COLLEGE DE PARIS SUPÉRIEUR

DAKARTECH

Programmation orientée objets [C++]

Dr. Pape Abdoulaye BARRO

La POO

- **Notion de Classe**
- Fonctions membres
- Constructeurs, destructeurs et initialisation d'objet
- Héritage simple
- Héritage multiple
- Fonctions virtuelles et polymorphisme
- La gestion des exceptions
- Généralités sur la STL

Notion de Classe

Dans cette section, nous abordons véritablement les possibilités de la P.O.O en C++, qui comme introduit initialement est entièrement basée sur le concept de Classe.

Une **classe** est un type de données dont le rôle est de rassembler sous un même nom à la fois données et traitements. La **notion de classe** n'est pas trop loin de la **notion de structure**:

- La déclaration d'une classe est presque similaire de celle d'une structure.
  - Il suffit de remplacer le mot clé ***struct*** par le mot clé ***class***;
  - Puis de préciser les fonctions ou données membres publics avec le mot clé ***public*** et les membres privés avec le mot clé ***private***.
- Syntaxe:

```
class X
{
        private :
                ...
        public :
                ...
};
```

Algo

```
Classe mon_objet
    attributs privés
        attr1 :entier
        attr2 :tableau[10] de réels
        ...
    méthodes publiques
        Constructeur mon_objet(<params>)
        Destructeur ~mon_objet()
        procédure afficher()
        procédure effacer()
.    ..
FinClasse
```

Classe

Déclaration

En C++, la programmation d'une classe se fait en trois phases: déclaration, définition et utilisation.

- **Déclaration**: c'est la **partie interface** de la classe. Elle se fait dans un fichier dont le nom se termine par .h ou .hpp, .H, ou .h++ (**appelé fichier d'entête**).
  - **La syntaxe est la suivante**:

```
class Nom_de_la_classe {
        public:
        // déclarations des données et fonctions-membres publiques
        private:
        // déclarations des données et fonctions-membres privées
};
```

  - **Exemple:**

```
// Personne.hpp
class Personne {
        public:
          Personne(string, string);
          void toString();
        private:
          string p_nom;
          string p_prenom;
};
```

Classe

Définition

- **Définition**: c'est la **partie implémentation** de la classe. Elle se fait dans un fichier dont le nom se termine par .cc, .c++, .c ou .cpp. Ce fichier contient les **définitions des fonctions-membres** de la classe.

  - **Exemple**:

```
//Personne.cpp
#include <iostream>
#include "Personne.hpp"

// constructeur (Il permet d'initialiser une nouvelle personne)
Personne::Personne(string prenom, string nom){
        p_prenom = prenom;
        p_nom = nom;
}

// une méthode (fonctions-membres)
void Personne::toString(){
        cout <<"Je m'appel"<<p_prenom<<" "<<p_nom<<endl;
}
```

Classe

Utilisation

- **Utilisation**: Elle se fait dans un fichier dont le nom se termine par .cc, .c++, .c ou .cpp. Ce fichier contient le **traitement principal** (la fonction ***main***).

  - **Exemple**: // test.cpp

```
#include <iostream>
#include "Personne.hpp"

// traitement principal
void main(){
        //  appel implicite du constructeur
        Personne p("Nabi" , "Barro");

        p.toString();
}
```

Classe

Constructeurs et destructeurs

- Un **constructeur** est une fonction-membre déclarée du même nom que la classe, et sans type.
  - **Syntaxe** : **Nom_de_la_classe(<paramètres>)**;
  - **Fonctionnement**: à l'exécution, l'appel au constructeur produit un nouvel objet de la classe, dont on peut prévoir l'initialisation des données-membres dans la définition du constructeur.
    - **Exemple**: personne p("Nabi" , "Barro");
  - Dans une classe, il peut y avoir plusieurs constructeurs à condition qu'ils différent par le nombre ou le type des paramètres.
    - *Un constructeur sans paramètre s'appelle **constructeur par défaut**.*
- Un **destructeur** est une fonction-membre déclarée du même nom que la classe mais précédé d'un **tilde** (**~**) et ***sans type ni paramètre***.
  - **Syntaxe** : **~Nom_de_la_ classe()**;
  - **Fonctionnement**: à l'issue de l'exécution d'un bloc, le destructeur est automatiquement appelé pour chaque objet de la classe déclaré dans ce bloc. Cela permet par exemple de programmer la restitution d'un environnement, en libérant un espace mémoire alloué par l'objet.

Classe

Constructeurs et destructeurs

Exemple:

L'exemple ci-dessous est un petit programme mettant en évidence les moments où sont appelés respectivement le **constructeur** et le **destructeur** d'une classe.

- Nous créons une classe test définissant que ces deux fonctions-membres et une donnée membre ***num*** qui sera initialisée par le constructeur nous permettant d'identifier l'objet en question.

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

class test
{
    public :
        int num ;
        test (int) ; // déclaration constructeur
        ~test () ; // déclaration destructeur
} ;
```

```
test::test (int n)
{
    num = n ;
    cout << ''Appel du constructeur, avec num='' << num<<endl;
}
```

```
test::~test ()
{
    cout << ''Appel du destructeur, avec num='' << num<<endl;
}
```

Classe

Constructeurs et destructeurs

Illustration du déroulement de la libération des ressources:

Nous créons des instances de type test à deux endroits différents:

- dans la fonction ***main*** d'une part,
- dans une fonction ***fct*** appelée par ***main*** d'autre part.

```
main()
{
        void fct (int) ;
        test a(1) ;

        for (int i=1 ; i<=2 ; i++)
             fct(i) ;
}

void fct (int n)
{
    test t(2*n) ;
}
```

```
-> Appel du constructeur, avec num=1
-> Appel du constructeur, avec num=2
-> Appel du destructeur, avec num=2
-> Appel du constructeur, avec num=4
-> Appel du destructeur, avec num=4
-> Appel du destructeur, avec num=1
```

Classe

Constructeurs et destructeurs

Quelques règles usuelles:

- Un **constructeur** peut éventuellement comporter un nombre quelconque d'arguments.
- un **constructeur** n'a pas de type de retour et par conséquent, ne renvoie pas de valeur. La présence de void est dans ce cas une erreur.
- Un **destructeur** n'a pas de type de retour et donc, ne renvoie pas de valeur. Ici aussi, la présence de void est une erreur.
- Les **constructeurs** et les **destructeurs** peuvent être **publics** ou **privés**. En pratique, à moins d'avoir de bonnes raisons de faire le contraire, il vaut mieux les rendre **publics**.

Classe

Visibilité des membres d'une classe

La **visibilité des attributs et méthodes** d'une classe est définie dans l'interface de la classe grâce aux mots-clés **public**, **private** ou **protected** qui permettent de préciser leurs types accès.

- **public**: autorise l'accès pour tous. Exemple: le constructeur ainsi que la méthode toString() de l'exemple précèdent peuvent être utilisés partout sur une instance de **personne**.
- **private**: restreint l'accès aux seuls corps des méthodes de cette classe. Sur l'exemple précédent, les attributs privés p_prenom et p_nom ne sont accessibles sur une instance de **personne** que dans les corps des méthodes de la classe. Ainsi, la méthode toString() a le droit d'accès sur ses attributs p_prenom et p_nom. Elle aurait aussi l'accès aux attributs p_prenom et p_nom d'une autre instance de **personne**.
- **protected**: comme private sauf que l'accès est aussi autorisé aux corps des méthodes des classes qui héritent de cette classe.

Classe

Objet

Un **objet** est une instance d'une classe (c'est-à-dire, une variable dont le type est une classe). Un objet occupe donc de l'espace mémoire. Il peut être alloué :

- ***statiquement***: dans ce cas, on met le nom de la classe suivi du nom de l'objet et éventuellement suivi par des arguments d'appel donnés à un constructeur de la classe.

  Exemple: Personne p("Nabi" , "Barro");

- ou ***dynamiquement***: dans ce cas, un pointeur sur la zone mémoire ou l'objet a été alloué est retourné. Lorsqu'on n'en a plus besoin, on le libère avec l'opérateur *delete*.

  Exemple:

```
personne * p = new personne("Nabi" , "Barro");
p->toString();
delete p;
```

Classe

Exemple complet

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

// déclaration de la classe
class personne
{
        public:
        personne(string, string);
        void toString();

        private:
        string p_nom;
        string p_prenom;
  };

// définition des membres publics
// constructeur
personne::personne(string prenom, string nom){
        p_prenom = prenom;
        p_nom = nom;
}
// une méthode
void personne::toString(){
        cout <<"Je m'appel"<<p_prenom<<" "<<p_nom<<endl;
}
```

statiquement

```
// Utilisation de la classe personne
void main(){
        //  appel implicite du constructeur
        personne p("Nabi" , "Barro");
        p.toString();
}
```

dynamiquement

```
// Utilisation de la classe personne
void main(){
        //  appel implicite du constructeur
        personne * p = new personne("Nabi" , "Barro");
            p->toString();
            delete p;
}
```

La POO

- Notion de Classe
- **Propriétés des fonctions membres**
- Constructeurs, destructeurs et initialisation d'objet
- Héritage simple
- Héritage multiple
- fonctions virtuelles et le polymorphisme
- La gestion des exceptions
- Généralités sur la STL

Propriétés des fonctions membres

Précédemment, nous avons vu les concepts fondamentaux de la notion de Classe. Ici, nous allons étudier en profondeurs les **fonctions membres** et les quelques possibilités offertes par C++.

- **Surdéfinition des fonctions membres**

Ces possibilités sont déjà offertes pour les fonctions (indépendantes). Il s'agit simplement d'une généralisation aux fonctions membres. Seulement, ici, il faut tenir compte de la visibilité (privée ou publique).

Exemple:

```
class personne
{
        public:
          personne();        // constructeur (sans arguments)
          personne(string); // constructeur (avec 1 argument)
          personne(string, string); // constructeur (avec 2 arguments)
          void toString();  // fonction membre (sans arguments)
          void toString(string); // fonction membre (avec 1 argument)

        private:
          string p_nom;
          string p_prenom;
 };
```

Propriétés des fonctions membres

- **Arguments par défaut**

Comme pour les fonctions indépendantes, les fonctions membres peuvent disposer d'arguments par défaut.

Exemple:

```
class personne
{
        public:
          personne();        // constructeur (sans arguments)
          personne(string); // constructeur (avec 1 argument)
          personne(string, string); // constructeur (avec 2 arguments)
          void toString(string s= "Bonjour"); // fonction membre (avec 1 argument par défaut)

        private:
          string p_nom;
          string p_prenom;
  };
```

définition

```
// la méthode
void personne::toString(string s){
        cout << s <<p_prenom<<" "<<p_nom<<endl;
}
```

Propriétés des fonctions membres

- **Fonctions membres en ligne**

C++ permet la définition de fonctions en ligne pour accroitre l'efficience d'un programme dans le **cas de fonctions courtes**. Cette propriété s'applique aussi au fonctions membres avec quelques nuances en ce qui concerne sa mise en œuvre.

Deux cas de figure s'offre en nous pour rendre en ligne une fonction membre.

- On peut fournir directement la définition de la fonction dans la déclaration. Dans ce cas, le qualificatif *inline* n'a pas à être utilisé;

```
class personne
    {
        public:
          personne() {p_nom =  " " ; p_prenom =  " " ; }                    // constructeur 1 « en ligne »
          personne(string np) {p_prenom =  p_prenom= np; }                  // constructeur 2 « en ligne »
          personne(string n, string p) {p_nom =  n; p_prenom =  p; }        // constructeur 3 « en ligne »
          void toString(string s= "Bonjour");

        private:
          string p_nom;
          string p_prenom;
    };
```

définition

```
// la méthode
void personne::toString(string s){
        cout << s <<p_prenom<<" "<<p_nom<<endl;
}
```

Propriétés des fonctions membres

- **Fonctions membres en ligne**
  - On peut aussi procéder comme pour une fonction ordinaire en fournissant une définition en dehors de la déclaration de la classe. Dans ce cas, le qualificatif *inline* doit apparaitre à la fois devant la déclaration et devant l'entête;

```
class personne
    {
        public:
          inline personne() ;          // constructeur « en ligne »
          …

        private:
          string p_nom;
          string p_prenom;
    };

    int main()
    {
          personne p ; // appel du constructeur
          p.toString(" Hello") ;
    }
```

définition

```
// le constructeur en ligne
inline personne::personne() {p_nom =  " " ; p_prenom =  " " ; }
```

Propriétés des fonctions membres

- **Transmission d'objets en argument à une fonction membre**

Une fonction membre peut recevoir un ou plusieurs arguments du type de sa classe. Par exemple, supposez que nous souhaitions, au sein d'une classe personne, introduire une fonction membre nommée coïncide, chargée de voir si deux personnes ont le même nom et prenom.

```
class personne
    {
        public:
          personne() {p_nom =  " " ; p_prenom =  " " ; }
          …
          bool coincide(personne);

        private:
          string p_nom;
          string p_prenom;
     };

     int main()
     {
          personne p1, p2 ;
          ….
          cout<< " p1 et p2 ont elles le même noms et le même prenom ? "<<p1.coincide(p2);
     }
```

définition

```
bool personne:: coincide(personne p)
{
    return ((p.p_nom == p_nom )&&  (p.p_prenom ==  p_prenom));
 }
```

Propriétés des fonctions membres

- **Mode de transmission des objets en argument**

Sur l'exemple précèdent, la personne p2 a été **transmis par valeur** à la fonction membre **coïncide**. Il serait donc possible de prévoir le mode de **transmettre par adresse** plutôt que la valeur, ou de mettre en place le mode de transmission par référence.

- **Transmission par adresse d'un objet**

Comme pour les fonctions ordinaires, le principe reste le même. Voici une adaptation de la fonction membre ***coincide***.

```
class personne {
        public:
          personne() {p_nom =  " " ; p_prenom =  " " ; }
          …
          bool coincide(personne *);

        private:
          string p_nom;
          string p_prenom;
};

int main(){
        personne p1, p2 ;
        ….
        cout<< " p1 et p2 sont elles des amies ? "<<p1.coincide(&p2);
}
```

définition

```
bool personne:: coincide(personne * p)
{
    return ((p->p_nom ==  p_nom )&&  (p->p_prenom ==  p_prenom));
 }
```

Propriétés des fonctions membres

- **Transmission par référence d'un objet**

Comme nous l'avons vu, l'emploi des références va permettre de mettre en place une transmission par adresse, sans avoir à le gérer soi-même. Voici une adaptation de la fonction membre ***coincide***.

```
class personne {
        public:
          personne() {p_nom =  " " ; p_prenom =  " " ; }
          …
          bool coincide(personne &);

        private:
          string p_nom;
          string p_prenom;
    };

    int main(){
          personne p1, p2 ;
          ….
          cout<< " p1 et p2 sont elles des amies ? "<<p1.coincide(p2);
    }
```

définition

```
bool personne:: coincide(personne &p)
{
    return ((p.p_nom ==  p_nom )&&  (p.p_prenom ==  p_prenom));
 }
```

Propriétés des fonctions membres

- **Une fonction renvoyant un objet**

Les propriétés qui s'applique sur les arguments d'une fonction membre s'applique également à sa valeur de retour qui peut être un objet. Cette objet pourra être du même type que la classe(*auquel cas la fonction aura accès à ses membres privés*) oubien un type différent de la classe(*auquel cas la fonction n'aura accès qu'à ses membres publics*).

- La transmission par valeur suscite la même remarque que précédemment: par défaut, elle se fait par simple recopie de l'objet.
- En revanche, la transmission par adresse ou la transmission par référence risquent de poser un problème qui n'existait pas pour les arguments. *Ici, il va falloir* ***éviter que l'objet de retour soit un objet local à la fonction*** *car l'emplacement de ce dernier sera libéré dès la sortie de celle-ci et la fonction appelante récupérera l'adresse d'un objet qui n'existe plus.*

définition

```
personne personne:: homonyme()
{
    personne p;
    p.p_nom =  p_nom;
    p.p_prenom =  p_prenom;
    return p;
 }
```

Il est donc déconseillé d'en prévoir une transmission par référence, en utilisant cet en-tête :

personne & personne::homonyme()

Propriétés des fonctions membres

- **Autoréférence**

  En C++, nous avons le mot clé ***this*** *utilisable uniquement au sein d'une fonction membre et désignant un pointeur sur l'objet l'ayant appelée.*

  - *Si on reprenait l'exemple avec la fonction membre* ***coincide****, on pourra le réécrire comme suit:*

```
bool personne:: coincide(personne * p)
{
    return ((this->p_nom == p->p_nom )&&  (this->p_prenom == p->p_prenom));
 }
```

  - *Il est applicable aussi dans un constructeur initialisant les membres données. exemple:*

```
class personne {
        public:
                personne(string n, string p) {this->p_nom =  n ; this->p_prenom =  p ; }
                …

        private:
                string p_nom;
                string p_prenom;
};
```

Propriétés des fonctions membres

- **Fonctions membres statiques**

C++ permet aussi de définir des fonctions membre statiques avec le mot clé *static*. Ces fonctions membres ont un rôle totalement indépendant d'un quelconque objet. On pourra les utiliser, par exemple, pour agir sur des membres données statiques.

```
class personne {
        public:
          personne() {p_nom =  " " ; p_prenom =  " " ; };
          …
          static void toString();

        private:
          static int age;
          string p_nom;
          string p_prenom;
};

int personne:: age= 25;
int main(){
        personne p;
        personne:: toString ();
}
```

définition

```
void personne:: toString ()
{
     cout << "Elle a" << age<< " ans " ;
 }
```

Propriétés des fonctions membres

- **Fonctions membres constantes**

Le concept de constance des variables s'étend aux classes, ce qui signifie qu'on peut définir des objets constants. Dans ce cas ,il va falloir préciser parmi les fonctions membres, lesquelles sont autorisés à opérer sur des objets constants, sinon le compilateur rejettera la requête. Ces fonctions membres sont accompagnés du mot cle *const*.

```
class personne {
        public:
          personne() {p_nom = " " ; p_prenom = " " ; }
          void toString() const;
          void affiche();
        private:
          static int age;
          string p_nom;
          string p_prenom;
};

int main(){
          personne p1;
          const personne p2;
          p1.affiche();      // OK
          p1.toString ();    // OK
          p2.affiche();      // Incorrect
          p2.toString ();    // OK
}
```

Propriétés des fonctions membres

- **Les membres mutables**

Il est impossible qu'une fonction membre constante puisse modifier les valeurs des membres non statiques. Pour que cela soit possible, la donnée membre doit être précéder du mot clé **mutable** pour désigner qu'elle est modifiable même par des fonctions membres constantes.

```
class personne {
        public:
          personne() {p_nom =  " " ; p_prenom =  " " ; }
          void toString() const {age=25;}
          void affiche();

        private:
          mutable int age;
          string p_nom;
          string p_prenom;
 };
```

Propriétés des fonctions membres

- **Les getters et les setteurs**

En se rappelant des étiquettes ***public***, ***private*** ou ***protected*** indiquant la visibilité des fonctions et données membres ou précisant si celles-ci sont accessibles à partir d'autres classes ou non... Et bien cette notion est appelée ***encapsulation*** des données et fonctions membres, qui est une notion essentiels du concept « orienté objet »

Ainsi, des données membres portant l'étiquette ***private*** ne peuvent pas être manipulées directement par les fonctions membres des autres classes.

Pour que ces dernières soient manipulables, il faut définir :

- des fonctions membres ***publique*** appelées ***accesseurs*** ou ***getter*** en anglais, permettant d'accéder aux données membres privées en vue de lire leur contenu;
- des fonctions membres ***publique*** appelées ***mutateurs*** ou ***setter*** en anglais permettant, de modifier les données membres privées en vue de changer leur valeur.

Propriétés des fonctions membres
Les getters et les setteurs

- **Getters**

Un Getter ou Accesseur est donc une fonction membre permettant de récupérer le contenu d'une donnée membre étiquetée ***private***. Pour sa définition:

- Son type de retour doit correspondre au type de la variable à renvoyer;
- Il ne doit pas naturellement posséder d'arguments;
- Son nom doit conventionnellement précéder par le préfixe **Get** afin de faire ressortir sa fonction première.

Syntaxe:

```
class NomClasse{
    private :
        TypeVariable NomVariable;
    public :
        TypeVariable GetVariable();
};
TypeVariable NomClasse::GetVariable(){
            return NomVariable;
}
```

Exemple:

```
class personne
{
        public:
            personne(string, string);
            string GetNom();
        private:
            string nom;
  };

string personne::GetNom(){
        return nom;
}
```

Propriétés des fonctions membres
Les getters et les setteurs

- **Setters**

Un Setter ou mutateur est donc une fonction membre permettant de **modifier** le contenu d'une donnée membre étiquetée ***private***. Pour sa définition:

- Il doit prendre en paramètre, la valeur (de même type) à assigner à la donnée membre;
- C'est un **void**, Il ne doit donc pas renvoyer de valeur;
- Son nom doit conventionnellement précéder par le préfixe **Set** afin de faire ressortir sa fonction première.

Syntaxe:

```
class NomClasse{
    private :
        TypeVariable NomVariable;
    public :
        void SetVariable(TypeVariable);
};
NomClasse::SetVariable(TypeVariable NomVar){
            NomVariable=NomVar;
}
```

Exemple:

```
class personne
{
        public:
            personne(string, string);
            void SetNom(string nom);
        private:
            string _nom;
  };

personne::SetNom(string nom){
    _nom=nom;
}
```

Propriétés des fonctions membres
Les fonctions amies

- **Les fonctions amies**

Avec l'encapsulation, une fonction non membre ne peut pas accéder aux données privées ou protégées d'un objet. Heureusement, en c++, il existe un mécanisme permettant d'accéder aux données privées ou protégées à partir de fonctions non membres: la notion de fonction et/ou de classe amie.

- Une **fonction indépendante amie** est une fonction normale déclaré dans une classe avec le mot clé ***friend*** et pouvant accéder aux données étiquetée ***private*** de la classe.

Syntaxe:

```
class NomClasse{
    public:
        friend typeDeRetour NomFonction(paramètres);
    private:
        TypeVariable NomVariable;
};
```

- A sa définition le mot clé **friend** n'est utilisé.

```
typeDeRetour NomFonction(paramètres){
    // instructions
}
```

Exemple:

```
#include <iostream>
using namespace std;
 class personne
 {
          public:
              personne(){nom="NoName"}
              friend void toString(personne&);
          private:
              string nom;
   };
 void toString(personne& p){
        cout << p.nom<< endl;
 }
 int main() {
     personne p;
     toString(p);
     return 0;
 }
```

Propriétés des fonctions membres
Les fonctions amies

- On peut également avoir une **fonction membre d'une classe**, qui est aussi **amie dans une autre classe**.

Exemple:

```
#include <iostream>
using namespace std;
 class B;
 class A
 {
        public:
            void toString(B&);
   };
 class B
 {
        public:
            B(){value=5;}
            friend void A::toString(B& b);
        private:
            int value;
   };
```

```
// suite

 void A::toString(B& b){
         cout << b.value << endl;
 }

 int main() {
     A a;
     B b;
     a.toString(b);

     return 0;
 }
```

Propriétés des fonctions membres
Les classes amies

- **Les classes amies**

Une classe amie comme une fonction amie peut accéder aux membres privés et protégés d'une autre classe dans laquelle elle est déclarée comme ami.

Exemple:

```
#include <iostream>
using namespace std;
 class B;
 class A
 {
        public:
            void toString(B&);
   };
 class B
 {
        public:
            B(){value=5;}
            friend class A;
        private:
            int value;
   };
```

```
// suite

 void A::toString(B& b){
          cout << b.value << endl;
 }

 int main() {
      A a;
      B b;
      a.toString(b);

      return 0;
 }
```

La POO

- Notion de Classe
- Fonctions membres
- **Constructeurs, destructeurs et initialisation d'objet**
- Héritage simple
- Héritage multiple
- fonctions virtuelles et le polymorphisme
- La gestion des exceptions
- Généralités sur la STL

Constructeurs, destructeurs et initialisation d'objet

Précédemment, nous avons vu les quelques propriétés des fonctions membres. Ici, nous allons étudier en profondeurs les constructeurs/destructeurs et l'initialisation des objets en C++.

- **Constructeurs et destructeurs**

Les constructeurs et le destructeur sont des fonctions membres particulières des classes.

- **Constructeurs :**

Un constructeur permet de définir un ou des comportements particuliers lors de l'instanciation d'une classe. Ils permet aussi d'initialiser correctement un nouvel objet. Ils portent tous le même nom que la classe avec des particularités sur le nombre et le type de paramètres.

Le constructeur par défaut ou constructeur sans paramètre qui est appelé lors de l'instanciation d'un objet sans argument d'appel.

Exemple: // appel du constructeur par défaut

- Personne p1 ; // oubien
- Personne* p2 = new Personne;

Un tel constructeur est aussi appelé lors de l'instanciation d'un tableau d'objets

Exemple: // pour chaque personne du tableau, appel du constructeur par défaut

- Personne tabP1[5]; // oubien
- Personne* tabP2 = new Personne[5];

Constructeurs, destructeurs et initialisation d'objet

- **destructeurs:**

Un constructeur permet la destruction ou la désallocation d'un objet. Il est définie implicitement pour toutes les classes. Il ne fait rien par défaut, mais on peut lui donner un comportement spécifique.

- **Appel des constructeurs statiquement**
- Supposons qu'un objet possède qu'un seul constructeur, sa déclaration doit obligatoirement comporter les arguments correspondants.

Exemple:

```
class personne {
        public:
          personne(string, string);
          …
        private:
          string p_nom;
          string p_prenom;
    };
```

- **La déclaration suivante est correcte**:
  personne p("Nabi" , "BARRO");
- **Les déclarations suivantes sont incorrectes**:
  personne p1;
  personne p2("Nabi");

Constructeurs, destructeurs et initialisation d'objet

- S'il existe plusieurs constructeurs, la déclaration doit comporter les arguments requis du constructeur appelé.

Exemple:

```
class personne {
        public:
          peronne();
          personne(string, string);
          …
        private:
          string p_nom;
          string p_prenom;
    };
```

- **Les déclarations suivantes sont correctes**:

```
personne p1;
personne p2("Nabi" , "BARRO");
```

- **La déclaration suivante est incorrecte**:

```
personne p("Nabi");
```

- S'il n'existe pas de constructeur, alors la déclaration suivante est acceptable:

- personne p; // il va falloir faire attention à ne pas faire *personne p()* car ceci est une déclaration pour fonction.

Constructeurs, destructeurs et initialisation d'objet

- **Appel des constructeurs dynamiquement**
- Reconsidérons la class personne (*sans constructeur*) suivante:

```
class personne {
        public:
          void toString();
          …
        private:
          string p_nom;
          string p_prenom;
    };
```

  - Il est donc possible de créer dynamiquement un emplacement mémoire de type personne.
    - Exemple: personne * p;
  - Puis maintenant d'affecter son adresse à ***p*** par:
    - p = new personne; // ou new personne()
  - On pourra, alors accéder aux fonctions membres soit via l'opérateur -> :
    - p -> toString();
  - Ou en faisant comme suit:
    - (*p).toString();
  - L'opérateur *delete* sera utilisé pour appeler le destructeur afin de libérer l'emplacement mémoire correspondant:
    - delete p;

Constructeurs, destructeurs et initialisation d'objet

- Reconsidérons la classe personne (*avec constructeur*) suivante:

```
class personne {
        public:
          personne(string, string);
          void toString();
          …
        private:
          string p_nom;
          string p_prenom;
    };
```

  - Il est donc possible de créer dynamiquement un emplacement mémoire de type personne.
    - Exemple: personne * p;
  - ici, pour que *new* puisse appeler un constructeur disposant d'arguments, il est nécessaire qu'il dispose des informations correspondantes.
    - p = new personne("Nabi" , "BARRO");

Constructeurs, destructeurs et initialisation d'objet

- **Initialisation d'objet**

En C++, munie d'un initialiseur, on peut fournir, sous une forme peu naturelle, des arguments pour un constructeur. Le langage n'impose aucune restriction sur le type de l'initialiseur qui pourra, éventuellement, être du même type que l'objet initialisé.

Supposons la classe personne avec un constructeur:

```
class personne {
        public:
                personne();
                personne(string prenom){p_prenom=prenom; p_nom="BARRO"; }
                personne (string, string);
                …
        private:
                string p_nom;
                string p_prenom;
};
```

Constructeurs, destructeurs et initialisation d'objet

- Les déclarations et initialisations suivantes sont valables et équivalentes:
  - personne p("zeynab");
  - personne p= "zeynab";
- on peut aussi initialiser un objet avec un autre de même type:
  - personne p1;
  - personne p2=p1;
  - personne p3(p1);
- Pour le cas d'un constructeur à deux ou plusieurs arguments, l'initialisation prévue est la suivante:
  - Personne p("Souleymane", "BARRO");

Notion de classe – fonctions membre – constructeurs et destructeurs

EXERCICES D'APPLICATIONS

- Application 28:

  Réaliser une classe *point* permettant de manipuler un point d'un plan.

  On prévoira :

  - un constructeur recevant en arguments les coordonnées (*float*) d'un point ;
  - une fonction membre *deplace* effectuant une translation définie par ses deux arguments (*float*) ;
  - une fonction membre *affiche* se contentant d'afficher les coordonnées cartésiennes du point.

  Les coordonnées du point seront des membres donnée privés.

  On écrira séparément :

  - un fichier source constituant la *déclaration* de la classe ;
  - un fichier source correspondant à sa *définition*.
  - un petit programme d'essai (*main*) déclarant un point, l'affichant, le déplaçant et l'affichant à nouveau.

- Application 29:

  Réaliser une classe *point*, analogue à la précédente, mais ne comportant pas de fonction *affiche*. Pour respecter le principe d'encapsulation des données, prévoir deux fonctions membre publiques (nommées *abscisse* et *ordonnée*) fournissant en retour l'abscisse et l'ordonnée d'un point. Adapter le petit programme d'essai précédent pour qu'il fonctionne avec cette nouvelle classe.

EXERCICES D'APPLICATIONS

Application 30:

On souhaite réaliser une classe vecteur3d permettant de manipuler des vecteurs à trois composantes. On prévoit que sa déclaration se présente ainsi :

```
class vecteur3d {
    float x, y, z ;
    .....
} ;
```

On souhaite pouvoir déclarer un vecteur, soit en fournissant explicitement ses trois composantes, soit en en fournissant aucune, auquel cas le vecteur créé possédera trois composantes nulles. Écrire le ou les constructeurs correspondants :

a. en utilisant des fonctions membre surdéfinies ;
b. en utilisant une seule fonction membre ;
c. en utilisant une seule fonction en ligne.

Application 31:

Soit une classe vecteur3d définie comme suit :

```
class vecteur3d {
    float x, y, z ;
    public :
        vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0) {
            x = c1 ; y = c2 ; z = c3 ;
          }
        .....
} ;
```

Introduire une fonction membre nommée *coincide* permettant de savoir si deux vecteurs ont les mêmes composantes :

a. en utilisant une transmission par valeur ;
b. en utilisant une transmission par adresse ;
c. en utilisant une transmission par référence.

Si v1 et v2 désignent 2 vecteurs de type vecteur3d, comment s'écrit le test de coïncidence de ces 2 vecteurs, dans chacun des 3 cas considérés ?

EXERCICES D'APPLICATIONS

- Application 32:

Soit une classe vecteur3d définie comme suit :

```
class vecteur3d {
    float x, y, z ;
    public :
        vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0) {
                x = c1 ; y = c2 ; z = c3 ;
        }
        .....
} ;
```

Introduire, dans cette classe, une fonction membre nommée normaux permettant d'obtenir, parmi deux vecteurs, celui qui a la plus grande norme. On prévoira trois situations :

a. le résultat est renvoyé par valeur ;

b. le résultat est renvoyé par référence, l'argument (explicite) étant également transmis par référence;

c. le résultat est renvoyé par adresse, l'argument (explicite) étant également transmis par adresse.

À suivre …

Feedback sur:
pape.abdoulaye.barro@gmail.com