# La programmation par Objets

Tour d'horizon avec C++

# Entrée / Sortie
# Lecture / Ecriture
# Cin / Cout

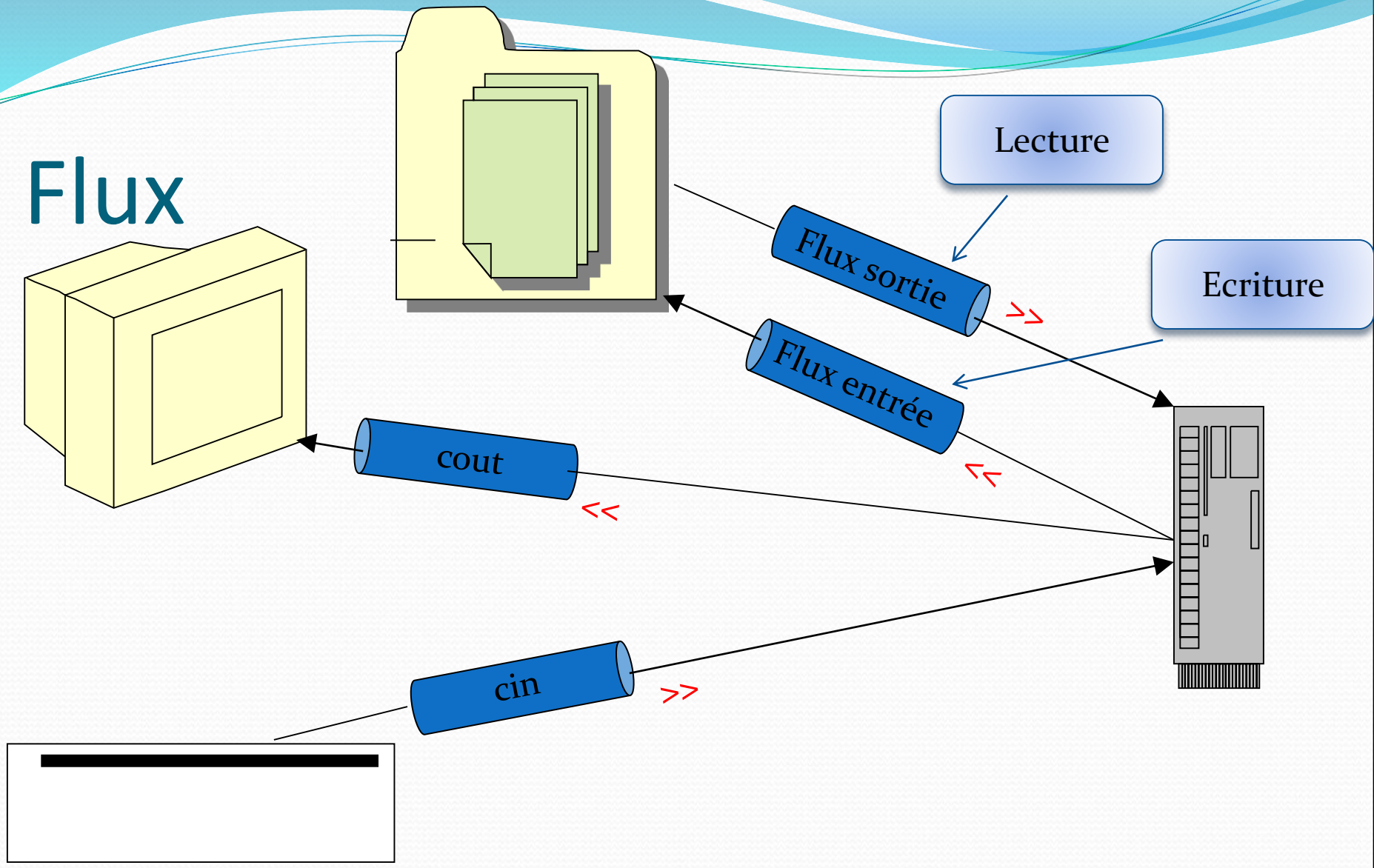
# Entrée / Sorties

- E/S par l'intermédiaires de **flots** (ou flux)

- Flot d'entrée :
  - **fournit** de l'information (**envoie** des valeurs)
  - opérateur : >> (lecture sur un flot d'entrée)
  - standard : **cin** (connecté par défaut au clavier)

- Flot de sortie :
  - **reçoit** de l'information (**récupère** des valeurs)
  - opérateur : << (écriture sur un flot de sortie)
  - standard : **cout** (connecté par défaut à l'écran)

# Flux



# Création de flux

- Inclure les fichiers d'en-tête :
  - **<fstream.h>**
  - **<iostream.h>**

- Créer un **flux d'entrée** (ouvrir un fichier en lecture) :
  - **ifstream** *nomFlux* ("*nomFichier*");
  - `ifstream fluxE("C:\MesDocs\Cpp\test.txt");`

- Créer un **flux de sortie** (ouvrir un fichier en écriture) :
  - **ofstream** *nomFlux* ("*nomFichier*");
  - `ofstream fluxS("test2.txt");//meme repertoire`

# Ouverture / Fermeture

- Par défaut, un flux de sortie remplace (supprime) les données existantes du fichier
- Il faut préciser s'il s'agit d'un ajout

  (ouvrir le fichier en mode ajout):
  - **fstream** *nomFlux* ("*nomFichier* ", **ios_base::app**);
  - `fstream fluxSA("test3.txt", ios_base::app);`

- Fermeture d'un flux (fermeture du fichier)
  - *nomFlux*.**close**();
  - `fluxSA.close();`

# Lecture/Écriture dans fichiers

- Écriture sur un flux (dans un fichier)
  - *nomFlux* << *expression1* [<<*expression2*<< ...];
  - **`fluxS << nom << " " << age << endl;`**
    **`(nom: string, age : int)`**

- Lecture sur un flux (dans un fichier)
  - *nomFlux* >> *variable1* [>> *variable2*>> ...];
  - **`fluxE >> nom >> age;`**
  - (espaces et le sauts de ligne sont des séparateurs de lecture)

# Lecture/Écriture dans fichiers

- Test de fin de fichier
  - *nomFlux*.**eof( )**
  - `b = fluxE.eof();  //b : bool`

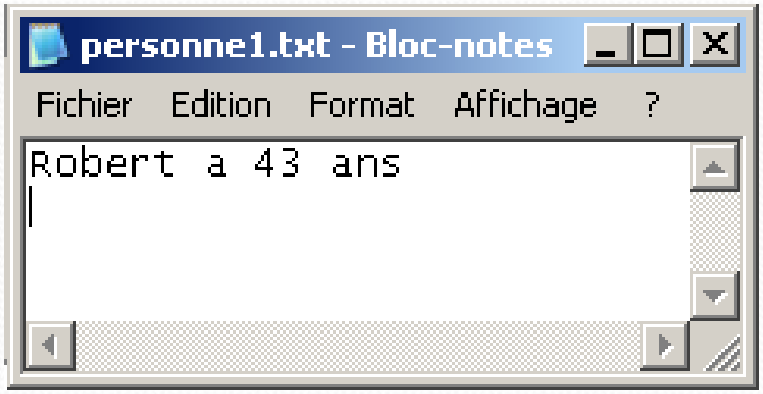
- Vrai si la lecture a **déjà** déclenché une erreur
  - Attention :

```
while ( !fluxE.eof() )
{
      fluxE >> nom >> tel ;
      cout << nom << "\t"<< tel << endl;
}
```

Fait une boucle de trop !
Il faut donc que la dernière chose de la boucle soit une lecture !

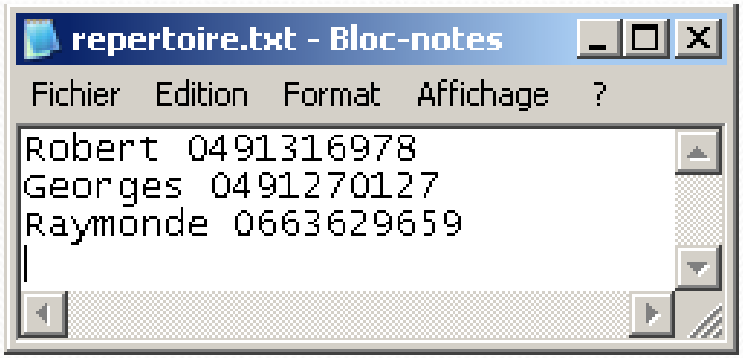
# Exemple écriture

```
#include<fstream>
#include<string>
#include<iostream>
using namespace std;
main ()
{
   string nomSaisi;
   int ageSaisi;
   cout << " quel est votre nom? ";
   cin >> nomSaisi;
   cout << " quel est votre age? ";
   cin >> ageSaisi;
   ofstream fluxS("personne1.txt", ios::in|ios::app);
   fluxS << nomSaisi << " a "<< ageSaisi << " ans " << "\n";
   fluxS.close();
}
```



# Exemple Lecture

```
#include<fstream>
#include<string>
#include<iostream>
using namespace std;
main ( )
{
   string nom;
   int tel ;
   ifstream fluxE("repertoire.txt");
   fluxE >> nom >> tel;
   while ( !fluxE.eof() )
   {
        cout << nom << "\t"<< tel << endl;
        fluxE >> nom >> tel ;
   }
   fluxE.close( );
}
```




Attention à l'ordre !

# Les classes

# Introduction

- Quelques définitions

Problème ? Algorithme ⟶ Programme

Un **algorithme** est une description non ambiguë d'un calcul ou de la résolution d'un problème en un nombre fini d'étapes.

Un **programme** est l'implantation d'un algorithme dans un langage de programmation

Un **langage de programmation** n'est donc qu'un moyen d'expression des traitements que peut réaliser une machine pour résoudre le problème.

# Introduction

- Traitement + Données = Programmes

- Priorité aux données :
  - SGBD + langages de requête
- Priorité aux traitements :
  - Algorithmes + langages de programmation structurés
- Approche combinée :
  - Structures de données & fonctions ⟶ Classes d'objets

# Introduction

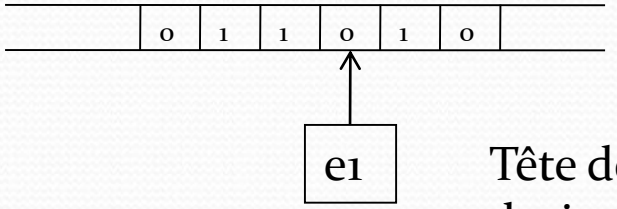
Aperçu des langages :

- Il existe un grand nombre de langages de programmation (>2000 !)
- Ces langages partagent beaucoup de concepts commun :
  - Nombre limité de **structures lexicales** (vocabulaire)
  - Nombre limité de **syntaxe** (grammaire)
  - Nombre limité de **sémantiques** (interprétation)
  - Nombre limité de mécanismes de **traduction** et **d'exécution**

# Introduction

Aperçu des langages :

Point commun ultime : équivalence des langages avec celui de la **machine de Turing universelle** (1936)




Bande magnétique

Tête de lecture pouvant lire, écrire et se déplacer à droite et à gauche

# Introduction

- La genèse des langages
  - Origine (<1955) : le langage machine
    - Seul langage compris par la machine (directement)
    - L'UC reçoit puis exécute 1 à 1 les instructions

Inintelligible pour les humains : Une somme

```
Binaire         hex

10001011        8B \
01000101        45  | lire premier entier dans le registre AX
00001010        0A /
00000011        03 \
01000101        45  | lire second entier et ajouter à AX
00010100        14 /
```

# Introduction

- La genèse des langages
  - Le langage assembleur
    - Un peu plus lisible
    - Difficile car nécessite une très bonne connaissance de l'architecture de la machine (registres du processeur…)

```
mov 10(%ebp), %ax    ; lire premier entier dans le reg. AX
Add 20(%ebp), %ax    ; lire second entier et ajouter à AX
```

Langage Bas niveau

# Introduction

- La genèse des langages
  - Le langage haut niveau
    - Notation plus lisible et familières
    - Portabilité (peut être exécuter sur différentes machines sans trop de modifications)
    - Détection plus simple d'erreurs

```
AX = X+Y;
```

# Introduction

- La genèse des langages :
  - Le langage à Objets
    - On mélange les structures avec des fonctions
    - Permet d'avoir des blocs autonomes (qui se chargent de tout : s'initialiser, modifier les valeurs de la structure, s'afficher…)

# Les objets

Les objets sont les éléments logiciels fondamentaux associés au paradigme de programmation par objets (sic!)

Définition : un objet représente une chose, une entité ou un concept et est caractérisé par un état, un comportement et une identité.

- L'état d'un objet est égal à la valeur des données qui lui sont associées
- Le comportement correspond à la responsabilité de l'objet
- L'identité précise l'existence de l'objet indépendamment de sa valeur et de la façon dont on le désigne

# Les objets

Pour simplifier :

- Un objet, c'est
  - Une structure
  - Des fonctions
- Donc dans un objet, on a
  - Des variables (comme dans une structure)
  - Des fonctions

# Les objets

Un objet représente une chose, une entité ou un concept et est caractérisé par un état, un comportement et une identité.

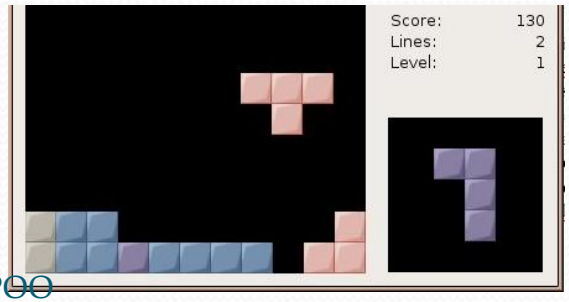
- Objet : association des données (attributs) et fonctions (méthodes) agissant sur ces données
  - Objet = Attributs + Méthodes
- Autre vocabulaire :
  - Attribut : **donnée membre**
  - Méthode : **fonction membre**

# Les objets

Exemple : Un jeu de Tetris-like représente chaque pièce par un objet

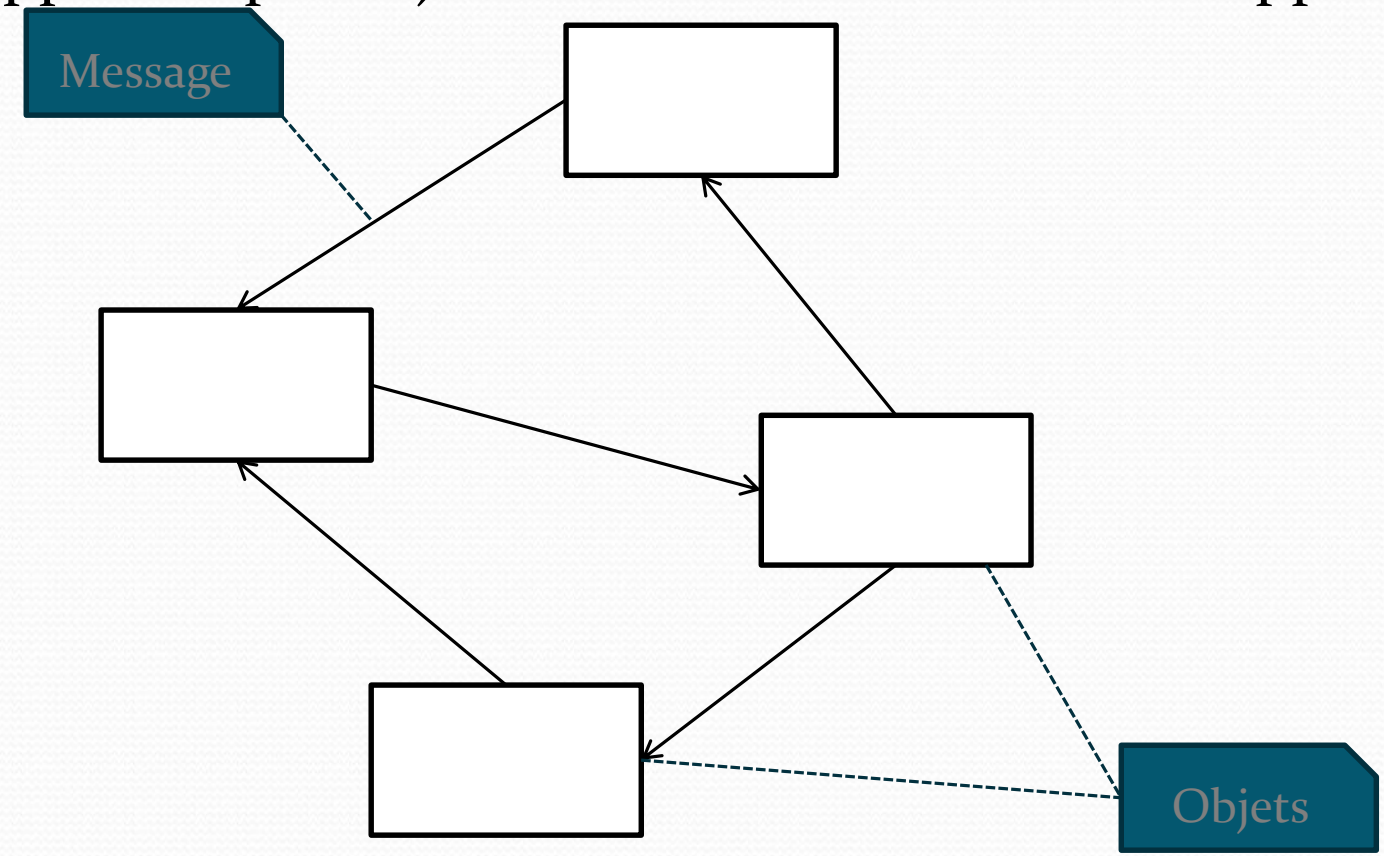
Les données associées à une pièce sont sa dimension (nombre de blocs composants), sa forme (disposition relative des blocs), son orientation, sa position dans le tableau et sa couleur.

La responsabilité d'une pièce inclut la capacité de se déplacer verticalement d'une ou plusieurs cases, de pivoter à gauche et à droite



# La programmation par objets

- L'approche par objets est basé sur une nouvelle approche



# La programmation par objets

- L'approche par objets est basé sur une nouvelle approche

Résoudre un problème en adoptant une démarche par objets consiste donc à **créer un certain nombre d'objets** correspondant aux éléments du problème, et à faire **communiquer ces objets** entre eux.

# La programmation par objets

- Exemple (Tetris) : Un objet pièce reçoit un message demandant un déplacement maximum vers le bas

1. L'objet envoie des messages à toutes les pièces déjà disposées dans le tableau, afin de leur demander d'évaluer leur position
2. L'objet calcule sa position finale en fonction des contraintes imposées par les autres pièces
3. L'objet réalise le déplacement en envoyant un message à chacun de ses blocs composants avec leurs nouvelles positions respectives

# Apports de la POO

- *Classe* : **description d'un type d'objet**
  - une variable de ce type est appelé instance
  - Encapsulation des données
    - Comme pour une structure, la variable contient plusieurs informations de plusieurs types différents
  - Héritage de classe
    - Permet de donner des fonctions à ses enfants (permet de ne pas avoir a tout réécrire)

# Encapsulation

- Il n'est **pas possible d'agir directement sur les attributs** d'un objet
  - Par défaut, les attributs sont privés, c'est-à-dire qu'ils ne sont pas accessibles à partir du `main()` !
- Il faut passer par l'intermédiaire de **méthodes dédiées**
  - En général précédé de **`set`** et **`get`**.
- Un objet se caractérise alors uniquement par les spécifications de ses méthodes

# Avantages principaux

- Facilite la **maintenance** : modification sans incidence
- Facilite la **réutilisabilité** : détails d'implémentation sont cachés

- L'encapsulation n'est pas obligatoire en C++ mais fortement conseillée
  - **c'est là-dessus que vous serez noté !**

# Héritage

- Permet de **définir une nouvelle classe à partir d'une classe existante**
  - La nouvelle classe « hérite » des attributs et méthodes de l'ancienne
  - Ajout de nouveaux attributs et méthodes
  - Peut-être réitéré autant de fois que nécessaire
    (C hérite de B qui hérite de A…)

- Facilite grandement la **réutilisation et l'extensibilité**

# Classes et Objets

- Attributs et Méthodes
- Constructeurs
- Initialisation
- Destruction
- Attributs et méthodes de classe
- Composition de classe

# Définitions

- Une classe est la **généralisation de la notion de type** défini par l'utilisateur

- Classe : association à la fois les **données et** les **fonctions**

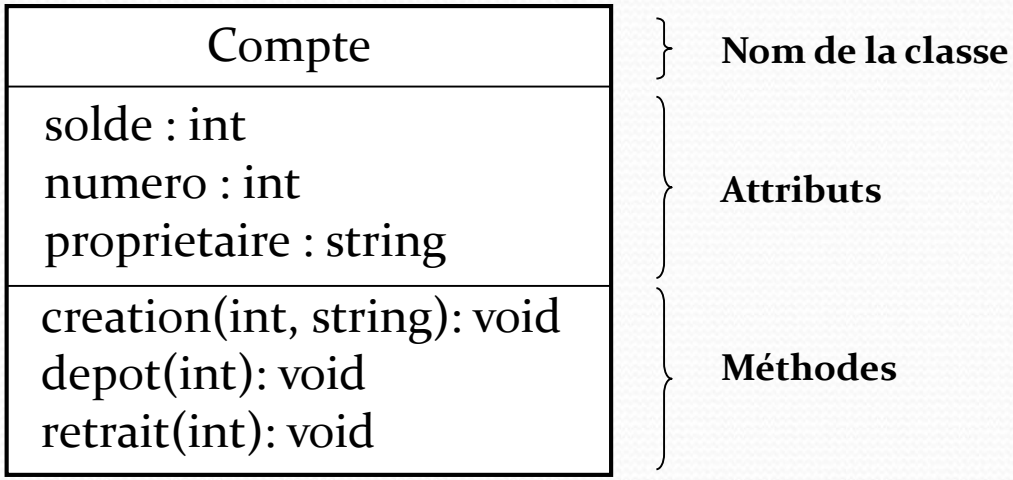
- Les objets sont des **instances** de classes
- Ils possèdent les mêmes caractéristiques que les **variables ordinaires**

# La structure d'un objet

- L'état d'un objet est défini par le **contenu** de ses variables
- Le comportement d'un objet est défini par un ensemble **d'opérations**, chacune étant représentée par une **méthode**
- Une demande d'effectuer une opération est transmise par un **message**, éventuellement accompagné de **paramètres**
  - Un message se traduit par l'appel d'une méthode de l'objet

# Objet

- Un objet se définit par :
  - *Ses **attributs*** (variables) : caractérisent son état variant au cours de l'exécution du programme
  - *Ses **méthodes*** (fonctions) : déterminent son comportement



# Classes et instances

- Il arrive bien souvent que plusieurs objets partagent les mêmes caractéristiques

Exemple : Bien que les pièces de Tetris ne sont pas strictement identiques, leur mémoire **possède la même structure**, i.e. une variable de dimension, de forme, d'orientation, …

De même, leur comportement est défini à partir des mêmes opérations telles que se déplacer d'une case, se déplacer de plusieurs cases, pivoter à gauche et pivoter à droite.

Définition : Une **classe** est un ensemble d'objets possédant des variables et des méthodes identiques.

Tout objet est une instance d'une classe

La méthode invoquée par un objet dès qu'il reçoit un message est déterminée sur la base de la **classe** de cet objet.

# Définition en C++

Compte.h

```
class Compte
{
   int solde;
   int numero;
   string proprietaire;

public :
   void creation(int, string);
   void depot(int);
   void retrait(int);
};
```

C'est ici que l'on déclare toutes les variables et toutes les fonctions que l'on pourra utiliser avec l'objet

Compte.cpp

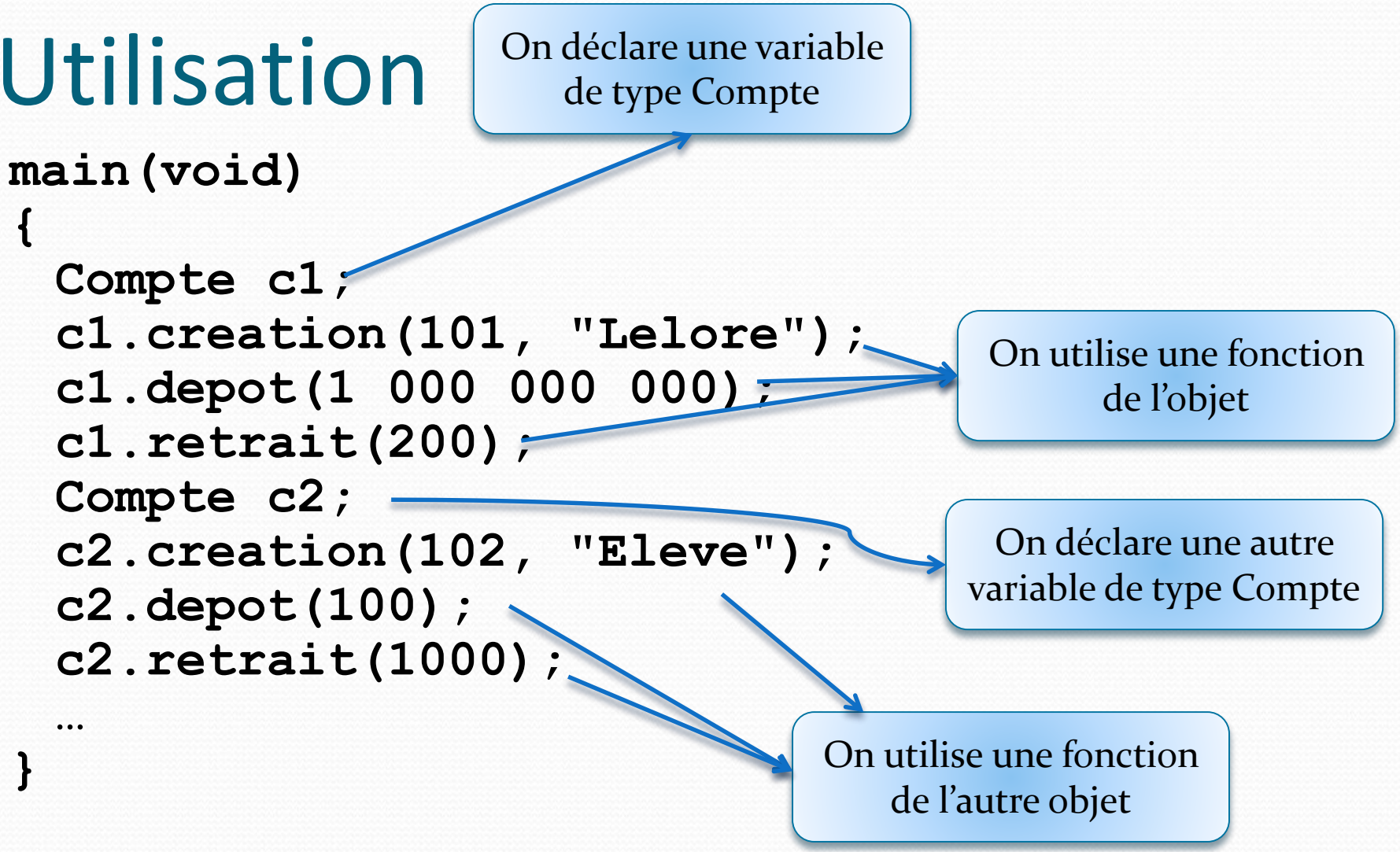
```
void Compte::creation(int num, string prop)
{
   numero = num;
   proprietaire = prop;
   solde = 0;
}

void Compte::depot(int montant)
{
   solde = solde + montant;
}

void Compte::retrait(int montant)
{
   solde = solde - montant;
}
```

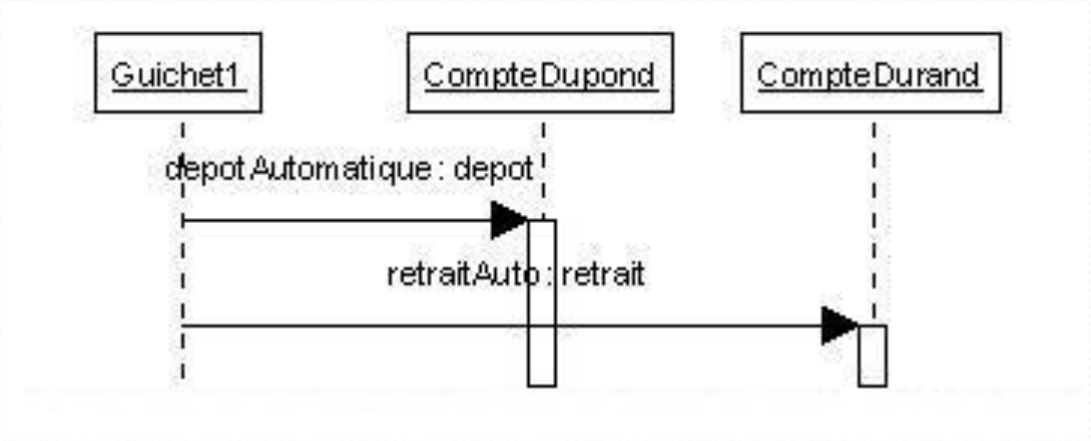
C'est ici que l'on écrit le code de toutes les fonctions que l'on a déclaré dans le fichier .h

# Utilisation



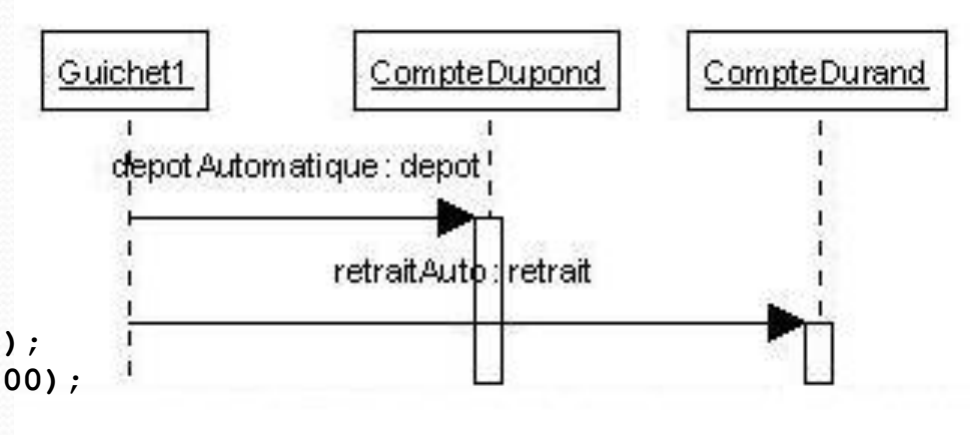
# Interaction

- Les objets interagissent entre eux par ***envoi de messages*** : demande d'exécution d'une de ses méthodes et éventuellement retour d'un résultat
- L'exécution d'un programme se traduit donc par une succession d'envois de messages



# Exemple

```
class Guichet
{
   int identifiant;
   public :
          void depotAutomatique (Compte&, int);
          void retraitAutomatique (Compte&, int);
};
void Guichet::depotAutomatique(Compte &c, int montant)
{
   c.depot(montant);
}
void Guichet::retraitAutomatique(Compte &c, int montant)
{
   c.retrait(montant);
}
main(void)
{
   Compte CptDupond, CptDurand;
   Guichet CredL;
   CompteDupond.creation(123, "Dupond");
   CompteDurand.creation(333, "Durand");
   CredL.depotAutomatique(CptDupond, 200);
   CredL.retraitAutomatique(CptDurand, 400);
}
```



# Appel entre méthodes

- D'un autre objet (instance) : préfixé de l'identifiant
- Sa propre méthode : mot clé « this »

- Exemple :

Un **&** car on veut modifier le compte

```
void Compte::virement(Compte &c, int montant)
{
  c.depot(montant);
  this->retrait(montant);
}
```

Une flèche plutôt qu'un point car c'est un pointeur… Plus de détails dans 1-2 mois

```
class Compte
{
   int solde;
   int numero;
   string proprietaire;

   public :
        void creation(int, string);
        void depot(int);
        void retrait(int);
        void virement(compte&, int);
};
…
main(void)
{
   Compte c1, c2;
   c1.creation(123, "Robert");
   c2.creation(333, "Michel");
   c2.depot(500);
   c2.virement(c1,300);
}
```

# Méthodes

- Chaque méthode étant liée à sa classe, il est possible que des méthodes ayant la **même signature** (même nom, même nombre et types de paramètres) soient définies dans des **classes différentes**
- La **surcharge** des méthodes est possible dans une classe (même nom, mais liste de paramètres différente)

# Affectation d'objet

- Il est possible d'affecter à un objet la valeur d'un autre objet :

```
Compte c1, c2;
…
c1 = c2;
```

- Recopie l'ensemble des valeurs des attributs
- Attention : comportement parfois incohérent

**Utilisation d'un constructeur de copie (voir plus loin)**

# Constructeur et Destructeur

- En général, il est nécessaire de faire appel à une méthode pour initialiser les valeurs des attributs (ex : `creation(int, string);`)
- Le constructeur permet de traiter ce problème à la déclaration de l'objet
- Le destructeur est appelé lorsque l'objet est supprimé

# Constructeur

Méthode appelée automatiquement à chaque création d'objet
Le constructeur porte le même nom que la classe

Exemple :

```
class Compte
{
  int solde;
  int numero;
  string proprietaire;

  public :
       Compte(int, string);
       void depot(int);
       void retrait(int);
};
```

```
Compte::Compte(int num, string prop)
{
        numero = num;
        proprietaire=prop;
        solde = 0;
}

main(void)
{
        Compte c1(123, "Robert");
}
```

# Utilisation

- À partir du moment où une classe possède un constructeur, il n'est plus possible de créer un objet sans fournir les arguments requis par son constructeur
  - La déclaration `Compte c1;` ne convient plus
  - L'utilisateur est obligé de donner des valeurs d'initialisation

L'utilisation de constructeur est fortement recommandée

# Utilisation

- Le travail réalisé par le constructeur peut-être plus élaboré qu'une simple initialisation
  - Affectation de valeur fabriquée au hasard
  - Allocation dynamique de mémoire (encore les pointeurs qu'on verra plus tard)

# Exemple

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

class Hasard
{
   double val[10];
   public:
        Hasard(int);
        void affiche();
};

Hasard::Hasard(int max)
{
   int i;
   for (i=0; i<10; i++)
        val[i] = double (rand()) / RAND_MAX * max;
}
```

```
main(void)
{
        Hasard suite1(5);
        suite1.affiche();
        Hasard suite2(12);
        suite2.affiche();
}
```

# Destructeur

- Méthode **appelée automatiquement** au moment de la destruction de l'objet (fin de la fonction, boucle… où l'objet a été créé)
- Porte le même nom que la classe précédé d'un tilde (~)
- Ne dispose **d'aucun argument** et ne renvoie pas de valeur
- Un destructeur devient **indispensable** lorsque l'objet **alloue de la mémoire dynamiquement**

# Exemple

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

class Matiere
{
   string nom;
   string prof;
   int nbEleves
   string *eleves;
   float *notes;
   public:
          Matiere(int, string, string);
          ~Matiere();
          bool validiteMotDePasse(string);
          void saisirNote();
          void modifierNote(string);
          void afficherNotes();
          void afficherMoyenne();
};
```

```
Matiere::Matiere(int nb, string n, string p)
{
   nbEleves = nb;
   eleves = new string [nbEleves ];
   notes = new float [nbEleves ];
   nom = n;
   prof = p;
}

Matiere ::~ Matiere()
{
   delete eleves;
   delete notes;
}

main(void)
{
   Matiere info(11, « info », « lelore »);

}
```

# Objets et tableaux

- Comme tous les types, il est possible de définir des tableaux d'objets
- Si des constructeurs ont été définis
  - C'est le constructeur sans arguments qui est appelé
    - il faut donc qu'il existe !
  - Il est possible de faire appel à un constructeur particulier, mais dans ce cas, il faut le préciser pour tout les éléments !

# Exemple

```
class maClasse
{
  string nom;
  public :
       maClasse();
       maClasse(string);
       void affiche();
};
maClasse::maClasse()
{
  nom="inconnue";
}
maClasse::maClasse(string s)
{
  nom = s;
}
```

```
void maClasse::affiche()
{
   cout << nom;
}
main()
{
maClasse tab1[5];
maClasse tab2[2]={
   maClasse("c0"),
   maClasse("c1") };
for (int i=0; i<5; i++)
   tab1[i].affiche();
for (int i=0; i<2; i++)
   tab2[i].affiche();
}
```

# Exemple

```
class maClasse
{
  string nom;
  public :
       maClasse();
       maClasse(string);
       void affiche();
};
maClasse::maClasse()
{
  nom="
}
maClass                    ing s)
{
  nom =
}
```

```
inconnue
inconnue
inconnue
inconnue
inconnue
c0
c1
```

```
void maClasse::affiche()
{
   cout << nom;
}
main()
{
maClasse tab1[5];
maClasse tab2[2]={
   maClasse("c0"),
   maClasse("c1") };
for (int i=0; i<5; i++)
   tab1[i].affiche();
for (int i=0; i<2; i++)
   tab2[i].affiche();
}
```

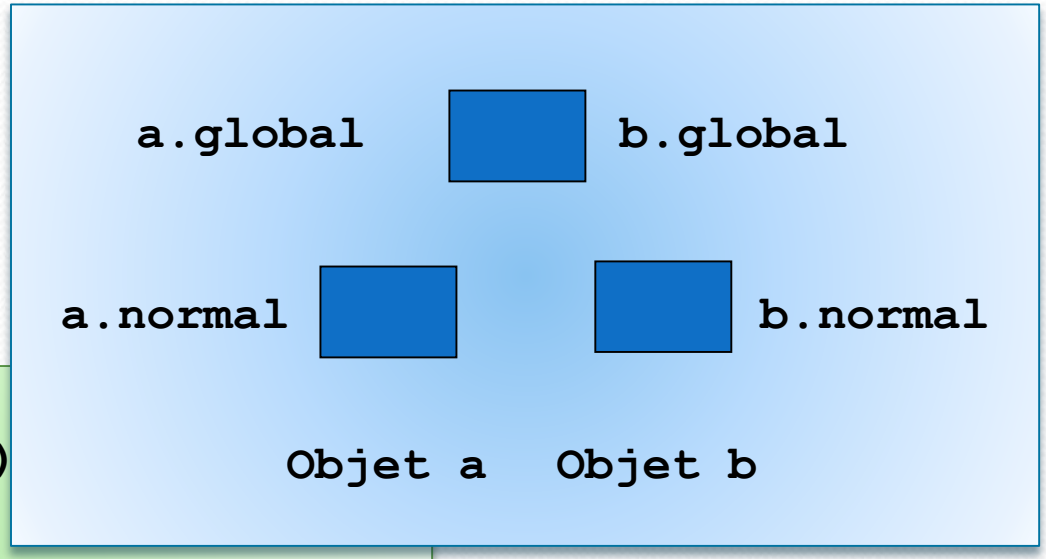
# Attributs de classe

- À priori, les différents objets d'une même classe possèdent leurs **propres attributs**
- Possibilité de permettre aux objets d'une même classe de partager une donnée : ***attributs de classe***
- Un attribut de classe est un attribut **indépendant** de toute instance de classe (Sorte de **variable globale** limitée à une classe)

# Utilisation

```
class MaClasse
{
int normal;
static string global;
public :
  MaClasse(int);
};
```

```
MaClasse a, b;
```




```
MaClasse::MaClasse(int a)
{
   normal = a;
}

string MaClasse::global = "MaClasse";
```

# Exemple

```
class Compte
{
   int solde;
   int numero;
   string proprietaire;
   static int nbComptes;
public :
   Compte(int, string);
   ~Compte();
   void depot(int);
   void retrait(int);
};


main(void)
{
   Compte c1(120,"Robert");
   Compte c2(333,"Georges");
}
```

```
int Compte::nbComptes = 0;


Compte::Compte(int num, string prop)
{
    numero = num;
    proprietaire = prop;
    solde = 0;
    nbComptes = nbComptes + 1;
    cout << nbComptes ;
}
Compte::~Compte (void)
{
    nbComptes = nbComptes - 1;
    cout << nbComptes ;
}
```

# Arguments par défaut

- Possibilité d'écrire une méthode en indiquant quelle valeur doit prendre l'argument si rien n'a été passé en paramètre
- Permet d'écrire une fonction au lieu de deux

# Exemple

```
class Compte
   {
   int solde;
   int numero;
   string proprietaire;
   static int nbComptes;

   public :
         Compte(int, string, int = 0);
         void depot(int);
         void retrait(int);
         void afficheSolde();
   };
```

```
void Compte::afficheSolde()
{
cout<<"solde="<<solde<<endl;
}

Compte::Compte(int num, string
prop, int s)
{
   numero = num;
   proprietaire = prop;
   solde = s;
   nbComptes = nbComptes + 1;
}
```

```
main(void)
{
  Compte c1(120,"Robert", 100);
  Compte c2(333,"Georgres");
  c1.afficheSolde();
  c2.afficheSolde();
}
```

```
100
0
```

# Objet argument d'une méthode

- Une méthode peut avoir en paramètre des arguments de type objet
- C'est un argument comme les autres

# Exemple par valeur

```
class Personne
{
  string nom;
  int portefeuille;
  public :
       Personne(string);
       void retireArgent(Compte);
};
…
void Personne::retireArgent(Compte c,int valeur)
{
   portefeuille = portefeuille + c.retrait(valeur);
}

main(void)
{
  Compte c1(120,"Robert");
  Personne p("Robert");
  p.retireArgent(c1,100);
}
```

# Exemple par référence

```
class Personne
{
  string nom;
  public :
       Personne(string);
       void deposerCompte(Compte&,int)
};
…
void Personne::deposerCompte (Compte &c, int somme)
{
   c.depot(somme);
}
main(void)
{
  Compte c1(120,"Robert");
  Personne p("Robert");
  p.deposerCompte(c1, 100);
}
```

# Objet en retour de fonction

- Exemple :

```
class Banque
   {
   string nom;
   public :
        Banque(string);
        Compte creerCompte();
        int newNum();
   };
…
Compte Banque::creerCompte()
{
   cout<< "quel est votre nom? " <<endl;
   string n;
   cin>>n;
   int num = newNum();
   Compte c(n,num);
   return c;
}
```

# Autoréférence

- Le mot clé **`this`** fait référence sur l'objet l'ayant appelé
  - Permet d'avoir l'adresse en mémoire de l'objet
  - Permet de résoudre les ambigüités

# Exemple

```
class Compte
  {
  int solde;
  int numero;
  string proprietaire;

  public :
       Compte(int, string);
       void changePropriétaire(string);
  };
…
void Compte::copie(string proprietaire)
  {
       this->proprietaire = proprietaire;
  }
```

# Méthode de classe

- Méthode appartenant à une classe
- Peut-être appelée en dehors de toute instance
- Permet de créer l'équivalent d'une fonction globale
- La fonction ne peut pas accéder au pointeur this !
  - Pas d'objet associé
  - N'a accès qu'aux variables globales (statiques) de la classe

# Exemple

```
class Compte
{
   int solde;
   int numero;
   string proprietaire;
   static int nbComptes;

public :
   Compte(int, string);
   ~Compte();
   static void affNbComptes();
};
```

```
int Compte::nbComptes = 0;

void Compte::affNbComptes()
{
        cout<<nbComptes<<endl;
}
```

```
main(void)
{
        Compte c1, c2;
        Compte::affNbComptes();
}
```

# Composition

- Une classe peut posséder un attribut d'une autre classe
- Si l'attribut ne possède pas de constructeur par défaut, un constructeur doit être défini
- Le constructeur de l'attribut est appelé avec des arguments du constructeur de la classe

# Exemple 1

```
class Compte
{
   int solde;
   int numero;
   string proprietaire;

   public :
   Compte(string);
};

Compte::Compte(string s)
{
   proprietaire = s;
…
}
```

```
class Personne
{
   string nom;
   Compte c;
   public :
        Personne(string);
};
Personne::Personne(string s) : c(s)
{
    nom=s;
}

void main()
{
   Personne p("Robert");
}
```

# Exemple 2

```
class Compte
{
   int solde;
   int numero;
   string proprietaire;

   public :
   Compte(string);
};

Compte::Compte(string s)
{
   proprietaire = s;
…
}
```

```
class Personne
{
        string nom;
        Compte c;
        public :
                Personne();
};
Personne:: Personne()
{
    cout << "quel est votre nom ? ";
    string n;
    cin>>n;
    nom=n;
    c=Compte(n);
}
main(void)
{
        Personne p;
}
```

# Exemple 3

```
class Compte
{
   int solde;
   int numero;
   string proprietaire;

   public :
   Compte(string);
};

Compte::Compte(string s)
{
   proprietaire = s;
…
}
```

```
class Personne
{
   string nom;
   Compte c1, c2;
   public :
        Personne();
};

Personne:: Personne(string s) : c1(s), c2(s)
{
   nom=s;
}

main(void)
{
   Personne p(" bob ");
}
```

# Héritage

# Notion d'héritage

- Un des fondements de la POO
- À la base de la réutilisation

- L'héritage autorise la définition d'une nouvelle classe dite « **dérivée** », à partir d'une classe existante dite « **de base** »

- La classe **dérivée** hérite des caractéristiques de la classe **de base** auxquelles sont ajoutées ses propres caractéristiques

# Notion d'héritage

- Plusieurs classes peuvent être dérivées d'une même classe de base
- Une classe dérivée peut devenir à son tour classe de base d'une autre classe
- L'héritage multiple est autorisé par le C++, mais reste controversé (et même déconseillé)
- Déclaration :

```
class classeDerivee : public classeDeBase
{
    …
};
```

# Déclaration d'une classe de base

```
class Point
{
  int x;
  int y;

public:
  void init(int, int);
  void deplace(int, int);
  void affiche();
};
```

# Déclaration d'une classe dérivée

```
class PointColore : public Point
{
  string couleur;

public:
  void colore(string c){couleur = c;}
};
```

# Conséquences

- le mot clé public signifie que les attributs et méthodes publics de la classe de base resteront publics (notion classique de l'héritage)
  - Précisé plus tard
- Chaque objet de type **`PointColore`** peut alors avoir accès :
  - Aux attributs et méthodes publics de **`PointColore`**
  - Aux attributs et méthodes publics de **`Point`**

# Exemple de programme

```
main()
{
  PointColore p;
  p.init(10, 10);
  p.colore("bleu");
  p.affiche();
  p.deplace(15, 30);
  p.affiche();
}
```

# Utilisation des attributs et méthodes de la classe de base

- Une classe dérivée n'a pas accès aux attributs et méthodes privés de la classe de base
  - Ex : impossible de définir une méthode dans la classe **`pointColore`** qui affiche directement les valeurs des attributs x et y

- Une classe dérivée (public) a accès aux attributs et méthodes publiques
  - Ex : ajouter une méthode d'affichage à la classe **`pointColore`**

# Exemple

```
class PointColore : public Point
{
  string couleur;

public:
  void colore(string c){couleur = c;}
  void afficheC();
};

void pointColore::afficheC()
{
  affiche();
  cout << "ma couleur est : " << couleur;
}
```

# Redéfinition de méthode

- Une méthode d'une classe de base peut être redéfinie dans une classe dérivée

```
class PointColore : public Point
{
   string couleur;

public:
   void colore(string c){couleur = c;}
   void affiche();
};

void pointColore::affiche ()
{
   Point::affiche();
   cout << "ma couleur est : " << couleur;
}
```

# Exemple

```
main()
{
  PointColore p;
  p.init(10, 10);
  p.colore("bleu");
  p.affiche();
  p.deplace(15, 30);
  p.affiche();
}
```

# Redéfinition d'attribut

- Un attribut de classe de base peut-être redéfini dans une classe dérivée
- Exemple :

```
class A
{
  string couleur;
…
};
class B : public A
{
  int couleur;
…
};
```

# Constructeurs et destructeurs

- Les constructeurs et destructeurs ne sont pas hérités
- Pour créer un objet de type dérivé :
  - Appel au constructeur de la classe de base
  - Appel au constructeur de la classe dérivée
- Lors de la destruction d'un objet de type dérivé :
  - Appel du destructeur de la classe dérivée
  - Appel du destructeur de la classe de base
- Automatique, pas d'appel explicite

# transmission d'information

- Si le constructeur de la classe de base nécessite des arguments, il faut lui transmettre à partir du constructeur de la classe dérivée
- Il faut alors utiliser une syntaxe précise :
  - On appelle le constructeur de la classe mère

# Exemple classe de base

```
class Point
{
  int x;
  int y;

public:
  Point(int, int);
  void deplace(int, int);
  void affiche();
};
```

# Exemple classe dérivée

```
class PointColore : public Point
{
  string couleur;
public:
  PointColore(int, int, string);
  void colore(string c){couleur = c;}
  void afficheC();
};

PointColore::PointColore(int a, int o , string c):Point(a, o)
{
  couleur = c;
}
```

# Contrôle des accès

- Héritage public :
  - La classe dérivée a accès aux attributs et méthodes publics de la classe de base
  - Les « utilisateurs » de la classe dérivée ont accès à ses attributs et méthodes publics, ainsi que ceux de sa classe de base
- Autorisation d'accès peut-être définie :
  - Lors de la conception de la classe de base
  - Lors de la conception de la classe dérivée

# Niveaux d'accès

- Publique (**`public`**): accessible de n'importe où
- Privé (**`private`**): accessible uniquement à l'intérieur de la classe où il est défini
- Protégé (**`protected`**):
  - Privé pour les utilisateurs de la classe
  - Public pour la classe dérivée elle même

# Exemple

```
class A
{
   string s1;
   protected string s2;
   …
};
class B : public A
{
   void affiche();
…
};
B::affiche()
{
   cout << s1;
   cout << s2;
}
```

# Dérivation privée

- Il est possible d'interdire à un utilisateur d'une classe dérivée l'accès aux membres publics de sa classe de base

```
class classeDerivee : private classeDeBase
    {
       …
    }
```

# Exemple

```
class Point
{…
public:
   Point(int, int);
   void deplace(int, int);
   void affiche();
};
class PointColore : private Point
{…
public:
   PointColore(int,int,string)
   void colore(string);
};
main()
{
   pointColore p (10, 10,"bleu");
   p.deplace(15, 30);
   p.affiche();
}
```

# Dérivation protégée

- Possibilité de dérivation intermédiaire entre dérivation publique et privée
- Les membres publics de la classe de base seront considérés comme protégés lors de dérivation ultérieures

# Surdéfinition d'opérateurs

# Surdéfinition d'opérateurs

- Surdéfinition de méthode (ou fonction) : **attribuer le même nom** à des méthodes (ou fonctions) différentes
- Lors d'un appel le **choix** de la méthode est fait par le **compilateur** en fonction du type et du nombre d'arguments
- Le C++ permet également de surdéfinir des opérateurs existant

# Mécanisme

- Définir une méthode de nom **`operator`** suivie du symbole de l'opérateur
- Elle prend en argument les objets sur lesquels l'opération a lieu et retourne le résultat de l'opération
- Exemple : définir la somme de 2 Points à l'aide de l'opérateur +

# Exemple

```
class Point
{
   int x;
   int y;
public:
   Point(int a=0, int o=0){x=a;y=o;}
   Point operator + (Point);
}
Point Point::operator + (Point a)
{
   Point p;
   p.x = x + a.getX();
   p.y = y + a.getY();
   return p;
}
main()
{
   Point p1(2,2);
   Point p2(5,3);
   Point p3;
   p3 = p1 + p2;
   p3 = p1 + p2 + p3;
}
```

# Polymorphisme

# Conversion

- Soit la classe *B* dérivée de *A*
  - les membres de *A* sont membres de *B*
  - Les membres publics de *A* peuvent être atteints à travers un objet *B*
- *tous les services offerts par un A sont offerts par un B*

=> là où un *A* est prévu, on doit pouvoir mettre un *B*

- la conversion (explicite ou implicite) d'un objet de type *B* vers le type *A* a le statut de conversion standard

# Exemple

```
class Point
{
   int x, y;
   public:
   Point(int, int);
   ...
};
class Pixel : public Point
{ // pixel = point coloré
   string couleur;
   public:
   Pixel(int, int, string);
   ...
};
Main()
{
   Pixel px(1, 2, "rouge");
   Point pt = px;
}
```