Programmation

Programmation orientée objets et évênementielle

FLORIAN LETOMBE
letombe@cril.univ-artois.fr
Département SRC – Bureau 105F

Le Cours

- Unité d'enseignement 3.2
- Module 3.23 : Outils et méthodes informatiques pour le multimédia
- Volume horaire : 30 h (6 h Cours, 6 h TD, 18 h TP)
- Objectifs :
  - acquérir les notions de base de la programmation orientée objets (POO) et de la programmation évênementielle
  - savoir utiliser une hiérarchie d'objets existante
  - mise en place d'animations
  - savoir concevoir et réaliser des interfaces ergonomiques pour des produits multimédia
- Pré-requis : Algorithmique I et II et bases de la programmation orientée objets et événementielle

Contenu

- notions d'objets, de classes, de hiérarchie, d'héritage, ...
- notions d'évênements et mise en place de gestionnaires d'évênements
- dynamiser, animer une interface utilisateur
- réalisation d'animations et de programmes interactifs
- implanter une animation simple, contrôle du tempo, du mouvement, de la vitesse
- mise en place d'interface homme/machine
- élaboration de systèmes intéractifs

Précisions

- Intervenant Cours & TDs :
  - F. Letombe
- Intervenants TPs :
  - J. Hondermarck
  - F. Letombe

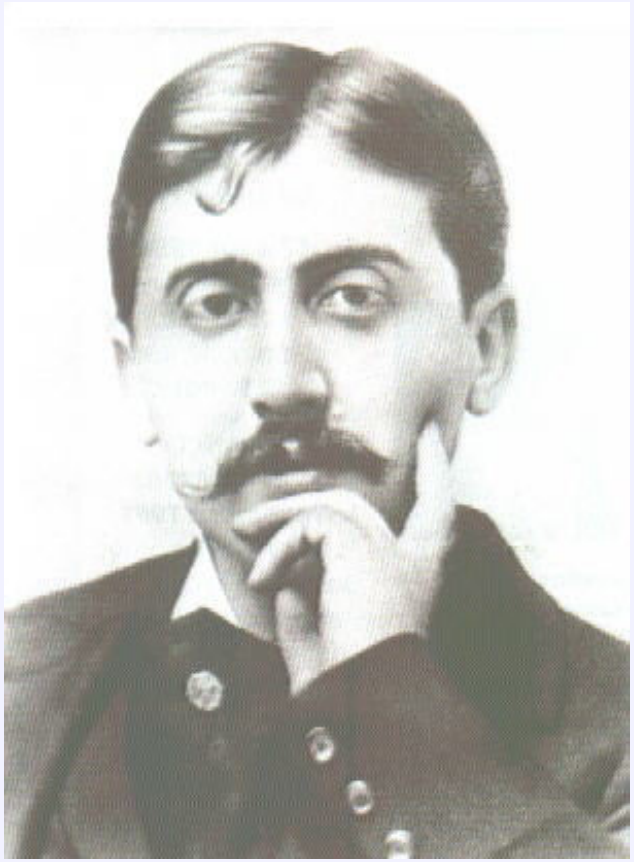
- Modalités de contrôle de connaissances :
  - Théorique :

$$max(\frac{DS + Examen}{2}; Examen)$$

  - Pratique : 1 Examen TP + Projet
- Calcul de la moyenne :

$$Moyenne = \frac{1}{2} Théorique + \frac{1}{2} Pratique$$

Citation



« *On ne peut regretter que ce qu'on se rappelle.* »
Marcel Proust
Extrait d'**Albertine disparue**

Programmation

Définition *(Programmation)*

- Art de debugger un fichier vide
- Le fait d'écrire des programmes ; certains pensent que ce n'est qu'une science, d'autres qu'il s'agit d'un art, ou encore de pure magie noire...
- Passe-temps similaire à celui qui consiste à se frapper la tête contre les murs, mais avec moins d'espoirs de récompense
- La programmation dans le domaine informatique est l'ensemble des activités qui permettent l'écriture des programmes informatiques ; c'est une étape importante de la conception de logiciels

Quelques définitions

Définition *(Programmation impérative)*

- La programmation impérative est un paradigme de programmation qui décrit les opérations en termes d'états du programme et de séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme
- Opposée à la programmation fonctionnelle

Exemple

Quelques langages impératifs

- Fortran
- Pascal
- C

Quelques définitions (suite ...)

Définition *(Programmation fonctionnelle)*

Tous les programmes sont des fonctions (pouvant être constituées de sous-fonctions)

Exemple

Quelques langages fonctionnels

- Lisp
- Scheme
- Haskell

Quelques définitions (suite et fin)

Définition *(Programmation Orientée Objets)*

La programmation orientée objet (POO, également appelée programmation à objets, OOP en Anglais), est une façon d'architecturer une application informatique en regroupant les données et les traitements de ces dernières au sein des mêmes entités, les objets

Exemple

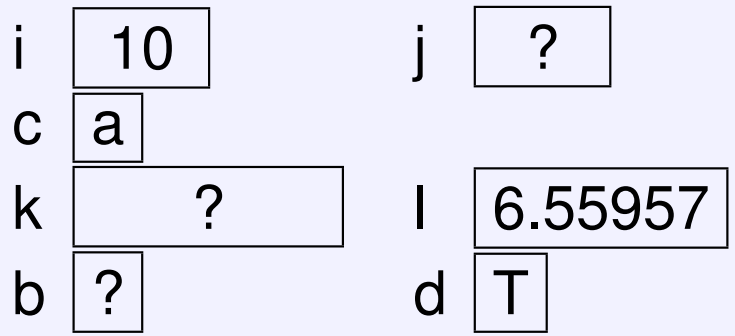
Quelques langages orientés objets

- Eiffel
- Smalltalk
- Java
- C++

Déclaration

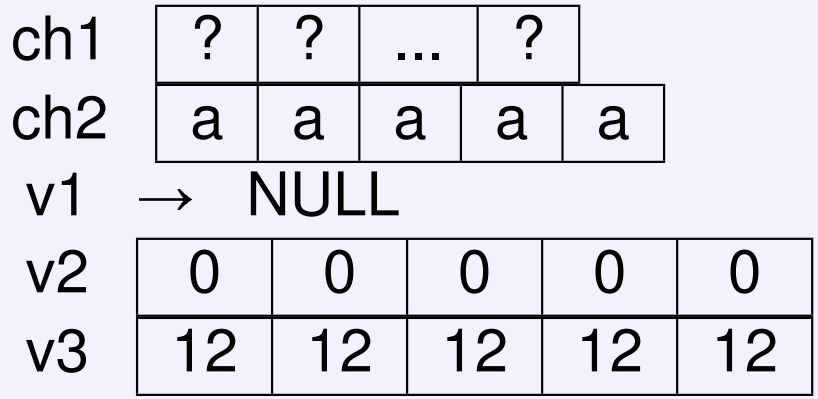
- Types simples :

```
int i = 10, j;
char c = 'a';
double k, l = 6.55957;
bool b, d = true;
```



- Types complexes :

```
string ch1,
       ch2(5, 'a');
vector<int> v1,
            v2(5),
            v3(5, 12);
```



Utilisation

```
string ch(5, 'a');
vector<int> v1(5),
            v2(5, 12);
int i = 10, j;
char c = 'a';
```

ch	a	a	a	a	a	
v1	0	0	0	0	0	
v2	12	12	12	12	12	
i	10			j	?	
c	a					

```
c = 'x';
ch[1] = c;
j = i++;
v1[0] = i;
v1[1] = j - 1;
v2[4] = v1[0];
```

c	x				
ch	a	x	a	a	a
i	11			j	10
v1	11	0	0	0	0
v1	11	9	0	0	0
v2	12	12	12	12	11

Conditionnelle

if (<expression logique>)
    <séquence d'instructions 1>
else <séquence d'instructions 2>

switch (<expression>) {
    case <expr const 1> : <séquence d'instructions 1>
    case <expr const 2> : <séquence d'instructions 2>
    ...
    case <expr const n> : <séquence d'instructions n>
    default : <séquence d'instructions n+1>
}

Itérative

while (<expression logique>)
    <séquence d'instructions>

for (<expression 1> ; <expr logique> ; <expression 2>)
    <séquence d'instructions>

do
        <séquence d'instructions>
while(<expression logique>)

Sur un exemple ...

```
#include <iostream>
#include <vector>

int main () {
    const int MAX = 10;
    vector<int> suite(MAX);
    int elem, i = 0;

    cout << "Suite de nombres positifs, terminée par 0 ?" << endl;
    do {
        cin >> elem;
        if (elem >= 0)
            suite[i++] = elem;
        else cout << "On a dit des nombres positifs" << endl;
    } while (i<MAX && (i==0 || suite[i-1]!=0));
}
```

Déclaration

Spécification

<identificateur> : <types de départ>→ <type de retour>
<liste de paramètres>↦ <élément retourné>

Syntaxe

```
<type de retour> identificateur(<liste de paramètres>) {
    <corps de la fonction>
    return <élément retourné> ;
}
```

Exemple ($f(x) = \frac{1}{x^2-4}$)

Spécification :

$$f\_x2\_4 : \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R} \\ x & \mapsto & \frac{1}{x^2-4} \end{array}$$

Code C++ :

```
double f_x2_4 (double x) {
    double resultat;
    if ((x != 2) && (x != -2))
        resultat = 1/(x*x-4);
    else resultat = 0;
    return resultat;
}
```

Appel

Syntaxe

identificateur(<liste de paramètres>);

Exemple ($f(x) = \frac{1}{x^2-4}$)

L'appel à f_x2_4 pour un x=3 se fait comme suit :

```
f_x2_4(3);
  ↳ 0.2
```

Bien différencier passage de paramètres

- par valeur (cas de base) : paramètres initialisés par une copie des valeurs des paramètres effectifs
- par référence (& entre le type du paramètre formel et son identificateur) : le paramètre formel devient synonyme du paramètre effectif

Citation

« *Le Style et la Structure sont l'essence d'un livre. Les grandes idées ne sont que foutaises.* »
Vladimir Nabokov

La déclaration de type

Syntaxe

typedef <caractéristiques du type> <nom du type>

Exemple

```
typedef vector<char> VectCarac;
int main () {
    VectCarac c(5);
    c[3] = 'a';
}
```

?	?	?	?	?
?	?	?	a	?

Les types énumérés

Syntaxe

enum identificateur { $enum_1, enum_2, \ldots, enum_n$ };

Exemple

```
enum JourDeLaSemaine
{ Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche };
...
JourDeLaSemaine j;
j = Mardi;
```

Les tableaux à deux dimensions

- Nous avons vu (et revu) comment déclarer et utiliser un tableau à une dimension
- Il est parfois nécessaire de manipuler de tableaux à deux dimensions appelés matrices

Exemple

L'instruction **vector**<**vector**<**int**> > t;

définit un tableau t à deux dimensions
Pour l'initialiser, on peut utiliser l'instruction

vector<**vector**<**int**> > t(100, **vector**<**int**>(50,1));

i.e. on initialise chacune des 100 cases de *t* avec un tableau de taille 50 rempli de 1
On accède à une case du tableau par une instruction du type t[i][j], avec i et j des entiers

Entrées/Sorties

- Nous savons déjà écrire sur la sortie standard (e.g. **cout** << n;)
- Utilisation de l'opérateur << à deux opérandes :
  - le « flot de sortie » concerné (ici **cout**)
  - l'expression dont on souhaite écrire la valeur (ici n)
- Nous savons lire sur l'entrée standard (e.g. **cin** >> x;)
- Utilisation de l'opérateur >> à deux opérandes :
  - le « flot d'entrée » concerné (ici **cin**)
  - la variable où on souhaite lire une information (ici x)
- Un flot peut être connecté à un périphérique ou à un fichier
- Par convention, le flot **cout** (resp. **cin**) est connecté à la « sortie standard » (resp. l'« entrée standard »)
- Généralement, l'entrée standard (resp. la sortie standard) correspond par défaut au clavier (resp. à l'écran)
- Il est possible de rediriger ces flots (cf. cours de système)

Les flots

- En dehors des flots prédéfinis (il en existe d'autre que ceux standard), l'utilisateur peut définir lui même d'autres flots qu'il pourra connecter à un fichier de son choix
- Un flot est un objet d'un type particulier :
  - **ostream** pour le flot de sortie
  - **istream** pour le flot d'entrée
- Il est possible d'utiliser l'opérateur << (resp. >>) sur les flots de sortie (resp. d'entrée)
- Il est nécessaire d'incorporer le fichier en-tête **iostream** comme pour les flots prédéfinis

#include <iostream>

Connexion d'un flot de sortie à un fichier

- Utiliser un objet de type **ofstream** (dérivé de ostream)
- Incorporer le fichier en-tête **fstream**, en plus du fichier iostream
- Tout objet de type ofstream nécessite à sa construction deux arguments :
  - le nom du fichier concerné (sous forme de chaîne de caractères)
  - un mode d'ouverture défini par une constante entière (cf. suite)

Exemple

```
#include <iostream>
#include <fstream>
...
ofstream sortie("toto.dat", ios::out);
```

Connexion d'un flot de sortie à un fichier (suite ...)

- L'objet *sortie* est donc associé au fichier nommé "toto.dat", ouvert en écriture
- Une fois construit un objet de type ofstream, l'écriture dans le fichier qui lui est associé peut se faire comme pour n'importe quel flot en faisant appel à toutes les facilités du type ostream
- Lorsque l'on a fini d'écrire dans un fichier, il est nécessaire de fermer le flot à l'aide de l'instruction **close()**

Exemple

Après la déclaration précédente de *sortie*, nous pouvons employer les instructions telles que

```
sortie << 10 << "blabla" << 20 << endl;
sortie.close();
```

Connexion d'un flot d'entrée à un fichier

- Utiliser un objet de type **ifstream** (dérivé de istream)
- Incorporer le fichier en-tête **fstream**, en plus du fichier iostream
- Tout objet de type ifstream nécessite à sa construction deux arguments :
  - le nom du fichier concerné (sous forme de chaîne de caractères)
  - un mode d'ouverture défini par une constante entière (cf. suite)

Exemple

```
#include <iostream>
#include <fstream>
...
ifstream entree("titi.dat", ios::in);
```

Connexion d'un flot d'entrée à un fichier (suite ...)

- L'objet *entree* est donc associé au fichier nommé "titi.dat", ouvert en lecture
- Une fois construit un objet de type ifstream, la lecture dans le fichier qui lui est associé peut se faire comme pour n'importe quel flot en faisant appel à toutes les facilités du type istream
- Lorsque l'on a fini de lire dans un fichier, il est nécessaire de fermer le flot à l'aide de l'instruction **close()**

Exemple

Après la déclaration précédente de *entree*, nous pouvons employer les instructions telles que

```
entree >> element;
entree.close();
```

Les différents modes d'ouverture d'un fichier

- Le mode d'ouverture est défini par un mot d'état dans lequel chaque bit correspond à une signification particulière
- Pour activer plusieurs modes d'ouverture, il suffit de faire appel à l'opérateur | (e.g. ios::out|ios::trunc)

Bit	Action
ios::in	Ouverture en lecture (obligatoire pour ifstream)
ios::out	Ouverture en écriture (obligatoire pour ofstream)
ios::app	Ouverture en ajout de données (écriture en fin de fichier)
ios::trunc	Si le fichier existe, son contenu est perdu
ios::ate	Ouverture en lecture et écriture en fin de fichier
ios::binary	Utilisé seulement dans les systèmes qui distinguent fichiers texte des autres

Principes de base

- Dans un tableau, toutes les composantes doivent être du même type
- Lorsque l'on souhaite regrouper dans un même type des valeurs ayant des types différents, on utilise la notion de structure
- Une structure contiendra un nombre fixe de composants appelés champs de types quelconques
- Syntaxe :

  struct identificateur {
  $champ_1$: $type_1$;
  ...
  $champ_n$: $type_n$;
  };
- où
  - **identificateur** est le nom de la structure
  - **champ$_i$**, $1 \leqslant i \leqslant n$ les champs de la structure

Précisions sur les structures

- Chaque champs est désigné par un sélecteur (identifiant) qui doit être différent des autres :

$$champ_1 \neq \cdots \neq champ_n$$

Exemple

```
const int NBEQUIP = 20;

struct match {
    int gagne, nul, perdu;
    int marques, encaisses;
};
```

```
struct club {
    string nom;
    match joues;
    int places;
};

typedef vector<club> division;
division d(NBEQUIP);
```

Opérations sur les structures

- Affectation (e.g. d[i] = d[j];)
- Utilisation d'une composante
  - Pour désigner un champ d'une variable de type enregistrement, on indique le nom de cette variable suivi d'un point puis du sélecteur de ce champ

Exemple

d[i].nom (de type **string**)
d[i].joues.gagne (de type **int**)

  - on peut alors effectuer toutes les opérations définies sur le type de la composante
- Fonctions membres : il est possible de déclarer des fonctions propres à une structure

Citation

« Objets inanimés, avez-vous donc une âme qui s'attache à notre âme et la force d'aimer ? »

Alphonse de Lamartine

Extrait de **Harmonies poétiques et religieuses**

Intérêts de la POO

- Le code est plus sûr
- Les programmes sont plus clairs
- La maintenance des applications est facilitée
- Le code est facilement réutilisable
- Il est facile de créer de nouveaux algorithmes légèrement différents par clonage d'un algorithme existant
- Il est facile de faire évoluer des programmes

Les caractéristiques de l'Objet

La 1ère étape consiste à déterminer

- les entités que l'on souhaite manipuler
- la description générique qui relie toutes ses entités

Exemple

- Prenons ces informations
  - 14 Juillet 1789 (Prise de la Bastille)
  - 11 Novembre 1918 (Armistice)
  - 25 Septembre 2006
- Ce sont des dates
- Chaque date se caractérise par
  - un *jour*
  - un *mois*
  - une *année*

Les dates

Date
- Jour
- Mois
- Annee

Prise de la Bastille
- 14
- Juillet
- 1789

Armistice
- 11
- Novembre
- 1918

Une Date
- 25
- Septembre
- 2006

Les classes

- Le *Jour*, le *Mois* et l'*Annee* sont les attributs d'une Date
- Cet ensemble d'attributs est appelé une Classe

Les objets

- Le *14 Juillet 1789* et le *11 Novembre 1918* sont chacune des instances de la classe Date
- Chacune de ces dates est appelée un Objet

Autre exemple : les planètes

- Saturne : planète gazeuse ; son diamètre est de 120.536 km ; elle est à une distance moyenne de 1.426.725.400 km du Soleil
- Mars : planète rocheuse ; son diamètre est de 6794 km ; elle est à une distance moyenne de 227.936.640 km du Soleil
- Jupiter : planète gazeuse ; son diamètre est de 142.984 km ; elle est à une distance moyenne de 779 millions de km du Soleil
- Terre : planète rocheuse ; son diamètre est de 12.756,28 km ; elle est à une distance moyenne de 150.000.000 km du Soleil

Les planètes (suite ...)

Ces planètes ont en commun

- Le Type : Rocheuse, Gazeuse
- La Distance au Soleil : 227936640, 779 millions, 1426725400, 150000000
- Le Diamètre : 12756.28, 120536, 142984, 6794

La classe Planete

Planete
- Type
- DistanceAuSoleil
- Diametre

Terre
- Rocheuse
- 150000000
- 12756,28

Saturne
- Gazeuse
- 1426725400
- 120536

Jupiter
- Gazeuse
- 779 millions
- 142984

Mars
- Rocheuse
- 227936640
- 6794

Encore une exemple : des étudiants

- Pierre, 19 ans, étudiant en SRC ; groupe 1, sous-groupe 2
- Nathalie, étudiante en informatique, a 18 ans ; elle est dans le sous-groupe 1 du groupe 2
- Paul, un garçon de 21 ans, est en GEA ; il est dans le groupe 3, sous-groupe 2

Les étudiants (suite ...)

Ici, un étudiant se caractérise par

- age
- département
- groupe
- sous-groupe
- sexe

Les étudiants – 1ère possibilité

Etudiant
- Age
- Departement
- Groupe
- SousGroupe
- Sexe

Pierre
- 19
- SRC
- 1
- 2
- Homme

Paul
- 21
- DEA
- 3
- 2
- Homme

Nathalie
- 18
- INFO
- 2
- 1
- Femme

Les étudiants – 2ème possibilité

Etudiant
- Prenom
- Age
- Departement
- Groupe
- SousGroupe
- Sexe

Etudiant1
- Pierre
- 19
- SRC
- 1
- 2
- Homme

Etudiant2
- Paul
- 21
- DEA
- 3
- 2
- Homme

Etudiant3
- Nathalie
- 18
- INFO
- 2
- 1
- Femme

Définir une classe en C++

```
class CDate
{
  int Jour;
  int Mois;
  int Annee;
};
```

```
class NomClasse
{
   typeAttribut1 nomAttribut1;
   typeAttribut2 nomAttribut2;
   ...;
   typeAttributn nomAttributn;
};
```

Définir un objet en C++

Comment créer un objet priseBastille de type CDate ?

$\Rightarrow$ La classe CDate est un type comme un autre (int, char...)

Exemple

```
CDate priseBastille;
```

Initialiser un Objet

Pour l'instant, notre objet priseBastille n'est pas encore le 14 Juillet 1789

$\Rightarrow$ Spécifier chaque attribut à la main

Exemple

```
priseBastille.Jour = 14;
priseBastille.Mois = 07;
priseBastille.Annee = 1789;
```

Initialiser un Objet (suite ...)

Autre exemple : la date armistice que l'on souhaite initialiser au 11 Novembre 1918

Exemple

```
CDate armistice;
armistice.Jour=11;
armistice.Mois=11;
armistice.Annee=1918;
```

Spécificité de la POO

Spécifier des comportements génériques communs à tous les objets d'une classe

Exemple

- s'initialiser
- s'afficher
- vérifier si c'est contemporain

Initialiser

```
class CDate
{
  int Jour;
  int Mois;
  int Annee;
```

```
  void InitDate (int jour,
                 int mois,
                 int annee)
  {
    Jour = jour;
    Mois = mois;
    Annee = annee;
  }
};
```

Afficher

```
class CDate
{
  int Jour;
  int Mois;
  int Annee;
  void InitDate (int jour,
                 int mois,
                 int annee)
  { ... }
```

```
  void Affiche ()
  {
    cout << Jour <<
      "/" << Mois <<;
      "/" << Annee;
  }
};
```

Contemporain

```
class CDate
{
  int Jour;
  int Mois;
  int Annee;
  void InitDate (int jour,
                 int mois,
                 int annee)
  { ... }
  void Affiche ()
  { ... }
```

```
  bool Contemporain ()
  {
    if (Annee >= 2000)
      return true;
    return false;
  }
};
```

Exemple d'utilisation

```
CDate priseBastille;
priseBastille.InitDate(14,07,1789);
CDate armistice; armistice.InitDate(11,11,1918);

cout << "La date "; priseBastille.Affiche();
cout << " est elle contemporaine ? "
     << priseBastille.Contemporaine() << endl;
          ↳ La date 14/07/1789 est elle contemporaine ? no

cout << "La date "; armistice.Affiche();
cout << " est elle contemporaine ? "
     << armistice.Contemporaine() << endl;
          ↳ La date 11/11/1918 est elle contemporaine ? no
```

Initialiser un Objet (bis)

- Il existe une autre méthode pour initialiser les attributs d'un objet
  $\Rightarrow$ utiliser les constructeurs
- Un constructeur est une « procédure » appelée automatiquement qui permet de spécifier chacun des attributs de l'objet en fonction de ses paramètres
- Un constructeur ne possède aucun type de retour, même pas **void**

Constructeur par défaut

```
class CDate
{
  int Jour;
  int Mois;
  int Annee;

  /* constructeur par défaut qui initialise
  l'objet date au 01 Janvier 2006 */
  CDate ()
  {
    Jour = 1; Mois = 1; Annee = 2006;
  }
};
```

Créer mon objet

```
{
   CDate uneDate;

   cout << "La date ";
   uneDate.Affiche();
   cout << " est elle contemporaine ? "
        << uneDate.Contemporaine();
              ↳ La date 01/01/2006 est elle contemporaine ? yes
}
```

- Le programme appelle automatiquement le constructeur (aucun paramètre) ⇒ l'objet uneDate est initialisé au 01 Janvier 2005
- Appel ensuite la fonction Affiche() pour afficher la date puis la fonction Contemporaine() pour savoir si elle est contemporaine de notre siècle

D'autres constructeurs possibles

```
CDate (int jour, int mois,
       int annee)
{
  Jour = jour;
  Mois = mois;
  Annee = annee;
}

CDate (int jour, int mois)
{
  Jour = jour;
  Mois = mois;
  Annee = 2006;
}
```

Il est également possible de donner des valeurs par défaut aux paramètres du constructeur (comme pour les fonctions)

```
CDate (int jour=1, int mois=1,
       int annee=2006)
{
  Jour = jour;
  Mois = mois;
  Annee = annee;
}
```

Usage

Exemple

```
CDate uneDate;
CDate priseBastille (14,07,1789);
CDate dateCetteAnnee(07,03);
CDate enJanvierCetteAnnee(16);

uneDate.Affiche();
        ↳ 01/01/2006
priseBastille.Affiche();
        ↳ 14/07/1789
dateCetteAnnee.Affiche();
        ↳ 07/03/2006
enJanvierCetteAnnee.Affiche();
        ↳ 16/01/2006
```

Écriture d'un constructeur

```
class MaClass
{
  typeA1 Attribut1;
  typeA2 Attribut2;
  ...
  MaClass (typeP1 parametre1, typeP2 parametre2, ...)
  {
    ...
  }
};
```

Sécurité

Comment empêcher un programmeur de définir la date 38/14/-362 ?

Exemple

```
CDate dateImpossible;
dateImpossible.Jour = 38;
dateImpossible.Mois = 14;
dateImpossible.Annee = -362;
dateImpossible.Affiche();
```

Mots clé

Le C++ met à votre disposition 3 mots clés

- public:
- private:
- protected:

public:

- Tous les attributs ou fonctions situés sous le mot clé public: sont accessibles en dehors de l'objet ou depuis n'importe quelle fonction de la classe
- Remarque : les fonctions précédentes ne fonctionnent qu'en ajoutant le mot clé public:

Exemple

```
class CDate {
  public:
    int Jour, Mois, Annee;
    void Affiche() {
      cout << Jour << "/"
           << Mois << "/"
           << Annee;
    }
};
```

```
{
  CDate uneDate;
  uneDate.Jour=06;
  uneDate.Mois=10;
  uneDate.Annee=1289;
  uneDate.Affiche();
      ↳ 06/10/1289
}
```

private:

- Tout attribut ou fonction situé sous le mot clé private: n'est accessible que depuis les fonctions de l'objet
- Comportement par défaut si aucun mot clé n'est spécifié

Exemple

```
class MaClasse {
  public:
    int NbreAccessible;
  private:
    int NbrePrive;
    void Affiche() {
      cout << NbrePrive << endl;
    }
  public:
    void LanceAffiche() {
      Affiche();
    }
};
```

```
{
  MaClasse objet;

  objet.NbreAccessible=3;
      ↳ OK
  objet.NbrePrive=7;
      ↳ Impossible
  objet.Affiche();
      ↳ Impossible
  objet.LanceAffiche();
      ↳ OK
}
```

protected:

- Utilisé de la même manière que les deux autres mots clé
- Définit un statut intermédiaire entre public et privé
- N'intervient que dans le cas d'héritage
- Nous reverrons cela au prochain chapitre

Sécurité

Problème : Comment empêcher l'accès à certains attributs tout en donnant la possibilité de les modifier/consulter ?

⇒ Solution : Définir des accesseurs

Accesseur en lecture

Un accesseur en lecture est une fonction publique de la classe permettant de récupérer le contenu d'un de ses attributs

Exemple

```
int GetMois()
{
    return Mois;
}
```

Accesseur en écriture

Un accesseur en écriture est une procédure publique de la classe permettant de modifier le contenu d'un de ses attributs

Exemple

```
void SetMois(int mois)
{
    Mois = mois;
}
```

Protéger les attributs de la classe CDate

```
class CDate {
  private:
    int Jour;
    int Mois;
    int Annee;

  public:
    int GetMois()
    {
      return Mois;
    }
                    ↪
    int SetMois()
    {
      if (mois < 1 || mois > 12)
        mois = 1;
      Mois = mois;
    }

    int GetJour() { ... }
    void SetJour(int jour) { ... }

    int GetAnnee() { ... }
    void SetAnnee(int jour) { ... }
};
```

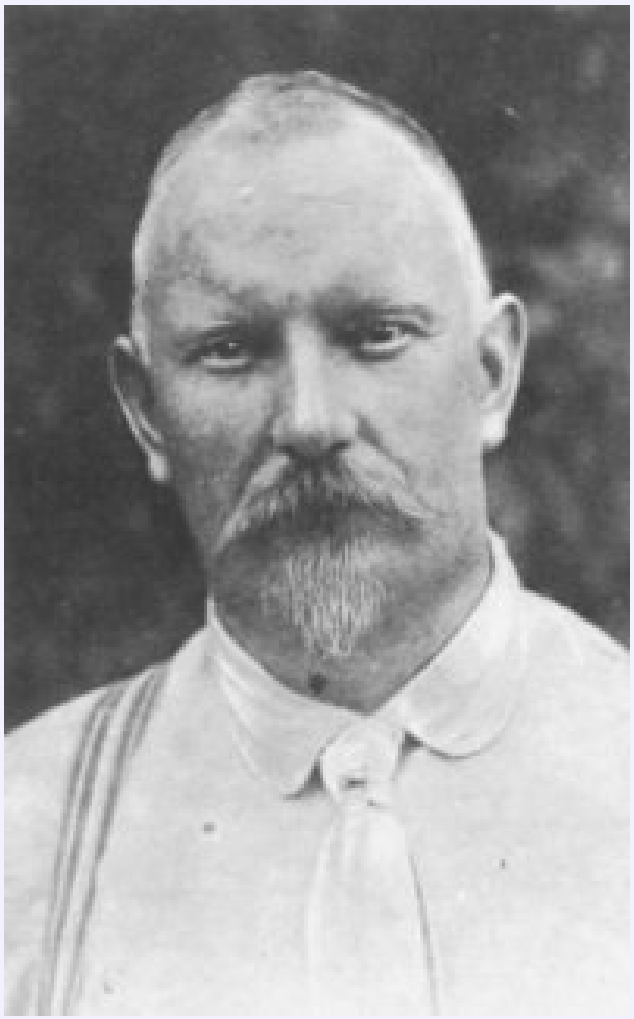
Reprenons notre exemple

Essayons de définir la date 38/14/-362

Exemple

```
{
  CDate dateImpossible;
  dateImpossible.SetJour(38);
                ↳ Jour = 1
  dateImpossible.SetMois(14);
                ↳ Mois = 1
  dateImpossible.SetAnnee(-362);
                ↳ Annee = 2006
  dateImpossible.Affiche();
                ↳ 01/01/2006
}
```

Citation



« Héritage. La mort nous prend un parent, mais elle le paie, et il ne nous faut pas beaucoup d'argent pour qu'elle se fasse pardonner. »
Marcel Proust
Extrait de **son Journal**

Exemple

- Dans l'avion Paris-Berlin, on peut trouver les personnes suivantes
  - Pierre : Pilote
    Paul : coPilote
  - Anne : Hôtesse n°1
    Nathalie : Hôtesse n°2
  - Laure : Passager siège n°1
    Frédéric : Passager siège n°2
    etc ...
- Chacune de ces personnes peut être représentée sous la forme d'objet
- Chacun de ces objets appartient à une de ces catégories
  - Pilote
  - Hôtesse
  - Passager

Organigramme

CPilote
• Prénom • N° de tel • Adresse • Âge • Nbre d'heures de vol
• les constructeurs • les accesseurs • **bool** EstFatigue()

CHotesse
• Prénom • N° de tel • Adresse • Âge • Nbre de langues
• les constructeurs • les accesseurs • **bool** ParlePlus3Langues()

CPassager
• Prénom • N° de tel • Adresse • Âge • N° de siège
• les constructeurs • les accesseurs • **bool** Chanceux()

Classe CPilote

```
class CPilote {
  private:
    string Prenom;
    int NTel;
    String Adresse;
    int Age;
    int NbHeureVol;

  public:
    CPilote() { ... }
                    ↪
    string GetPrenom()
    { ... }
    void SetPrenom(string prenom)
    { ... }
    ...
    bool EstFatigue()
    {
      return (GetNbHeureVol()>8);
    }
};
```

Classe CHotesse

```
class CHotesse {
  private:
    string Prenom;
    int NTel;
    String Adresse;
    int Age;
    int NbLangues;

  public:
    CHotesse() { ... }
    string GetPrenom()
    { ... }
    void SetPrenom(string prenom)
    { ... }
    ...
    bool ParlePlus3Langues()
    {
      return (GetNbLangues()>3);
    }
};
```

Classe CPassager

```
class CPassager {
  private:
    string Prenom;
    int NTel;
    String Adresse;
    int Age;
    int NumSiege;

  public:
    CPassager() { ... }
    string GetPrenom()
    { ... }
    void SetPrenom(string prenom)
    { ... }
    ...
    bool Chanceux()
    {
      return (GetNumSiege()!=13);
    }
};
```

Programme principal

Je peux maintenant créer mes objets

Exemple

```
{
  CPilote pilote("Pierre",...,5);
  CPilote coPilote("Paul",...,3);
  CHotesse hotesse1("Anne",...,4);
  CHotesse hotesse2("Nathalie",...,2);
  CPassager passager1("Laure",...,24);
  CPassager passager2("Frédéric",...,17);
  cout << pilote.GetPrenom() << endl;
  cout << passager2.Dort() << endl;
}
```

Ouf, enfin fini !!!

N'aurait on pas pu gagner du temps en remarquant et en exploitant que ces 3 classes avaient des attributs et des méthodes communes ?

Organigramme

CPilote
• Prénom • N° de tel • Adresse • Âge • Nbre d'heures de vol
• les constructeurs • les accesseurs • **bool** EstFatigue()

CHotesse
• Prénom • N° de tel • Adresse • Âge • Nbre de langues
• les constructeurs • les accesseurs • **bool** ParlePlus3Langues()

CPassager
• Prénom • N° de tel • Adresse • Âge • N° de siège
• les constructeurs • les accesseurs • **bool** Chanceux()

En bleu, les caractéristiques communes à nos trois classes

CPersonne

Ces caractéristiques communes peuvent représenter une personne

CPersonne
• Prénom • N° de tel • Adresse • Âge
• les constructeurs • les accesseurs

Classe CPersonne

```
class CPersonne {
  private:
    string Prenom;
    int NTel;
    String Adresse;
    int Age;

  public:
    CPersonne() { . . . }
                    ↪
    string GetPrenom() { . . . }
    void SetPrenom(string prenom)
    { . . . }
    int GetNTel() { . . . }
    void SetNTel(int ntel)
    { . . . }
    string GetAdresse() { . . . }
    void SetAdresse(string adresse)
    { . . . }
    int GetAge() { . . . }
    void SetAge(int age)
    { . . . }
};
```

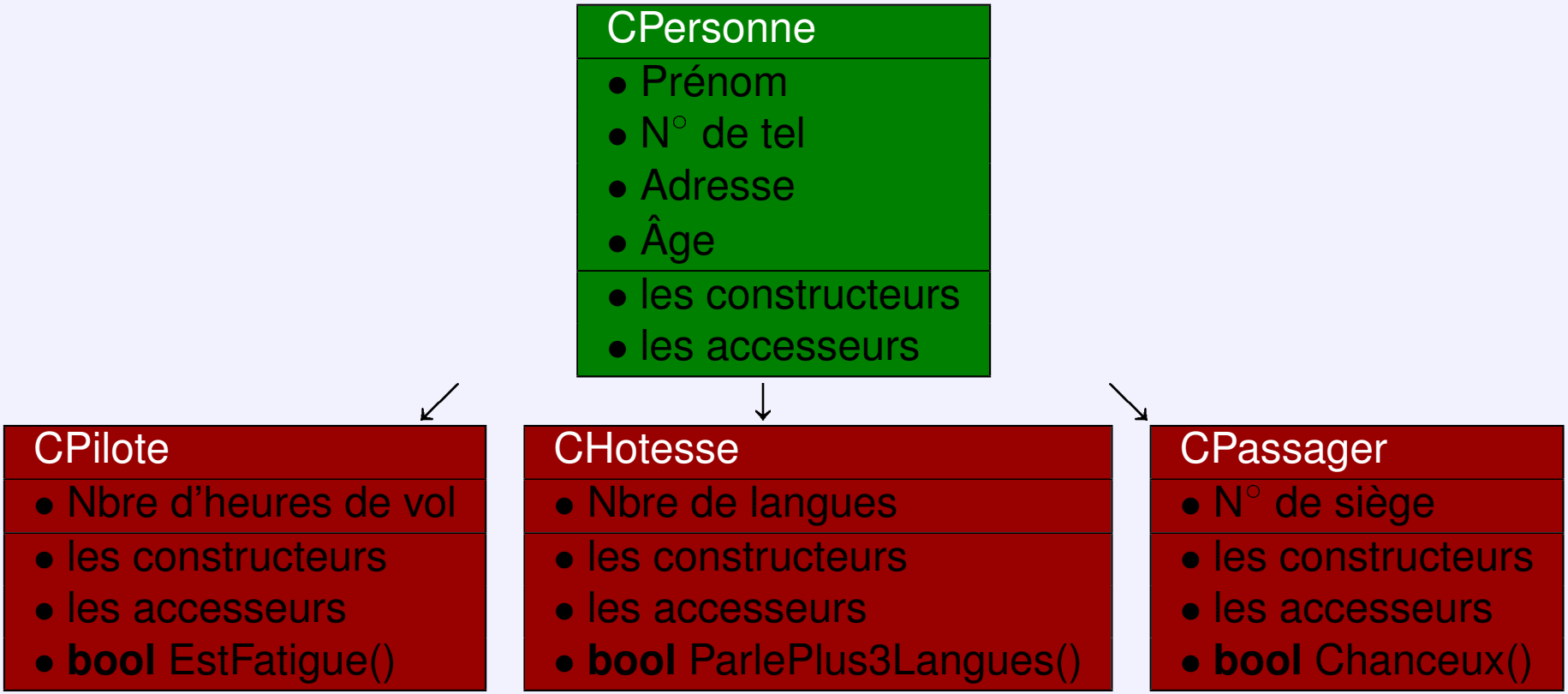
Factorisation

Maintenant, on peut dire que

- un pilote est une personne
- une hôtesse est aussi une personne
- un passager est aussi une personne

Donc un pilote, une hôtesse et un passager possèdent aussi un prénom, un n° de téléphone, une adresse, et un age

Organigramme



Classe CPilote

```
class CPilote {
  private:
    int NbHeureVol;

  public:
    CPilote() { ... }
                    ↪
    int GetNbHeureVol()
    {
      return NbHeureVol;
    }
    void SetNbHeureVol(int nbhv)
    {
      NbHeureVol = nbhv;
    }
    bool EstFatigue()
    {
      return (GetNbHeureVol()>8);
    }
};
```

Classe CHotesse

```
class CHotesse {
  private:
    int NbLangues;

  public:
    CHotesse() { ... }
                    ↪
    int GetNbLangues()
    {
      return NbLangues;
    }
    void SetNbLangues(int nbl)
    {
      NbLangues = nbl;
    }
    bool ParlePlus3Langues()
    {
      return (GetNbLangues()>3);
    }
};
```

Classe CPassager

```
class CPassager {
  private:
    int NumSiege;

  public:
    CPassager() { . . . }
                    ↪
    int GetNumSiege()
    {
      return NumSiege;
    }
    void SetNumSiege(int nums)
    {
      NumSiege = numsI;
    }
    bool Chanceux()
    {
      return (GetNumSiege()!=13);
    }
};
```

Héritage

La classe Fille hérite des attributs et des méthodes de la classe Mère

```
class Mère
{



};
```

```
class Fille : public Mère
{



};
```

Programme principal

À l'usage, rien n'a changé

Exemple

```
{
  CPilote pilote("Pierre",...,5);
  CPilote coPilote("Paul",...,3);
  CHotesse hotesse1("Anne",...,4);
  CHotesse hotesse2("Nathalie",...,2);
  CPassager passager1("Laure",...,24);
  CPassager passager2("Frédéric",...,17);
  cout << pilote.GetPrenom() << endl;
  cout << passager2.Dort() << endl;
}
```

Constructeur de CPilote

```
CPilote(string prenom, int nTel, string adresse, int age,
        int nbHeure)
{
  SetPrenom(prenom);
  SetNumTel(nTel);
  SetAdresse(adresse);
  SetAge(age);
  SetNbHeureVol(nbHeure);
}
```

Les constructeurs de CHotesse et de CPersonne diffèrent de celui de CPilote sur le dernier paramètre

Constructeur de CPersonne

Pour CPersonne, on n'a besoin d'initialiser que ses attributs

```
CPersonne(string prenom, int nTel, string adresse, int age)
{
  SetPrenom(prenom);
  SetNumTel(nTel);
  SetAdresse(adresse);
  SetAge(age);
}
```

Héritage du constructeur

J'utilise le constructeur de CPersonne pour m'aider à « construire » CPilote

```
CPilote(string prenom, int nTel, string adresse, int age,
        int nbHeure)
        : CPersonne(prenom,nTel,adresse,age)
{
  SetNbHeureVol(nbHeure);
}
```

Exemple *(Usage)*

```
CPilote pilote1("Pierre",0321175413,"home",54,9);
```

Héritage

La classe Fille hérite de la classe Mère

```
class Mère
{
  Mère(type_1 param_1,
       ...,
       type_n param_n)
  {
    ...
  }
};
```

```
class Fille : public Mère
{
  Fille(type_1 param_1, ...,
        type_n param_n, ...)
        : Mère(param_1, ..., param_n)
  {
    ...
  }
};
```

Héritage du constructeur

```
CHotesse(string prenom, int nTel, string adresse, int age,
                int nbLangues) : CPersonne(prenom,nTel,adresse,age)
{
  SetNbLangues(nbLangues);
}
```

```
CPassager(string prenom, int nTel, string adresse, int age,
                int numSiege) : CPersonne(prenom,nTel,adresse,age)
{
  SetNumSiege(numSiege);
}
```

Public/Private/Protected

- Tous les attributs ou fonctions situés sous le mot clé public: sont accessibles en dehors de l'objet ou depuis n'importe quelle fonction de la classe
- Tout attribut ou fonction situé sous le mot clé private: n'est accessible que depuis les fonctions de l'objet
- Tous les attributs ou fonctions situés sous le mot clé protected: ne sont accessibles que depuis les méthodes de la classe mère et de ses filles

Remarque : c'est une sorte de private: étendue aux classes filles

Classe CPersonne

Supposons qu'on ajoute un protected à la classe CPersonne

```
class CPersonne {
  private:
    string Prenom;
    int NTel;
    String Adresse;
    int Age;

  public:
    CPersonne() { . . . }
                    ↪
                                  protected:
                                    string GetPrenom() { . . . }
                                    void SetPrenom(string prenom)
                                    { . . . }
                                    int GetNTel() { . . . }
                                    void SetNTel(int ntel)
                                    { . . . }
                                    string GetAdresse() { . . . }
                                    void SetAdresse(string adresse)
                                    { . . . }
                                    int GetAge() { . . . }
                                    void SetAge(int age)
                                    { . . . }
                                };
```

Accessibilité

Les méthodes de la classe CPersonne situées après le mot clé « protected »

- GetPrenom, SetPrenom
- GetNTel, SetNTel
- GetAdresse, SetAdresse
- GetAge, SetAge

ne sont accessibles que dans

- CPersonne, CPilote, CHotesse, CPassager
- Toutes classes qui héritent de CPilote, CHotesse et CPassager
- et ainsi de suite....