Introduction C++

Master 2 IF App

Julien Saunier

LEPSiS, INRETS/LCPC
saunier@inrets.fr

Le langage C++

(Julien Saunier, fortement inspiré du cours de Maude Manouvrier)

- Correspondances avec le C
- Programmation objet
- Templates
- STL

Qu'est-ce que le C++ ?

- **Permettre la programmation orientée-objet**
- **Compatibilité C/C++ [Alard, 2000] :**
  - ***C*++ = sur-ensemble de C**
  - **C++ ⇒ ajout en particulier de l'orienté-objet (classes, héritage, polymorphisme)**
  - **Cohabitation possible du procédural et de l'orienté-objet en C++**
- *Différences C++/Java [Alard, 2000] :*
  - *C++ : langage compilé / Java : langage interprété par la JVM*
  - *C/C++ : passif de code existant / Java : JNI (Java Native Interface)*
  - *C++ : pas de machine virtuelle et pas de classe de base / java.lang.object*
  - *C++ : "plus proche de la machine" (gestion de la mémoire)*

Bibliothèques

- math.h => cmath
- stdlib.h => cstdlib

- stdio.h => iostream
- Char* => string

Premiers pas en C++ (1)

```
#include <iostream>

int main()
{
    int iEntier1;
    cout << "Saisir un entier : " << endl; // Affiche à l'écran
    cin >> iEntier1; // Lit un entier

    cout << "iEntier vaut " << iEntier1 << endl; // endl = saut de
    ligne
    return 0;
}
```

Premiers pas en C++ (2)

- Héritage des mécanismes de bases du C (pointeurs inclus)
- Entrées/sorties fournies à travers la librairie *iostream*
  - ***cout << expr$_1$ << … << expr$_n$***
    - Instruction affichant *expr$_1$* puis *expr$_2$, etc.*
    - `cout` : « flot de sortie » associé à la sortie standard (*stdout*)
    - << : opérateur binaire associatif à gauche, de première opérande `cout` et de 2ème l'expression à afficher, et de résultat le flot de sortie
    - << : opérateur surchargé (ou sur-défini) ⇒ utilisé aussi bien pour les chaînes de caractères, que les entiers, les réels etc.
  - ***cin >> var$_1$ >> … >> var$_n$***
    - Instruction affectant aux variables *var$_1$*, *var$_2$, etc.* les valeurs lues (au clavier)
    - `cin` : « flot d'entrée » associée à l'entrée standard (*stdin)*
    - >> : opérateur similaire à <<

Premiers pas en C++ (3)

- Programme C++ généralement constitué de plusieurs modules, compilés séparément
- Fichier entête – d'extension .h (ou .hh ou .hpp)
  - Contenant les déclarations de types, fonctions, variables et constantes, etc.
  - Inclus via la commande `#include`
- Fichier source – d'extension .cpp ou .C

MonFichierEnTete.h

```
extern char* MaChaine;
extern void MaFonction();
```

MonFichier.cpp

```
#include <iostream>
#include "MonFichierEnTete.h"
void MaFonction()
{
 cout << MaChaine << « \n « ;
}
```

MonProgPirncipal.cpp

```
#include "MonFichierEnTete.h"
char *MaChaine="Chaîne à afficher";
int main()
{
  MaFonction();
}
```

Premiers pas en C++ (4)

- Utilisation d'**espaces de noms** (*namespace*) lors de l'utilisation de nombreuses bibliothèques pour éviter les conflits de noms
- **Espace de noms** : association d'un nom à un ensemble de variable, types ou fonctions

  Ex. Si la fonction *MaFonction()* est définie dans l'espace de noms *MonEspace*, l'appel de la fonction se fait par *MonEspace::MaFonction()*

- Pour être parfaitement correct :

```
std::cin
std::cout
std::endl
```

- **Pour éviter l'appel explicite à un espace de noms** : `using`

```
using std::cout
using namespace std;
```

Premiers pas en C++ (5)

- **Type booléen** : `bool`
- **Librairie STL (*Standard Template Library*)** : incluse dans la norme C++ - définition de conteneurs (liste, vecteur, file etc.)
- Un bon nombre de méthodes utiles (www.cppreference.com)

```
#include <string> // Pour utiliser les chaînes de caractères
#include <iostream>

using namespace std ;

int main()
{
   string MaChaine="ceci est une chaine";
  cout << "La Chaine de caractères \""<< MaChaine
   << "\" a pour taille " << MaChaine.size() << "." << endl;

  string AutreChaine("!!");
  cout << "Concaténation des deux chaines : \""
    << MaChaine + AutreChaine<<"\".«  << endl ;

  return 0;
}
```

Premiers pas en C++ (6)

- 2 opérateurs supplémentaires : **`new`** et **`delete`**

```
float *PointeurSurReel = new float;
// Équivalent en C :
//   PointeurSurReel = (float *) malloc(sizeof(float));
int *PointeurSurEntier = new int[20];
// Équivalent en C :
// PointeurSurEntier = (int *) malloc(20 * sizeof(int));
delete PointeurSurReel; // Équivalent en C : free(pf);
delete [] PointeurSurEntier; // Équivalent en C :  free(pi);
```

- **`new type`** : définition et allocation d'un pointeur de type *`type*`*
- **`new type [n]`** : définition d'un pointeur de type *`type*`* sur un tableau de *`n`* éléments de type *`type`*
- En cas d'échec de l'allocation, `new` déclenche une exception du type `bad_alloc`

Premiers pas en C++ (7)

- Appel de fonction toujours précédé de la déclaration de la fonction sous la forme de prototype
- Une et une seule définition d'une fonction donnée mais autant de déclaration que nécessaire
- **Par défaut, passage des paramètres par valeur** (comme en C) ou **par référence**
- **Possibilité de surcharger ou sur-définir une fonction**

```
int carre (int x) {return x * x;}
double carre  (double y) {return y * y;}
```

- **Possibilité d'attribuer des valeurs par défaut aux arguments**

```
void MaFonction(int i=3, int j=5); // Déclaration
int x =10, y=20;
MaFonction(x,y);      // Appel avec 2 argument
MaFonction(x);        // Appel avec 1 argument
MaFonction();               // Appel sans argument
```

Les arguments concernés doivent obligatoirement être les derniers de la liste

A fixer dans la déclaration de la fonction pas dans sa définition

Premiers pas en C++ (8)

- **Possibilité de définir une variable de type *référence***

```
int i = 5;
int & j = i; // j reçoit i
   // i et j désignent le même emplacement mémoire
```

Impossible de définir une référence sans l'initialiser

- **Déréférencement automatique** : Application automatique de l'opérateur d'indirection * à chaque utilisation de la référence

```
int i = 5;
int & j = i;    // j reçoit i
int k = j;      // k reçoit 5
j += 2;  // j reçoit i+2 (=7)
j = k;   // j reçoit k (=5)
```

Une fois initialisée, une référence ne peut plus être modifiée – elle correspond au même emplacement mémoire

- Initialisation : 2 notations possibles

```
int i = 5;
int i(5);
```

Premiers pas en C++ (9)

Passage des paramètres par référence

```
#include <iostream>

void echange(int&,int&);

int main()
{

   int n=10, p=20;
   cout << "avant appel: " << n << " " << p << endl;
   echange(n,p);   // attention, ici pas de &n et &p
   cout << "apres appel: " << n << " " << p << endl;
}

void echange(int& a, int& b)
{
   int c;
   cout << "debut echange : " << a << " " << b << endl;
   c=a; a=b; b=c;
   cout << "fin echange : " << a << " " << b << endl;
}
```

```
avant appel: 10 20
debut echange: 10 20
fin echange: 20 10
apres appel: 20 10
```

Classes et objets (1) : définitions

- **Classe :**
  - Regroupement de données (attributs ou champs) et de méthodes (fonctions membres)
  - Extension des structures (`struct`) avec différents niveaux de visibilité (`protected`, `private` et `public`)
- En programmation orientée-objet pure: encapsulation des données et accès unique des données à travers les méthodes
- **Objet** : instance de classe
  - Attributs et méthodes communs à tous les objets d'une classe
  - Valeurs des attributs propres à chaque objet
- **Encapsulation**
  - Caractérisation d'un objet par les spécifications de ses méthodes : interface
  - Indépendance vis à vis de l'implémentation

Classes et objets (2) : 1er exemple de classe

```
// Exemple de classe repris de [Deitel et Deitel,
  2001]

class Horaire{

  private : // déclaration des membres privés
            // private: est optionnel (privé par
  défaut)
      int heure;      // de 0 à 24
      int minute;     // de 0 à 59
      int seconde;    // de 0 à 59

  public : // déclaration des membres publics
      Horaire();       // Constructeur
      void SetHoraire(int, int, int);
      void Afficher();
};
```

Interface de la classe

Classes et objets (3) : 1er exemple de classe

```
// Exemple repris de [Deitel et Deitel, 2001]
Horaire::Horaire() {heure = minute = seconde = 0;}

void Horaire::SetHoraire(int h, int m, int s)
{
  heure = (h >= 0 && h < 24) ? h : 0 ;
  minute = (m >= 0 && m < 59) ? m : 0 ;
  seconde = (s >= 0 && s < 59) ? s : 0 ;
}
void Horaire::Afficher()
{
  cout << (heure < 10 ? "0"  : "" ) << heure << ":"

   << (minute < 10 ? "0"  : "" ) << minute;
}
```

Classes et objets (4) : 1er exemple de classe

```
int main()
{ Horaire h;    // Appel du constructeur qui
                   n'a pas de paramètre
  h.SetHoraire(40,10,12);
  h.affiche();
  //cout << « heure: » <<c.heure << endl; //erreur
}
```

Si on avait indiqué dans la définition de la classe :

```
Horaire (int = 0, int = 0, int = 0);
```

- **Définition du constructeur :**

```
Horaire:: Horaire (int h, int m, int s)
  { SetHoraire(h,m,s);}
```

- **Déclaration des objets :**

```
Horaire h1, h2(8), h3 (8,30), h4 (8,30,45);
```

Quelques règles de programmation

1. **Définir les classes, inclure les librairies etc. dans un fichier d'extension .h**
2. **Définir le corps des méthodes, le programme main etc. dans un fichier d'extension .cpp (incluant le fichier .h)**
3. **Compiler régulièrement**
4. **Pour déboguer :**
   - **Penser à utiliser les commentaires et les `cout`**
   - **Utiliser le débogueur**

Notions de constructeurs et destructeur (1)

Constructeur : Méthode appelée automatiquement à la création d'un objet

- **Constructeurs**
  - De même nom que le nom de la classe
  - Définition de l'initialisation d'une instance de la classe
  - Appelé implicitement à toute création d'instance de la classe
  - Méthode pouvant être surchargée
- **Destructeur**
  - De même nom que la classe mais précédé d'un tilde (~)
  - Définition de la désinitialisation d'une instance de la classe
  - Appelé implicitement à toute disparition d'instance de la classe
  - Méthode non typée et sans paramètre
  - Ne pouvant pas être surchargé

Notions de constructeurs et destructeur (2)

```
// Programme repris de [Delannoy, 2004] - pour montrer les
   appels du constructeur et du destructeur

class Exemple
{
    public :
      int attribut;
      Exemple(int); // Déclaration du constructeur
      ~Exemple();   // Déclaration du destructeur
} ;

Exemple::Exemple (int i)  // Définition du constructeur
{ attribut = i;
   cout << "** Appel du constructeur - valeur de l'attribut = "
   << attribut << "\n";
}

Exemple::~Exemple()  // Définition du destructeur
{ cout << "** Appel du destructeur - valeur de l'attribut = "
   << attribut << "\n";
}
```

Notions de constructeurs et destructeur (3)

On se sert du destructeur principalement lorsqu'il y a besoin de désallouer la mémoire!

```
class TableauDEntiers
{
        int nbElements;
        int * pointeurSurTableau;
   public:
        TableauDEntiers(int, int);   // Constructeur
        ~ TableauDEntiers();         // Destructeur
        …
}
```

```
// Constructeur allouant dynamiquement de la mémoire pour nb entiers
TableauDEntiers::TableauDEntiers (int nb, int max)
{ pointeurSurTableau = new int [nbElements=nb] ;
  for (int i=0; i<nb; i++) // nb entiers tirés au hasard
    pointeurSurTableau[i]= double(rand())/ RAND_MAX*max;
} // rand() fournit un entier entre 0 et RAND_MAX
```

```
TableauDEntiers::~TableauDEntiers ()
{ delete pointeurSurTableau ; // désallocation de la mémoire
}
```

Notions de constructeurs et destructeur (4)

Constructeur par recopie (*copy constructor*) :

- Constructeur créé par défaut mais pouvant être redéfini
- Appelé lors de l'**initialisation d'un objet par recopie** d'un autre objet, lors du **passage par valeur d'un objet** en argument de fonction ou en **retour d'un objet** comme retour de fonction

```
MaClasse c1;
MaClasse c2=c1; // Appel du constructeur par recopie
```

- Possibilité de définir explicitement un constructeur par copie si nécessaire :
  - Un seul argument de type de la classe
  - Transmission de l'argument par référence

```
MaClasse(MaClasse &);
MaClasse(const MaClasse &) ;
```

Nécessaire quand type complexe (tableaux, pointeurs) -> recopie 1 à 1

Méthodes de base d'une classe

- **Constructeur**
- **Destructeur**
- **Constructeur par copie**
- **Opérateur d'affectation (=)**

Attention aux implémentations par défaut fournies par le compilateur

Si une fonctionnalité ne doit pas être utilisée alors en interdire son accès en la déclarant `private`

Propriétés des méthodes (1)

- **Surcharge des méthodes**

```
MaClasse();                Afficher();
MaClasse(int);             Afficher(char* message);
```

- Possibilité de définir des **arguments par défaut**

```
MaClasse(int = 0);         Afficher(char* = "" );
```

- Possibilité de définir des **méthodes en ligne**

```
inline MaClasse::MaClasse() {corps court};

class MaClasse
{ …
 MaClasse() {corps court};
}
```

Définition de la méthode dans la déclaration même de la classe

Propriétés des méthodes (2)

Méthode retournant un objet

- **Transmission par valeur**

```
Horaire Horaire::HeureSuivante()
 { Horaire h;
    if (heure<24) h.SetHeure(heure+1);
       else h.SetHeure(0);
    h.SetMinute(minute); h.SetSeconde(seconde);
    return h; // Appel du constructeur par recopie
 }
```

- **Transmission par référence**

```
Horaire & Horaire::HeureSuivante()
```

La variable locale à la méthode est détruite à la sortie de la méthode – Appel automatique du destructeur!

Auto-référence

Auto-référence : pointeur `this`

- Pointeur sur l'objet (i.e. l'adresse de l'objet) ayant appelé
- Uniquement utilisable au sein des méthodes de la classe

```
Horaire::AfficheAdresse()
{ cout << "Adresse : " << this << endl ;
cout << "heure : " << this->heure << endl ;
}
```

Surcharge d'opérateurs (1/3)

Possibilité en C++ de redéfinir n'importe quel opérateur unaire ou binaire : =, ==, +, -, *, \, [], (), <<, >>, ++, -- , +=, -=, *=, /= , & etc.

```
class Horaire
{ …
  bool operator== (const Horaire &);
}

bool Horaire::operator==(const Horaire& h)
{
  return((heure==h.heure) && (minute == h.minute)
  && (seconde == h.seconde));
}
```

Fonction main

```
Horaire h1, h2;
…
if (h1==h2) …
```

Surcharge d'opérateurs (2/3)

```
// Fonction extérieure à la classe
bool operator==(const Horaire& h1, const Horaire& h2)
{
  return((h1.GetHeure()==h2.GetHeure()) && (h1.GetMinute()
  == h2.GetMinute()) && (h1.GetSeconde() ==
  h2.GetSeconde()) );
}
```

Fonction main

```
Horaire h1, h2;
…
if (h1==h2) …
```

Surcharge d'opérateurs (3/3)

```
class Horaire
{ …
  const Horaire& operator= (const Horaire &);
}

const Horaire& Horaire::operator=(const Horaire& h)
{
  heure=h.heure;
  minute = h.minute;
  seconde= h.seconde;
  return *this;
}
```

Fonction main

```
   Horaire h1(23,16,56),h2;
   …
   h2=h1; …
```

Objet membre (1/2)

Possibilité de créer une classe avec un membre de type objet d'une classe

```
// exemple repris de [Delannoy, 2004]
class point
 {
   int abs, ord ;
   public :
    point(int,int);
 };

 class cercle
 {
   point centre; // membre instance de la classe point
   int rayon;
   public :
    cercle (int, int, int);
 };
```

Objet membre (2/2)

```
#include "ObjetMembre.h"

point::point(int x=0, int y=0)
{
   abs=x; ord=y;
   cout << "Constr. point " << x << " " << y << endl;
}

cercle::cercle(int abs, int ord, int ray) : centre(abs,ord)
{
   rayon=ray;
   cout << "Constr. cercle " << rayon << endl;
}

int main()
{
   point p;
   cercle c (3,5,7);
}
```

Affichage :

```
Constr. point 0 0
Constr. point 3 5
Constr. cercle 7
```

Héritage simple (1/3)

- **Héritage** [Delannoy, 2004] :
  - Un des fondements de la P.O.O
  - A la base des possibilités de réutilisation de composants logiciels
  - Autorisant la définition de nouvelles classes « dérivées » à partir d'une classe existante « de base »
- **Super-classe** ou classe mère
- **Sous-classe** ou classe fille : spécialisation de la super-classe - héritage des propriétés de la super-classe
- Possibilité d'héritage multiple en C++

Héritage simple (2/3)

```
class CompteBanque
{
   long ident;
   float solde;
   public:
    CompteBanque(long id, float so = 0);
    void deposer(float);
    void retirer(float);
    float getSolde();
};

class ComptePrelevementAuto : public CompteBanque
{
   float prelev;
   public:
    void prelever();
    ComptePrelevementAuto(long id, float pr, float so);
 };
```

Héritage simple (3/3)

```
void transfert(CompteBanque cpt1, ComptePrelevementAuto cpt2)
{
    if (cpt2.getSolde() > 100.00)
    {
        cpt2.retirer(100.00);
        cpt1.deposer(100.00);
    }
}
```

```
void ComptePrelevementAuto::prelever()
```

```
void ComptePrelevementAuto::prelever()
{
    if (getSolde() > 100.00)
    {
      // La sous-classe a accès aux méthodes publiques de
      // sa super-classe - sans avoir à préciser à quel objet
      // elle s'applique
    }
}
```

Une sous-classe n'a pas accès aux membres privés de sa super-classe!!

Héritage simple et constructeurs (1/4)

```
class Base
{
    int a;
    public:
     Base() : a(0) {}
     Base(int A) : a(A) {}
};

class Derived : public Base
{
    int b;
    public:
     Derived() : b(0) {} // appel implicite à Base()
     Derived(int i, int j) : Base(i), b(j) {} // appel explicite
};
```

`Derived obj;` ⇒ « construction » d'un objet de la classe `Base` puis d'un objet de la classe `Derived`

Destruction de `obj` ⇒ appel automatique au destructeur de la classe `Derived` puis à celui de la classe `Base` (ordre inverse des constructeurs)

Héritage simple et constructeurs (2/4)

```
// Exemple repris de [Delannoy, 2004] page 254
#include <iostream>
using namespace std ;

// ************ classe point *********************
class point
{
   int x, y ;
   public :
    // constructeur de point ("inline")
    point (int abs=0, int ord=0)
     { cout << "++ constr. point : " << abs << " " << ord << endl;
       x = abs ; y =ord ;
     }
    ~point () // destructeur de point ("inline")
     { cout << "-- destr.  point :    " << x << " " << y << endl ;
     }
} ;
```

Héritage simple et constructeurs (3/4)

```
// ************ classe pointcol ******************
class pointcol : public point
{
   short couleur ;
   public :
    pointcol (int, int, short) ;  // déclaration constructeur pointcol
   ~pointcol ()                   // destructeur de pointcol ("inline")
     { cout << "-- dest. pointcol - couleur : " << couleur << endl ;
     }
} ;

pointcol::pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
{
    cout << "++ constr. pointcol : " << abs << " " << ord << " " << cl
      << endl ;
    couleur = cl ;
}
```

Héritage simple et constructeurs (4/4)

```
// ************ programme d'essai ****************
int main()
{
    pointcol a(10,15,3) ;                // objets non dynamiques
    pointcol b (2,3) ;
    pointcol c (12) ;
    pointcol * adr ;
    adr = new pointcol (12,25) ;         // objet dynamique
    delete adr ;
}
```

Résultat :

```
++ constr. point :    10 15
++ constr. pointcol : 10 15 3
++ constr. point :    2 3
++ constr. pointcol : 2 3 1
++ constr. point :    12 0
++ constr. pointcol : 12 0 1
++ constr. point :    12 25
++ constr. pointcol : 12 25 1
-- dest. pointcol - couleur : 1
-- destr.  point :    12 25
-- dest. pointcol - couleur : 1
-- destr.  point :    12 0
-- dest. pointcol - couleur : 1
-- destr.  point :    2 3
-- dest. pointcol - couleur : 3
-- destr.  point :    10 15
```

Héritage simple et constructeurs par copie (1/4)

```
#include <iostream>
using namespace std ;

class point
{
   int x, y ;
   public :
    point (int abs=0, int ord=0)  // constructeur usuel
      { x = abs ; y = ord ;
        cout << "++ point    " << x << " " << y << endl ;
      }
    point (point & p)  // constructeur de recopie
      { x = p.x ; y = p.y ;
        cout << "CR point    " << x << " " << y << endl ;
      }
} ;
```

<u>Rappel</u> : appel du constructeur par copie lors

- **de l'initialisation d'un objet par un objet de même type**
- **de la transmission de la valeur d'un objet en argument ou en retour de fonction**

Héritage simple et constructeurs par copie (2/4)

```
class pointcol : public point
{
   int coul ;
   public :
    // constructeur usuel
   pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord)
     {
       coul = cl ;
       cout << "++ pointcol " << coul << endl ;
     }
   // constructeur de recopie
   // il y aura conversion implicite de p dans le type point
    pointcol (pointcol & p) : point (p)
      {
       coul = p.coul ;
       cout << "CR pointcol " << coul << endl ;
     }
} ;
```

Si pas de constructeur par copie défini dans la sous-classe ⇒ Appel du constructeur par copie par défaut de la sous-classe et donc du constructeur par copie de la super-classe

Héritage simple et constructeurs par copie (3/4)

```
void fct (pointcol pc)
{
   cout << "*** entree dans fct ***"  << endl ;
}

int main()
{
   void fct (pointcol) ; // Déclaration de f
   pointcol a (2,3,4) ;
   fct (a) ; // appel de fct avec a transmis par valeur
}
```

Résultat :

```
++ point     2 3
++ pointcol 4
CR point     2 3
CR pointcol 4
*** entree dans fct ***
```

pointcol a (2,3,4) ;

fct (a) ;

Héritage simple et constructeurs par copie (4/4)

Soit une classe B, dérivant d'une classe A :

```
B b0;
B b1 (b0); // Appel du constructeur par copie de B
B b2 = b1 ; // Appel du constructeur par copie de B
```

- Si aucun constructeur par copie défini dans B :
  - ⇒ Appel du constructeur par copie par défaut faisant une copie membre à membre
  - ⇒ Traitement de la partie de b1 héritée de la classe A comme d'un membre du type A
    - ⇒ Appel du constructeur par copie de A
- Si un constructeur par copie défini dans B :
  - **B ([const] B&)**

    ⇒ Appel du constructeur de A sans argument ou dont tous les arguments possède une valeur par défaut
  - **B ([const] B& x) : A (x)**

    ⇒ Appel du constructeur par copie de A

Le constructeur par copie de la classe dérivée doit prendre en charge l'intégralité de la recopie de l'objet et également de sa partie héritée

Contrôle des accès (1)

Trois qualificatifs pour les membres d'une classe : `public, private` et `protected`

- **`Public`** : membre accessible non seulement aux fonctions membres (méthodes) ou aux fonctions amies mais également aux clients de la classe

- **`Private`** : membre accessible uniquement aux fonctions membres (publiques ou privées) et aux fonctions amies de la classe

- **`Protected`** : comme **`private`** mais membre accessible par une classe dérivée

Contrôle des accès (2)

```
class Point
{
    protected: // attributs protégés
      int _x;
      int _y;
    public:
      Point (...);
      affiche();
      ...
};

class Pointcol : public Point
{
    short _couleur;
    public:
      void affiche()
      { // Possibilité d'accéder aux attributs protégés
        // _x et _y de la super-classe dans la sous-classe
       cout << "je suis en " << _x << " " <<  _y << endl;
       cout << " et ma couleur est " << _couleur << endl;
      }
};
```

Contrôle des accès (3)

Membre protégé d'une classe :

- Équivalent à un membre privé pour les utilisateurs de la classe
- Comparable à un membre public pour le concepteur d'une classe dérivée
- Mais comparable à un membre privé pour les utilisateurs de la classe dérivée
- Possibilité de violer l'encapsulation des données

Contrôle des accès (4)

Plusieurs modes de dérivation de classe :

- Possibilité d'utiliser **`public`**, **`protected`** ou **`private`** pour spécifier le mode de dérivation d'une classe
- Détermination, par le mode de dérivation, des membres de la super-classe accessibles dans la sous-classe
- Dérivation privée par défaut

Contrôle des accès (5)

Dérivation publique :

- Conservation du statut des membres publics et protégés de la classe de base dans la classe dérivée
- Forme la plus courante d'héritage modélisant : « une classe dérivée est une spécialisation de la classe de base »

```
class Base
{
  public:
    void méthodePublique1();
  protected:
    void méthodeProtégée();
  private:
    void MéthodePrivée();
};
```

```
class Derivee : public Base
{
  public:
    int MéthodePublic2()
      {
        méthodePublique1(); // OK
        méthodeProtégée();  // OK
        MéthodePrivée();    // KO
    }
};
```

Contrôle des accès (6)

Dérivation publique :

statut dans la classe de base	accès aux fonctions membre et amies de la classe dérivée	accès à un utilisateur de la classe dérivée	nouveau statut dans la classe dérivée
public	oui	oui	public
protégé	oui	non	protégé
privé	non	non	privé

Héritage simple

constructeurs/destructeurs/constructeurs par copie

- Pas d'héritage des constructeurs et destructeurs ⇒ il faut les redéfinir
- Appel implicite des constructeurs par défaut des classes de base (super-classe) avant le constructeur de la classe dérivée (sous-classe)
- Possibilité de passage de paramètres aux constructeurs de la classe de base dans le constructeur de la classe dérivée par appel explicite
- Appel automatique des destructeurs dans l'ordre inverse des constructeurs
- Pas d'héritage des constructeurs de copie et des opérateurs d'affectation

Héritage simple et redéfinition/sur-définition

```
class Base
{
   protected :
   int a;
   char b;
   public :
    …
    void affiche();
};

class Derivee : public Base
{
   float a; // redéfinition de l'attribut a
   public :
    …
    void affiche(); // redéfinition de la méthode affiche
    float GetADelaClasseDerivee(){return a;} ;
    int GetADeLaClasseDeBase() {return Base::a;}
};
```

```
void Base::affiche()
{ cout << a << b << endl; }

void Derivee::affiche()
{  // appel de affiche
   // de la super-classe
   Base::affiche();
   cout << " a est un ";
}
```

Héritage simple et amitié

- **Mêmes autorisations d'accès pour les fonctions amies d'une classe dérivée que pour ses méthodes**

Pas d'héritage au niveau des déclarations d'amitié

```
class A
{   friend class ClasseAmie;
    public:
      A(int n=0) : attributDeA(n) {}
    private:
      int attributDeA;
 };
class ClasseAmie
{   public:
      ClasseAmie(int n=0) : objetMembre(n) {}
      void affiche1() {cout << objetMembre. attributDeA << endl;}
                          // OK: Cette classe est est amie de A
    private:
      A objetMembre;
};
```

Compatibilité entre classe de base et classe dérivée (1/2)

- Possibilité de **convertir implicitement une instance d'une classe dérivée en une instance de la classe de base**, si l'héritage est public
- L'inverse n'est pas possible : impossibilité de convertir une instance de la classe de base en instance de la classe dérivée

```
ClasseDeBase a;
ClasseDérivée b;
a=b; // légal et appel de l'opérateur = de la classe de Base
     // s'il a été redéfini ou de l'opérateur par défaut sinon
b=a; // illégal
     // error: no match for 'operator=' in 'b = a'
```

Compatibilité entre classe de base et classe dérivée (2/2)

- Possibilité de convertir un pointeur sur une instance de la classe dérivée en un pointeur sur une instance de la classe de base, s'il l'héritage est public

```
ClasseDeBase o1, * p1=NULL;
ClasseDérivée o2, * p2=NULL;

p1=&o1; p2=&o2;

p1->affiche() ; // Appel de ClasseDeBase::affiche();
p2->affiche() ; // Appel de ClasseDérivée::affiche();

p1=p2; // legale: ClasseDérivée* vers ClasseDeBase*

p1->affiche() ; // Appel de ClasseDeBase::affiche();
                // et non de ClasseDérivée::affiche();

p2=p1; // erreur sauf si on fait un cast explicite
// error: invalid conversion from `ClasseDeBase*' to `ClasseDérivéé*'
p2= (ClasseDérivée*) p1; // Possible mais Attention les attributs membres
                         // de la classe dérivée n'auront pas de valeur
```

Héritage simple et opérateur d'affectation (1/6)

- Si pas de redéfinition de l'opérateur = dans la classe dérivée :
  - ⇒Affectation membre à membre
  - ⇒Appel implicite à l'opérateur = sur-défini ou par défaut de la classe de base pour l'affectation de la partie héritée
- Si redéfinition de l'opérateur = dans la classe dérivée :
  - ⇒Prise en charge totale de l'affectation par l'opérateur = de la classe dérivée

Héritage simple et opérateur d'affectation (2/6)

```
#include <iostream>
using namespace std ;

class point
{  protected :
     int x, y ;
 public :
     point (int abs=0, int ord=0)
       { x=abs ; y=ord ;}
     point & operator = (const point & a)
      { x = a.x ; y = a.y ;
        cout << "operateur = de point" << endl;
        return * this ;
      }
} ;
```

Héritage simple et opérateur d'affectation (3/6)

```
class pointcol : public point
{
  int couleur ;
  public :
  pointcol (int abs=0, int ord=0, int c=0);
  // Pas de redéfinition de l'opérateur = dans la classe dérivée
};

pointcol::pointcol(int abs, int ord, int c) : point(abs,ord)
{ couleur=c;}

int main()
{
   pointcol c, d;
   d=c;
}
```

operateur = de point

Héritage simple et opérateur d'affectation (4/6)

```
class pointcol : public point
{
  int couleur ;
  public :
  pointcol (int abs=0, int ord=0, int c=0);
  // Redéfinition de l'opérateur = dans la classe dérivée
  pointcol & operator = (const pointcol & a)
     {  couleur=a.couleur;
        cout << "operateur = de pointcol" << endl;
        return * this ;
      }
};
pointcol::pointcol(int abs, int ord, int c) : point(abs,ord)
{ couleur=c;}

int main()
{  pointcol c, d;
   d=c;
}
```

```
operateur = de pointcol
```

Héritage simple et opérateur d'affectation (5/6)

```
// Redéfinition de l'opérateur = dans la classe dérivée
// Avec appel explicite à l'opérateur = de la classe de base
// en utilisant des conversions de pointeurs
pointcol & pointcol::operator = (const pointcol & a)
 {  point * p1;
    p1=this; // conversion d'un pointeur sur pointcol
             // en pointeur sur point
    const point *p2= &a;  // idem
    *p1=*p2; // affectation de la partie « point » de a
    couleur=a.couleur;
    cout << "operateur = de pointcol" << endl;
    return * this ;
 }

int main()
{  pointcol c, d;
   d=c;
}
```

```
operateur = de point
operateur = de pointcol
```

Héritage simple et opérateur d'affectation (6/6)

```
// Redéfinition de l'opérateur = dans la classe dérivée
// Avec appel explicite à l'opérateur =
// de la classe de base
pointcol & pointcol::operator = (const pointcol & a)
 {  // Appel explicite à l'opérateur = de point
    this->point::operator=(a);
    couleur=a.couleur;
    cout << "operateur = de pointcol" << endl;
    return * this ;
 }

int main()
{
   pointcol c, d;
    d=c;
}
```

```
operateur = de point
operateur = de pointcol
```

Fonction/Méthode virtuelle et typage dynamique (1)

- **Liaison statique** : **type de l'objet pointé déterminé au moment de la compilation.**

  De même pour les méthodes à invoquer sur cet objet

  ⇒Appel des méthodes correspondant au type du pointeur et non pas au type effectif de l'objet pointé

- **Polymorphisme** ⇒ possibilité de choisir dynamiquement (à l'exécution) une méthode en fonction de la classe effective de l'objet sur lequel elle s'applique – **liaison dynamique**

- **Liaison dynamique obtenue en définissant des méthodes virtuelles**

Fonction/Méthode virtuelle et typage dynamique (2)

```
class Personne
{
  string nom;
  string prenom;
  int age;
  char sexe;

  public :
   // Constructeur
   Personne(string n, string p, int a, char s)
   { nom=n; prenom=p; age=a; sexe=s;
     cout << "Personne::Personne("<<nom<< "," <<
       prenom << "," << age << ","  <<  sexe << ")"  << endl;
   }

  // Affichage
  void Affiche()
  { if (sexe == 'M') cout << "Monsieur "
     else cout << "Madame/Mademoiselle " ;
    cout << prenom  << " " << nom << " agée de " <<
           age << " ans." << endl;
  }

  // Destructeur
  ~Personne() {cout << "Personne::~Personne()" << endl;}

};
```

Fonction/Méthode virtuelle et typage dynamique (3)

```
class Etudiant : public Personne
{ int note;
  public:
  // Constructeur
  Etudiant(string nm, string p, int a, char s, int n):Personne(nm,p,a,s)
  { note = n;
   cout << "Etudiant::Etudiant(" <<GetNom()<< "," << GetPrenom() <<
    ","<< GetAge() << ","  <<  GetSexe() << "," << note << ")" << endl;
  }
  void Affiche() // Affichage
  { Personne::Affiche();
  cout << "Il s'agit d'un étudiant ayant pour note :" << note << "." << endl;
  }
  // Destructeur
  ~Etudiant() {cout << "Etudiant::~Etudiant()" << endl;}

};
```

```
// Pas d'appel de la méthode Affiche() ou du
// destructeur de la classe Etudiant
int main()
{ Personne * p1 = new Etudiant("GAMOTTE","Albert",34,'M',13);
  p1->Affiche();
  delete p1;
}
```

```
Personne::Personne(GAMOTTE,Albert,34,M)
Etudiant::Etudiant(GAMOTTE,Albert,34,M,13)
Monsieur Albert GAMOTTE agé de 34 ans.
Personne::~Personne()
```

Fonction/Méthode virtuelle et typage dynamique (4)

```
class Personne
{
  string nom;
  string prenom;
  int age;
  char sexe;

  public :
   // Constructeur
   Personne(string n, string p, int a, char s)
   { nom=n; prenom=p; age=a; sexe=s;
     cout << "Personne::Personne("<<nom<< "," <<
       prenom << "," << age << ","  <<  sexe << ")"  << endl;
   }

  virtual void Affiche() // Affichage
  { if (sexe == 'M') cout << "Monsieur "
     else cout << "Madame/Mademoiselle " ;
    cout << prenom  << " " << nom << " agée de " <<
           age << " ans." << endl;
  }

  // Destructeur
  virtual ~Personne() {cout << "Personne::~Personne()" << endl;}

};
```

Fonction/Méthode virtuelle et typage dynamique (5)

```
int main()
{
  Personne * p1 = new Etudiant("GAMOTTE","Albert",34,'M',13);

  // Appel de la méthode Affiche() de la classe Etudiant
  // La méthode étant virtuelle dans la classe Personne
  p1->Affiche();

  // Appel du destructeur de la classe Etudiant
  // qui appelle celui de la classe Personne
  delete p1;
}
```

```
Personne::Personne(GAMOTTE,Albert,34,M)
Etudiant::Etudiant(GAMOTTE,Albert,34,M,13)
Monsieur Albert GAMOTTE agé de 34 ans.
Il s'agit d'un étudiant ayant pour note :13.
Etudiant::~Etudiant()
Personne::~Personne()
```

Fonction/Méthode virtuelle et typage dynamique (6)

Toujours déclarer virtuel le destructeur d'une classe de base destinée à être dérivée pour s'assurer que une libération complète de la mémoire

- Pas d'obligation de redéfinir une méthode virtuelle dans les classes dérivées
- Possibilité de redéfinir une méthode virtuelle d'une classe de base, par une méthode virtuelle dans une classe dérivée
- **Nécessité de respecter le prototype de la méthode virtuelle redéfinie dans une classe dérivée (même argument et même type retour)**

Fonction virtuelle pure et classe abstraite (1/2)

- **Classe abstraite** : classe sans instance, destinée uniquement à être dérivée par d'autres classes
- **Fonction virtuelle pure** : fonction virtuelle déclarée sans définition dans une classe abstraite et devant être redéfinie dans les classes dérivées

```
class MaClasseAbstraite
{ …
  public :
  // Définition d'une fonction virtuelle pure
  // =0 signifie qu'elle n'a pas de définition
  // Attention, c'est différent d'un corps vide : { }
  virtual void FonctionVirtuellePure() = 0;
  …
}
```

Fonction virtuelle pure et classe abstraite (2/2)

- **Toute classe comportant au moins une fonction virtuelle pure est abstraite**
- Toute fonction virtuelle pure doit
  - Être redéfinie dans les classes dérivées
  - Ou être déclarée à nouveau virtuelle pure
    $\Rightarrow$ Classe dérivée abstraite
- Pas de possibilité de définir des instances d'une classe abstraite
- Mais **possibilité de définir des pointeurs et des références sur une classe abstraite**

Patrons de fonctions (1/8)

Patron de fonctions : fonction générique exécutable pour n'importe quel type de données

```
int minimum(int a, int b)
{ if(a<b)return a;
   else return b;
}
```

```
float minimum(float a, float b)
{ if(a<b)return a;
   else return b;
}
```

```
// création d'un patron de fonctions
template <typename T> T minimum (T a, T b)
{   if (a < b) return a ;
     else return b ; // ou return a < b ? a : b ;
}
```

Patrons de fonctions (2/8)

- **Définition d'un patron :** `template <typename T>` ou `template <class T>`
- **Paramètre de type quelconque :** `T`

```
// Exemple d'utilisation du patron de fonctions minimum
// repris de [Delannoy, 2004]
int main()
{
   int n=4, p=12 ;
   float x=2.5, y=3.25 ;
   cout << "minimum (n, p) = " << minimum (n, p) << endl ;
   cout << "minimum (x, y) = " << minimum (x, y) << endl ;

   char * adr1 = "monsieur", * adr2 = "bonjour" ;
   cout << "minimum (adr1, adr2) = " << minimum (adr1, adr2)
        << endl; ;
}
```

```
minimum (n, p) = 4
minimum (x, y) = 2.5
minimum (adr1, adr2) = monsieur
```

Patrons de fonctions (3/8)

```
// Exemple d'utilisation du patron de fonctions minimum
// repris de [Delannoy, 2004]

class vect
{  int x, y ;
  public :
   vect (int abs=0, int ord=0) { x=abs ; y=ord; }
   void affiche () { cout << x << " " << y ; }
   friend int operator < (vect, vect) ;
} ;

int operator < (vect a, vect b)
{  return a.x*a.x + a.y*a.y < b.x*b.x + b.y*b.y ;}

int main()
{
   vect u (3, 2), v (4, 1), w ;
   w = minimum (u, v) ;
   cout << "minimum (u, v) = " ; w.affiche() ;
}
```

```
minimum (u, v) = 3 2
```

Patrons de fonctions (4/8)

- **Mécanisme des patrons :**

  ⇒ Instructions utilisées par le compilateur pour fabriquer à chaque fois que nécessaire les instructions correspondant à la fonction requise

- **En pratique, placement des définitions de patron dans un fichier approprié d'extension .h**

- **Possibilité d'avoir plusieurs paramètres de classes différentes** dans l'en-tête, dans des déclarations de variables locales ou dans les instructions exécutables

- Mais **nécessité que chaque paramètre de type apparaisse au moins une fois dans l'en-tête du patron** pour que le compilateur soit en mesure d'instancier la fonction nécessaire

Patrons de fonctions (5/8)

```
// Exemple repris de [Delannoy, 2004]
template <typename T, typename U>
 void fct(T a, T* b, U c)
{
   T x; // variable locale x de type T
   U* adr; // variable locale adr de type pointeur sur U
   ...
   adr = new U[10]; // Allocation dynamique
   ...
   int n=sizeof(T);
   ...
}
```

- Nécessité de passer un ou plusieurs arguments au constructeur des objets déclarés dans le corps des patrons de fonctions

```
template <typename T> void fct (T a)
{
  T x(3); // Si on appelle fct avec un paramètre int
   ....    // le compilateur sait exécuter int x(3);
}
```

Patrons de fonctions (6/8)

Possibilité de redéfinir les patrons de fonctions

```
// Exemple repris de [Delannoy, 2004]
#include <iostream.h>

template <typename T> T minimum (T a, T b)   // patron I
{
   if(a<b) return a;
    else return b;
}

template <typename T> T minimum (T a, T b, T c) // patron II
{ return minimum(minimum(a,b),c);}

int main()
{
   int n=12, p=15, q=2;
   float x=3.5, y=4.25, z=0.25;
   cout << minimum(n,p) << endl;      // patron I
   cout << minimum(n,p,q) << endl;    // patron II
   cout << minimum(x,y,z) << endl;    // patron II
}
```

```
12
2
0.25
```

```
cout << minimum (n, x) << endl ;
// => BUG car error: no matching function for
// call to `minimum(int&,float&)
```

Patrons de fonctions (7/8)

Possibilité de redéfinir les patrons de fonctions

```
// Exemple repris de [Delannoy, 2004]

// patron numéro I
template <typename T> T minimum (T a, T b)
{  if (a < b) return a ;
      else return b ;
}

// patron numéro II
template <typename T> T minimum (T * a, T b)
{  if (*a < b) return *a ;
      else return b ;
}

// patron numéro III
template <typename T> T minimum (T a, T * b)
{  if (a < *b) return a ;
      else return *b ;
}
```

Patrons de fonctions (8/8)

Possibilité de redéfinir les patrons de fonctions

```
// Exemple repris de [Delannoy, 2004]
int main()
{
  int n=12, p=15 ;
  float x=2.5, y=5.2 ;

  // patron numéro I    int minimum (int, int)
  cout << minimum (n, p) << endl ;
  // patron numéro II   int minimum (int *, int)
  cout << minimum (&n, p) << endl ;
  // patron numéro III  float minimum (float, float *)
  cout << minimum (x, &y) << endl ;
  // patron numéro I    int * minimum (int *, int *)
  cout << minimum (&n, &p) << endl ;
}
```

```
12
12
2.5
0x22eeb0
```

Ne pas introduire d'ambiguïté

```
// Ambiguité avec le premier template
// pour minimum (&n,&p)
template <typename T> T minimum (T* a, T * b)
{  if (*a < *b) return *a ;
      else return *b ;
}
```

Patrons de classes (1)

Patron de classes : Définition générique d'une classe permettant au compilateur d'adapter automatiquement la classe à différents types

```
// Définition d'un patron de classes
template <class T> class Point
{ T x;
  T y;
  public:
    Point (T abs=0, T ord=0){x=abs; y=ord; }
    void affiche();
};


// Corps de la méthode affiche()
template <class T> void Point<T>::affiche()
{ cout << "Coordonnées: " << x << " " << y << endl;}
```

Patrons de classes (2)

```
int main()
{
  // Déclaration d'un objet
  Point<int> p1(1,3);
  // => Instanciation par le compilateur de la
  // définition d'une classe Point dans laquelle le
  // paramètre T prend la valeur int

  p1.afficher();

  // Déclaration d'un objet
  Point <double> p2 (3.5, 2.3) ;
  // => Instanciation par le compilateur de la
  // définition d'une classe Point dans laquelle le
  // paramètre T prend la valeur double

  p2.affiche () ;
}
```

Patrons de classes (3)

Possibilité d'avoir un nombre quelconque de paramètres génériques :

```
template <class T, class U, class V> class Essai
{
   T x; // Membre attribut x de type T
   U t[5]; // Membre attribut t de type tableau *
           // de 5 éléments de type U
   ...
   V fml(int, U); // Méthode à deux arguments,
                  // un de type entier et l'autre
                  // de type U et retournant
                  // un résultat de type V
   ...
};

Essai <int,float,int> ce1;
Essai <int, int*,double> ce2;
Essai <float, Point<int>, double> ce3;
Essai <Point<int>,Point<float>,char*> ce4;
```

Généralités sur la STL

- **STL (*Standard Template Library*)** : patrons de classes et de fonctions
- Définition de structures de données telles que les conteneurs (vecteur, liste), itérateurs, algorithmes généraux, etc.

```
#include <vector>
#include <stack>
#include <list>
int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } ;
vector<int> v1(4, 99) ;  // vecteur de 4 entiers egaux à 99
vector<int> v2(7, 0) ;   // vecteur de 7 entiers
vector<int> v3(t, t+6) ; // vecteur construit a partir de t
// Pile d'entiers utilisant un conteneur vecteur
stack<int, vector<int> > q ;
cout << "taille initiale  : " << q.size() << endl ;
for (i=0 ; i<10 ; i++) q.push(i*i) ;
list<char> lc2 ; // Liste de caractères
list<char>::iterator il1 ; // itérateur sur une liste de char
il2 = lc2.begin() ;
for (il1=lc1.begin() ; il1!=lc1.end() ; il1++) { …}
```

- « *STL - précis et concis* » de Ray Lischner, O'Reilly (Français), février 2004, ISBN: 2841772608

Les flots

- **Flot** : « Canal »
  - **Recevant de l'information – flot de sortie**
  - **Fournissant de l'information – flot d'entrée**
- **`cout`** connecté à la « sortie standard »
- **`cin`** connecté à l'« entrée standard »
- 2 opérateurs **`<<`** et **`>>`** pour assurer le transfert de l'information et éventuellement son formatage

Templates: algorithm

- Opérations sur les ensembles (union...)
- Opérations de recherche
- Opérations de copie

Templates: structures

- # C++ Vectors: listes
- # C++ Double-Ended Queues (double chainage)
- # C++ Lists : listes chainées
- # C++ Queues : FIFO (file)
- # C++ Stacks : pile
- # C++ Sets : ensemble
- Itérateurs!

Vector

- Constructeur vide possible: vector()
- Opérations de construction:
  - push_back(&obj), obj pop_back()
  - erase(iterator)
  - Clear()
- Opérations d'information:
  - Int size(), bool empty()
  - Iterator begin(), iterator end()