



Eléments du langage C++

Christine REGIS

2022-2023

M1 SRI

Equipe SMAC - IRIT - Université Paul Sabatier -Toulouse



Organisation

- Programme

- 19h dont

- 5h cours

- 6h TD

- 8h TP

- Exams

- Contact

- Mme Régis Christine – IRIT2- bureau 370 - 05 61 55 61 76 – christine.regis@irit.fr



Plan du cours

- Rappel du modèle objet
- Une classe en C++
- Constructeurs, destructeurs, copie
- Opérateurs
- Héritage
- Virtualité, polymorphisme, classes abstraites



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
- 1994 : Python 1.0
- 1995 : Java (Sun)
- 2000 et 2008 : Python 2.0 et Python 3.0



Comparaison JAVA/C++

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



Comparaison Python/C++

- Orientés objets
- Interprété/Compilé
- C++: puissance de calcul, mais développement coûteux, pour les applications critiques
- Python: langage moderne, lisible, portable, types de base très puissant, pas de gestion de la mémoire



Rappel du modèle objet

Chapitre 1



Monde = collection d'objets

- Entités physiques de taille variable
 - grains de sable, étoiles, êtres vivants, ...
- Concepts
 - univers, compte en banque, équation mathématique,...
- Les objets naissent, vivent et meurent au travers de leurs interactions

Un objet « Moto »

Etat

- Couleur
- Cylindrée
- Vitesse maximale

Comportement

- Démarrer
- Accélérer
- Freiner
- Stopper



Un objet « Livre »

Données

- titre
- auteur
- texte

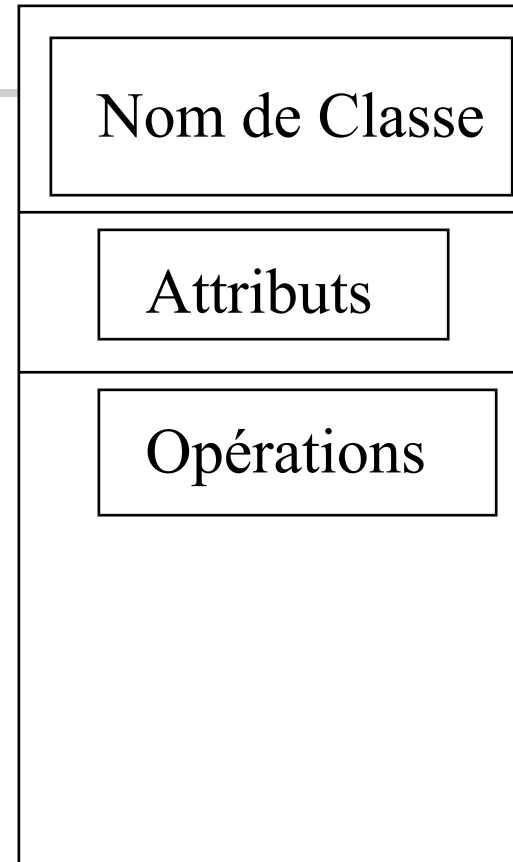
Fonctions

- saisir
- afficher
- lire

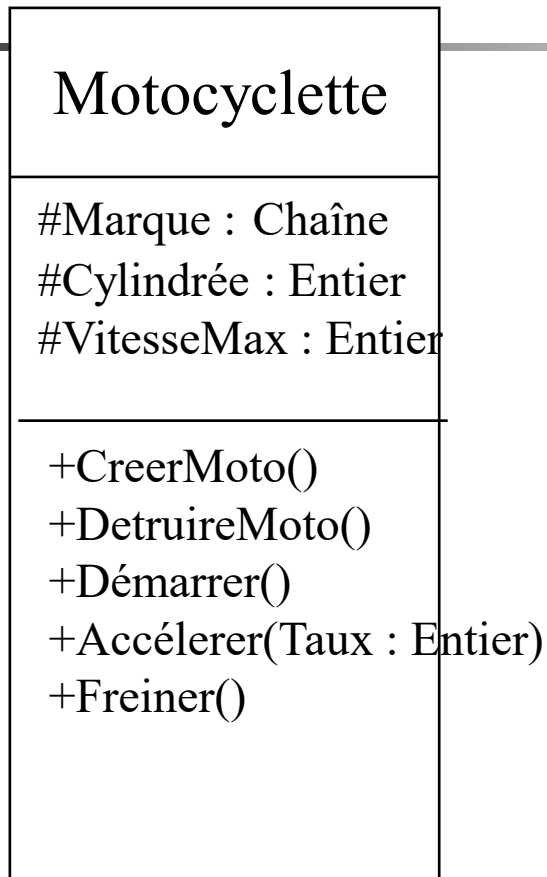


Les classes

- **Regroupement** d'objets qui se ressemblent pour réduire la complexité du monde réel
- Une classe définit un **moule** pour générer ses instances (les objets)
- Démarche **d'abstraction** : identifier les caractéristiques communes à un ensemble d'éléments



Notation UML d'une classe



Classes et objets : instantiation

- Chaque objet (instance) est formé à partir d'une classe
- Les objets contiennent la valeur des attributs déclarés dans la classe
- Ils partagent l'implantation des méthodes

```
void main(){  
  Livre leLion (« Le lion », »Kessel »);  
}
```

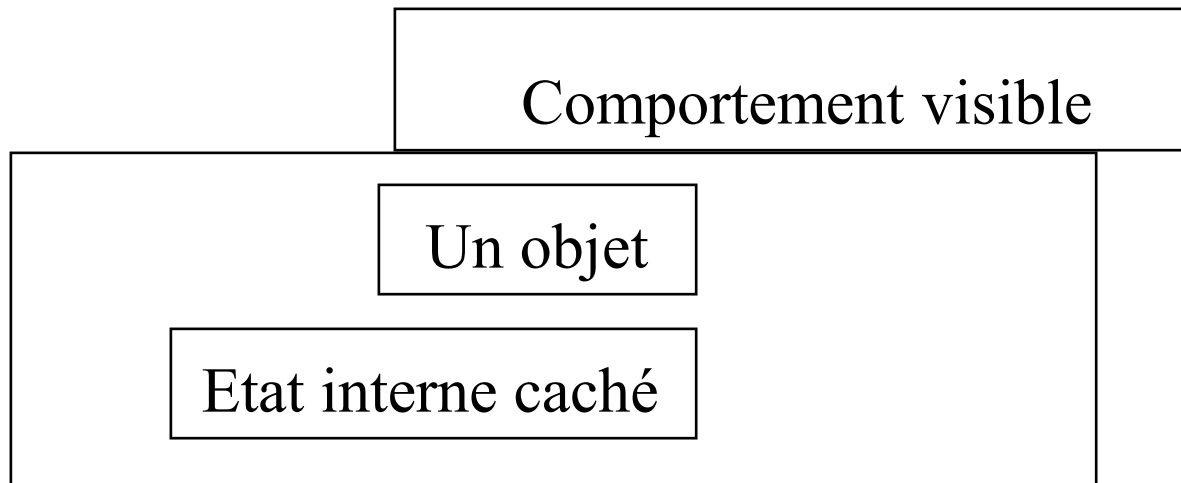
**objet « Le lion »
issu de la classe Livre**





L'encapsulation protège les données

- Accès aux données privées de l'objet par l'intermédiaire des procédures publiques
- L'objet est accessible via son comportement et non par ses données





Une classe en C++

Chapitre 2



Définition d'une classe « Livre »

```
class Livre {  
    string titre ;  
    string auteur;  
    string resume;  
  
    public :  
        void saisirInfos();  
        void afficherInfos();  
};
```

Livre.h



Interface d'une classe : NomDeClasse.h

```
class NomDeClasse
```

```
{
```

```
private :
```

```
    // variables cachées appelées “attributs”
```

```
    // accessibles depuis les méthodes de la classe
```

```
protected :
```

```
    // variables accessibles depuis la classe et ses sous-classes
```

```
public :
```

```
    // fonctions qui manipulent les variables cachées
```

```
    // appelées “méthodes”
```

```
};
```



Attributs publics et privés

- **private** indique que les éléments qui suivent sont cachés aux fonctions externes à la classe

```
void main() {  
    Livre monLivre;  
    cout << monLivre.titre; // accès illégal mais syntaxe correcte }  
}
```

- **public** indique que les fonctions qui suivent sont accessibles depuis l'extérieur de la classe
- Mode par défaut
 - public dans une structure
 - private dans une classe



Coder les méthodes dans Livre.cpp

```
#include « Livre.h »

void Livre::afficherInfos() {
    cout << titre;
    cout << auteur;
    cout << resume;
}

void Livre::saisirInfos() {
    cin >> titre;
    cin >> auteur;
    cin >> resume;
}
```



Entrées / Sorties

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

```
cout << « bonjour » <<endl;  
int entier;  
cin >> entier;
```



L'opérateur d'écriture <<

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

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



L'opérateur de lecture >>

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

```
#include <iostream>
using namespace std;
void main() {
    int nb, nb1, nb2; char c; float price;
    cout<<"Enter an integer : "; cin >> nb; cout <<"Integer nb : "<<nb<<endl;
    cout<<"Enter a letter : "; cin >> c; cout <<"Letter c : "<<c<<endl;
    cout<<"Enter a price : "; cin >> price; cout <<"Price : "<< price <<endl;
    cout<<"Enter two integers : ";
    cin >> nb1 >> nb2;
    cout<<"Integers nb1 : "<<nb1<< "\t nb2 : " <<nb2<<endl;
}
```



Un programme principal

Main.cpp

```
#include « Livre.h »  
int main()  
{  
    Livre monLivre;  
  
    monLivre.saisirInfos();  
    monLivre.afficherInfos();  
  
}
```



Autre programme pour saisir MAX livres

```
#include « Livre.h »Main.cpp  
#define MAX 10  
int main()  
{  
    Livre biblio[MAX]; // tableau de MAX objets Livre  
    int i;  
    for (i=0;i<MAX;i++)  
        biblio[i].saisirInfos();  
};
```




Le mot-clé this

- **this** est un pointeur sur l'objet courant qui a appelé la fonction

```
void main() {  
    Livre L ;  
  
    // la variable L est l'objet courant  
    // sur lequel s'applique la fonction  
    // saisir  
    L.saisirInfos();  
}
```

Main.cpp

- Deux codes possibles pour une même fonction

```
void Livre::saisirInfos() {  
    cin >> titre;  
    cin >> auteur;  
    cin >> resume;  
}  
  
void Livre::saisirInfos() {  
    cin >> this->titre;  
    cin >> this-> auteur;  
    cin >> this-> resume;  
}
```

Livre.cpp



Fonctions d'accès aux attributs

- Elles autorisent l'accès en lecture et /ou en écriture aux attributs privés d'une classe
- Appelées aussi accesseurs
- Utilisées par une autre classe ou dans le programme principal



Accesseurs dans la classe Livre (Livre.cpp)

```
#include « Livre.h »
```

```
// Fonctions d'accès en lecture
```

```
    string Livre::getTitre() { return titre;}
```

```
    string Livre::getAuteur() { return auteur;}
```

```
    string Livre:: getResume() { return resume;}
```

```
// Fonctions d'accès en écriture
```

```
    void Livre:: setTitre ( string t) { titre = t;}
```

```
    void Livre:: setAuteur ( string a) { auteur = a;}
```

```
    void Livre:: setResume(string r) {resume = r;}
```

L
i
v
r
e



Livre.h

```
class Livre {  
    string titre ;  
    string auteur;  
    string resume;  
public :  
    void saisirInfos();  
    void afficherInfos();  
    string getTitre() { return titre;}  
    string getAuteur() { return auteur;}  
    string getResume() { return resume;}  
    void setTitre ( string t) { titre = t;}  
    void setAuteur ( string a) { auteur = a;}  
    void setResume(string r) { resume = r;}  
};
```



Compilation puis exécution

- Compiler la classe Livre

`g++ -c Livre.cpp`

- Compiler le programme principal

`g++ -c Main.cpp`

- Faire l'édition de liens de tous les fichiers pour produire le fichier exécutable

`g++ Livre.o Main.o -o Livre`

- Lancer le fichier exécutable

`./Livre`



Spécification d'une classe Compte

- Un compte a un solde (float) initialisé à 0 //constructeur
- On peut faire un dépôt sur le compte
- On peut faire un retrait sur le compte
- On peut afficher le solde
- On peut obtenir la valeur du solde // getSolde()
- On peut virer de l'argent vers un compte destinataire



Différences entre C et C++

Chapitre 3



Les paramètres d'une fonction

- En C++, l'objet courant est toujours accessible dans la fonction via la variable `this`

`L.saisirInfos();`

- En C, on passe toujours l'objet sur lequel on travaille en paramètre de la fonction

`saisirInfos(L);`



Les déclarations de variables

- Une déclaration est une instruction.
- Il n'est plus obligatoire de les regrouper en début de bloc
- La déclaration d'une variable peut se faire juste avant son utilisation.

```
int i =0;
```

```
i= i+1;
```

```
int j ;
```

```
j= i % 2; //modulo
```



Union, struct, enum

- En C :
 - un identificateur de type est introduit par **typedef**.
`typedef struct {string nom;} Personne ;`
`Personne Paul;`
- En C++, l'étiquette représente le type
`enum Couleur {Rouge, Vert, Bleu};`
`Couleur c;`
`struct Personne {string nom;};`
`Personne Paul;`



Paramètres avec valeurs par défaut (1)

- Possibilité de donner des valeurs par défaut aux paramètres d'une fonction

```
float prixTTC(float prixHT, float TTC = 0.05) {  
    return (prixHT*(1+TTC));  
}  
  
void main() {  
    cout << prixTTC(100) << endl;  
    cout << prixTTC(100, 0.196) << endl;  
}
```



Paramètres avec valeurs par défaut (2)

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

```
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

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



Paramètres avec valeurs par défaut (3)

- Soit la déclaration de la fonction suivante :
`int exemple1(int a, int b=5, int c=10);`

- Les appels ci-dessous sont-ils corrects?
 - `exemple1(2);`
 - `exemple1(2,10);`
 - `exemple1(2,10,15);`
 - `exemple1(2, ,15);`
 - `int exemple1(int a, int b=5, int c);`



La surcharge de fonctions (1)

- Capacité de donner le même nom à plusieurs fonctions
- Mais fonctions différenciées par le type et/ou le nombre de paramètres

`void afficher (int tab[], int taille);`

`void afficher (float tab[], int taille);`

`void afficher (float mat[] [], int dim1, int dim2);`

Sens des fonctions identiques \Leftrightarrow même nom



La surcharge de fonctions (2)

(1) `int abs (int n) { return (n<0)? -n : n; }`

(2) `float abs (float f) { return (f<0.0)? -f : f; }`

```
int N; float F;
```

```
abs(N);        // appel de (1)
```

```
abs(F);        // appel de (2)
```

```
abs('a'); // appel de (1) après conversion de 'a' en int
```



Surcharge + valeurs par défaut

Main.cpp

```
void test (int n = 0, double x = 0); // test1
void test (double y = 0, int p = 0); // test2
void main() {
    int N; double Z;
    test(N, Z);           // appel de test1
    test(Z, N);           // appel de test2
    test(N);              // appel de test1
    test(Z);              // appel de test2
    test();               // erreur car ambiguïté
}
```




Constructeur

Chapitre 4



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;  
    string texte;  
public :  
    Livre(string t, string a, string  
    texte) ;  
};
```

L

```
Livre::Livre(string t, string a,  
    string 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);
    Complexe (float Re); // Constructeur surchargé
    Complexe (); // Constructeur sans paramètre
};
```

Complexe.h

```
Complexe ::Complexe (float Re, float Im) {re = Re; im = Im;}
Complexe ::Complexe (float Re) {re = Re; im = 0;}
Complexe ::Complexe () { re = 0.0; im = 0.0;}
```

Complexe.cpp



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 », string texte =« ») {  
    this->titre = t;  
    this->auteur = a;  
    this->texte = texte;  
}
```

Livre.cpp



Exemples d'appels

```
void main() {  
    // 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;  
}
```

Main.cpp



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

```
void main() {
```

Main.cpp

```
    Complexe C; //OUI, appel à Complexe()
```

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

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

```
}
```




Attention !

- Si aucun constructeur n'est défini dans la classe, C++ en utilise un d'office qui va réserver de la mémoire pour toutes les variables de la classe



Références, constructeur copie destructeur

Chapitre 5



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 adresse

```
void appel_par_pointeur (int * a)
{ *a = 3;}
// a est modifiée durablement 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
}
```



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 \rightarrow , $*$, $\&$)
 - 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**)



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;`



Destructeur

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

`~Classe()`

- 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



Constructeur copie (clonage)

- Construction d'un objet identique à un autre

`Nom_classe (Nom_classe &);`

`Nom_classe (const Nom_classe &);`

- 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

```
#include " Chaine.h"
```

Main.cpp

```
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
```

```
}
```



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



Coder une classe Tableau

- Tableau d'entiers (`int *valeurs`, `int nbElem`, `int tailleMax`)
- Constructeur
- Constructeur copie
- Destructeur



Tableau.h

```
class Tableau {  
    Int *valeurs;  
    Int tailleMzax;  
    Int nbElem;  
    Public:  
    Tableau(int = 100);  
    Tableau(Tableau&);  
    ~Tableau();  
};
```



Tableau.cpp

```
#include « Tableau.h »
```

```
Tableau::Tableau(int t) {
```

```
    this->maxTaille = t;
```

```
    this->nbElem=0;
```

```
    this->valeurs=new int[t];
```

```
}
```

```
Tableau::Tableau(Tableau& copie) {
```

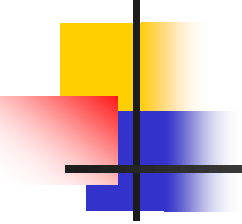
```
    this->maxTaille = copie.maxTaille;
```

```
    this->nbElem=copie.nbElem;
```

```
    this->valeurs=new int[maxTaille];
```

```
    for (int i=0; i<nbElem; i++) this->valeurs[i] = copie.valeurs[i];
```

```
}
```



```
Tableau::~~Tableau() {  
    delete [] valeurs;  
}
```



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



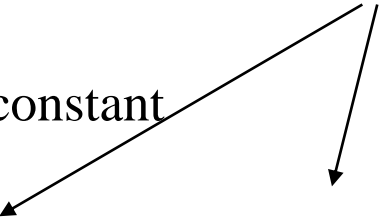
Constructeur avec liste d'initialisation (2)

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

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

Tableau.h

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

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

```
Tableau:: ~Tableau() {A CODER ;}
```

Tableau.cpp