant C++11?

- Que fait l'instruction suivante?

```
LaClasse F()
{ return LaClasse(8); }
LaClasse a; a=F();
```

- La valeur de retour de F est stockée dans un temporaire (*xvalue*) qui va aussitôt être « copié » dans a avant d'être détruit
- Si certains de ses champs requièrent de l'allocation mémoire, il aurait été préférable que la copie dans a puisse directement récupérer les espaces alloués par le temporaire ...

82

82

## Depuis C++11, la notion de référence devient plus précise!

- Le concept de référence à un temporaire (ie à une *xvalue*) est introduit par la norme!  
Syntaxe : **&&** (mais on dit *rvalue reference*!)
- La **construction par déplacement** devient possible

```
class LaClasse
{ BigT *ptr;
public:
    LaClasse(const LaClasse &x)
        : ptr(new BigT>(*x.ptr)) {} //COPIE
    ~LaClasse() { delete ptr; } // DESTRUCTION
    LaClasse(LaClasse &&x) : ptr(x.ptr)
        { x.ptr = nullptr; } //TRANSFERT
};
```

83

83

## Depuis C++11, la notion de référence devient plus précise!

- Constructeur par déplacement choisi à la place du constructeur de recopie en cas de copie d'un objet temporaire (sur le point d'être supprimé)
- **L'opération d'affectation par déplacement** devient également envisageable

```
LaClasse & LaClasse::operator=(LaClasse
&&x)
{ if (this!=&x)
    { delete ptr;
      ptr=x.ptr;
      x.ptr=nullptr; }
}
```

84

84

## La notion de référence devient plus précise!

- C++11 introduit également le « sémantique move »
  - la fonction `std::move()` retourne une référence sur un temporaire (on dit *rvalue référence*) ie sur une *xvalue*
  - Même quand elle prend en paramètre une *lvalue*!
  - Permet de « caster » en T&&

```
template <typename T>
typename remove_reference<T>::type&&
move(T&& a) {
    return a;
}
```

85

85

## La notion de référence s'étend!

- La fonction `move()` modifie la nature de la référence passée en paramètre (donc sans préserver les données de sa source!)
- Permet de réécrire `swap()`, sans recopie!

```
void swap(LaClasse& a, LaClasse& b)
{
    LaClasse tmp(std::move(a)); //init avec &&
    a = std::move(b); //affectation avec &&
    b = std::move(tmp); //affectation avec &&
}
```

Aucune copie profonde de faite, seulement un transfert!

86

86

## La notion de référence s'étend!

- A comparer avec l'implantation classique qui nécessite 3 copies (1 constructeur par copie et 2 affectations)

```
void swap(LaClasse & a, LaClasse & b)
{
    LaClasse c(a);
    a=b;
    b=c;
}
```

87

87

## Depuis C++11, la notion de référence devient plus précise!

- Attention CEPENDANT!!!!!! lorsqu'on utilise une rvalue reference nommée a
- Même si on installe bien la référence a sur une xvalue
- Si on appelle ensuite une fonction sur a, la fonction considèrera a comme une lvalue, puisque elle a désormais un nom!
- C'est normal : tout le temps où on utilise notre objet on n'a pas envie qu'il se fasse dépouiller!!

88

88

## Constructeur par déplacement et héritage

- Attention!!! Quand on ne prend pas le constructeur par déplacement généré par le langage!

```
Derivee::Derivee(Derivee &&arg) :  
Base(std::move(arg)),  
member(std::move(arg.member)) {}
```

- En effet, arg et arg.member sont considérés comme des lvalue lors de leur appel dans la liste d'initialisation!!!!
- En l'absence du std::move c'est une construction par copie qui est faite de arg (upcasté en Base&) et de arg.member

89

89

## Depuis C++11, Rule of 3 a été remplacée par Rule of 5

- Constructeur par copie  
`LaClasse::LaClasse(const LaClasse &)`
- Constructeur par déplacement  
`LaClasse::LaClasse(LaClasse &&)`
- Opérateur d'affectation par copie  
`LaClasse &`  
`LaClasse::operator=(const LaClasse &)`
- Opérateur d'affectation par déplacement  
`LaClasse &`  
`LaClasse::operator=(LaClasse &&)`
- Destructeur  
`LaClasse::~~LaClasse()`

90

90

## Sémantique du constructeur « par copie » depuis C++11

- Restriction de la philosophie du langage pour une prise de risque réduite
- Constructeur pour initialiser un objet à partir d'un objet du même type ...
  - C++ considère désormais que cela devrait obligatoirement être pour en faire une copie....
  - D'ailleurs ça s'appelle un constructeur par copie!!

91

91

## Sémantique du constructeur par copie depuis C++11

- `LaClasse(LaClasse(LaClasse(LaClasse(8)))`  
copie de copie de copie d'un temporaire ...
- Même chose que directement que de considérer directement le temporaire `LaClasse(8)...`

92

92

## Sémantique du constructeur par copie depuis C++11

- Copy elision
  - Le compilateur fait cette simplification même si vous avez placé un effet de bord dans votre constructeur par copie.
- Moralité : lorsque vous faites le code d'un constructeur « par copie », vous devez vous engager à faire une copie...
- Si on ne veut pas ça, -fno-copy-elisions, non autorisé en C++17

93

93