Les classes et les objets

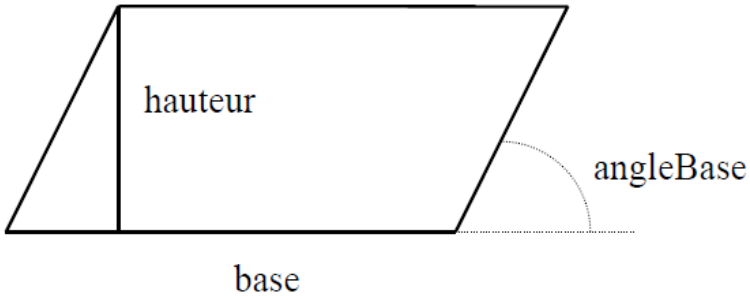
Les structures en C++

Les structures en C++ permettent de regrouper en une seule entité logique des données de types différents. Selon la nécessité de programmation, on peut :

- Considérer la structure dans son ensemble ;
- Considérer les différents champs de la structure.

Ainsi si on désire manipuler des parallélogrammes en C, on définira une structure Para du type suivant :

```
struct Para
{
     float base, hauteur ;
     double angleBase ;
     char* nom ;
};
```



La déclaration d'une telle structure se fait de la manière suivante en C++ :

```
Para monPa ;  // contrairement au « struct Para monPa ; » en C
```

La manipulation d'une telle structure se fait au moyen de fonctions indépendantes qui doivent bien sûr connaître la composition de la structure.

Par exemple, si on désire pouvoir calculer l'aire d'un parallélogramme :

```
float aire(Para p) {
     return p.base * p.hauteur ;
}
```

Ce qui donnera :

```
void main()
{
      Para monPa ;
      monPa.base = 5.7 ;
      monPa.hauteur = 7.0 ;
      cout << « L'aire du parallélogramme vaut : «  <<
      aire(monPa) << endl ;
}
```

Cette **séparation des données et de leur traitement** est ressentie comme un **obstacle à la modularité des programmes**. La grande idée du C++ est précisément d'**incorporer dans la structure les fonctions qui permettent leur manipulation**.

Ce qui donnera :

```
struct Para
{
      float base, hauteur ;
      double angleBase ;
      float aire() { return base*hauteur ; }
      float peri()
      { return 2*(base+hauteur/sin(angleBase*M_PI/180)) ; }
};
```

La déclaration d'une telle structure se fera toujours de la manière suivante :

```
Para monPa;
```

Exemple :

```
#include <math.h>
#include <iostream>
using namespace std;

// #define M_PI 3.1416

struct Para
{
  float base, hauteur;
  double angleBase;
  float aire() { return base * hauteur; };
  float peri() { return 2*(base+hauteur/sin(angleBase*M_PI/180)); };
};
```

```
int main()
{
  Para monPa;

  cout << "Base : "; cin >> monPa.base;
  cout << "Hauteur : "; cin >> monPa.hauteur;
  cout << "Angle a la base : "; cin >> monPa.angleBase;

  cout << "Perimetre = " << monPa.peri() << endl;
  cout << "Aire = " << monPa.aire() << endl;

  return 0;
}
```

Un exemple d'exécution fournit :

```
bash-3.00$ a.out
Base : 4
Hauteur : 8
Angle a la base : 40
Perimetre = 32.8916
Aire = 32
bash-3.00$
```

Les classes

Cependant, en C++ :

- On ne parle plus de *structure* mais de **classe**
- On ne parle plus non plus de déclaration mais d'**instanciation**
- On ne parle plus de variable « monPa » mais d'un **objet** « monPa »

On dit maintenant : « **L'objet monPa instancie la classe Para** ».

De plus,

- Les champs sont appelés les « **variables membres** »
- Les fonctions sont appelées les « **fonctions membres** » ou « **méthodes** »

Les classes sont donc des structures de données qui se suffisent à elles-mêmes.

Ce qui nous permet d'atteindre le niveau de **modularité** et d'**abstraction** recherché en POO :

- **Modularité** : une classe se suffit à elle-même puisqu'un objet qui l'instancie contient ses données et ses traitements.

- **Abstraction** : une classe devient un type de données analogue aux types prédéfinis.

Enfin, idéalement les données de la classe ne devraient être accessibles, et donc manipulées, que par les fonctions membre de celle-ci (→ **Encapsulation** à l'intérieur de la classe)

Ce qui donnera dans notre exemple :

```
…
class Para
{
  float base, hauteur;
  double angleBase;
  float aire() { return base * hauteur; };
  float peri() { return 2*(base+hauteur/sin(angleBase*M_PI/180)); };
};

int main()
{
  Para monPa;  // Tout est declare inacessible, car dans une classe,
               // les membres sont prives par defaut
  cout << "Base : "; cin >> monPa.base;
  cout << "Hauteur : "; cin >> monPa.hauteur;
  cout << "Angle a la base : "; cin >> monPa.angleBase;
  cout << "Perimetre = " << monPa.peri() << endl;
  cout << "Aire = " << monPa.aire() << endl;
  return 0;
}
```

Mais cela conduit à des erreurs à la compilation ☹ !!!...

```
bash-3.00$ g++ Exemple2.cxx
Exemple2.cxx: In function `int main()':
Exemple2.cxx:9: error: `float Para::base' is private
Exemple2.cxx:18: error: within this context
Exemple2.cxx:9: error: `float Para::hauteur' is private
Exemple2.cxx:19: error: within this context
```

```
Exemple2.cxx:10: error: `double Para::angleBase' is private
Exemple2.cxx:20: error: within this context
Exemple2.cxx:12: error: `float Para::peri()' is private
Exemple2.cxx:21: error: within this context
Exemple2.cxx:11: error: `float Para::aire()' is private
Exemple2.cxx:22: error: within this context
bash-3.00$
```

Ceci est logique dans la mesure où, par défaut, dans une classe, **les membres ne sont accessibles que par d'autres membres**. Ils sont **privés** à la classe. C'est l'**encapsulation** maximale.

Les membres publics et privés

En réalité, l'accès aux membres d'une classe peut être défini de trois manières différentes. Pour le moment, seulement deux de celles-ci nous intéressent :

- **Privé (private)** : les membres déclarés ainsi ne sont pas accessibles à l'extérieur de la classe.

- **Publique (public)** : les membres déclarés ainsi sont accessibles par tout le monde.

Pour rendre le code du programme précédent compilable, on déclare tous les membres de la classe Para étant publiques.

```
…
class Para
{
  public :
    float base, hauteur;
    double angleBase;
    float aire() {return base * hauteur;};
    float peri() {return 2*(base+hauteur/sin(angleBase*M_PI/180));};
};

int main()
{
  Para monPa;
  cout << "Base : "; cin >> monPa.base;
  cout << "Hauteur : "; cin >> monPa.hauteur;
```

```
  cout << "Angle a la base : "; cin >> monPa.angleBase;
  cout << "Perimetre = " << monPa.peri() << endl;
  cout << "Aire = " << monPa.aire() << endl;
  return 0;
}
```

- Cela compile à nouveau mais on a utilisé une classe comme une structure.
- Solution « intermédiaire » pour respecter les principes de la POO et l'encapsulation des données en particulier.

On place :

- En **privé** ce qui doit être interne à la classe
- En **publique** ce qui doit être accessible par tout le monde externe.

Ce qui donne dans notre exemple :

```
…
class Para
{
  private :
    float base, hauteur;
    double angleBase;
  public :
    float aire() {return base * hauteur;};
    float peri() {return 2*(base+hauteur/sin(angleBase*M_PI/180));};
};

int main()
{
  Para monPa;
  cout << "Base : "; cin >> monPa.base;
  cout << "Hauteur : "; cin >> monPa.hauteur;
  cout << "Angle a la base : "; cin >> monPa.angleBase;
  cout << "Perimetre = " << monPa.peri() << endl;
  cout << "Aire = " << monPa.aire() << endl;
  return 0;
}
```

Mais de nouveau, il y a erreur à la compilation ☹… :

```
bash-3.00$ g++ Exemple3.cxx
Exemple3.cxx: In function `int main()':
Exemple3.cxx:10: error: `float Para::base' is private
Exemple3.cxx:20: error: within this context
Exemple3.cxx:10: error: `float Para::hauteur' is private
```

```
Exemple3.cxx:21: error: within this context
Exemple3.cxx:11: error: `double Para::angleBase' is private
Exemple3.cxx:22: error: within this context
bash-3.00$
```

En effet, le programme principal essaie d'accéder directement aux variables membres, qui sont devenues « privées », de l'objet monPa.

- Il faut doter la classe **Para** d'une fonction membre permettant de saisir les valeurs de ses composantes :

```
…
class Para
{
  private:
    float base, hauteur;
    double angleBase;
  public:
    float aire() {return base * hauteur;};
    float peri() {return 2*(base+hauteur/sin(angleBase*M_PI/180));};
    void inputDim()
    {
      cout << "Base : "; cin >> base;
      cout << "Hauteur : "; cin >> hauteur;
      cout << "Angle a la base : "; cin >> angleBase;
    }
};

int main()
{
  Para monPa;
  monPa.inputDim();
  cout << "Perimetre = " << monPa.peri() << endl;
  cout << "Aire = " << monPa.aire() << endl;
  return 0;
}
```

Le programme compile à nouveau ☺ ! Un exemple d'exécution fournit :

```
bash-3.00$ a.out
Base : 3
Hauteur : 7
Angle a la base : 30
Perimetre = 34
Aire = 21
bash-3.00$
```

Le but recherché est finalement atteint :

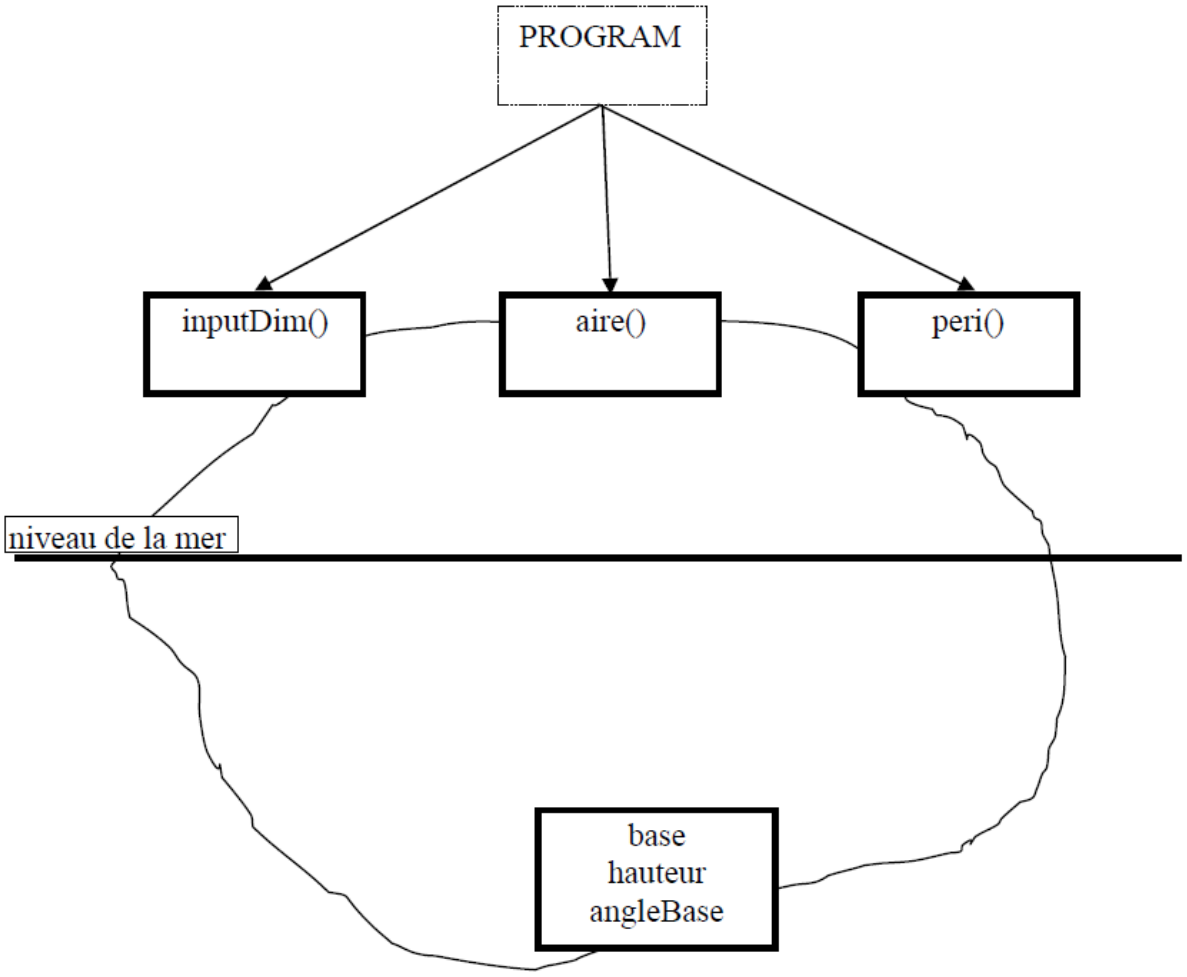
- Les données sont **encapsulées** dans la classe et sont donc **protégées** contre les assauts d'autres fonctions non membres de la classe. Elles sont invisibles pour le monde extérieur.

- Les fonctions membres publiques sont utilisables pour tout le monde.

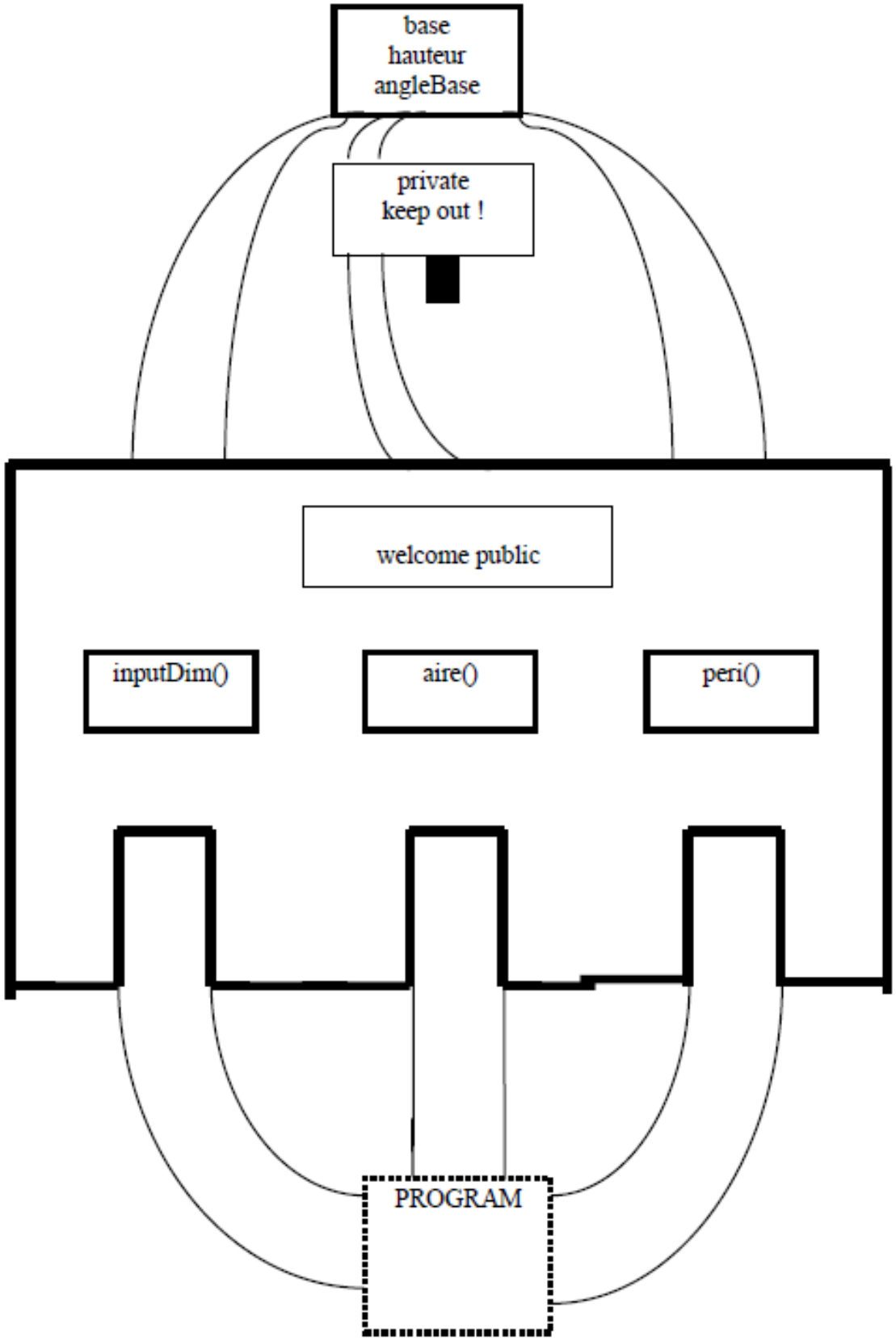
On dit encore que :

- La **partie publique** d'une classe en constitue l'**interface**
- La **partie privée** d'une classe en constitue l'**implémentation**

Représentation imagée sous forme « d'iceberg » :



Représentation imagée sous forme de « jardin secret » :



Remarque concernant l'utilisation des classes et/ou des structures en C++ :

Type	Caractéristiques	Utilisation
Classe	Membres privés par défaut	Pour des objets ayant des variables membres et des fonctions membres
Structure	Membres publics par défaut	Pour des objets n'ayant que des variables membres

Fonction membre utilisant une autre fonction membre

Calcul du volume du prisme dont la base est notre parallélogramme :

```
class Para
{
  private:
    float base, hauteur;
    double angleBase;

  public:
    float aire() {return base * hauteur;};
    float peri() {return 2*(base+hauteur/sin(angleBase*M_PI/180));};
    void inputDim()
    {
      cout << "Base : "; cin >> base;
      cout << "Hauteur : "; cin >> hauteur;
      cout << "Angle a la base : "; cin >> angleBase;
    }
    float volume(float hP) { return aire() * hP; }
};

int main()
{
  Para monPa;
  monPa.inputDim();
  cout << "Perimetre = " << monPa.peri() << endl;
```

```
  cout << "Aire = " << monPa.aire() << endl;
  float hautPri;
  cout << "Hauteur du prisme : "; cin >> hautPri;
  cout << "Volume du prisme = " << monPa.volume(hautPri) << endl;
  return 0;
}
```

Exemple d'exécution :

```
bash-3.00$ a.out
Base : 3
Hauteur : 8
Angle a la base : 30
Perimetre = 38
Aire = 24
Hauteur du prisme : 4
Volume du prisme = 96
bash-3.00$
```

Les fonctions membres classiques

Les données étant encapsulées à l'intérieur de la classe, leur accès (en lecture/écriture) doit se faire via des fonctions membres. On retrouve en général pour une donnée encapsulée :

- Une méthode « **get…()** » qui renvoie la valeur
- Une méthode « **set…()** » qui permet de spécifier la valeur

Ce qui donne dans notre exemple :

```
class Para
{
  private:
    float base, hauteur;
    double angleBase;

  public:
    float getBase() {return base;}
    float getHauteur() {return hauteur;}
    double getAngleBase() {return angleBase;}
```

```
    void setBase(float b) { base = b; }
    void setHauteur(float h) { hauteur = h; }
    void setAngleBase(double a) { angleBase = a; }

    float aire() { return getBase() * getHauteur(); };
    float peri()
 {return 2*(getBase()+getHauteur()/sin(getAngleBase()*M_PI/180));};
    …
};
```

L'intérêt majeur de cette façon de faire est qu'**une modification de l'implémentation** (ici, par exemple, remplacer la base et la hauteur par les diagonales) **ne modifiera en rien l'interface** d'utilisation : il suffira de réécrire les méthodes get… et set… ! C'est le véritable intérêt de l'**encapsulation** !

Le pointeur this

Chaque objet d'un même type (et donc instanciant une même classe) possède ses propres données mais partage le même code avec chaque objet de ce type. Comment le compilateur fait-il alors pour générer un code qui retrouve l'objet auquel les méthodes évoquées s'appliquent ?

En fait, il y réussit car il passe un paramètre supplémentaire, et ce **de manière implicite**, à chaque fonction : c'est le pointeur **this** qui est un pointeur sur l'objet en question.

La méthode

```
void setBase(float b) ;
```

est « vue » par le compilateur comme :

```
void setBase(float b,Para* const this) ;
```

Accès au pointeur this :

```
class Para
{
  …
  void setBase(float b)
  {
    this->base = b ;
```

```
    // est totalement equivalent à base = b;
  }
  …
};
```

Cette notation permet même d'écrire

```
class Para
{
  …
  void setBase(float base)
  {
    this->base = base ;
  }
  …
};
```

sans confusion entre la variable membre et le paramètre de la méthode.

La définition des fonctions membres

Lorsque la taille des fonctions devient plus importante, la taille du code de la classe peut devenir prohibitive et diminuer la lisibilité du code. Il est préférable dans ce cas (voir même dans tous les cas lorsque l'on travaillera en fichiers séparés) d'utiliser la **définition rapportée des méthodes**. Cela signifie que :

- La fonction est seulement **déclarée dans la classe** (le prototype)
- La fonction est **définie à l'extérieur de la classe** ; il faut cependant spécifier que cette fonction fait partie de cette classe → on utilise l'**opérateur de résolution de portée « :: »**

Ce qui donne dans notre exemple :

```
…
class Para
{
  private:
    float base, hauteur;
    double angleBase;

  public:
    float getBase() {return base;}
    float getHauteur() {return hauteur;}
```

```
    double getAngleBase() {return angleBase;}

    void setBase(float b) { base = b; }
    void setHauteur(float h) { hauteur = h; }
    void setAngleBase(double a) { angleBase = a; }

    float aire() {return getBase() * getHauteur();};
    float peri()
{return 2*(getBase()+getHauteur()/sin(getAngleBase()*M_PI/180));};
    void  setDim(float b,float h,double a); // declaration
    float volume(float hP) { return aire() * hP; }
};

void Para::setDim(float b,float h,double a) // definition rapportee
{
  setBase(b);
  setHauteur(h);
  setAngleBase(a);
}

int main()
{
  Para monPa;

  cout << "Base au depart = " << monPa.getBase() << endl;
  cout << "Hauteur au depart = " << monPa.getHauteur() << endl;
  cout << "Angle au depart a la base = " << monPa.getAngleBase() << endl;

  float maB,maH;
  double monA;
  cout << "Base : "; cin >> maB;
  cout << "Hauteur : "; cin >> maH;
  cout << "Angle a la base : "; cin >> monA;
  monPa.setDim(maB,maH,monA);

  cout << "Base = " << monPa.getBase() << endl;
  cout << "Hauteur = " << monPa.getHauteur() << endl;
  cout << "Angle a la base = " << monPa.getAngleBase() << endl;
  cout << "Perimetre = " << monPa.peri() << endl;
  cout << "Aire = " << monPa.aire() << endl;

  float hautPri;
  cout << "Hauteur du prisme : "; cin >> hautPri;
  cout << "Volume du prisme = " << monPa.volume(hautPri) << endl;
  return 0;
}
```

Un exemple d'exécution fournit :

```
bash-3.00$ a.out
Base au depart = 4.00465e-34
Hauteur au depart = 4.03503e-34
Angle au depart a la base = 4.83796e-270
Base : 4
Hauteur : 7
Angle a la base : 45
Base = 4
Hauteur = 7
Angle a la base = 45
Perimetre = 27.799
Aire = 28
Hauteur du prisme : 3
Volume du prisme = 84
bash-3.00$
```

Le constructeur

Au même titre que les types de données traditionnelles, un objet peut être initialisé lors de sa création.

Le **constructeur** d'une classe est la **fonction qui est appelée lors de la création d'un objet instanciant cette classe**. C'est dans cette fonction qu'on détermine les actions à entreprendre afin d'**initialiser l'objet** :

- Allocation des espaces mémoires nécessaires
- Initialisations
- Calcul de la valeur de départ de certaines variables membres
- Ouverture de fichiers, de connexions réseau, etc…

Syntaxiquement, le constructeur est une fonction :

- Qui porte le **même nom que la classe** (il s'appelle donc Para dans notre exemple)
- Qui **n'a pas de type** (même pas void)
- Qui **peut avoir des paramètres en nombre et en type quelconque**

Le polymorphisme des fonctions du C++ s'applique aussi au constructeur d'une classe. Par conséquent,

Une classe peut posséder **plusieurs constructeurs**

Ceux-ci différent entre eux par le nombre et le type de leurs paramètres.

Le constructeur par défaut

Le constructeur par défaut est celui qui ne possède **aucun paramètre**.

Le C++ crée **implicitement** un constructeur par défaut pour chaque classe qui ne possède pas de constructeur **explicite**. → cela signifie que si vous n'en créez pas un vous-même, C++ vous en crée un mais **vide**

Jusque maintenant, lorsque l'on écrivait

```
Para monPa;
```

on évoquait par défaut le constructeur par défaut implicite.

Ce constructeur ne fait **rien de plus que de réserver l'espace mémoire des variables** statiques. Leur **valeur est folklorique**, sauf si l'objet est global.

Il convient donc d'écrire un constructeur par défaut explicite pour chaque classe développée.

Exemple :

```
class Para
{
  private :
    float base, hauteur ;
    double angleBase ;
  public :
    Para() { base = hauteur = angleBase = 0 ; }
    float getBase() { return base ; }
    …
};

void main()
{
  Para monPa ;  // appel au constructeur par défaut explicite
}
```

Cette dernière ligne :

1. Déclare une variable **monPa** du type **Para**
2. Réserve un espace mémoire pour les variables membres base, hauteur et angleBase
3. Exécute le constructeur par défaut **Para()**

Le constructeur d'initialisation

Le constructeur d'**initialisation** est le constructeur qui permet d'**initialiser les données membres de l'objet en cours de création** avec des **valeurs distinctes** d'un objet à l'autre.

Les valeurs permettant d'initialiser les données membres de l'objet sont passées en paramètres au constructeur.

Il n'y a **pas de constructeur d'initialisation implicite**.

Exemple :

```
Para::Para(float b,float h,double a)
{
  Base = b ;
  Hauteur = h ;
  angleBase = a ;
}
```

qui permettra d'écrire

```
Para tonPara(20,10,45) ;
```

on créée ainsi un objet Para avec la base initialisée à 20, la hauteur à 10 et l'angle à la base à 45.

Le constructeur de copie

Le constructeur de **copie** permet de créer un objet comme étant une **copie d'un autre objet du même type**. Un tel constructeur **existe aussi de manière implicite**. Il se contente d'affecter membre à membre les champs de l'objet copié à ceux du nouvel objet.

Exemple :

```
Para monPa(10,16,45) ;

Para autrePa(monPa) ;

Para monAutrePa = monPa ; // le constructeur de copie est aussi appelé !
```

Tout comme le constructeur par défaut, **il convient d'écrire un constructeur de copie pour chaque classe développée** (sous peine de gros souci… surtout en cas de variable membre de type pointeur).

Le constructeur de copie **est appelé** :

- Lors de la création d'un objet à partir d'un autre du même type (appel explicite)
- Lors du **passage par valeur** d'un objet dans une fonction (appel implicite)
- Lorsqu'un objet est **renvoyé par valeur** par une fonction (appel implicite)

Exemple :

```
Para::Para(const Para& p)
{
  base = p.base ;
  hauteur = p.hauteur ;
  angleBase = p.angleBase ;
}
```

Remarque : un constructeur de conversion

Il est parfois intéressant de créer un objet à partir d'un objet d'un autre type.

Exemple :

Supposons avoir défini la classe suivante :

```
class Rect
{
  private:
    float longueur, largeur;

  public:
    Rect(float lo,float la) { longueur = lo; largeur = la; }
    float getLongueur() { return longueur; }
    float getLargeur() { return largeur; }
};
```

Un rectangle est en fait un parallélogramme ayant un angle droit. Donc, un objet Para peut être construit à partir d'un objet Rect :

```
Para::Para(Rect r)
{
  base = r.getLongueur();  // pas le choix !
  hauteur = r.getLargeur();
  angleBase = 90.0;
}
```

On pourra ainsi écrire :

```
Rect monRect(20,40);

Para sonPa(monRec);
```

Classe Para avec l'ensemble de ses constructeurs :

```
class Para
{
  private:
    float base, hauteur;
    double angleBase;

  public:
    Para();
    Para(float b,float h,double a);
    Para(const Para& p);
    Para(Rect r);

    float getBase() {return base;}
    float getHauteur() {return hauteur;}
    double getAngleBase() {return angleBase;}

    void setBase(float b) { base = b; }
    void setHauteur(float h) { hauteur = h; }
    void setAngleBase(double a) { angleBase = a; }

    float aire() { return getBase() * getHauteur(); };
    float peri()
{ return 2*(getBase()+getHauteur()/sin(getAngleBase()*M_PI/180)); };
    void  setDim(float b,float h,double a);
    float volume(float hP) { return aire() * hP; }
};
```

Définitions rapportées des constructeurs de la classe Para :

```
Para::Para()
{
  base = hauteur = angleBase = 0.0;
}

Para::Para(float b,float h,double a)
{
  base = b;
  hauteur = h;
  angleBase = a;
}

Para::Para(const Para& p)
{
  base = p.base;
  hauteur = p.hauteur;
  angleBase = p.angleBase;
}
```

```
Para::Para(Rect r)
{
  base = r.getLongueur();
  hauteur = r.getLargeur();
  angleBase = 90.0;
}
```

Une variable membre pointeur

Afin d'illustrer davantage l'utilité de surcharger le constructeur d'une classe, ajoutons-y une variable membre pointeur.

```
class Para
{
  private :
    …
    char *nom ;
  public :
    …
    char* getNom() { return nom ; }
    …
    void setNom(const char* nom);
};
```

Il faut allouer le pointeur nom !!! Les constructeurs de la classe Para deviennent :

```
const int L_NOM = 20;

class Para
{
  private :
    …
    char *nom ;
  public :
    Para();
    Para(float b,float h,double a,const char* s=nullptr);
    …
    char* getNom() { return nom ; }
    …
    void setNom(const char* nom);
};

Para::Para()
{
  cout << "Constructeur par defaut" << endl;
```

```
  base = hauteur = angleBase = 0.0;
  nom = new char[L_NOM];
  strcpy(nom,"");
}

Para::Para(float b,float h,double a,const char* s)
{
  cout << "Constructeur d'initialisation" << endl;
  base = b; hauteur = h; angleBase = a;
  if (s != nullptr)
  {
    nom = new char[strlen(s)+1];
    strcpy(nom,s);
  }
  else
  {
    nom = new char[L_NOM];
    if (angleBase == 90.0) strcpy(nom,"rectangle");
    else strcpy(nom,"parallelogramme");
  }
}

Para::Para(const Para& p)
{
  cout << "Constructeur de copie" << endl;
  base = p.base;
  hauteur = p.hauteur;
  angleBase = p.angleBase;
  nom = new char[strlen(p.nom) + 1];
  strcpy(nom,p.nom);
}
```

Le destructeur

Le **destructeur** d'une classe est une **fonction membre particulière**. C'est la fonction qui est appelée **lorsqu'on détruit l'objet** (objet dynamique) ou **lorsque l'on quitte sa portée** (objet statique).

Son rôle est de détruire proprement l'objet, c'est-à-dire

- **Libérer l'espace mémoire** occupé par l'objet
- Fermer proprement les fichiers ouverts
- Mettre fin à une connexion
- Sauvegarder des données de l'objet dans un fichier
- Etc…

Syntaxiquement, le destructeur est une fonction :

- Qui porte **le même nom que la classe précédé du symbole « ~ »**
- Qui **n'a pas de type** (même pas void)
- Qui **ne peut avoir de paramètres**
- Qui **ne peut être surchargé** : le destructeur d'une classe est **unique**

Exemple :

```
Para::~Para()
{
  cout << "Destructeur" << endl;
  if (nom) delete nom;
}
```

```
int main()
{
  cout << "Entree dans le bloc" << endl;
  {
    Para monPa;
    cout << "Base = " << monPa.getBase() << endl;
    cout << "Hauteur = " << monPa.getHauteur() << endl;
    cout << "Angle a la base = " << monPa.getAngleBase() << endl;
    cout << "On va sortir du bloc" << endl;
  }
  cout << "On est sorti du bloc" << endl << endl;

  cout << "Allocation dynamique d'un Para" << endl;
  Para *pPara = new Para(10,30,60);
  cout << "Base = " << pPara->getBase() << endl;
  cout << "Hauteur = " << pPara->getHauteur() << endl;
  cout << "Angle a la base = " << pPara->getAngleBase() << endl;
  cout << "Liberation de la memoire" << endl;
  delete pPara;

  cout << "Fin du programme" << endl;
  return 0;
}
```

Un exemple d'exécution fournit :

```
bash-3.00$ a.out
Entree dans le bloc
Constructeur par defaut
Base = 0
Hauteur = 0
Angle a la base = 0
On va sortir du bloc
Destructeur
On est sorti du bloc

Allocation dynamique d'un Para
Constructeur d'initialisation
Base = 10
Hauteur = 30
Angle a la base = 60
Liberation de la memoire
Destructeur
Fin du programme
bash-3.00$
```

Les objets dynamiques

Comme on vient de le voir, il est bien sûr possible de définir des objets dynamiquement, c'est-à-dire désigné par un pointeur (pPara dans notre exemple).

Tout se passe exactement comme pour une structure pointée :

- On accède aux champs (et aux méthodes) avec l'opérateur **« -> »** au lieu de **« . »**
- Un objet est créé en utilisant l'opérateur **new** ; **celui-ci appelle un constructeur qui permet d'initialiser l'objet**
- Il est détruit en utilisant le l'opérateur **delete** ; **celui-ci appelle le destructeur puis libère l'espace mémoire**.

Lorsqu'il s'agit d'un **tableau**, on utilise plutôt les opérateurs new et delete avec crochets :

```
pointeur = new type[dimension] ;
```

et

```
delete [] pointeur ;
```

Le constructeur est appelé pour chaque élément du tableau lors du **new** et le destructeur est appelé pour chaque élément lors du **delete**.

Exemple :

```
int main()
{
  Para *tab = new Para[4];
  cout << "delete sans crochet" << endl;
  delete tab;

  tab = new Para[4];
  cout << "delete avec crochets" << endl;
  delete [] tab;

  return 0;
}
```

L'exécution fournit :

```
bash-3.00$ a.out
Constructeur par defaut
Constructeur par defaut
Constructeur par defaut
Constructeur par defaut
delete sans crochet
Destructeur

Constructeur par defaut
Constructeur par defaut
Constructeur par defaut
Constructeur par defaut
delete avec crochets
Destructeur
Destructeur
Destructeur
Destructeur
bash-3.00$
```

On remarque que

- « delete tab » ne désalloue que la première case ☹ !
- « delete [] tab » désalloue toutes les cases du tableau ☺

Les méthodes constantes d'une classe

Les méthodes membres d'une classe **ne modifiant pas les variables membres** de celle-ci sont appelées **méthodes constantes de cette classe**. Il est d'usage d'utiliser le **mot protégé « const »** du C++ **à la fin de la déclaration/définition** d'une telle fonction pour préciser ce caractère constant.

Exemple :

```
float getBase() { return base ; }
```

devient

```
float getBase() const { return base; }
```

Code complet de la classe Para (avec les « const » correctement placés) :

```
#include <math.h>
#include <iostream>
using namespace std;

class Para
{
  private:
    float base, hauteur;
    double angleBase;
    char *nom;

  public:
    Para();
    Para(float b,float h,double a,const char* s=nullptr);
    Para(const Para& p);
    ~Para();

    float getBase() const;
    float getHauteur() const;
    double getAngleBase() const;
    char* getNom() const;

    void setBase(float b);
    void setHauteur(float h);
    void setAngleBase(double a);
    void setNom(const char* s);

    float aire() const;
    float peri() const;
```

```
    float volume(float hP) const;
};

// ***** Constructeurs *****
Para::Para()
{
  setBase(0.0);
  setHauteur(0.0);
  setAngleBase(0.0);
  nom = nullptr;
  setNom("");
}

Para::Para(float b,float h,double a,const char* s)
{
  setBase(b);
  setHauteur(h);
  setAngleBase(a);
  nom = nullptr;
  if (s != nullptr) setNom(s);
  else
  {
    if (angleBase == 90.0) setNom("rectangle");
    else setNom("parallelogramme");
  }
}

Para::Para(const Para& p)
{
  setBase(p.getBase());
  setHauteur(p.getHauteur());
  setAngleBase(p.getAngleBase());
  nom = nullptr;
  setNom(p.getNom());
}

// ***** Destructeur *****
Para::~Para()
{
  if (nom) delete nom;
}

// ***** getXXX / setXXX *****
float Para::getBase() const { return base; }
float Para::getHauteur() const { return hauteur; }
double Para::getAngleBase() const { return angleBase; }
char* Para::getNom() const { return nom; }

void Para::setBase(float b) { base = b; }
void Para::setHauteur(float h) { hauteur = h; }
```

```
void Para::setAngleBase(double a) { angleBase = a; }
void Para::setNom(const char* s)
{
  if (nom) delete nom;
  nom = new char[strlen(s) + 1];
  strcpy(nom,s);
}

// ***** Autres methodes de l'interface *****
float Para::aire() const { return getBase() * getHauteur(); };
float Para::peri() const { return
2*(getBase()+getHauteur()/sin(getAngleBase()*M_PI/180)); };
float Para::volume(float hP) const { return aire() * hP; }

// ***** Programme principal *****
int main()
{
  Para monPa(10,30,90);
  cout << "Base = " << monPa.getBase() << endl;
  cout << "Hauteur = " << monPa.getHauteur() << endl;
  cout << "Angle a la base = " << monPa.getAngleBase() << endl;
  cout << "Nom = " << monPa.getNom() << endl;

  return 0;
}
```

Compilation en fichiers séparés

Le code du programme précédent devant assez long, il est d'usage de travailler en **fichiers séparés**. Pour chaque classe, on crée deux fichiers :

- Un premier d'extension **.h** contenant la **déclaration** de la classe et de ses variables/méthodes membres
- Un second d'extension **.cpp** contenant la **définition** des méthodes de la classe.

Nous allons prendre pour exemple notre classe Para.

Fichier « Para.h » :

```
#ifndef PARA_H
#define PARA_H

class Para
{
```

```
  private:
    float base, hauteur;
    double angleBase;
    char *nom;
  public:
    Para();
    Para(float b,float h,double a,const char* s=nullptr);
    Para(const Para& p);
    ~Para();

    float getBase() const;
    float getHauteur() const;
    double getAngleBase() const;
    char* getNom() const;

    void setBase(float b);
    void setHauteur(float h);
    void setAngleBase(double a);
    void setNom(const char* s);

    float aire() const;
    float peri() const;
    float volume(float hP) const;
};

#endif
```

Fichier « Para.cpp » :

```
#include <math.h>
#include <iostream>
using namespace std;

#include "Para.h"

// ***** Constructeurs *****
Para::Para()
{
  setBase(0.0);
  setHauteur(0.0);
  setAngleBase(0.0);
  nom = nullptr;
  setNom("");
}

Para::Para(float b,float h,double a,const char* s)
{
  setBase(b);
  setHauteur(h);
  setAngleBase(a);
  nom = nullptr;
```

```
  if (s != nullptr) setNom(s);
  else
  {
    if (angleBase == 90.0) setNom("rectangle");
    else setNom("parallelogramme");
  }
}

Para::Para(const Para& p)
{
  setBase(p.getBase());
  setHauteur(p.getHauteur());
  setAngleBase(p.getAngleBase());
  nom = nullptr;
  setNom(p.getNom());
}

// ***** Destructeur *****
Para::~Para()
{
  if (nom) delete nom;
}


// ***** getXXX / setXXX *****
float Para::getBase() const { return base; }
float Para::getHauteur() const { return hauteur; }
double Para::getAngleBase() const { return angleBase; }
char* Para::getNom() const { return nom; }

void Para::setBase(float b) { base = b; }
void Para::setHauteur(float h) { hauteur = h; }
void Para::setAngleBase(double a) { angleBase = a; }
void Para::setNom(const char* s)
{
  if (nom) delete nom;
  nom = new char[strlen(s) + 1];
  strcpy(nom,s);
}

// ***** Autres methodes de l'interface *****
float Para::aire() const { return getBase() * getHauteur(); };
float Para::peri() const { return
2*(getBase()+getHauteur()/sin(getAngleBase()*M_PI/180)); };
float Para::volume(float hP) const { return aire() * hP; }
```

Et le fichier du programme principal « Exemple11.cpp » devient :

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

#include "Para.h"

int main()
{
  Para monPa(10,30,90);
  cout << "Base = " << monPa.getBase() << endl;
  cout << "Hauteur = " << monPa.getHauteur() << endl;
  cout << "Angle a la base = " << monPa.getAngleBase() << endl;
  cout << "Nom = " << monPa.getNom() << endl;

  return 0;
}
```

La compilation se fait alors en **deux étapes** :

1. Compilation de la classe Para avec l'option –c afin de créer le fichier objet Para.o
2. Compilation du programme principal et édition de lien avec Para.o

Ce qui correspond à

```
bash-3.00$ g++ Para.cpp -c
bash-3.00$ g++ Exemple11.cpp Para.o -o Exemple11
bash-3.00$ Exemple11
Base = 10
Hauteur = 30
Angle a la base = 90
Nom = rectangle
bash-3.00$
```

Les variables membres statiques

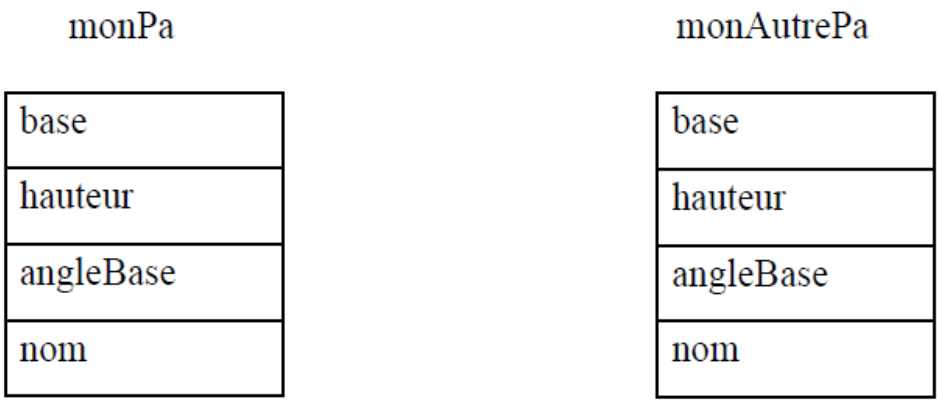
Chaque fois qu'un objet instancie une classe, le système crée pour cet objet un nouvel ensemble de variables membres.

Exemple :

La déclaration des objets suivants :

```
Para monPa, monAutrePa ;
```

donne en mémoire centrale :



Il est cependant possible de définir une **variable qui appartient à tous les objets instanciant la classe Para**. Il lui correspond alors **un seul emplacement mémoire**. Il s'agit d'une variable **statique**.

Exemple :

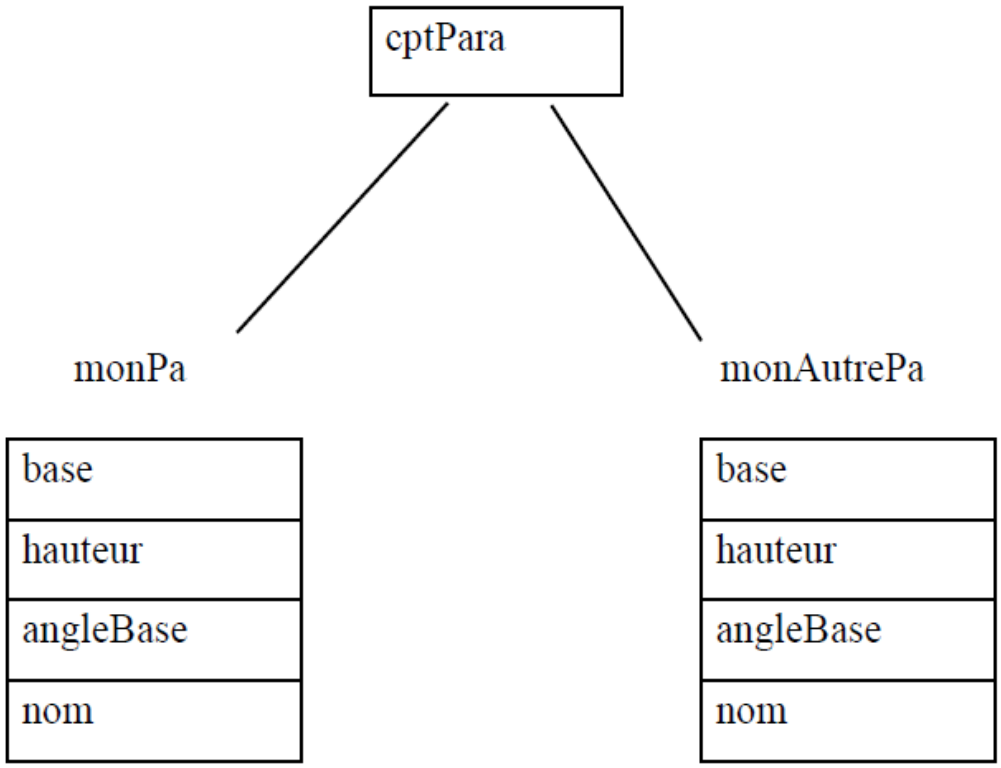
Ajoutons une variable permettant de compter le nombre d'objets instanciant la classe Para. Cette variable doit être la même pour chaque objet, elle doit être unique.

Déclaration de la variable :

```
class Para
{
  private :
    float base, hauteur ;
    double angleBase ;
    char *nom ;

  public :
    static int cptPara ;
    Para() ;
    …
};
```

Ce qui donne schématiquement en mémoire



Définition et initialisation de la variable :

Elle s'effectue en dehors de la classe par

```
int Para::cptPara = 0 ;   // dans le fichier .cpp !!!
```

C'est à ce stade, et uniquement à ce stade, que l'emplacement mémoire est réservé.

Accès à la variable à l'intérieur de la classe :

```
Para::Para()
{
  …
  cptPara++ ;
}

Para::~Para()
{
  …
  cptPara-- ;
}
```

Accès à la variable à l'extérieur de la classe :

```
#include "Para.h"

int main()
{
  cout << "cptPara = " << Para::cptPara << endl;
  Para monPa;
  cout << "cptPara = " << Para::cptPara << endl;
  cout << "cptPara = " << monPa.cptPara << endl;
  Para *pPara = new Para();
  cout << "cptPara = " << Para::cptPara << endl;
  cout << "cptPara = " << monPa.cptPara << endl;
  cout << "cptPara = " << pPara->cptPara << endl;
  delete pPara;

  return 0;
}
```

Ce qui donne à l'exécution :

```
bash-3.00$ Exemple12
cptPara = 0
cptPara = 1
cptPara = 1
cptPara = 2
cptPara = 2
cptPara = 2
bash-3.00$
```

On observe que :

- La variable statique est donc une **sorte de variable globale** qui existe tout au long du programme → elle est donc accédée directement, c'est-à-dire sans passer par le pointeur this d'un objet de la classe

- L'**accès à la variable statique publique** : on utilise l'opérateur de résolution de portée « :: » avec le nom de la classe

- On peut **utiliser cette variable statique avant même qu'un seul objet de cette classe n'existe** !

Les fonctions membres statiques

Il est possible de définir des fonctions membres statiques.

Quel intérêt ? Protéger l'accès des données statiques de la même manière que les autres données de la classe.

Ce qui donnera :

```
class Para
{
  private :
    float base, hauteur ;
    double angleBase ;
    char *nom ;
    static int cptPara ;

  public :
    Para() ;
    …
    static int getCptPara() { return cptPara ; }
    …
};
```

Tandis que dans le main :

```
#include "Para.h"

int main()
{
  cout << "cptPara = " << Para::getCptPara() << endl;
```

```
  Para monPa;
  cout << "cptPara = " << Para::getCptPara() << endl;
  cout << "cptPara = " << monPa.getCptPara() << endl;
  Para *pPara = new Para();
  cout << "cptPara = " << Para::getCptPara() << endl;
  cout << "cptPara = " << monPa.getCptPara() << endl;
  cout << "cptPara = " << pPara->getCptPara() << endl;
  delete pPara;

  return 0;
}
```

Récapitulatif des 2 types de variables et de fonctions d'une classe :

Accès aux variables	Variable classique	Variable statique
Fonction membre classique	**OUI**	**OUI**
Fonction membre statique	**NON**	**OUI**

Les fonctions membres statiques **ne peuvent pas** accéder **aux variables membres qui ne sont pas statiques** !

C'est logique dans la mesure où les fonctions membres statiques

- peuvent être appelées sans faire référence à un objet et, même plus,
- peuvent être invoquées dans un programme sans même avoir créé d'objets auparavant !

Les fonctions membres statiques ne peuvent donc accéder qu'aux variables statiques.

Une autre utilisation du mot réservé « delete »

Comme nous l'avons vu précédemment, le mot réservé « **delete** » permet de détruire proprement un objet alloué dynamiquement en appelant son destructeur et en libérant l'espace mémoire de l'objet.

Depuis C++ 11, il est devenu possible de marquer explicitement des fonctions comme supprimées en utilisant **= delete**, offrant ainsi un contrôle plus explicite et précis sur l'interface des classes et des fonctions.

Actuellement, on pourrait imaginer

- décorer tous les constructeurs d'une classe avec **« = delete »** afin d'empêcher l'instanciation du moindre objet de cette classe → cela permettrait par exemple de créer une classe dite « boîte à outils » qui ne contiendrait que des méthodes et variables statiques
- décorer le constructeur de copie avec **« = delete »** → cela permettrait d'empêcher les copies d'un objet une fois instancié → très pratique lorsque l'on veut instancier un objet unique appelé « singleton » dans une application

Ce décorateur se met uniquement dans le fichier **.h**. En effet, la méthode étant supprimée, elle n'apparaît pas dans le fichier **.cpp**.

Exemple d'une classe n'ayant pas de constructeur :

Soit le fichier **MyClass.h** :

```
class MyClass {
public:
    MyClass() = delete;  // Empêche la création d'instances
    MyClass(const MyClass&) = delete; // Empêche la copie d'instance

    // Méthode statique exemple
    static int calcul(int) ;
};
```

Dans le fichier **MyClass.cpp** :

```
#include "MyClass.h"

int MyClass::calcul(int val) {
    return 3 * val ;
}
```

Ainsi le C++ ne créera pas implicitement un constructeur par défaut et un constructeur de copie, ce qu'il ferait sans la décoration « delete ».

Considérons le fichier **main1.cpp** suivant

```
#include "MyClass.h"

int main() {
     MyClass obj;
     MyClass obj2(obj);
}
```

Tentons de compiler la classe et ce programme :

```
[student@moon Delete]$ g++ MyClass.cpp -c
[student@moon Delete]$ g++ main1.cpp MyClass.o -o main1
main1.cpp: Dans la fonction « int main() »:
main1.cpp:4:10: error: utilisation de la fonction supprimée
« MyClass::MyClass() »
  MyClass obj;
          ^~~
In file included from main1.cpp:1:
MyClass.h:3:5: note: déclaré ici
     MyClass() = delete;                // Empêche la création
d'instances
     ^~~~~~
main1.cpp:5:18: error: utilisation de la fonction supprimée
« MyClass::MyClass(const MyClass&) »
  MyClass obj2(obj);
                  ^
In file included from main1.cpp:1:
MyClass.h:4:5: note: déclaré ici
     MyClass(const MyClass&) = delete;  // Empêche la copie d'instance
     ^~~~~~
[student@moon Delete]$
```

On observe que

- la classe **MyClass** compile sans aucun souci bien qu'il n'y ait aucun code de constructeur dans le fichier **MyClass.cpp**
- le fichier **main1.cpp** ne compile pas car ont tente d'instancier des objets en utilisant les constructeurs par défaut et de copie

Considérons à présent le fichier **main2.cpp** suivant

```
#include <iostream>
#include "MyClass.h"

int main() {
	std::cout << "resultat = " << MyClass::calcul(7) << std::endl;
}
```

Ce qui donne à présent

```
[student@moon Delete]$ g++ MyClass.cpp -c
[student@moon Delete]$ g++ main2.cpp MyClass.o -o main2
[student@moon Delete]$ main2
resultat = 21
[student@moon Delete]$
```

On observe que

- la compilation se fait sans souci à présent
- la méthode statique est parfaitement utilisable
- la classe **MyClass** peut être considéré une boîte à outils → on peut ainsi regrouper des méthodes statiques (et des variables si on voulait) par thème, ce thème étant imposé par le nom de la classe → en C++, on préfère travailler ainsi au lieu de déclarer des méthodes et des variables globales

Il est également possible de décorer un destructeur avec **« = delete »** afin d'empêcher la destruction d'un objet alloué dynamiquement. Nous retrouverons également le décorateur **« = delete »** lors de la surcharge d'opérateurs qui sont créés implicitement par le C++.

Les associations entre classes

Dans une application, les classes ne sont pas totalement indépendantes les unes des autres. Le plus souvent, elles coopèrent entre elles.

Une **phase d'analyse** réfléchie doit précéder toute écriture d'une ligne de code : c'est le rôle de la conception orientée objet que de mettre en place la structure des relations entre les classes ainsi que les interactions entre les objets correspondants.

Prenons l'exemple d'une banque faisant intervenir des clients, des comptes, des placements, etc…

Classes « Client » et « Compte » :

```
class Client
{
  private:
    char nom[30];
    int  numClient;

  public:
    Client(const char* n,int num):numClient(num) { strcpy(nom,n); }
    const char* getNom() const { return nom; }
    int getNumClient() const { return numClient; }
    void Affiche() const
    {
      cout << "Nom = " << getNom() << endl;
      cout << "NumClient = " << getNumClient() << endl;
    }
};

class Compte
{
  private:
    float solde;

  public:
    Compte():solde(0.0) {}
    float getSolde() const { return solde; }
    void credit(float m) { solde += m; }
    void debit(float m) { solde -= m; }
    void Affiche() const
    { cout << "Solde du compte = " << solde << endl; }
};
```

Un point intéressant est qu'il serait souhaitable qu'un objet instanciant Compte puisse fournir comme renseignement les coordonnées du client.

Agrégation par valeur

On ajoute à la classe Compte une variable membre instanciant le Client : on parle de relation de type « **par valeur** » ou encore de « **relation d'agrégation par valeur** ».

La classe Compte devient alors

```
class Compte
{
  private:
    float solde;
    Client client;

  public:
    …
    void Affiche() const
    {
      cout << "Solde du compte = " << solde << endl;
      cout <<  "Client :" << endl ;
      client.Affiche() ;
    }
};
```

Cependant, l'objet encapsulé n'existe que pendant que l'objet contenant existe également.

- Ceci pose problème dans le cas où un client n'a pas encore de compte. Un objet client peut exister sans compte.
- Un seul et même client peut avoir plusieurs comptes…

Agrégation par référence

L'idée est qu'un objet Client peut être référencé par plusieurs objets Compte différents. On ajoute à la classe Compte **un pointeur** désignant le client. On parle alors de relation du type « **par référence** » ou encore de « **relation d'agrégation par référence** ».

La classe Compte devient alors

```
class Compte
```

```
{
  private:
    float solde;
    Client* pClient;

  public:
    …
    void Affiche() const
    {
      cout << "Solde du compte = " << solde << endl;
      cout <<  "Client :" << endl ;
      pClient->Affiche() ;
    }
};
```

Cette fois, le client peut exister indépendamment du fait d'avoir un compte ou pas. De plus, un compte bancaire non encore attribué à un client possède un pointeur pClient nullptr.

Les utilisations

On peut imaginer que le placement d'un capital arrive à terme et que l'on souhaite créditer le compte de ce capital et des intérêts acquis. On peut donc imaginer la classe Placement suivante :

```
class Placement
{
  private:
    float capital;
    int   tauxAnnuel;

  public:
    Placement(float c,int t):capital(c),tauxAnnuel(t) {}
    float interet(int nbMois)
    { return capital*((float)tauxAnnuel/100)*(float)nbMois/12; }
};
```

Ajoutons à notre classe Compte une nouvelle méthode PlacementATerme. Cette méthode pourrait s'écrire

```
class Compte
{
  private:
```

```
    …

  public:
    …
    void PlacementATerme(float capital,int tauxA,int nbMois)
    {
      Placement pl(capital,tauxA);
      credit(capital);
      credit(pl.interet(nbMois));
    }
};
```

On peut remarquer que cette méthode créée une variable local pl, instance de Placement, dans le but d'utiliser la méthode interet(). On parle de « **relation d'utilisation** ». On entend par là que la classe Compte utilise un « **service** » de la classe Placement.

Code complet illustrant l'agrégation par référence et la relation d'utilisation :

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

class Client
{
  private:
    char nom[30];
    int  numClient;

  public:
    Client(const char* n,int num):numClient(num) { strcpy(nom,n); }
    const char* getNom() const { return nom; }
    int getNumClient() const { return numClient; }
    void Affiche() const
    {
      cout << "Nom = " << getNom() << endl;
      cout << "NumClient = " << getNumClient() << endl;
    }
};

class Placement
{
  private:
    float capital;
    int   tauxAnnuel;

  public:
    Placement(float c,int t):capital(c),tauxAnnuel(t) {}
```

```
    float interet(int nbMois)
    { return capital*((float)tauxAnnuel/100)*(float)nbMois/12; }
};

class Compte
{
  private:
    float solde;
    Client* pClient;

  public:
    Compte():solde(0.0),pClient(nullptr) {}

    float getSolde() const { return solde; }

    Client* getClient() const { return pClient; }
    void setClient(Client *pC) { pClient = pC; }

    void credit(float m) { solde += m; }
    void debit(float m) { solde -= m; }
    void Affiche() const
    {
      cout << "Solde du compte = " << solde << endl;
      cout << "Client :" << endl;
      if (pClient) pClient->Affiche();
    }
    void PlacementATerme(float capital,int tauxA,int nbMois)
    {
      Placement pl(capital,tauxA);
      credit(capital);
      credit(pl.interet(nbMois));
    }
};

int main()
{
  Compte compte;
  compte.Affiche();
  cout << endl;

  Client *pC = new Client("Wagner",17);
  compte.setClient(pC);
  compte.Affiche();
  cout << endl;

  compte.debit(150.0);
  compte.PlacementATerme(1000.0,3,6);
  compte.Affiche();

  delete pC;
```

```
  return 0;
}
```

L'exécution de ce code fournit :

```
bash-3.00$ a.out
Solde du compte = 0
Client :

Solde du compte = 0
Client :
Nom = Wagner
NumClient = 17

Solde du compte = 865
Client :
Nom = Wagner
NumClient = 17
bash-3.00$
```

Les namespaces en C++

Introduction

Dès qu'un programme C++ commence à prendre de l'ampleur, le nombre de classes peut également commencer à grandir et on n'est pas à l'abri de créer des classes (ou fonctions / variables) dont les noms sont déjà utilisés dans d'autres librairies (et donc fichiers .h et .cpp). Ce qui engendrerait des conflits de nom.

En C++, les **namespaces** (**espaces de noms**) sont utilisés pour organiser le code et éviter les conflits de noms, surtout dans les projets de grande envergure ou lors de l'utilisation de bibliothèques externes. Ils permettent de regrouper des entités comme des classes, des fonctions, des variables et des objets sous un même nom, créant ainsi un espace de noms distinct.

Création d'un espace de nom

Pour créer un **namespace**, le plus propre est de travailler en fichiers séparés (.h et .cpp).

Prenons l'exemple d'un **namespace « myspace »** contenant

- Une variable globale **compteur** de type int

- Une fonction **hello()**, et
- Une classe **MaClasse**

Nous travaillons en fichiers séparés et nous rassemblons la variable **compteur** et la fonction **hello()** dans le même module. Voici donc le fichier **utils.h** :

```
#ifndef UTILS_H
#define UTILS_H

namespace myspace {
  extern int compteur;

  void hello();
} // end of namespace myspace

#endif
```

où la variable globale **compteur** est déclarée « **extern** » car on ne déclare pas des variables dans un fichier .h. Le fichier **utils.cpp** est quant à lui :

```
#include <iostream>
#include "utils.h"

namespace myspace {
  int compteur = 3;

  void hello() {
    std::cout << "Hello namespace !" << std::endl;
  }
} // end of namespace myspace
```

La classe **MaClasse** est fournie dans le fichier **MaClasse.h** :

```
#ifndef MA_CLASSE_H
#define MA_CLASSE_H

namespace myspace {

class MaClasse {
  private:
    int val;

  public:
    MaClasse(int);

    void display() const;
};
```

```
} // end of namespace myspace

#endif
```

et le fichier **MaClasse.cpp** :

```
#include "MaClasse.h"
#include <iostream>

namespace myspace {

MaClasse::MaClasse(int val) {
  this->val = val;
}

void MaClasse::display() const {
  std::cout << "[MaClasse val=" << this->val << "]" << std::endl;
}

} // end of namespace myspace
```

Ces deux modules se compilent de manière tout à fait classique :

```
[student@moon Namespaces]$ g++ MaClasse.cpp -c
[student@moon Namespaces]$ g++ utils.cpp -c
[student@moon Namespaces]$ ls -l *.o
-rw-rw-r--. 1 student student 3072 15 jui 08:39 MaClasse.o
-rw-rw-r--. 1 student student 2784 15 jui 08:39 utils.o
[student@moon Namespaces]$
```

On observe donc que le **nom d'un namespace** est tout à fait indépendant du nombre et du nom des fichiers qui le constituent.

Utilisation d'un namespace

Voici un premier exemple d'un programme (**main1.cpp**) qui utilise les éléments qui se trouvent dans le namespace « **myspace** » :

```
#include <iostream>
#include "utils.h"
#include "MaClasse.h"

int main() {
  std::cout << "Compteur avant = " << myspace::compteur << std::endl;
  myspace::compteur++;
```

```
  std::cout << "Compteur apres = " << myspace::compteur << std::endl;

  myspace::hello();

  myspace::MaClasse obj(7);
  obj.display();

  return 0;
}
```

où on observe que

- le nom de chaque élément du namespace a du être préfixé du nom du **namespace myspace** concaténé avec l'opérateur de résolution de portée **::**
- ici il en a été de même pour **cout** et **endl** du namespace **std**

La compilation et l'exécution de ce programme fournissent :

```
[student@moon Namespaces]$ g++ main1.cpp -c
[student@moon Namespaces]$ g++ main1.o utils.o MaClasse.o -o main1
[student@moon Namespaces]$ main1
Compteur avant = 3
Compteur apres = 4
Hello namespace !
[MaClasse val=7]
[student@moon Namespaces]$
```

Utilisation d'un namespace avec le mot clé « using »

Lorsqu'il n'y a pas d'ambiguïté possible, on peut utiliser le mot clé **using** afin d'inclure d'un coup tous les éléments présents dans un **namespace**.

En reprenant le **namespace myspace**, voici un second programme de test (**main2.cpp**) :

```
#include <iostream>
#include "utils.h"
#include "MaClasse.h"

using namespace std;
using namespace myspace;

int main() {
  cout << "Compteur avant = " << compteur << endl;
  compteur++;
  cout << "Compteur apres = " << compteur << endl;
```

```
  hello();

  MaClasse obj(7);
  obj.display();

  return 0;
}
```

dont la compilation et l'exécution fournissent exactement les mêmes résultats. Le fait d'utiliser cette méthode permet d'éviter de recopier le nom du namespace précédent de **::** devant les éléments du namespace, mais ouvre la porte aux ambiguïtés de nom.

Une ambiguïté qui pose problème

Considérons cet exemple de programme de test (**main3.cpp**) :

```
#include <iostream>
#include "utils.h"
#include "MaClasse.h"

using namespace std;
using namespace myspace;

int compteur = 100;

int main() {
  cout << "Compteur avant = " << compteur << endl;
  compteur++;
  cout << "Compteur apres = " << compteur << endl;

  hello();

  MaClasse obj(7);
  obj.display();

  return 0;
}
```

dans lequel nous avons déclaré une variable globale **compteur** du même nom que celle du **namespace myspace**. La compilation fournit :

```
[student@moon Namespaces]$ g++ main3.cpp -c
main3.cpp: Dans la fonction « int main() »:
main3.cpp:11:34: error: la référence à « compteur » est ambiguë
   cout << "Compteur avant = " << compteur << endl;
                                  ^~~~~~~~
In file included from main3.cpp:2:
```

```
utils.h:5:14: note: les candidats sont : « int myspace::compteur »
   extern int compteur;
              ^~~~~~~~
main3.cpp:8:5: note:                       « int compteur »
 int compteur = 100;
     ^~~~~~~~
main3.cpp:12:3: error: la référence à « compteur » est ambiguë
   compteur++;
   ^~~~~~~~
In file included from main3.cpp:2:
utils.h:5:14: note: les candidats sont : « int myspace::compteur »
   extern int compteur;
              ^~~~~~~~
main3.cpp:8:5: note:                       « int compteur »
 int compteur = 100;
     ^~~~~~~~
main3.cpp:13:34: error: la référence à « compteur » est ambiguë
   cout << "Compteur apres = " << compteur << endl;
                                 ^~~~~~~~
In file included from main3.cpp:2:
utils.h:5:14: note: les candidats sont : « int myspace::compteur »
   extern int compteur;
              ^~~~~~~~
main3.cpp:8:5: note:                       « int compteur »
 int compteur = 100;
     ^~~~~~~~
[student@moon Namespaces]$
```

où on observe que le compilateur est incapable de faire la différence entre la variable globale **compteur** et la variable globale **compteur** du **namespace myspace**. La compilation échoue donc.

Lever l'ambiguïté

Voici un nouveau programme de test (**main4.cpp**) :

```
#include <iostream>
#include "utils.h"
#include "MaClasse.h"

using namespace std;
using namespace myspace;

int main() {
  int compteur = 100;

  cout << "Compteur avant = " << compteur << endl;
  compteur++;
```

```
  cout << "Compteur apres = " << compteur << endl;

  hello();

  MaClasse obj(7);
  obj.display();

  return 0;
}
```

dans lequel nous avons déclaré une variable locale **compteur** qui porte le même nom que la variable globale **compteur** du **namespace myspace**. La compilation et l'exécution de ce programme fournissent :

```
[student@moon Namespaces]$ g++ main4.cpp -c
[student@moon Namespaces]$ g++ main4.o utils.o MaClasse.o -o main4
[student@moon Namespaces]$ main4
Compteur avant = 100
Compteur apres = 101
Hello namespace !
[MaClasse val=7]
[student@moon Namespaces]$
```

L'ambiguïté est donc levée mais la variable locale **compteur** « couvre » complètement la variable globale **compteur** du **namespace myspace**.

Une autre manière de lever l'ambiguïté est fournie dans le programme de test (**main5.cpp**) suivant :

```
#include <iostream>
#include "utils.h"
#include "MaClasse.h"

using namespace std;
using namespace myspace;

int main() {
  int compteur = 100;

  cout << "Compteur avant = " << ::compteur << endl;
  ::compteur++;
  cout << "Compteur apres = " << ::compteur << endl;

  hello();

  MaClasse obj(7);
  obj.display();

  return 0;
}
```

dont la compilation et l'exécution fournissent :

```
[student@moon Namespaces]$ g++ main5.cpp -c
[student@moon Namespaces]$ g++ main5.o utils.o MaClasse.o -o main5
[student@moon Namespaces]$ main5
Compteur avant = 3
Compteur apres = 4
Hello namespace !
[MaClasse val=7]
[student@moon Namespaces]$
```

L'utilisation de l'opérateur de résolution de portée **::** permet de récupérer l'accès à la variable globale **compteur** du **namespace myspace**.