Modélisation, Conception et Programmation Orientées Objets - Partie : C++

Christine REGIS

M1 – DL UE18
MCPOO2

IRIT - Université Paul Sabatier -Toulouse

Plan du cours

- Introduction à C++ (7h cours, 8h TP)
  - Classes, références
  - Opérateurs
  - Héritage et polymorphisme
  - Généricité
  - Gestion mémoire
- Design Patterns (8h cours, 8h TD)
  par J.P. Arcangeli

Historique

- 1970 : Langage C de bas niveau conçu pour UNIX
  - Kernighan , Ritchie
- 1979 : C++, extension du C, classes, héritage
  - Stoustrup, AT & T Bell Labs
- 1983 : liaison dynamique
- 1985 : surcharge, opérateurs, constantes, références, opérateurs new et delete, commercialisation (version 1.0)
- 1989 : héritage multiple (version 2.0)
- 1990 : généricité, gestion d'exception
- 1995 : Java (Sun)

Comparaison JAVA/C++

- Pointeurs, références
- Libération mémoire non transparente
- Surcharge d'opérateurs
- Héritage multiple
- Rapidité d'exécution
- Langage très technique
- Java est +simple à apprendre et utiliser
- Site web « bjarne stroustrup faq »

Les bases du C++

Chapitre 1

Interface d'une classe : Livre.h

Livre.h

```
class Livre {
// privé par défaut
      string titre ;
      string auteur;
      Texte * texte;
public :
      void saisir();
      void afficher();
      void lire();
      };
```


Interface d'une classe : Texte.h

Texte.h

```
class Texte {
      // attributs …
public :
      void saisir();
      void lister();
      void lire();
      };
```


Coder les méthodes dans Livre.cpp

```
#include "Livre.h"

void Livre::afficher() {
    cout << titre;
    cout << auteur;
    texte->lister();
}
void Livre::saisir() {
    cin >> titre;
    cin >> auteur;
    texte->saisir();
}
```

```
void Livre::lire() {
    texte->lire();
}
```

- Toutes les méthodes sont précédées du nom de la classe et de l'opérateur de résolution de portée ::

Déclaration et utilisation d'objets

Main.cpp

```
#include "Livre.h"
void main()
{
    Livre mon_livre;          // déclaration sans new
    mon_livre.lire();

    Livre *chevet;            // pointeur sur un objet Livre
    chevet = new Livre();
    chevet->lire();

    }
```


Entrées / Sorties

- Inclure le fichier <iostream>
- **cout** : flux sur la sortie standard stdout (écran)
- **cin** : flux sur l 'entrée standard stdin (clavier)

- << : opérateur d'écriture **(opérateur d'insertion)**
  - L'opérande gauche est un flux en sortie
  - L'opérande droit est une expression (tout type, valeur à afficher)
  - Surcharge possible

- >> : opérateur de lecture **(opérateur d'extraction)**
  - L'opérande gauche est un flux en entrée
  - L'opérande droit est une expression (tout type, valeur à lire)

L'opérateur d'écriture <<

```
#include <iostream>
using namespace std;
void main() {
        float price;
        cout << "  price :  " << price << endl;
    }
```


L'opérateur de lecture >>

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

void main() {
    int nb, nb1, nb2; char c;
    cout<<"Enter an integer : ";
    cin >> nb;

    cout<<"Enter a letter : ";
    cin >> c;

    cout<<"Enter two integers : ";
     cin >>  nb1 >> nb2;
}
```

Déclaration d'un objet Livre

- Variable non initialisée, contenu indéterminé

  int i;
- Initialisation par appel automatique du constructeur de l'objet

  Livre L;
- Aucun constructeur appelé

  Livre *ptr;
  - Utiliser new pour allouer de la mémoire
- A la fin d'un bloc, appel automatique du destructeur de l'objet

Constructeur

- Initialisation des attributs de la classe
- Fonction membre de la classe
- Porte toujours le nom de la classe
- Ne retourne rien

Livre.h

```
class Livre {
        string titre , auteur;
        Texte * texte;
public :
      Livre(string t, string a, Texte
     *texte) ;
        };
```

L

```
#include "Livre.h "

Livre::Livre(string t, string a, Texte
     *texte) {
      this->titre = t;
      this->auteur = a;
     this->texte = texte;
}
```

Livre.cpp

Plusieurs constructeurs

```
class Complexe {
    float re,im;
public :
Complexe (float Re, float Im) {re = Re; im = Im;}

// Constructeur surchargé
Complexe (float Re) {re = Re; im = 0;}

// Constructeur sans paramètre
Complexe () { re = 0.0; im = 0.0;}
};
```


Exemples d'appels des constructeurs

```
// appel du constructeur avec 2 paramètres
Complexe C1(3, 4);

// appel du constructeur surchargé
Complexe C2(1); ou Complexe C2 = 1;

// Attention, appel du constructeur copie
Complexe C3 = C2;
```

Constructeur par défaut

- Constructeur :
  - sans paramètre
  - ou tous les paramètres ont des valeurs par défaut
- Si aucun constructeur n'est défini dans la classe, il est défini automatiquement par C++.
- Sinon il faut le définir.

Appels du constructeur par défaut

Complexe C; //OUI, appel à Complexe()

Complexe C();// NON, déclaration de fonction

Complexe tab[10]; // 10 appels à Complexe()

Complexe *C4 = new Complexe; // appel de Complexe()

Complexe *p = new Complexe[10]; // 10 appels à Complexe()

Paramètres avec valeurs par défaut (1)

- En C, le nombre de paramètres formels doit être égal au nombre de paramètres effectifs
  - int produit(**int a**, **int b**); //paramètres formels
  - int res = produit(**12**, **34**); // paramètres effectifs
- En C++, mécanisme d'attribution de valeurs par défaut aux paramètres pour s'affranchir de cette règle

Paramètres avec valeurs par défaut (2)

```
void essai (int, int = 3);

int main() {
int n=1, p = 2;

//appel normal
essai (n, p);

// appel avec un seul argument,
// le deuxième argument a la valeur 3
// par défaut
essai (n);

essai ();                    // NON  rejeté
}
```

- Tous les paramètres peuvent avoir des valeurs par défaut

```
void essai (int = 2, int = 3); //OK
```

- Les paramètres avec valeur par défaut doivent être les derniers de la liste de paramètres

```
void essai (int = 2, int );      //KO
```

Constructeurs uniques avec paramètres par défaut

```
Complexe::Complexe (float Re = 0.0, float Im = 0.0) {
                this->re = Re;
                this->im = Im;}
```
Complexe.cpp

```
Livre::Livre(string t = "Le lion", string a = "Kessel",
                                        Texte *texte =NULL) {
    this->titre = t;
    this->auteur = a;
   this->texte = texte;
}
```
Livre.cpp

Destructeur

- Destruction des objets par libération de la mémoire allouée
- Fonction membre portant le nom de la classe, ne retournant rien, sans paramètres

  ~Livre()
- S'il n'est pas défini dans la classe, il y a appel d'un destructeur par défaut géré par C++
- Appel automatique du destructeur en sortie de bloc

Codage ou non du destructeur

- Il n'est pas nécessaire de le coder si le programmeur n'alloue pas de la mémoire dans le constructeur
  - Ne pas coder ~Complexe() !
- Constructeur avec allocation -> coder un destructeur

```
Livre::Livre(string t = « Le lion », string a = « Kessel », Texte *texte =NULL) {
      this->titre = t;
      this->auteur = a;
      this->texte = new Texte(*texte);          // constructeur copie Texte(Texte&)
}

Livre::~Livre() {
     delete texte;    // appel de ~Texte()
}
```
Livre.cpp

Les opérateurs new et delete

New

1. Réservation de l 'espace mémoire pour un objet
2. Initialisation de l'objet par appel du constructeur

   Livre *pt = new Livre; // constructeur par défaut

   Livre *pt = new Livre("Lambeaux", "Juliet"); // constructeur avec paramètres

Delete

1. Appel du destructeur sur l'objet
2. Libération de l'espace mémoire

   delete pt;

Les opérateurs new[] et delete[]

- Pour allouer ou détruire un tableau d'objets

Livre *pt = new Livre [taille];

- Appel du constructeur par défaut de la classe Livre pour chaque élément du tableau

delete [] pt;

- Appel du destructeur de Livre pour chaque élément du tableau

Constructeur copie (clonage)

- Construction d'un objet identique à un autre

  Complexe (Complexe &);

  Complexe(const Complexe &);

- Recopie des attributs

```
Complexe::Complexe(const Complexe& copie) {
    this->re = copie.re;
    this->im = copie.im;
}
```


Constructeur copie par défaut

- Créé automatiquement par C++ s'il n'est pas programmé dans la classe
- Effectue une copie membre à membre des attributs de la classe

  -> problème si pointeurs car copie des pointeurs uniquement

```
class Chaine {
        char *pt;
        public:
        Chaine(char * s);
        Chaine (const Chaine &copie) ;
        };
```

Chaine.h

```
#include "Chaine.h "

Chaine::Chaine (const Chaine &copie)
{   pt = new char [strlen(copie.pt) +1];
    strcpy(pt, copie.pt);
}
```

Chaine.cpp

Appels automatiques du constructeur copie

- pour initialiser un objet avec un autre objet de même classe

```
Chaine nom1 = « Paul »;
Chaine nom2 = nom1 ;
```

- à chaque appel de fonction avec passage de paramètre par valeur

```
Chaine f(Chaine mot) {          // passage par valeur
 // traitement sur mot
return mot; //retour de résultat
}
```

- pour un retour de résultat

Exemples d'appels du constructeur copie

Main.cpp

```
#include " Chaine.h"

void f(Chaine beta) // passage par valeur
{ // traitement sur beta }

void main()
{   Chaine premier,
    deuxieme(premier),      // 1er appel, initialisation
    troisieme = premier;    //2ième appel, initialisation

    f(premier);     // 3ième appel, recopie du paramètre effectif dans son
                    // paramètre formel
}
```


Variable de type référence (1)

- Une référence ou alias est un deuxième nom pour une même variable

```
int i;
int & ri = i; // déclaration d'une référence sur i
```

- Tout accès à ri est un accès à i => déréférenciation automatique
- La variable ri est synonyme de i

```
i = 2;
cout << ri;        // → 2
ri = ri + 1;
cout << i;         // → 3
```


Variable de type référence (2)

- Une référence désigne toujours le même objet
  (!= des pointeurs)
- Elle doit être initialisée
- Elle ne peut être modifiée
- ri et i sont synonymes → une seul emplacement mémoire
- Pas d 'équivalent de pointeur NULL

Comparaison avec les pointeurs

- **Simplification** (pas de →, *, &)
  - à l'écriture de la fonction
  - et pour son utilisation
- **Gain de temps** car évite la copie de la donnée
- **Plus sécurisé** : on référence un seul et unique objet
- **MAIS** : effets de bord : l'utilisateur ne sait pas si la référence va être modifiée ou non (à moins de préciser **const**)

Passage de paramètre par valeur

```
void appel_par_valeur (int a)
{ a = 3;}
// a est modifiée localement dans la fonction

void main() {
    int val = 0;
    appel_par_valeur (val);
    // val vaut 0
}
```


Passage de paramètre par pointeur

```
void appel_par_pointeur (int * a)
{ *a = 3;}
// a est modifiée localement dans la fonction

void main() {
    int val = 0;
    appel_par_pointeur (&val);
    // val est modifié, il vaut 3
}
```


Passage de paramètre par référence

```
void appel_par_référence (int & a)
{ a = 3;}

void main() {
    int val = 0;
    appel_par_référence (val);
    // val est modifié, il vaut 3
}
```


Les différentes significations du caractère '&'

- Adresse d'une variable s'il préfixe le nom d'une variable
  - Exemple : int i=1;cout<<"adresse de i"<<&i<<endl;
- Paramètre par référence quand il suffixe un type dans la déclaration du paramètre d'une fonction
  - Exemple : void echange (int & a, int & b);
- Référence quand il suffixe le nom d'un type lors de la déclaration d'une variable
  - Exemple : int cpt1; int & cpt2 = cpt1;

Constructeur avec liste d'initialisation (1)

Complexe.cpp

```
Complexe::Complexe (float Re, float Im) : re ( Re),  im ( Im) {}
```

- Uniquement dans les constructeurs
- Pour initialiser les attributs membres d'une classe
- Pour éviter une affectation : re = Re;
- Utile pour les données membres de type référence ou constants qui ne peuvent être qu'initialisées

Constructeur avec liste d'initialisation (2)

1. Selon l'ordre de déclaration des membres, allocation mémoire et appels des constructeurs par défaut pour tous les attributs de la classe
2. Initialisation à partir de la liste d'initialisation si elle existe
3. Exécution du corps du constructeur

Initialisation des références et des constantes

```
class Biblio {
    Livre *tab;
    Livre & chevet;  // référence sur un livre
    int index ;
    const int code; // constant
public :
     Biblio (Livre &L) : chevet(L) , code(1234) {
          tab = new Livre [10]; index = 0;
    }
    void add(Livre & L) {
          tab[index ++] = L;
    }};
```


Fonction membre constante

- fonction de consultation de l'objet, qui ne modifie pas l'objet courant

```
#include "Complexe.h "

// fonction constante
ostream& Complexe::afficher(ostream& out) const {
out <<re << « i »<<im<<endl;
return out;}

// fonction non constante
void Complexe::modifier() { re+=10; }
```

Complexe.cpp

Utilisation de la fonction constante

```
void Complexe::testFonctions(const Complexe & c)
    {
    // on ne doit pas modifier c car constant
     c.afficher(cout);          // possible
     c.modifier();              // impossible
    }
```


Classe amie

- Si une classe B est déclarée amie d'une classe A, toutes les méthodes de la classe B ont accès aux données privées de la classe A.

```
class Livre {
   string titre; ….

   // Biblio a accès à toutes les données privées de Livre
   friend class Biblio;
   …
};
class Biblio {              …
   public:
            void edition() {
                     for (int i =0; i< indiceLibre; i++) {
                              cout << « Livre : « <<  tab[i].titre << tab[i].auteur ;
                              cout << endl;
                              }
   }};
```


Classe amie

- Le principe d'encapsulation n'est plus respecté
- C'est un compromis satisfaisant les contraintes de temps d'exécution
- Contrôle plus fin des droits d'accès : on peut spécifier un utilisateur privilégié

Variable statique

- Les objets, variables ou fonctions déclarés **static** sont communs à tous les objets de la classe
- Une variable de classe existe en un seul exemplaire pour tous les objets de la classe
- Elle est initialisée à l'extérieur de la classe (en début du .cpp)

```
class S { static int n ;
          int x;

   …};
                    S.h
```

```
#include « S.h »
 int S::n = 2;
void main() {
          S s1, s2;
…}
                    S.cpp
```

Fonction statique

- Non attachée à un objet
- Ne dispose pas du pointeur this
- Ne peut référencer que les fonctions et les membres statiques

Exemple

- Dans une gestion d'agendas, il est intéressant de connaître tous les agendas existants (liste statique)
- A chaque création, le pointeur this est ajouté à la liste.
- Méthode statique permettant d'éditer tous les objets d'une classe, …

```
static void tout_editer();
```

- Accès direct par

```
Agenda::tout_editer();
Agenda a; a.tout_editer();
```


Code

```
// Agenda.h
class Agenda {
    static list<Agenda *> agendas;
public :
    Agenda();
    static void toutEditer();
};
// Agenda.cpp
Agenda::Agenda() {…;
    Agenda::agendas.push_back(this);}

void Agenda:: toutEditer() {
    for (list<Agenda*> ::iterator it=Agenda::agendas.begin();
                                        it!= Agenda::agendas.end();it++)
            (*it)->afficher();
}
```


Compilation conditionnelle

- Pour éviter les définitions en double lors d'inclusions multiples

```
#ifndef CELLULE_H
#define CELLULE_H
class Cellule {
…
};
#endif

#ifndef LISTE_H
#define LISTE_H
class Liste {
…
};
#endif
```

Déclaration différée (1)

```
struct Impair;  // Déclaration différée
                // Pointeurs et références sur Impair autorisés
struct Pair {
        Impair *suiv;};
struct Impair {
        Pair *suiv;};
```


Déclaration différée (2)

```
class  Livre;  // Déclaration différée de Livre
               // Pointeurs et références sur Livre autorisés
class Biblio {
        Livre *tab;
…
};
```

Biblio.h

- Dans un fichier .h, remplacer les inclusions de types (i.e.classes) par des déclarations différées
- Les inclusions de .h sont retardés : ils sont plutôt inclus dans les fichiers .cpp quand on a réellement besoin de connaître un type de données

Espace de noms

- Pour ranger du code dans des boites virtuelles
- Regroupement logique d'identificateurs
- Eviter les conflits de noms entre plusieurs parties d'un même projet

Espace de nom *std*

- La STL (Standard Template Library) est définie à l'intérieur de l'espace de nom *std*
- Chaque classe, fonction ou variable doit donc être préfixée par std::

```
std::string nom("toto");
std::cout << nom;
```

- Ou bien il faut importer l'ensemble des symboles par la directive using namespace

```
#include <iostream>
using namespace std;
string nom("toto");
cout << nom;
```

La classe Tableau

Tableau.h

```
class Tableau {
    int taille, indice_libre;
    int * valeurs;
public:
    Tableau(int t = 100) ;
    Tableau(const Tableau& t);          //constructeur copie
    ~Tableau() ;                        //destructeur
};
```

```
Tableau::Tableau(int t) { A CODER }

Tableau:: Tableau(const Tableau& t) :  A CODER  {
            A CODER }
 Tableau:: ~Tableau() {A CODER ;}
```

Tableau.cpp