Langage de Programmation Orientée Objet : C++

Khalid GABER

Passage du C au C++

Commentaire et Type construit

- Commentaires :

  // ceci est un commentaire

- type énuméré, n-uplet :

  enum *Couleur* {rouge,vert,orange};

  struct *Compte* { int code, float solde};

  ...

  Couleur feux ; // var. de type énuméré

  Compte cpte; // var. de type n-uplet

Prototype de fonctions

- le type **void**
  - ☞ indique qu'une fonction ne renvoie pas de valeur.

```
void f(int c) { .... };
```

- prototype de fonctions :
  - ☞ signature de la fonction
  - ☞ permet un contrôle

```
void echanger(int, int);
```

Surcharge (surdéfinition) des fonctions

- C++ permet de distinguer deux fonctions de mêmes noms sur le type de leurs arguments d'appels :
  - void init(float r) { ... };
  - void init(char *chaine) { ... };
- paramètre par défaut :
  - void test(int i, float x = 6.23) {... };
  - test(5);
  - test(9, 80.4);

Passage de Paramètres par Copie

- Ce mode de passage consiste à ne déposer que l'adresse de l'argument dans la pile d'exécution au moment de l'appel.
  - ☞nécessite l'utilisation de pointeurs pour modifier un argument.

```
// version adresses explicites :
    void echange( int *x, int *y)
    { int z = *x;
      *x = *y; *y = z; };
// utilisation :
    d=4; e=6;
    echange( &d, &e);
    cout << d  << " " << e << endl;
```

Passage de Paramètres par référence

- Manipulation implicite des adresses en rendant le mode de passage transparent au client.

```
// version adresses implicites :
void echange(int &x, int &y)
{  int z = x;
   x = y;   y = z;
}
d=4; e=6;
echange(d,e);
cout << d  << " " << e << endl;
```

Gestion de la mémoire dynamique

- Opérateurs **new** et **delete**
  - ☞ new : allocation de mémoire sur le tas

```
float* r = new  float;                  // var. élémentaire
float* r = new ( float );               // variante

float* t = new float[20];               //  var. tableau
```

  - ☞ delete : restitution de la mémoire

```
delete r;          // var. élémentaire
delete [] t;       // var. tableau
```

Sécurité des fonctions

- « **const** » permet de :
  - ☞ protéger le résultat d'une fonction :

```
const int lire_code(Employe  e);
```

  - ☞ spécifier qu'un argument ne peut être que LU :

```
void maj_nom(Employe &e, const char *n);
```

Entrée- Sortie

- Flots (stream)
  - ☞ entrée : cin (stdin en C)
  - ☞ sortie : cout (stdout en C)
  - ☞ erreur : cerr (sterr en C)
- Opérateur d'écriture <<

```
cout << "Nom : " << emp.nom << endl;
```

- Opérateur de lecture >>

```
cin >> emp.code;
```

- Les entrées/sorties sont formatées par défaut.

E/S : gestion des délimiteurs

- En C++ les séparateurs :
  - ☞ne sont pas traités en tant que caractère par l'opérateur "**>>**"
  - ☞mais servent à séparer 2 valeurs consécutives dans le fichier d'entrée standard
- get(char &) traite les séparateurs comme des caractères normaux :

```
#include <iostream>
void main()
{  char c;
   while (cin.get(c))  cout << c;   }
```

E/S : fichier (1)

- librairie : fstream.h

- 3 types prédéfinies de fichier
  - ☞ifstream lecture seule
  - ☞ofstream écriture seule
  - ☞fstream lecture/écriture

- toutes les primitives de manipulations de flot peuvent être appliqués à des flots auxquels on a rattaché un fichier.

E/S : fichier

- Association d'un fichier et d'un flot
  - open(char *nomDuFichier)
    - ouvre un fichier en l'associant à un flot déclaré précédemment
  - close()
    - ferme le fichier associé au flot
    - coupe la liaison flot/fichier

E/S : fichier (Exemple)

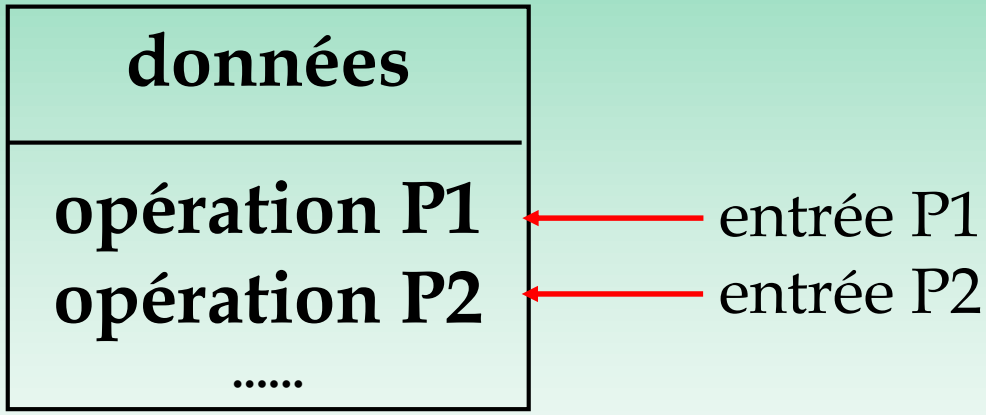
```
#include <fstream.h>
#include <stdlib.h>

void main()
{ ifstream monFichier;

  monFichier.open("test.txt");
  while (monFichier >> car)  nbre++;
  monFichier.close();
  ….
}
```

Encapsulation ?

- regroupement sous un même NOM :
  - ☞des données
  - ☞des opérations manipulant les données



- **une opération appartenant à une encapsulation ne peut manipuler que les données de cette encapsulation.**

Une encapsulation est

- Vue de l'extérieur
  - ☞une région mémoire
  - ☞une entité unique et indépendante
  - ☞une liste de points d'entrée

- Vue de l'intérieur
  - ☞des données structurées locales
  - ☞des opérations locales correspondant chacune à un point d'entrée et qui manipulent ces données

Prototypes et exemplaires

- un Prototype représente les objets des propriétés communes :
  - ☞factuelles : attributs,
  - ☞comportementales : méthodes.

Vocabulaire en C++

- **Classe** : prototype
- **Instances** : exemplaires créés à partir du prototype

Classe et Instance en C++

- classe :

```
class CCercle {
        CPoint  centre;          // attributs
        CPositif rayon  ;

        void init ();            // méthodes
        void dessinesToi();
      };
```

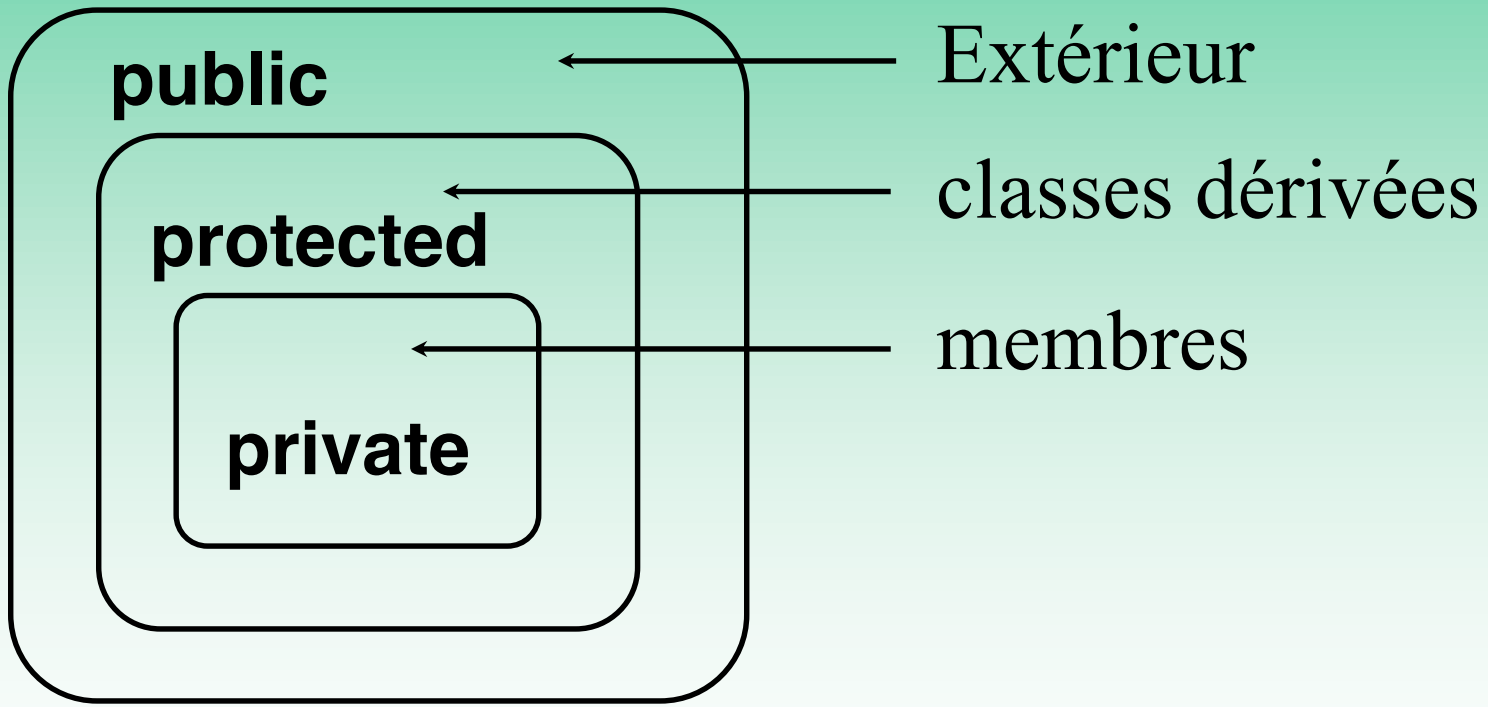
- instance :

```
CCercle  C;
```

- initialisation :

```
C.init();
```

Structure d'une Classe



Déclaration d'une Classe

- fournir un nouveau type d'Objets.

- déclarer les **MEMBRES** de la classe :
  - ☞**Attributs** : données-membres;
  - ☞**Méthodes** : fonctions-membres.

- déclarer des fonctions "**amies**" :
  - ☞ n'appartenant pas à la classe;
  - ☞ se comportant comme des « méthodes ».

Méthodes

- lors de la déclaration de la classe, les méthodes seront :
  - ☞déclarée :

    signature : type et arguments.

  - ☞déclarée ET définie (**inline**) :

    description du corps de la fonction.

Méthodes

- Une Fonction membre peut :
  - ☞ recevoir tous les types d'arguments
  - ☞ retourner :
    - ♦ des expressions de tous types,
    - ♦ des pointeurs sur des objets de types divers,
    - ♦ des objets de type référence sur des types divers.

Méthodes inline

```
class Employe
{
   public:
     int getMatricule()  { return matricule; }   // fonction inline
     void setMatricule(int);                     // descriptions séparées
     void voir();
   private:
     int matricule;
};
```

Définir une méthode

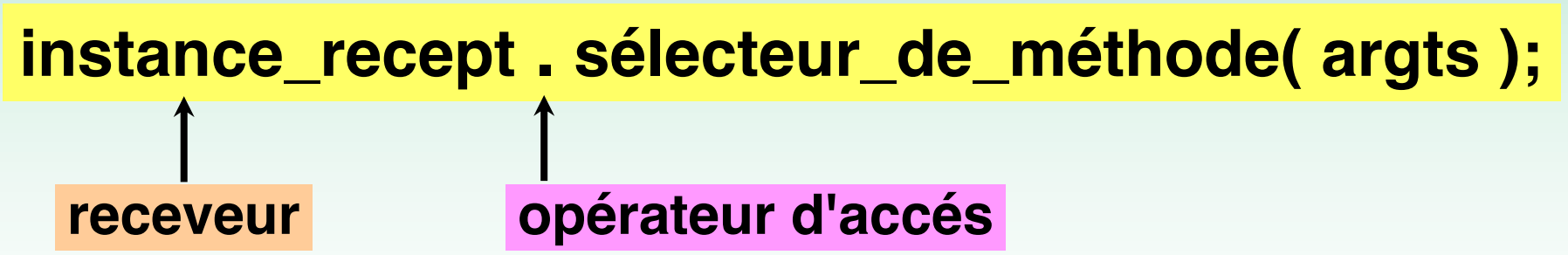
nom_classe :: nom_méthode(arguments) { ... }

```
void Employe::voir()
{  cout << "matricule : " << matricule << endl; }

void Employe::setMatricule(int m)
{  (*this).matricule = m; }
```

Message

- Définition d'une transmission de message :
  - ☞Receveur,
  - ☞Sélecteur de méthodes,
  - ☞Arguments.



Accès à un membre

- Opérateurs d'accès à membre (attributs ou méthodes) :
  - ☞. pour les instances de la classe
  - ☞-> pour les pointeurs sur des instances.

```
Employe martin;
Employe *pDupond;

martin.getMatricule();    // instance
pDupond->voir();          // pointeur sur une instance
```

Constructeur

- permet **l'initialisation** automatique d'une **instance** de **classe**, lors de sa déclaration.

- un **constructeur** est une fonction membre :
  - ☞qui ne renvoi pas de valeur,
  - ☞qui porte le nom de la classe à laquelle elle appartient.

- un constructeur peut être **surchargé** (comme toute fonction).

Constructeur (exemple)

Allocation dynamique d'un certain nombre de caractères au moment de la définition d'un objet de type Chaîne :

```
class Chaine {
                char* ch;            // champ privé
        public :
                Chaine();             // constructeur par défaut
                Chaine(int);          //constructeur Spécialisé
                void saisie(char *);
                void affiche()  { cout <<ch<<endl; }
                ~Chaine();            //destructeur
};
Chaine::Chaine()
{  ch = new char[100];
   cout <<« constructeur par defaut\n »; }
```

Constructeur (exemple)

```
Chaine::Chaine(int taille)
{ ch = new char[taille];
  cout <<« constructeur Spesialise\n »;
 }
void Chaine::saisie(char *s)
{   strcpy(ch,s);
}
Chaine::~Chaine()
{ delete [] ch;
  cout << « Destructeur ……..\n »;
}
```

Constructeur - suite

le constructeur est surchargé deux fois:

- une fois pour allouer par défaut 100 octets,
- une seconde fois pour allouer un nombre d'octets donnés

Constructeur (exemple)

```
main ()
{
   Chaine une;            // constr. par défaut
   Chaine deux(20);       // constr. spécialisé
...
}
```

Constructeur de Copie

- **initialiser** une instance avec une autre instance de la même Classe.
- **indispensable** pour le passage de paramètre par Copie.

Chaine s1 = s2;

- copie standard bit à bit, de *s2* dans *s1*
- si l'un des champs est dynamique, on risque de détruire 2 fois ce champ

Constructeur de Copie

```
class Chaine {
 private:
        char *ch;     // champ privé
 public:
        Chaine();    // constructeur par défaut
        Chaine(int);              // constructeur spécialisé
        Chaine( const Chaine &);     // constructeur de copie
};
Chaine::Chaine(const Chaine& e) {
         this->ch = new char[strlen(e.ch) +1];            // nouvel objet dynamique
         strcpy(this->ch,e.ch);          // recopie de la valeur de e
       Cout << « constructeur de copie ……\n»
}
```

Surcharges des Opérateurs et Fonctions Amies