

**PARTIE II**

**Rappels de C++**

**Christophe Duhamel Bruno Bachelet**

## Caractéristiques générales

### Historique

* + Héritage des autres langages
* POO en C++
  + Définition d'une classe
  + Cycle de vie des objets
  + Relations entre classes
* Autres concepts
  + Généricité
  + Exceptions
  + Surcharge d'opérateurs
* Origines
  + Travaux de Bjarne Stroustrup (AT&T Bell)
  + *«C with classes»* (80) → C++ (83)
  + Normalisation en 98 (ISO/IEC 98-14882), norme C++ 98
  + Depuis 2003, norme C++ 03
  + Mi-2011, nouvelle norme C++ 11
  + Mi-2014, nouvelle norme C++ 14

### 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)

**Généalogie**

C#

Delphi

Java

90

### Eiffel

80

70

Pascal

généricité, exceptions

classes

Ada

C++

### Simula

base

opérateurs

### Objective C

Smalltalk

C

60

**ment logiciel - ISIM**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Algol |  |  |
| **Méthodes et outils de développe** | | **A / ZZ3 - 2018-2019 32** | |

# En C, paramètres uniquement passés par valeur

## Passage en mode *in/out*  «passage par adresse»

##### Passe (par valeur) l'adresse de la variable

* + Conséquences
    - Code peu lisible, passage de pointeurs, source d'erreurs
* En C++, utilisation de références (**&**)
  + Référence = nouvel alias d'une variable
  + Utilisation identique à une variable
  + Pointeur masqué, simulant le passage par référence
  + Pour les méthodes *inline*, vrai passage par référence
* A la mode C  A 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;**

**}**

**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;**

**}**

* Avantages
  + Code plus lisible
  + Appel plus simple
  + Moins d'erreurs
  + Efficace
* Inconvénients
  + Syntaxe ambiguë à cause de «**&**»
  + Peu évident à comprendre au départ
* Déclaration d'une référence
  + Se déclare «comme» un pointeur
  + Se comporte comme un alias sur l'objet
  + Nécessite un objet référencé à la déclaration
  + Ne peut changer d'objet par la suite

**int i = 5; int & j = i;**

**j = 4; // maintenant i=4 !**

* Référencer quoi ?
  + Une référence non constante est toujours liée à une variable
  + Une référence constante peut être liée à une constante
    - **const int & j = 4;**
  + La référence nulle n'existe pas !

**Règles d'usage des types: *const*, référence ?**

Passage d'arguments

|  |  |  |
| --- | --- | --- |
|  | Type primitif T | Classe C |
| Argument variable | **T & arg** | **C & arg** |
| Argument constant | **T arg**  ou  **const T & arg** | **const C & arg** |

Retour de variable

|  |  |  |
| --- | --- | --- |
|  | Type primitif T | Classe C |
| Retour (mode lecture) d'un attribut | **T m(…) const;**  ou  **const T & m(…) const;** | **const C & m(…) const;** |
| Retour (mode lecture/écriture) d'un attribut | **T & m(…);** | **C & m(…);** |
| Retour d'un résultat produit par une méthode | **T m(…) const;** | **C m(…) const;** |

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

**int \* iptr = new int;**

**…**

**delete iptr;**

* + - Pour allouer un tableau

**int \* iptr = new int[10];**

**…**

**delete[] iptr;**

* + - Réalisent aussi la construction / destruction
      * **new** = allocation mémoire + appel constructeur
      * **delete** = appel destructeur + libération mémoire
    - Plus de **malloc** / **free** !
* En C, couple **printf** / **scanf** (et consorts)
* En C++, mécanisme de flux

### Bibliothèque standard (namespace «**std**»)

* + Flux standard **std::cin**, **std::cout** et **std::cerr**
  + Inclusion de **<iostream>**
    - Pour éviter de rappeler le namespace: **using namespace std;**

### Pour lire depuis le flux en entrée

**double x; int j; cin >> x >> j;**

* + Pour envoyer dans le flux en sortie

**double x; int j;**

**cout << x << " + " << j << " = " << x + j << endl;**

* + Pour les fichiers: **<fstream>**
    - Les classes en C++

##### Déclaration / définition

* + - * Cycle de vie des objets
    - Les relations entre classes
      * Agrégation
      * Héritage
      * Association
    - La généricité
      * Fonctions
      * Classes
    - Les exceptions
    - Les opérateurs
* Mot-clé «**class**»

##### Contient les attributs et les prototypes des méthodes

* Modificateurs d'accès
  + **public**: membre accessible par tous
    - Réservé exclusivement aux méthodes de l'interface
  + **private**: membre accessible aux méthodes de la classe
    - Pour les attributs
    - Pour les méthodes non destinées à l'utilisateur
  + **protected**: membre accessible aux méthodes de la classe et de ses sous-classes
    - Assouplit l'accès privé à des fins de redéfinition dans les sous-classes
* Modificateur «**static**»
  + Définit un membre de classe

**class Point**

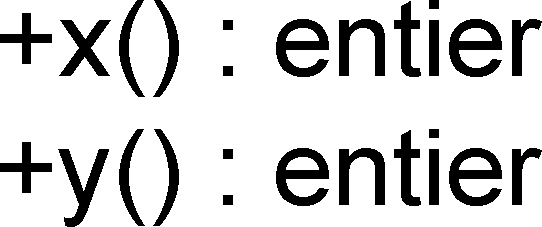
**{**

**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 !

**Point::Point(int x, int y) { absc = x;**

**ordo = y; nb\_points++;**

**}**

**int Point::x(void) const { return absc; } void Point::move(int incX, int incY) {**

**absc += incX; ordo += incY;**

**}**

**int Point::nbPoints(void) { return nb\_points; } int Point::nb\_points = 0; // Attribut de classe**

* Appel méthode  coût d'exécution
* Parfois, dommage d'utiliser un appel de méthode
  + Pour récupérer la valeur d'un attribut
  + Pour un traitement simple
* Méthode *«inline»*: développée «comme» une macro
  + S'applique aussi aux fonctions
* Avantage
  + Rapidité d'exécution (coût appel + optimisation supplémentaire)
* Inconvénients
  + Augmentation taille exécutable
    - A utiliser donc sur des méthodes courtes
  + Implémentation dans la partie déclaration de la classe
    - Ou dans un fichier d'entête

## Implémentation

##### Définition avec la déclaration

**class Point {**

**…**

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

**…**

**};**

* + Utilisation du mot-clé «**inline**» (indication au compilo)

**inline int Point::x(void) const { return absc; }**

* Fichier entête
  + Déclaration de la classe
  + Définition méthodes *«inline»*

##### Fichier implémentation

* + Définition variables de classe
  + Définitions méthodes

**#ifndef**

**#define**

**CLASSE\_H**

**CLASSE\_H**

**// includes**

**// déclarations anticipées**

**class Classe**

**{**

**// attributs**

**// proto méthodes**

**// méthodes inline**

**};**

**#endif**

**#include "classe.h"**

**// init. des variables**

**// de classe**

**// définition des méthodes**

**// externalisées**

# Construction

## Réservation mémoire

* + - * Appel d'un constructeur

 Vie

## Appel des méthodes

* + - Destruction
      * Appel du destructeur
      * Libération mémoire
* Rôle: initialiser les objets
* Syntaxe
  + Même nom que la classe
  + Pas de type de retour
  + Surcharge à volonté
  + Une particularité: la liste d'initialisation
* Exemples

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

##### Syntaxe

* + *nom\_classe****(****…****)* :** *liste\_initialisation* **{***…***}**
  + Liste = *nom\_attribut***(***valeur***),***nom\_attribut***(***valeur***)**…
  + Les valeurs peuvent être des expressions
    - Calcul, appel de fonction…

##### Rôle: initialisation des attributs d'un objet

* + Même sans liste, initialisation avant le bloc de code

##### Construction de chacun des attributs

* + Dans l'ordre de déclaration
  + Donc, il faut lister les attributs dans l'ordre de déclaration
  + Si un attribut est omis dans la liste  construction par défaut
  + Les attributs de type référence obligatoirement dans la liste
* Respecter l'ordre des attributs
* Initialisation plus complexe
  + Ajout d'un attribut *distance*
  + Distance du point à l'origine

avec code

**class Rationnel**

**{**

**private: int num; int den;**

**};**

**public:**

**Rationnel(int n=0, int d=1)**

**: den(d), num(n)**

**{}**

**Point::Point(int x, int y) : absc(x), ordo(y)**

**{**

**dist = sqrt(x\*x+y\*y);**

**}**

Solution ou tout dans la liste

**Rationnel::Rationnel(int n=0,**

**int d=1)**

**: num(n), den(d)**

**{}**

**Point::Point(int x, int y) :**

**absc(x), ordo(y), dist(sqrt(x\*x+y\*y))**

**{}**

## Trois types d'allocation (comme en C)

##### Statique: variable globale, variable locale statique

* + - * Automatique: variable locale sur la pile
      * Dynamique: variable allouée sur le tas
        + **new** = allocation mémoire + appel constructeur
        + **delete** = appel destructeur + libération mémoire

## Gestion mémoire

##### Statique et automatique: par le système

* + - * Dynamique: par le programmeur
* Moment de la construction
  + Variables globales: avant l'exécution du «**main**»
  + Variables locales: à l'entrée dans le bloc
    - Variables locales statiques: à la 1ère entrée

##### Variables dynamiques: à l'exécution de «**new**»

* Moment de la destruction
  + Variables statiques: après la sortie du «**main**»
    - Même chose pour les variables locales statiques

##### Variables locales sur la pile: à la sortie du bloc

* + Variables dynamiques: à l'exécution de «**delete**»
* Utilisation du mot-clé «**const**» en fin de prototype
* Indique les méthodes ne modifiant pas l'objet
  + Qui ne modifient pas les attributs
* Limité aux méthodes d'instance
* Avantages
  + Seules méthodes utilisables sur un objet constant
    - Une méthode «non constante» ne peut pas être exécutée

##### La méthode ne peut pas modifier les attributs

* + Contrôlé à la compilation
* Signification plus subtile
  + «**const**» fait partie de la signature
  + Possibilité de définir deux versions
* Définition d'accesseurs (version 1 – recommandée)

**class Exemple { protected: string s;**

**public:**

**const string & getS(void) const { return s; } void setS(const string & x) { s=x; }**

**};**

* Utilisation d'accesseurs

**Exemple e1; const Exemple e2;**

**e1.setS("nawouak"); ** ok

**e2.setS("nawouak"); ** problème

**std::cout << e1.getS() << std::endl; ** ok

**std::cout << e2.getS() << std::endl; ** ok

* Définition d'accesseurs (version 2 – non recommandée)

**class Exemple { protected: string s;**

**public:**

**const string & getS(void) const { return s; } string & getS(void) { return s; }**

**};**

* Utilisation d'accesseurs

**Exemple e1; const Exemple e2;**

**e1.getS() = "nawouak"; ** ok

**e2.getS() = "nawouak"; ** problème **std::cout << e1.getS() << std::endl; ** ok **std::cout << e2.getS() << std::endl; ** ok

* Regrouper un ou plusieurs objets dans un autre = les attributs
* Trois manières d'agréger / trois types d'attributs
  + Attribut objet: construit en même temps que l'objet
  + Attribut référence: initialisation obligatoire dans le constructeur
    - Pas de changement par la suite
  + Attribut pointeur: peut être initialisé n'importe quand
    - Attention à la forme normale de Coplien
    - Si la mémoire de l'attribut est gérée par la classe

##### Vie de l'objet agrégé

* + Objet construit par l'agrégeant
    - Attribut objet ou pointeur
  + Objet en provenance de l'extérieur
    - Recopie: attribut objet
    - Référence: attribut pointeur ou référence
      * Une classe peut hériter d'une ou plusieurs autres
        + **class** *derivee* : *modificateur mere1***,** *modificateur mere2*…
        + Modificateur  limitation de visibilité
      * Visibilité de l'héritage : qui voit l'héritage ?
        + **public**  tout le monde
        + **protected**  classes filles uniquement
        + **private**  classe mère uniquement
        + Perte du lien de parenté  plus de conversion ascendante
      * Visibilité des membres de la classe mère

|  |  |  |  |
| --- | --- | --- | --- |
| Visibilité dans classe mère | Visibilité dans classe fille | | |
| Héritage  «**public**» | Héritage  «**protected**» | Héritage  «**private**» |
| **public** | **public** | **protected** | **private** |
| **protected** | **protected** | **protected** | **private** |
| **private** | **private** | **private** | **private** |

* Modificateur d'accès **protected**

##### Visible des classes fille mais pas de l'extérieur

* Utilisation classique de l'héritage
  + Attributs **protected** + héritage **public**
* Passer les attributs **private** en **protected** ?

##### Avantage: accessibles directement

* + Inconvénient: violation de l'encapsulation
    - Problèmes de maintenabilité si héritage en cascade
    - Solution: méthodes protégées pour l'accès aux attributs
* Héritage privé  perte de l'interface
* Utilisation 1: s'approprier l'implémentation

### Mais l'héritage n'a pas forcément de sens

* + L'agrégation peut être utilisée à la place
  + A éviter donc dans ce but
* Utilisation 2: proposer une nouvelle interface
  + Modéliser un *«wrapper»*

### Solution possible: l'agrégation

* + Héritage privé  solution sans agrégation
* Rendre une méthode polymorphe (virtuelle): **virtual**
  + Virtuelle un jour, virtuelle toujours !
    - Mot-clé «**virtual**» pas nécessaire dans les sous-classes

### Peut être redéfinie dans les sous-classes

* Classe abstraite en C++
  + Pas de mot-clé
  + Classe abstraite  au moins une méthode abstraite
  + Méthode abstraite = méthode virtuelle pure
    - Pas de code
    - **virtual** *type\_retour nom\_méthode***(***arguments***) = 0;**

### Redéfinir impérativement dans les sous-classes

* + - Car tant qu'une méthode est abstraite  pas d'instanciation
      * Appeler l'implémentation de la classe mère
        + *classe\_mère***::***nom\_méthode***(***arguments***)**
      * Exemple : compléter l'implémentation de la classe mère

**class Personne {**

**…**

**virtual void afficher(void) const**

**{ cout << nom << " " << prenom; }**

**…**

**};**

**class Etudiant : public Personne {**

**…**

**void afficher(void) const { Personne::afficher();**

**cout << " " << ecole;**

**}**

**…**

**};**

* Les constructeurs ne peuvent pas être virtuels
  + Pas d'héritage des constructeurs
  + Mais séquence de construction prédéfinie
* Exemple: B hérite de A
  + Construction B = Construction A, puis construction attributs de B
  + **class A {**

**protected: string s;**

**public:**

**A() { s=…; }**  **A() : s() { s=…; }**

**A(const string & ss) : s(ss) {}**

**};**

**class B : public A { protected: string t;**

**public:**

**B() { s=…; t=…; }**  **B() : A(),t() { s=…; t=…; }**

**B(const string & ss, const string & tt)**

**: A(ss),t(tt) {}**

**};**

* Méthode virtuelle  destructeur virtuel

##### Destruction impérativement polymorphe

* + Exemple

**vector<Point \*> v;**

**…**

**for (int i=0; i<v.size(); ++i) delete v[i];**

* Si destructeur polymorphe
  + Appel destructeur sous-classe
  + Puis appel destructeur super-classe
* Si destructeur non-polymorphe
  + Appel destructeur super-classe  incohérent !
* Héritage en diamant

**A**

**B**

**C**

**C**

**D**

* Duplication des attributs
  + **class A**

**{ A(…) {} };**

**class B : public A**

**{ B(…) : A(…) {} };**

**class C : public A**

**{ C(…) : A(…) {} };**

**class D : public B, public**

**{ D(…) : B(…), C(…) {} };**

* + 2 appels au constructeur de A dans D

 attributs de A dupliqués dans D

* Collision des noms de méthode (ou attribut)
  + Exemple: méthode **A::x()**
  + **D::x()** signifie appel sur l'objet A issu de B ou de C ?
    - Distinction possible via **B::x()** ou **C::x()** ou conversion vers **B &** ou **C &**

##### Solution: héritage «virtuel»

* + **class A**

**{ A(…) {} };**

**A**

**virtual virtual**

**class B : virtual public A**

**{ B(…) : A(…) {} }; B C**

**class C : virtual public A**

**{ C(…) : A(…) {} };**

**class D : public B, public C D**

**{ D(…) : A(…), B(…), C(…) {} };**

* + Une seule copie de A
  + Appel explicite au constructeur de A dans D
  + Paramètres destinés à A ignorés dans les constructeurs de B et C

##### Autres solutions: héritage d'interfaces ou délégation

###### Constructeurs

* + Constructeur par défaut
    - **A(void);**
  + Constructeur de copie
    - **A(const A &);**

###### Affectation (méthode)

* + **A & operator = (const A & x) {**

**… // Recopie de x dans «this» return (\*this);**

**}**

* + Retour de l'objet pour chaînage: **a = b = c;**

###### Opérations arithmétiques / logiques binaires (fonctions)

* + **A operator + (const A & x, const A & y) { A resultat;**

**… // Calcul de x+y return resultat;**

**}**

* + **bool operator == (const A & x, const A & y);**
* Opérations arithmétiques unaires (méthodes)
  + Préfixé
    - **A & operator ++ () {**

**… // Incrémentation de «this» return \*this;**

**}**

* + - Retour de l'objet: **a = ++b;**

## Postfixé

#### A operator ++ (int) { A copie = \*this;

**… // Incrémentation de «this» return copie;**

**}**

##### Retour d'une copie avant incrément: **a = b++;**

* Opérateurs de flux (fonctions)
  + Ecriture
    - **ostream & operator << (ostream & flux,const A & x)**

**{**

**… // Ecriture de x dans le flux return flux;**

**}**

* + - Retour du flux: **f << a << b;**

##### Lecture

* + - **istream & operator >> (istream & flux, A & x) {**

**… // Lecture du flux dans x return flux;**

**}**

* + Ne jamais passer un flux par copie !
* Autres symboles
  + **()**, **[]**, **\***, **,** …