Le langage C++

ISIMA - ZZ2 - 2009

Christophe Duhamel
Andréa Duhamel

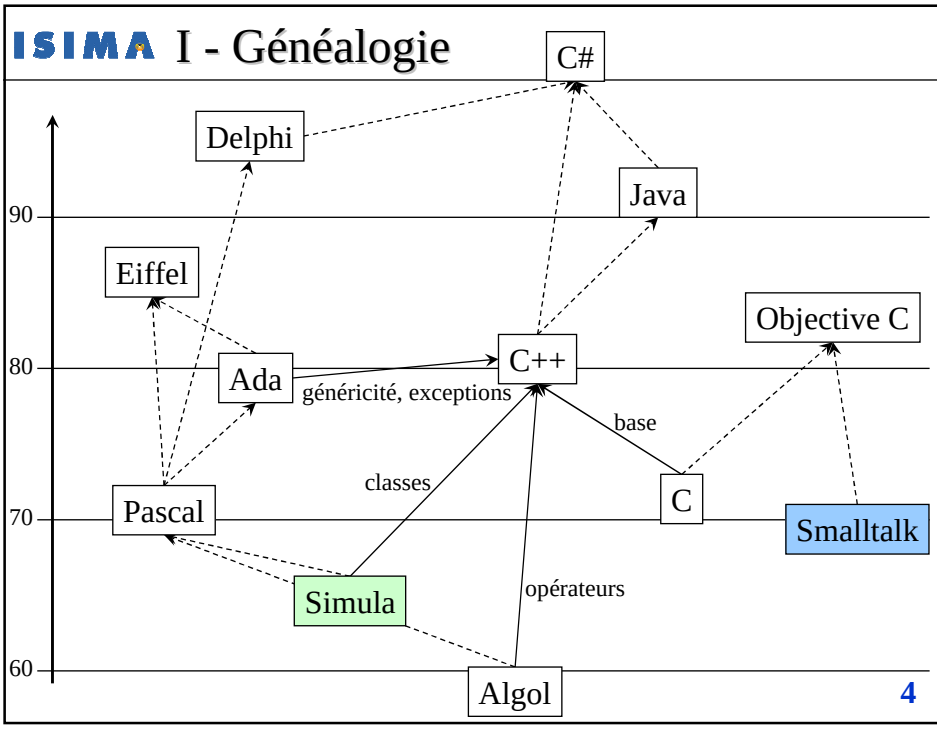
ISIMA Plan général

- Caractéristiques générales
- Les améliorations par rapport au C
- La POO en C++
  - classes
  - organisation
  - cycle de vie des objets
  - agrégation
  - héritage

ISIMA I - Caractéristiques générales

- Origines
  - travaux de Bjarne Stroustrup (AT&T Bell)
  - « C with classes » (80) → C++ (83)
  - normalisation en 98 (ISO/IEC 98-14882)
  - norme actuelle C++ 03, draft C++ 0x
- Langage orienté objet (← SIMULA 67)
  - typage fort
  - maintien des types primitifs et des fonctions
- Support de la généricité et des exceptions (←ADA 79)
- Surcharge des opérateurs (←ALGOL 68)

ISIMA I - Généalogie



ISIMA II - Améliorations sur le C

- inclusions
- commentaires
- surcharge de fonctions
- valeur par défaut de paramètres
- paramètres muets
- constantes
- type référence
- variables de boucle
- gestion de la mémoire

- opérateurs de transtypage efficaces
- gestion des types à l'exécution
- espaces de nommage

en troisième année !

ISIMA II.a - Inclusions

- Suppression de l'extension (normalisation)
  - avant (C) `#include <iostream.h>`
  - après (C++) `#include <iostream>`

- Librairies de la bibliothèque standard C renommées
  - ajout d'un 'c' devant le nom
  - avant (C) `#include <stdio.h>`
  - après (C++) `#include <cstdio>`

- Attention : certains (vieux compilateurs) ne respectent pas cette norme

ISIMA II.b - Commentaires (1/2)

- Nouveau commentaire monoligne
  - syntaxe `… // reste de la ligne commenté`
  - utile pour commenter le code
  - ne pas les utiliser en C !

- Commentaire multiligne du C toujours valide
  - syntaxe

```
/* Début de commentaire
…
// commentaire monoligne imbriqué
…
Fin de commentaire */
```

  - ne pas imbriquer
  - utile pour désactiver une portion de code

ISIMA II.b - Commentaires (2/2)

- Utilisation des directives du préprocesseur
  - exemple avec `#if` / `#endif`

```
#if 0
  Portion de code non évaluée
#endif
```

  - code non évalué car la condition du `#if` est nulle
  - ne pas trop utiliser, en général

- Utilité en C++
  - éviter les inclusions multiples d'en-tête
  - trio `#ifndef` / `#define` / `#endif`

ISIMA II.c – Surcharge de fonction

- Forme faible du polymorphisme
  - signature = type de retour + nom + paramètres + modificateurs
  - surcharge lorsque même nom et même type de retour
  - différence sur le nombre / type des arguments (+ `const`)
  - exemple

```
float moyenne (float, float);
float moyenne (float, float, float);
float moyenne (int, float[]);
```

  - recherche du meilleur matching par le compilateur
  - attention aux promotions de types !

ISIMA II.d - Valeur par défaut (paramètre)

- Simplifier les appels en omettant les paramètres lorsque la valeur de ceux-ci est la plus courante
- Syntaxe
  - préciser la valeur par défaut dans la déclaration

```
void show (Window * wnd, int mode = DEFAULT_SIZE);
```

  - ne pas la rappeler dans la définition

```
void show (Window * wnd, int mode)
{
  // implémentation de l'affichage
}
```

  - appels

```
show (wnd); // mode par défaut
show (wnd, ICONIFIED);
```

ISIMA II.d - Valeur par défaut (paramètre)

- Règle
  - si un paramètre possède une valeur par défaut, alors tous ses successeurs aussi
  - raison : lever les ambiguïtés possibles à la compilation
  - exemple

```
float f (float x = 0., float y , float z = 0.);
```

    - pourrait être appelé comme

```
res = f (0.); // ok
res = f (1., 1.); // pbm : qui reçoit 1 ? x ou z ?
```


ISIMA II.e - paramètres anonymes

- Avoir des paramètres muets
  - compatibilité d'appel d'une version sur l'autre
  - éviter les warnings de compilation
  - respect de conventions (API, `main()`, …)
- Syntaxe

```
int func (type p1, type, type p2) {…}
```

- Grand classique
  - la fonction `main()` prend 2 paramètres (voire plus)
  - si on ne les utilise pas

```
int main (int, char **) {…}
```

  - plus de manipulation de `argc` et `argv` → sécurité

ISIMA II.f - Constantes

- Utilité de `const` limitée en C

```
char c;
const char * p = ''toto'';
char * const q = &c;
// p = ptr sur caract. const.
// q = ptr const. sur caract.
```

- Usage renforcé en C++
  - définir de véritables constantes numériques
  - connues par le compilateur (typechecking…)
  - exemple

```
const int TAILLE = 5;
int tableau[TAILLE];
```

- Autres usages avec les objets

Plus jamais de `#define` pour les constantes !!!

ISIMA II.g – Déclaration de variables

- En C (C90)
  - variables locales déclarées en début de fonction
  - nécessite de penser à toutes les variables au préalable
- En C++
  - peut déclarer les variables à tout moment
  - conseillé de les déclarer en début de bloc

```
if (dblptr == 0)
{
  int n = computeSize ();
  dblptr = new double[n][n];
  …
}
```

  - portée (et durée de vie) limitée au bloc
- Repris dans le C99

ISIMA II.g – Déclaration de variables

- Déclarer un compteur dans le premier bloc du `for`
  - variable limitée à la boucle
  - une seule variable possible

```
int i = 5;
int j = 2;
for (int i = 0; i < 10; i++)
{
  j += i; // le i local
}
cout << i << endl; // i=5 ! le i global
```

- Autres possibilités
  - dans les conditions du `if`, `while` et `switch`
  - attention à la lisibilité !

ISIMA II.h - Type référence (1/4)

- En C, paramètres uniquement passés par valeur
  - pour un passage en mode in/out
    - passe (par valeur) l'adresse de la variable
    - terme « passage par adresse » impropre
  - conséquences
    - code peu lisible, passage de pointeurs, source d'erreurs
- En C++, utilisation de références (`&`)
  - pointeur masqué, simulant le passage par référence
  - utilisation identique à une variable
  - pas repris en C99

ISIMA II.h - Type référence (2/4)

- à la mode C

```
void swap (int * a, int * b)
{
  int c = *b;
  *b = *a;
  *a = c;
}

int main (int, char **)
{
  int i = 5, j = 6;
  swap (&i, &j);
  return 0;
}
```

- à la mode C++

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

int main (int, char **)
{
  int i = 5, j = 6;
  swap (i, j);
  return 0;
}
```


ISIMA II.h - Type référence (3/4)

- Avantages
  - code plus lisible
  - appel plus simple
  - moins d'erreurs
  - efficace
- Inconvénients
  - syntaxe ambiguë à cause de &
  - peu évident à comprendre au départ

ISIMA II.h - Type référence (4/4)

- Variable référence
  - se déclare « comme » un pointeur
  - se comporte comme un alias sur l'objet
  - nécessite un objet référencé
  - ne peut changer d'objet

```
int i = 5;
int & j = i;
j = 4; // maintenant i=4 !
```

- Type référence
  - une référence est toujours liée à une variable
  - elle ne peut être liée à une constante
  - la référence nulle n'existe pas !

ISIMA II.i – Allocation dynamique

- En C, couple **`malloc`** / **`free`**
- En C++, couple **`new`** / **`delete`**
  - pour allouer une donnée

```
int * iptr = new int;
…
delete iptr;
```

  - pour allouer un vecteur

```
int * iptr = new int[10];
…
delete[] iptr;
```

  - syntaxe plus simple
  - réalise aussi l'appel au constructeur (cf. objets)

Plus de **`malloc`** / **`free`** !!!

ISIMA II.i – Entrée / Sorties

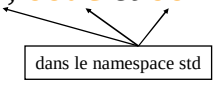
- En C, couple **printf** / **scanf** (et consorts)
- En C++
  - mécanisme de flux (objets) d'E/S : **cin**, **cout** et **cerr**
  - définis dans **iostream**
  - pour lire depuis le flux en entrée

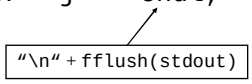
```
double x;
int j;
cin >> x >> j;
```

  - pour envoyer dans le flux en sortie (afficher)

```
double x;
int j;
cout << x << " + " << j << " = " << x + j << endl;
```

  - syntaxe plus simple
  - formatage évolué avec les **ios**





ISIMA II.j – Divers…

- Type booléen
  - existe de manière implicite en C (corrigé en C99)
  - type **bool**
  - mots-clé **true** et **false**
- Transtypage
  - en C `(type)expression`
  - en C++ `type(expression)`
  - notation fonctionnelle plus conforme à la sémantique
- Pointeur vide
  - en C, macro **NULL** `int * iptr = NULL;`
  - en C++, valeur **0** `int * iptr = 0;`

ISIMA Questions

```
#include <iostream.h>
#include <cstdio>

/* On ne nous dit pas tout */
// On ne nous dit pas tout

const int TAILLE = 10;
char chaine[TAILLE];

for (int i=0; i<10; i++)
{
      j+=i;
}

cout << " Bonjour " << endl;
```


ISIMA III - La POO en C++

- Les classes en C++
- Les relations entre classes
  - héritage
  - agrégation
  - association ?
- La généricité
  - fonctions
  - classes
- Les exceptions
- Eléments avancés ⇨ troisième année !

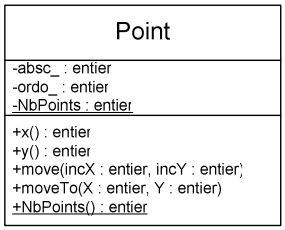
ISIMA III.a - Les classes en C++

- Syntaxe générale
- Déclaration
- Définition/ implémentation des méthodes
- Quelques améliorations
- Cycle de vie des objets

ISIMA Déclaration de la classe

- Débute par `class NomDeLaClasse`
- Contient les attributs et les prototypes des méthodes
- Modificateurs d'accès :
  - `public` : membre accessible universellement
    - ⇨réservé exclusivement aux méthodes de l'interface
  - `private` : visible seulement par les méthodes de la classe
    - ⇨tous les attributs
    - ⇨méthodes non destinées à l'utilisateur
  - `protected` : cf. plus loin

ISIMA Déclaration de classe




```
class Point
{
  private:
    int absc;
    int ordo;
    static int nb_points;
  public:
    Point(int x, int y);
    int x(void) const;
    int y(void) const;
    void move(int,int);
    void moveTo(int,int);
    static int nbPoints(void);
};
```

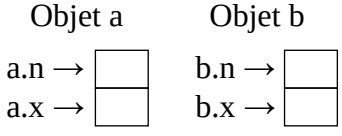
attention!

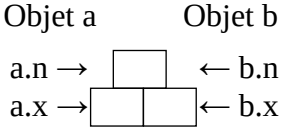
ISIMA Membres statiques (static)

- Ce sont des membres dont la portée est limitée à la classe
- Il n'existe qu'un exemplaire, indépendamment des objets de la classe

```
class exemple1
{
    int n;
    float x;
    …
};
exemple1 a,b;
```

```
class exemple2
{
    static int n;
    float x;
    …
};
exemple2 a,b;
```





ISIMA Membres statiques (static)

- Un membre statique doit être initialisé explicitement (à l'extérieur de la déclaration de la classe)

```
class exemple2
{
    static int n=6;
    float x;

    …
};
```

```
class exemple2
{
    static int n;
    float x;

    …
};
int exemple2::n=5;
```

Erreur

ISIMA Déclaration de classe

```
class Point
{
  private:
    int absc;
    int ordo;
    static int nb_points;

  public:
    Point(int x, int y);
    int x(void) const;
    int y(void) const;
    void move(int,int);
    void moveTo(int,int);
    static int nbPoints(void);
};
```

ISIMA Définition de classe

```
Point::Point(int x, int y)
{
  absc = x;
  ordo = y;
  Point::nb_points++; // ou aussi nb_points++
}
int Point::x(void) const
{
  return absc;
}
void Point::move(int incX, int incY)
{
  absc += incX;
  ordo += incY;
}
static int Point::nbPoints(void)
{
  return nb_points;
}
int Point::nb_points = 0; // attribut de classe
```

Opérateur de résolution de portée

ISIMA Les méthodes constantes

- Utilisation du mot clef **`const`** en fin de signature
- Méthodes ne modifiant pas l'objet (attributs)
- Limité aux méthodes d'instance

- Avantages
  - utilisable sur un objet **`const`**
  - la méthode ne peut modifier les attributs
  - contrôlé à la compilation

- Signification plus subtile

ISIMA Les méthodes constantes

- Utilisable pour un objet déclaré constant
- Les instructions dans sa définition ne doivent pas modifier la valeur de l'objet concerner.

```
Point a;
const Point b;

a.x(…);          // OK
b.x(…);          // OK
a.move(…);       // OK
b.move(…);       /* ERREUR, b est constante
                    et pas Point::move()
                 */
```


ISIMA Amélioration : méthodes inline

- Dommage de réaliser un appel de méthode pour
  - récupérer la valeur d'un attribut
  - effectuer un traitement simple
- Méthode `inline` : développée comme une macro
- S'applique aussi aux fonctions

👍 Rapidité d'exécution

👎 augmentation taille exécutable ⇨ méthodes courtes

👎 implémentation dans partie déclaration

ISIMA Implémentation inline

- Deux solutions
  - définition avec la déclaration

```
int x(void) const
{
  return absc;
}
```

  - utilisation du mot clef `inline`

```
inline int y(void) const; //déclaration
...
inline int Point::y(void) const //définition
{
  return ordo;
}
```


ISIMA Amélioration : structure du code source

- Fichier header
  - déclaration de la classe
  - méthodes inline

```
#ifndef __CLASSE_H__
#define __CLASSE_H__

// includes
// forward déclaration

class Classe
{
  // attributs
  // proto méthodes
  // méthodes inline
};
#endif
```

- Fichier implémentation
  - définition variables de classe
  - définition méthodes

```
#include "classe.h"

// init. des variables
// de classe

// définition des méthodes
// externalisées
```

ISIMA Cycle de vie des objets

1. Construction
   - réservation mémoire
   - appel d'un constructeur

2. Vie
   - appel des méthodes

3. Destruction
   - appel du destructeur
   - libération mémoire

ISIMA Les constructeurs

- Rôle : initialiser les objets
- Syntaxe
  - même nom que la classe
  - pas de type de retour (ni même `void`)
  - surcharge à volonté
  - pas d'héritage, mais appel (implicite ou explicite) à un constructeur de la classe mère
  - un élément particulier, la liste des initialisations

```
Point::Point() {…}
Point::Point(int x, int y) {…}
Point::Point(const Point & p) {…}
```

ISIMA Les constructeurs

- On suppose que Point a le(s) constructeur(s) suivante(s) … quelles déclarations sont correctes ou incorrectes ?

EXEMPLE 1

```
// Constructeur
Point (int, int);

// Declarations
Point a;        // Erreur
Point b(3);     // Erreur
Point c(1,7);// OK
```

EXEMPLE 2

```
// Constructeurs
Point ();
Point (int, int);

// Declarations
Point a;        // OK
Point b(5);     // Erreur
Point c(1,7);// OK
```


ISIMA Les constructeurs

- On suppose que Point a le(s) constructeur(s) suivante(s) … quelles déclarations sont correctes ou incorrectes ?

EXEMPLE 3

```
// Constructeurs
Point (int x=0, int y=0);

// Declarations
Point a;        // OK  arguments 0 et 0
Point b(5);     // OK  arguments 5 et 0
Point c(1,7);   // OK  arguments 1 et 7
```

ISIMA Exemple de flot de vie d'un objet

```
Point::Point(int x=0, int y=0)
{
  …
  cout << " Création du point : " << x << "\t " << y << endl;

}

Point::~Point()
{
  cout << " Destruction du point : " << x << "\t " << y << endl;
}

Point::affiche()
{
  cout << " Je suis à la position : " << x << "\t " << y << endl;
}
```


ISIMA Exemple de flot de vie d'un objet

```
main ()
{
  Point a(5,2);
  a.affiche();
  Point b(1,-1);
  b.affiche();
  a.move(-2,4);
  a.affiche();
}
```

Création du point : 5 2
Je suis à la position : 5 2
Création du point : 1 -1
Je suis à la position : 1 -1
Je suis à la position : 3 6
Destruction du point : 1 -1
Destruction du point : 3 6

ISIMA La liste d'initialisation du constructeur

- Avantage : les initialisations sont faites à la création des objets (début de vie). D'autres avantages cf. plus loin.
- Syntaxe

```
[: attribut(expression){, attribut(expression)}^n]
```

```
class Point
{
  private:
    int absc;
    int ordo;
    static int nb_points;

  …
};
```

```
Point::Point(int x, int y) :
  absc(x), ordo(y), nb_points++
{
…
}
```


ISIMA La liste d'initialisation du constructeur

- Propriétés
  - traitée avant le code du constructeur
  - expression pas limitée à une valeur
    - expressions arithmétiques
    - fonctions
  - initialisation des attributs **dans l'ordre de déclaration**
  - possibilité d'utiliser la liste et le code
  - **obligatoire** pour les attributs de type référence (voir plus loin)
  - c'est l'endroit pour appeler le constructeur de la classe mère (voir plus loin) ; **si pas mis, appel au constructeur par défaut de la classe mère**

ISIMA La liste d'initialisation du constructeur

```
class Rationnel
{
  private:
    int num;
    int den;

  public:
    Rationnel(int n=0, int d=1):
      den(d), num(n)
    {…}
};
```

- initialisation plus complexe
  - ajout d'un attribut distance
  - distance du point à l'origine

avec code

```
Point::Point(int x, int y) :
  absc(x), ordo(y)
{
  dist = sqrt(x*x+y*y);
}
```

solution

```
Rationnel::Rationnel(int n=0,
                     int d=1):
  num(n), den(d)
{…}
```

ou tout dans la liste

```
Point::Point(int x,int y) :
  absc(x), ordo(y),
  dist(sqrt(x*x+y*y))
{}
```


ISIMA Création d'objets

- Trois classes d'allocation (comme en C)
  - automatique : variable locale sur la pile
  - statique : variable globale, variable locale statique
  - dynamique : variable allouée sur le tas
    - allocation avec **new** + constructeur
    - destruction avec **delete**
- Gestion mémoire
  - statique et automatique : système
  - dynamique : construction / destruction à la main

ISIMA Exemples d'instanciation

```
Point p1(120, 13);          // objet statique
const int TAILLE = 5;       // constante entière

int main (int, char **)
{
  Point p2(12, 15);         // objet automatique
  Rationnel r1;             // objet automatique, valeur par défaut
  Rationnel r5[TAILLE];     // tableau de taille TAILLE 5
                            // tous initialisés par défaut
  Rationnel r2();           // prototype de fonction !

  Rationnel * r;            // pointeur
  r = new Rationnel (10,25); // création sur le tas
  delete r;                 // destruction

  return 0;
};
```


ISIMA Création et destruction d'objets

```
class Point {…};
Point c(1,1);
main ()
{
  cout << "Début du main" << endl;
  Point b(10,10);
  for(int i=0;i<2;i++)
  {
    cout << "iter" << i << endl;
    Point b(i,2*i);
  }
  cout << "Fin du main" << endl;
}
```

Création du point : 1 1
Début du main
Création du point : 10 10
iter 0
Création du point : 0 0
Destruction du point : 0 0
iter 1
Création du point : 1 2
Destruction du point : 1 2
Fin du main
Destruction du point : 10 10
Destruction du point : 1 1

ISIMA Les destructeurs

- Rôle : finalise la vie d'un objet
- Syntaxe
  - même nom que la classe
  - pas de type de retour (ni même `void`)
  - pas possible de réaliser surcharge (pourquoi?)
  - pas d'héritage, mais appel au destructeur de la classe mère à la fin
  - il existe une définition par défaut

```
~Point() {…}
~Pixel() {…}
```