L’orienté objet en C++

Contenu de la première partie de l'UE

- Les concepts de base orientés objet et héritage simple
- Héritage multiple et droits d'accès
- Généricité et STL
- Gestion des exceptions et ++

Les concepts de base orientés objet en C++

Brève histoire de C++

- B. Stroustrup s'inspirant de Simula 67 ...
- C car efficace et très répandu
- *C with classes* (1979)
- C++ (1983)
- normalisation ANSI/ISO en 1998
- évolutions récentes de la norme : C++ 2011, C++ 2014
- un langage complexe ... ici dans un style objet

Classes

- Réifier les concepts
  - d'un domaine (*CompteBancaire*)
  - techniques (*Liste*, *Fenêtre*)
- Classe
  - structure : attributs
  - comportement : opérations

Représentation de la poésie : Vers et Strophes

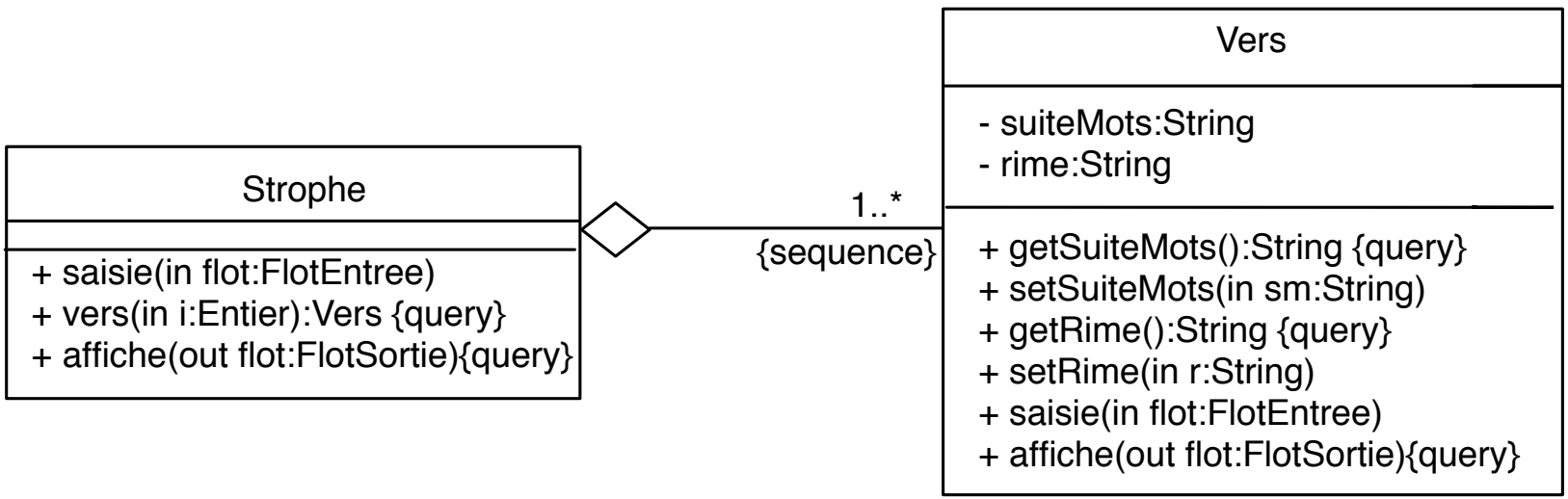


Figure : Eléments de la représentation des strophes

Eléments de la notation UML qui seront traduits

- `query` : l'opération ne modifie pas l'objet receveur du message,
- association de type agrégation (losange évidé) : *se compose de*. L'agrégation autorise le partage, c'est-à-dire qu'un vers peut apparaître dans plusieurs strophes,
- multiplicité 1..* contre la classe `Vers` : une strophe se compose d'(au moins) un ou plusieurs vers,
- `{sequence}` : les vers sont ordonnés ; un même vers peut figurer plusieurs fois dans la même strophe,
- protection (visibilité) : attributs privés (symbole -), opérations publiques (symbole +).

Description recommandée de la classe en deux fichiers

- L'interface (fichier *header*, muni d'une extension `.h`)
- L'implémentation (fichier d'extension `.cc` ou `.cpp`)

L'interface (fichier .h)

Contenu :

- attributs (non initialisés par défaut)
- déclarations (signatures ou entêtes) des méthodes
- déclaration de fonctions associées à la classe (souvent des surcharges d'opérateurs, le moins possible)

Macros classiques dans l'entête :

- `#define vers_h` : définition de la variable
- `#ifndef vers_h` ... `#endif` : encadre le texte du programme pour éviter une double analyse

L'interface de la classe Vers (fichier Vers.h)

Pour la POO, toutes les méthodes sont `virtual` , y compris le destructeur.

Au moins une méthode `virtual` est présente. Seuls les constructeurs n'ont pas de `virtual`.

```
#ifndef vers_h  #define vers_h
class Vers{
private:
 string suiteMots;     // suiteMots, attribut de type string
 string rime;              // rime, attribut de type string
public:
 Vers(); explicit Vers(string s); Vers(string s, string r);
 virtual ~Vers();
 virtual string getSuiteMots()const;
 virtual void setSuiteMots(string sm);
 virtual string getRime()const; virtual void setRime(string r);
 virtual void saisie(istream& is);
 virtual void affiche(ostream& os)const;
};
#endif
```

Pour la POO - utilisation de types pointeurs pour manipuler les objets

Dans la Strophe, on désire déclarer une suite de vers.

`Vers* pv` (à éviter pour la POO)
= `pv` adresse d'un emplacement mémoire contenant une instance de vers
= `*pv` est une instance de vers, avec $*$ opérateur de déréférencement
= ou tableau de vers d'une taille qui n'est pas encore connue et qui sera alloué dynamiquement

`Vers** pv` (solution adoptée pour la POO)
= tableau (ou pointeur) de pointeurs vers des instances de vers
= l'agrégation spécifiée en UML qui indique qu'une strophe se compose de vers

L'interface de la classe Strophe

```
#ifndef Strophe_h
#define Strophe_h
class Strophe
{
private:
  Vers ** suiteVers;
      // attribut de type tableau de pointeurs vers des Vers
      // implémente l'agrégation "une strophe se compose de vers"
  int nbVers;
public:
  Strophe();
  virtual ~Strophe();
  virtual void saisie(istream& is);
  virtual Vers* vers(int i)const;
  virtual void affiche(ostream& os)const;
};
#endif
```

L'implémentation d'une classe (fichier .cc)

Contenu :

- corps des méthodes
- définition de certains attributs (attributs de classes, `static`)
- définition de fonctions associées à la classe (souvent des opérateurs, le moins possible)

Entête classique :

- inclusion de fichiers de déclaration de la librairie standard
  `#include<string>`
- directive pour simplifier le nommage
  `using namespace std;`
- inclusion de fichiers utilisateur
  `#include"Vers.h"`

L'implémentation de la classe Vers (fichier Vers.cc)

```
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"

Vers::Vers(){}
Vers::Vers(string sm){suiteMots=sm;}
Vers::Vers(string sm, string r){suiteMots=sm;rime=r;}
Vers::~Vers(){}

.........
```

L'implémentation de la classe Vers (fichier Vers.cc)

const : l'objet receveur this ne sera pas modifié (contrôlé par le compilateur)

```
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"

......

string Vers::getSuiteMots()const
{return suiteMots;}
void Vers::setSuiteMots(string sm)
{suiteMots=sm;}
string Vers::getRime()const
{return rime;}
void Vers::setRime(string r)
{rime=r;}
```

L'implémentation de la classe Vers (fichier Vers.cc)

```
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"

.........

void Vers::saisie(istream& is)
{cout <<"vers puis rime" <<endl;is>>suiteMots>>rime;}

void Vers::affiche(ostream& os)const
{os<<"<<"<<suiteMots<<">>";}
```

L'implémentation de la classe Strophe (fichier Strophe.cc)

```
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"
#include"Strophe.h"

Strophe::Strophe(){suiteVers=NULL; nbVers=0;}
Strophe::~Strophe(){if (suiteVers) delete[] suiteVers;}
                  // on ne détruit que le tableau, pas les vers.

Vers* Strophe::vers(int i)const
{if (i>=0 && i<nbVers) return suiteVers[i]; else return NULL;}

........
```

Figure : Strophe.cc

L'implémentation de la classe Strophe (fichier Strophe.cc)

```
.........
#include"Vers.h"
#include"Strophe.h"
........
void Strophe::saisie(istream& is){
  if (suiteVers) delete[] suiteVers;
  cout << "Entrer le nombre de vers : " << endl;
  is>>nbVers; suiteVers = new Vers*[nbVers];
  for (int i=0; i<nbVers; i++)
    {Vers *v=new Vers(); v->saisie(is); suiteVers[i]=v;}
}
void Strophe::affiche(ostream& os)const{
  for (int i=0; i<nbVers; i++)
    {suiteVers[i]->affiche(os); os << endl;}
}
```

Figure : Strophe.cc

Encapsulation - Objectifs principaux :

- cacher les détails d'implémentation (pour pouvoir en changer sans influer sur les utilisateurs de la classe)
- pour contrôler les accès à l'état (attributs) des objets et le garder cohérent

En C++ une première approximation :

- `public:` accès depuis n'importe quelle autre partie du code
- `private:` accès depuis des parties de code de la même classe
- `protected:` voir le cours sur l'héritage

classes versus *struct*

- dans les `class` tout est privé par défaut
- dans les `struct` tout est public par défaut

Visibilité dans Strophe.h

```
class Strophe
{
private:
  Vers ** suiteVers;
      // suiteVers pourrait être implémentée autrement
     // on contrôle l'accès par les méthodes
  int nbVers;
    // on pourrait faire des méthodes d'accès en lecture
    // mais surtout pas de modification
public:
  ......
  virtual void saisie(istream& is);
  virtual Vers* vers(int i)const;
  virtual void affiche(ostream& os)const;
};
```

Figure : Strophe.h

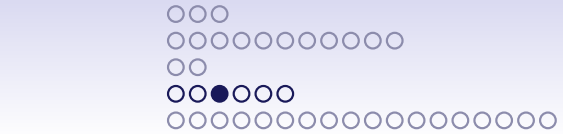
Création des objets

- créer un objet = instancier une classe, pour la POO on réalise une allocation dynamique (avec un `new`). Exception : `string`
- demande de gérer la mémoire (pas de ramasse-miettes comme en Java)
  - avantageux pour les logiciels où la maîtrise de la gestion mémoire est nécessaire (embarqué, scientifique, temps réel, soit très coûteux en espace, soit dans des contextes où on a peu de mémoire)
  - désavantageux et dangereux pour les logiciels complexes et en constante évolution, où le programmeur doit être déchargé des problèmes de bas niveau
- Des ramasse-miettes existent comme le *garbage collector* de Boehm (package Debian libgc)

Création des objets

Trois modes d'allocation :

- statique
- automatique
- dynamique, recommandé pour la POO (avec `new`)



L'allocation statique

- L'objet est créé lorsque le flot de contrôle atteint l'instruction de création et pour toute la durée du programme.
- exemple : attributs `static`.

L'allocation automatique

- l'objet est créé localement dans un bloc
  - dans une déclaration de classe
  - dans une méthode (variable locale, paramètre)
- durée de vie : depuis l'instruction de création jusqu'à la fin du bloc

```
{ // début du bloc
. . .
Vers v; // v est une variable de type Vers -- création de v
Strophe s; // s est une variable de type Strophe -- création de s
. . .
} // fin du bloc, disparition de v et s
```

Figure : Exemple d'allocation automatique (dans la pile)

L'allocation dynamique

- L'objet est référencé par un pointeur
- créé grâce à l'opérateur `new`
- détruit par l'usage de l'opérateur `delete`

```
Vers *pv=new Vers(); // variable pointeur sur Vers
Strophe *ps = new Strophe(); // variable de type pointeur sur Strophe
 . . .
delete pv; // destruction de pv
delete ps; // destruction de ps
```

Figure : Allocation dynamique (dans le tas)

L'allocation dynamique

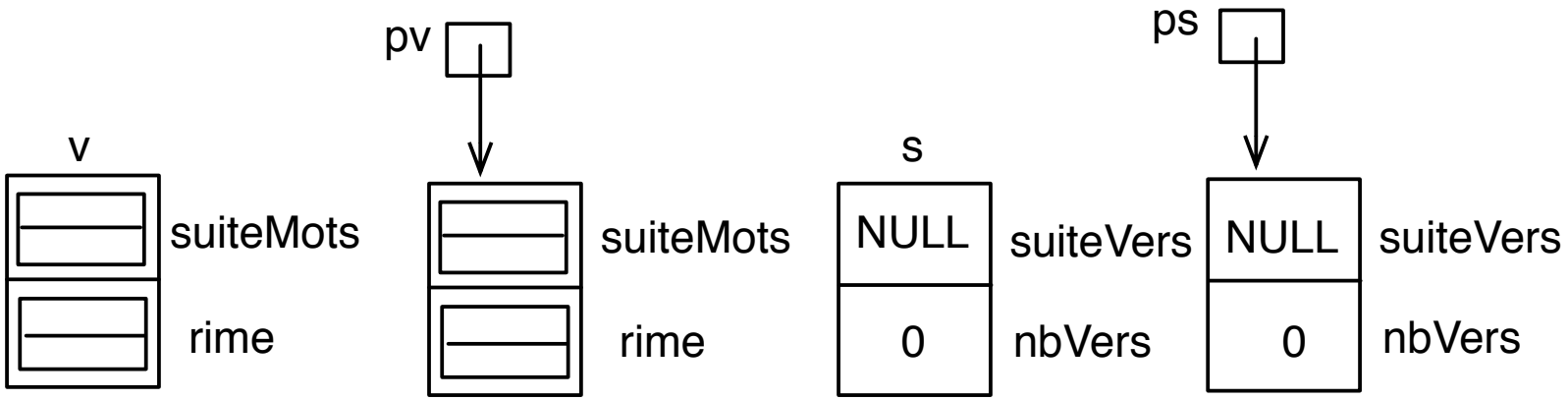


Figure : Structure en mémoire après création des vers et strophes

Opérations en C++

- fonctions
- méthodes de classes avec liaison statique
- méthodes d'instance avec liaison statique
- à utiliser dans le cadre de la programmation par objets :
  - méthodes d'instance avec liaison dynamique (base du polymorphisme), introduites par `virtual`
  - méthodes spéciales : constructeurs, destructeurs

Passage de paramètres et valeur retournée

Une seule sémantique = celle de **l'initialisation**

Lorsque la fonction ou la méthode est appelée, un emplacement est réservé pour chacun des paramètres formels (ceux de la signature) et chaque paramètre formel est initialisé avec le paramètre réel (celui de l'expression d'appel) correspondant

Même mécanisme pour la valeur retournée

Trois formes d'initialisation

```
void f(int fi, int* fpi, int &fri){
fi++; // incrémente fi, copie locale du premier paramètre réel
(*fpi)++; // incrémente le contenu de la zone pointée
                //par le deuxième paramètre réel
fri++; // incrémente le troisième paramètre réel
}
main(){
int i = 4;
int *pi = new int;
*pi = 4;
int j=4;
cout << "i=" << i << "  *pi=" << *pi << "  j=" << j << endl;
// i=4  *pi=4  j=4
f(i, pi, j);
cout << "i=" << i << "  *pi=" << *pi << "  j=" << j << endl;
// i=4  *pi=5  j=5
}
```

Trois formes d'initialisation

- passage *par valeur* lors de l'appel `f(i, pi, j)`, la valeur de `i` est copiée dans `fi`, puis `fi` est incrémenté, ce qui est sans effet sur `i`
- passage *par adresse* (pour la POO) lors de ce même appel, la valeur de `pi` (qui est l'adresse d'une zone de type entier) est copiée dans `fpi`, puis la valeur pointée par `fpi` est incrémentée, et cette valeur est toujours pointée par `pi` donc apparaît bien modifiée lorsqu'on termine `f`
- passage *par référence* après l'initialisation d'une référence telle que `fri` par une variable telle que `j`, il faut comprendre que `fri` est un alias pour `j`, quand on incrémente `fri` on incrémente donc `j` puisque c'est la même entité avec deux noms différents.

Trois formes d'initialisation

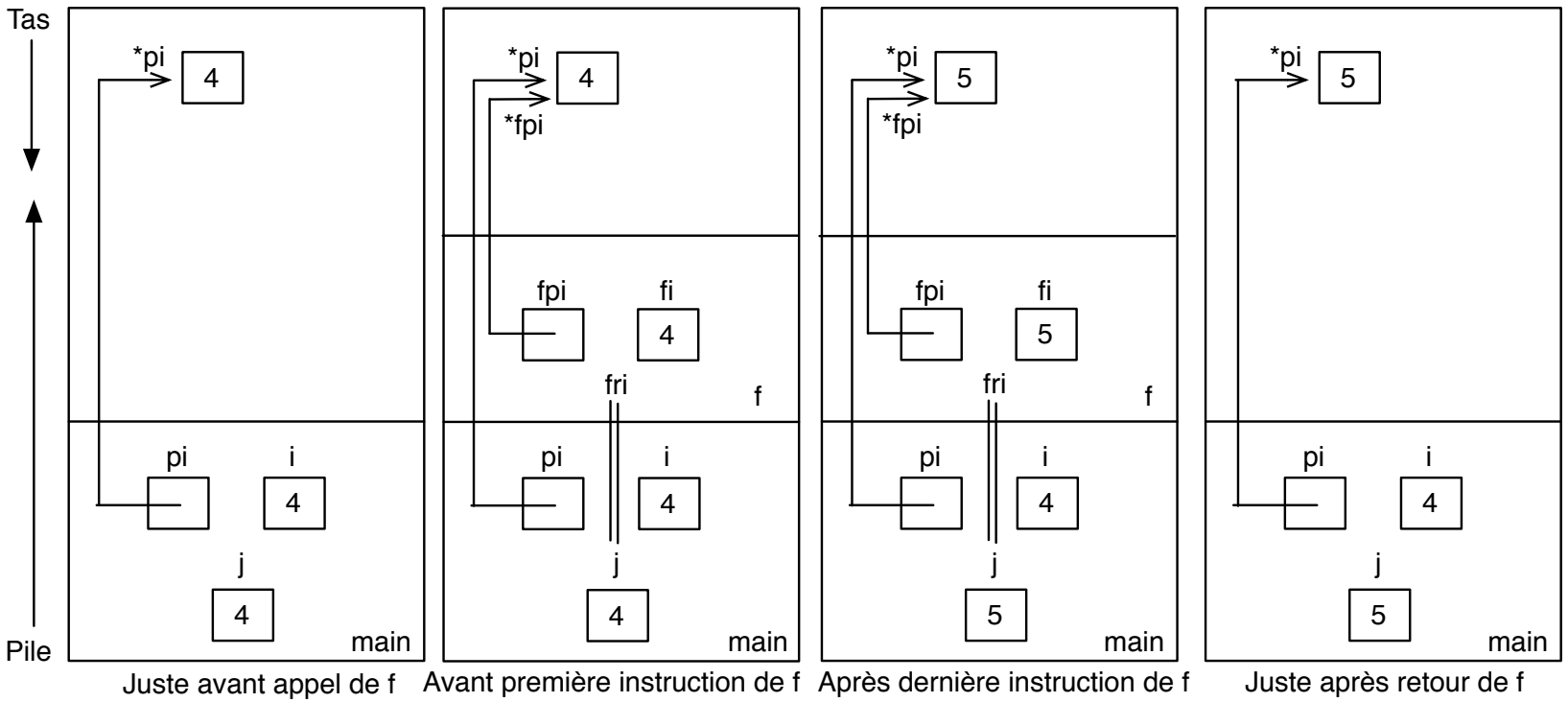


Figure : Evolution de la mémoire lors des passages de paramètres et retour

L'utilisation de const (1)

Pour un paramètre : l'objet est passé en paramètre sans copie mais reste constant pendant l'exécution.

- la méthode peut être utilisée sur des constantes (ex. `const Vers v= ..`)
- le compilateur vérifie l'absence de modifications

Le mode de passage dit *par référence constante* sera parfois utilisé dans le cadre de l'approche par objets en C++ (c'est un passage d'adresse) :

```
......
bool Vers::compareAvec(const Vers& autreVers)
{.......}
..........
```

L'utilisation de const (2)

L'objet receveur du message (l'objet sur lequel on applique la méthode) reste constant pendant l'exécution (traduction du query d'UML).

- la méthode peut être utilisée sur des constantes (ex. `const Vers v= ..`)
- le compilateur vérifie l'absence de modifications

```
// Vers.cc
// ----------------
#include"Vers.h"
......
string Vers::getRime()const
{return rime;}
..........
```

Moi (l'objet receveur)

- Pseudo-variable `this` = désigne l'objet auquel on a envoyé un message pendant l'exécution de la méthode correspondante
- En C++, `this` est un pointeur constant sur l'objet, ex. dans la classe Vers, de type `Vers* const`
- Il est sous-entendu pour l'accès aux propriétés (attributs et méthodes) de l'objet.

Par exemple, la méthode `getSuiteMots()` peut se réécrire comme suit :

```
string Vers::getSuiteMots()const
{return this->suiteMots;}    // équivaut a return (*this).suiteMots;
```

Pendant l'exécution de `getSuiteMots()`, `this` est de type statique `const Vers* const` (pointeur constant vers un vers constant).

Constructeur

Méthodes spéciales, appelées automatiquement lors de la création d'un objet

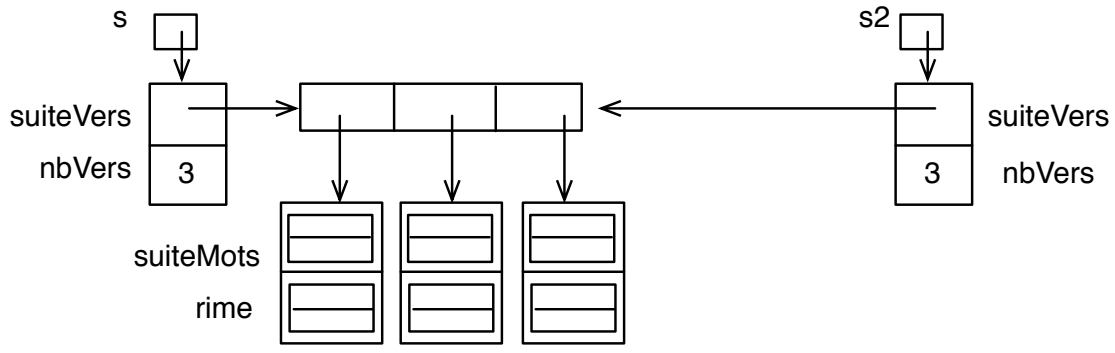
- même nom que la classe
- jamais virtuelles
- acquisition de ressource (mémoire, ouverture fichier, connexion, etc.)
- initialisation des attributs (jamais automatique)
- constructeur vide et sans paramètre : créé par défaut s'il n'y en a aucun autre

Constructeur par copie

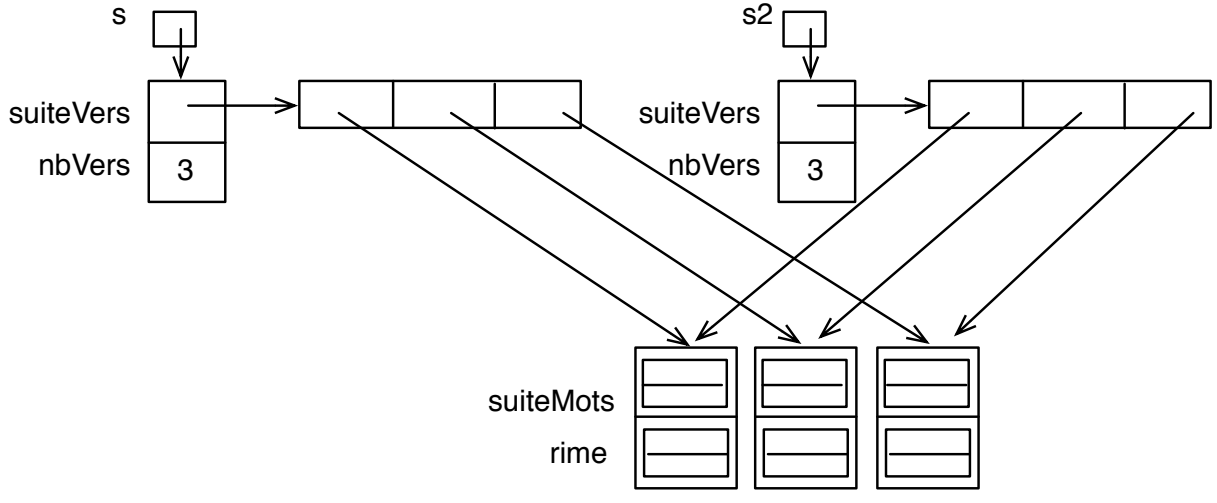
Appelé lors de la copie (ex. passage d'un paramètre par valeur)

- par défaut, copie superficielle : copie champ par champ (avec la sémantique de la copie liée au champ)
- selon les besoins, copie plus ou moins profonde : par exemple pour ne pas partager des ressources mémoire

Exemple : on décide que lors d'une copie, deux strophes ne partagent pas le même tableau de vers, mais les vers restent partagés, ce qui correspond à la sémantique de l'agrégation en UML.



Avec une copie champ par champ (par défaut)



Avec une copie plus profonde (constructeur par copie défini dans le cours)

Petit programme avec des copies de Strophes

```
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"
#include"Strophe.h"

Strophe *s = new Strophe();
s->saisie(cin);
s->affiche(cout);

Strophe *s2 = new Strophe(*s);
s2->saisie(cin);
s2->affiche(cout);
s->affiche(cout);
```

Constructeur par copie

Copie superficielle pour Strophe (ce qui se passe si on n'écrit aucun code)

```
Strophe::Strophe(const Strophe& autreStrophe)
{
   suiteVers=autreStrophe.suiteVers;
       // *this et autreStrophe partagent alors le meme tableau
   nbVers=autreStrophe.nbVers;
}
```

Copie profonde pour Strophe (à écrire pour changer les choses)

```
Strophe::Strophe(const Strophe& autreStrophe)
{
   nbVers=autreStrophe.nbVers;
   suiteVers=new Vers*[nbVers];
   for (int i=0; i<nbVers; i++)
   suiteVers[i]=autreStrophe.suiteVers[i];
}
```

Destructeur

- unique, `virtual`
- même nom que la classe derrière le préfixe `~`
- restitution des ressources acquises par l'objet (mémoire, fichier, connexion, etc.)
- restitution de l'espace mémoire conformément à la sémantique donnée au partage

exemple : dans le cas des strophes, on ne détruit que le tableau, pas les vers conformément à la sémantique de l'agrégation en UML, et de manière cohérente avec le fait que deux strophes peuvent partager les mêmes vers.

```
Strophe::~Strophe(){if (suiteVers) delete[] suiteVers;}
```

Accesseurs

- Importance pour le contrôle de l'état (attributs)
- attributs privés, accesseurs publics mais pas pour tous les attributs !
  Exemple : pas d'accès en écriture au nombre de vers d'une strophe
- accesseur en écriture : en profiter pour effectuer des contrôles sur les valeurs

Accesseurs typiques pour la classe Vers (fichier Vers.cc)

```
......
string Vers::getSuiteMots()const
{return suiteMots;}

void Vers::setSuiteMots(string sm)
{suiteMots=sm;}

string Vers::getRime()const
{return rime;}

void Vers::setRime(string r)
{rime=r;}
..........
```

Initialisation en C++ 2011

Valeurs initiale des attributs

```
class Client {
private:
      string nom ="inconnu"; int nbComptes = 0;
}
```

Délégation d'un constructeur à un autre

```
class Client {
public:
   Client(int nbC) {this->nbComptes = nbC;}
   Client(string n, int nbC) : Client(nbC) {this->nom = n;}
}
```

Initialisation des attributs lors de la création de l'objet (ici appel du constructeur)

```
Client nouveauClient {"jacques", 3};
nouveauClient =  {"jules", 2};
```

Pointeurs en C++ 2011

- Pointeur null : nullptr
- Type `nullptr_t`
- Ne peut plus être comparé à 0

```
// initialisation propre possible en déclarant les attributs
  Vers ** suiteVers = nullptr;
  int nbVers = 0;

// code utilisant la valeur nulle de pointeur dans le constructeur
Strophe::Strophe(){suiteVers=nullptr; nbVers=0;}

// code du destructeur n'utilisant plus le fait que 0 et false
// sont convertis l'un en l'autre automatiquement
Strophe::~Strophe(){if (suiteVers!=nullptr) delete[] suiteVers;}
```

Héritage Simple en C++

Sommaire

Classification et approches à objets

Rapprocher monde réel et représentation informatique
Idée 1-

- se baser sur la notion de « concept »
  - *extension* l'ensemble des objets couverts par le concept
  - *intension* l'ensemble des prédicats vérifiés par les objets couverts
  - classes et interfaces en Java ; classes en C++

Exemple

Le concept de « rectangle »
extension = l'ensemble des rectangles
intension = posséder quatre côtés parallèles deux à deux, posséder deux côtés consécutifs formant un angle droit, etc.

Classification et approches à objets

Rapprocher monde réel et représentation informatique
Idée 2-

- classification / organisation des concepts par spécialisation
  - inclusion des *extensions*
  - raffinement des *intensions*
  - représenté dans les langages par l'héritage

Exemple

Le concept de « carré » spécialise le concept de « rectangle »
l'ensemble des carrés est inclus dans l'ensemble des rectangles (inclusion des extensions).
les propriétés du concept « rectangle » s'appliquent au concept « carré » et se spécialisent (raffinement des intensions), ex. largeur et hauteur sont égales par exemple, le calcul du périmètre et de l'aire sont simplifiés, etc.

Classification et approches à objets

Formes de la classification

- arborescence, héritage simple (Smalltalk, Java-classes)
- graphe sans circuit, héritage multiple (Eiffel, C++, Java-interfaces)

Dans ce cours : héritage simple en C++

Sommaire

Cas d'étude

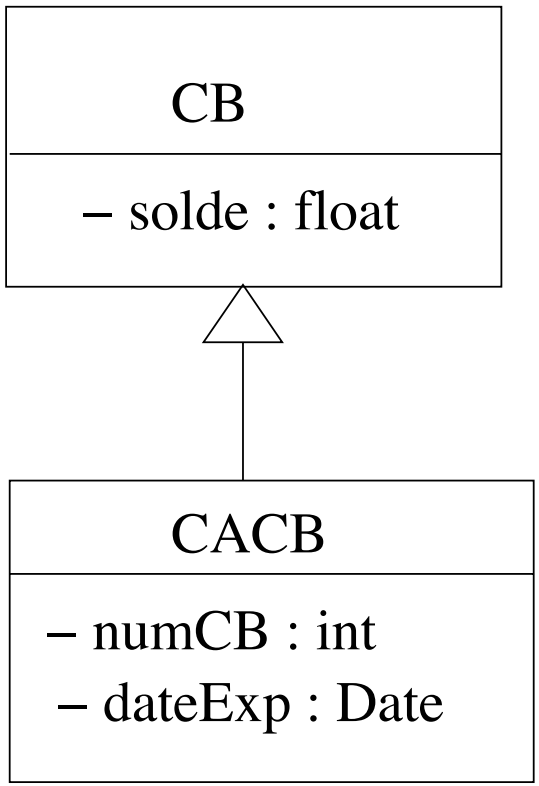


Figure : Comptes bancaires simplifiés

Déclaration dans les fichiers *header*

```
// Compte bancaire
class CB
{private:
  float solde;
 public:
  CB();
  virtual ~CB();
};

// Compte bancaire avec carte bleue
class CACB : public virtual  CB
{private:
  int numCB; Date dateExp;
 public:
  CACB();
  virtual ~CACB();
};
```

Sommaire

Ordre d'appel

- les constructeurs des classes sont appelés du haut vers le bas de la hiérarchie d'héritage (depuis les super-classes vers les sous-classes)
- les constructeurs sont appelés pour les attributs dont le type est une classe (pas un pointeur sur une classe), et ceci juste avant le constructeur de la classe qui déclare l'attribut
- la destruction se fait toujours en sens inverse de la construction

Ordre d'appel

`CACB *pc = new CACB();` ou `CACB c;`

Ordre d'appel des constructeurs :

CB()

Date() ... pour l'attribut dateExp

CACB()

Ordre inverse pour les destructeurs :

~CACB()

~Date() ... pour l'attribut dateExp

~CB()

Passage des paramètres

```
//...... CB.h .......
class CB {
private:
  float solde;
public:
  CB(float s);
};
```

```
//...... CB.cc

CB::CB(float s){solde=s;}
```

```
//...... CACB.h .......
class CACB : public virtual  CB{
private:
  int numCB;
  Date dateExp;
public:
 CACB(float s, int n, Date d);
};
```

```
//...... CACB.cc

CACB::CACB(float s,
  int n, Date d) :CB(s)
{numCB=n; dateExp=d;}
```

Initialisation des attributs

Possible dans le corps du constructeur, sémantique d'affectation

```
CACB::CACB(float s, int n, Date d)
: CB(s)
{numCB=n; dateExp=d;}
```

Possible dans l'entête du constructeur, sémantique de copie

```
CACB::CACB(float s, int n, Date d)
: CB(s),numCB(n),dateExp(d)
{} .
```

Sommaire

Liaison

- Liaison dynamique `virtual`
  préconisée pour l'objet
- Liaison statique sans indication dans le code
  préconisée pour les applications très sensibles au coût en temps et espace et sous la condition qu'il n'y aura pas d'extensions basées sur les mécanismes objets

illustration - débit de frais

```
//...... CB.h .......
class CB
{
private:
  float solde;
  static float fraisGestion;
public:
  ...
  virtual void changeAvec(float f);

  virtual float getSolde()const;

  static float getFraisGestion();

  virtual void debitFrais();
 };
```

```
//...... CB.cc

float CB::fraisGestion=10;

void CB::changeAvec(float f)
                {solde+=f;}
float CB::getSolde()const
               {return solde;}
float CB::getFraisGestion()
{return fraisGestion;}
void CB::debitFrais()
{changeAvec(-fraisGestion);}
```

illustration - débit de frais

```
//...... CACB.h .......
class CACB : public virtual  CB
{
private:
  ...
  static float fraisCB;
public:
  ...

  virtual void debitFrais();
};
```

```
//...... CACB.cc

float CACB::fraisCB=5;
void CACB::debitFrais()
{changeAvec(-(getFraisGestion()
 +fraisCB));}
```

illustration - débit de frais

```
CB **dossierComptes = new CB*[2];
dossierComptes[0] = new CB(200);
dossierComptes[1] = new CACB(74,1212,d);

for (int i=0; i<2; i++)
  dossierComptes[i]->debitFrais();
```

illustration - débit de frais

Avec `virtual`

dossierComptes[0]->debitFrais();

$\longrightarrow$ *appel de debitFrais de CB*

dossierComptes[1]->debitFrais();

$\longrightarrow$ *appel de debitFrais de CACB*

Sans `virtual`

dossierComptes[0]->debitFrais();

$\longrightarrow$ *appel de debitFrais de CB*

dossierComptes[1]->debitFrais();

$\longrightarrow$ *appel de debitFrais de CB!!!*

Pas de liaison dynamique dans les constructeurs

```
CB::CB(float s){solde=s; debitFrais();}
```

Les deux constructeurs suivants exécutent `debitFrais` de CB

```
dossierComptes[0] = new CB(200);
dossierComptes[1] = new CACB(74,1212,d);
```

Explication (discutable) : Pendant l'exécution du constructeur, l'objet n'est pas encore tout à fait construit et ne serait pas en mesure d'exécuter correctement un comportement qui nécessite qu'il soit intégralement initialisé.

Appel de la méthode héritée

Mécanisme de désignation explicite qui se base sur l'opérateur de portée `::`

```
void CACB::debitFrais()
{
  CB::debitFrais();
  changeAvec(-fraisCB);
}
```

- Danger : permet d'appeler une méthode située plus haut dans la hiérarchie, en sautant une méthode plus spécialisée.

Redéfinition de méthode

Règle de redéfinition

- même nom
- même liste de types d'arguments
- type de retour
  - identique pour les types primitifs
  - spécialisé pour pointeur ou référence vers une classe

Exemple :

`virtual CB* debitFrais()` dans `CB`

`virtual CACB* debitFrais()` dans `CACB`

Sommaire

Classes et méthodes abstraites

Classe abstraite = sans instance propre
méthode abstraite = sans corps

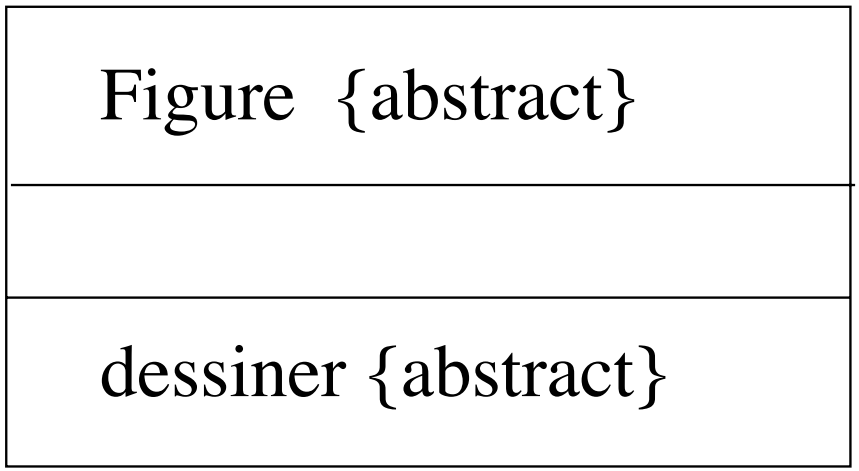


Figure : Une classe et une méthode abstraite

méthode abstraite $\Rightarrow$ classe abstraite

Classes et méthodes abstraites

En C++

Classe abstraite = pas de syntaxe particulière
méthode abstraite = méthode *virtuelle pure*

```
class Figure
{
public:
  virtual void dessine()=0;
};
```

Pas d'implémentation dans Figure.cc
`Figure *f=new Figure();` ne compile pas!!

Sommaire

Affectations entre variables dont le type est une classe

```
CB compteCB(10); CACB compteCACB(20,777,d);
compteCB = compteCACB;
// INTERDIT PAR DEFAUT ..... compteCACB = compteCB;
```

- Appelle l'opérateur `operator=`
- Sauf redéfinition, effectue une copie champ par champ (copie des champs communs)

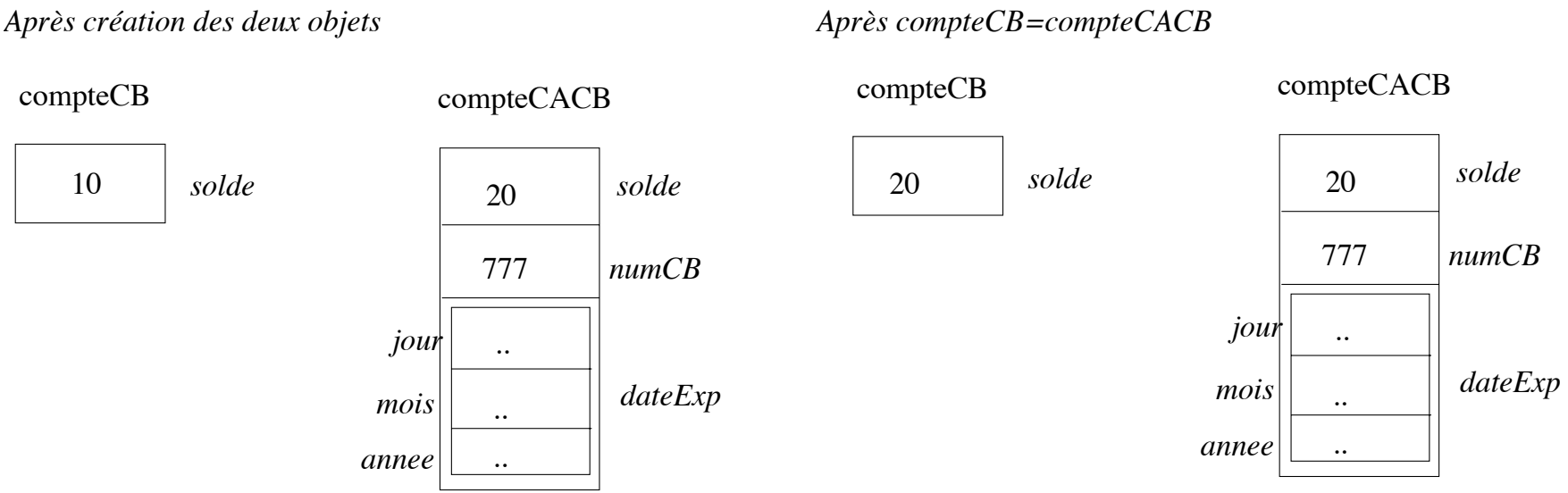


Figure : Affectation entre objets (1)

Affectations entre variables dont le type est une classe

Pour rendre possible l'affectation inverse, la définir !

```
class CACB : virtual public CB
{
 public:
  virtual CACB& operator=(const CB&);
};
........
CACB& CACB::operator=(const CB& compte)
{
  if (this != &compte)
     {changeAvec(compte.getSolde()); numCB=0; dateExp=Date();}
  return *this;
}
```

Affectations entre variables dont le type est une classe

Effet de l'affectation inverse

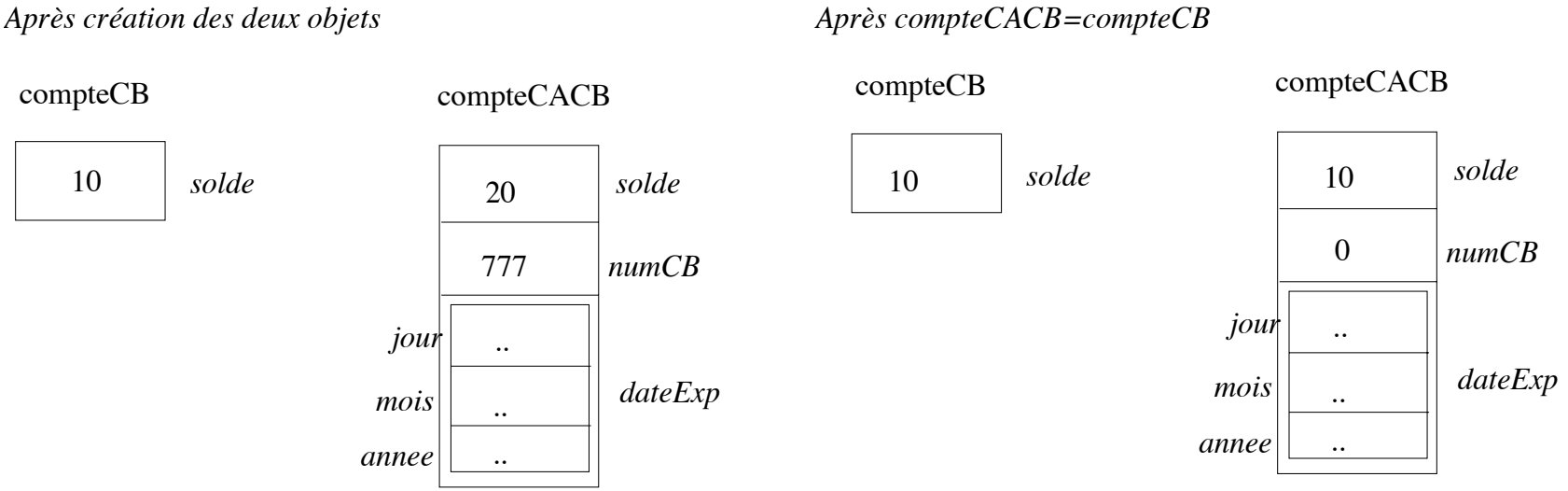


Figure : Affectation entre objets (2)

Affectation entre pointeurs (affectation polymorphe)

```
CB *pcompteCB = new CB(10); CACB *pcompteCACB = new CACB(20,777,d);
pcompteCB = pcompteCACB;
//INTERDIT ....  pcompteCACB = pcompteCB;
```

Pourtant la deuxième peut avoir du sens dans certaines situations

Affectation entre pointeurs (affectation polymorphe)

L'opérateur `dynamic_cast`

- effectue une vérification de type à l'exécution
- retourne `NULL` si la coercition est incorrecte (le pointeur n'est pas du type attendu)
- existe aussi pour les références, en cas d'erreur une exception est signalée

```
CB *pcompteCB = new CACB(10,333,d);
CACB *pcompteCACB = new CACB(20,777,d);
pcompteCACB = dynamic_cast<CACB*>(pcompteCB);
```

Affectation entre pointeurs (affectation polymorphe)

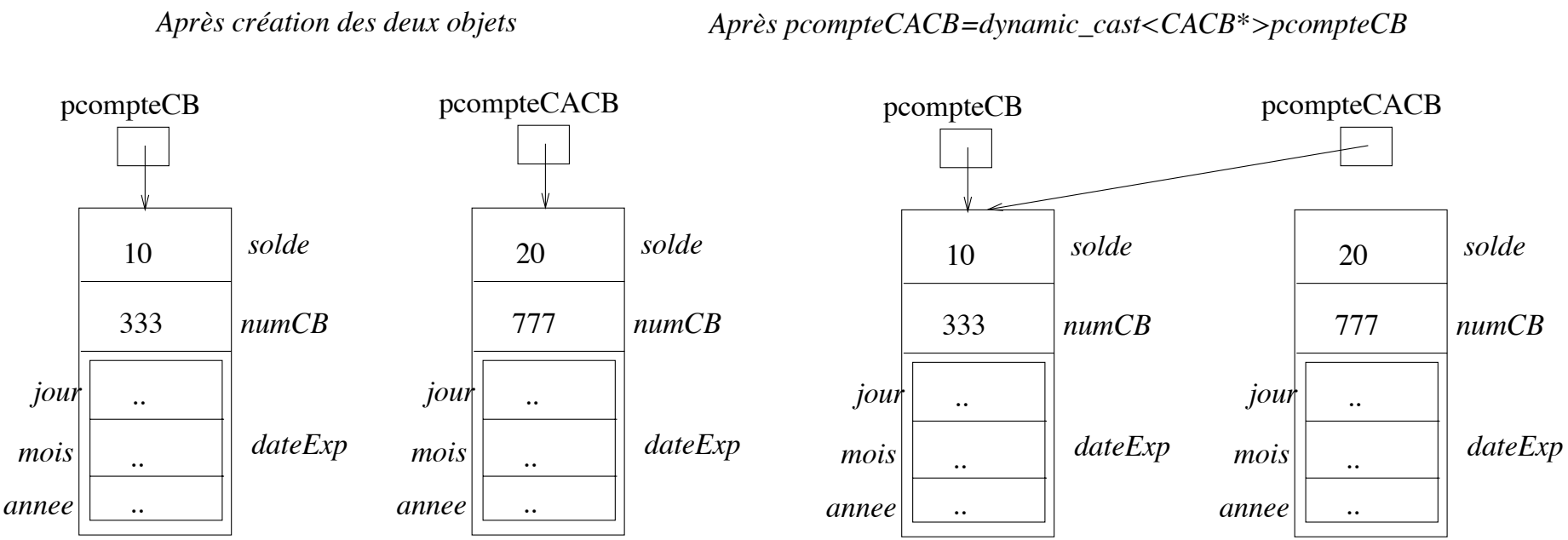


Figure : Affectation entre pointeurs vers des objets

Sommaire

Première approche de la protection

- `public`, `protected` et `private`
- se placent sur les propriétés (attributs, méthodes)
- se placent sur la déclaration d'héritage

Dans une approche simplifiée

- les propriétés publiques sont accessibles partout
- les propriétés privées d'une classe C seulement dans les méthodes de C
- les propriétés protégées de C sont accessibles pour une sous-classe C' de C dans ses méthodes soit sur des instances de C' soit sur des instances des sous-classes de C' (jamais en remontant)
- la déclaration sur l'héritage s'ajoute à celle de la propriété héritée

Les détails dans un prochain cours !

Introspection et réflexivité

Introspection. capacité d'un langage à inspecter, en cours d'exécution, les éléments d'un programme, par exemple pour connaître la classe exacte de l'objet stocké dans une variable ou encore la liste des attributs d'une classe

Réflexivité. les éléments du programme peuvent être véritablement manipulés ; par exemple on peut créer une nouvelle classe pendant l'exécution d'un programme ou encore instancier une classe en donnant seulement son nom sous forme d'une chaîne de caractères.

En C++ : RTTI

RTTI. Run Time Type Information

- opérateur `typeid`, retourne pour un objet *o*, une instance de `type_info` qui décrit le type (la classe) de *o*.
- `type_info`
  - opérateurs $!=$ et $==$ pour comparer les types de deux objets
  - méthode `name` qui retourne une chaîne contenant le nom de la classe de l'objet.

Affichage du nom de la classe `CACB`

```
CB *pcb=new CACB();
cout << typeid(*pcb).name() << endl;
```