Université Claude Bernard Lyon 1

LIFAPSD – Algorithmique, Programmation et Structures de données

Nicolas Pronost

LIRIS

Chapitre 3
Classes et objets

Retour sur les structures

- Dans une structure, vous définissiez uniquement des champs (variables typées)
- Les fonctions et procédures manipulant les champs étaient définies en dehors de la structure
  - la structure manipulée devrait être passée en paramètre (mode donnée ou mode donnée-résultat)

```
Structure Date
   jour : entier
   mois : entier
   annee : entier
Fin structure

Procédure saisirDate (d : Date)
Parametre en mode donnée-résultat: d
Debut
   afficher("Saisir le jour:")
   saisir(d.jour)
   afficher("Saisir le mois:")
   saisir(d.mois)
   afficher("Saisir l'année:")
   saisir(d.annee)
Fin saisirDate
```

```
struct Date{
   int jour, mois, annee;
};



void saisirDate (Date & d) {
   cout << "Saisir le jour:";
   cin >> d.jour;
   cout << "Saisir le mois:";
   cin >> d.mois;
   cout << "Saisir l'annee:";
   cin >> d.annee;
}
```

Principe d'une classe

- Ca serait utile que les fonctions/procédures manipulant une structure soit automatiquement associée à cette structure
  - plus besoin de la passer en paramètre
  - plus besoin de son nom dans les manipulations (nom.champ)
- C'est le principe général d'une classe: une extension d'une structure où on peut définir des « champs » qui sont des fonctions et des procédures
- On ne les appelle plus « champs » mais membres
  - Les données membres (⟺ champs d'une structure: variables typées)
  - Les fonctions membres (fonctions et procédures manipulant les données membres)

Principe d'une classe

- On appelle les variables issues d'un type définit par une classe un objet ou une instance de classe
  - d'où la dénomination programmation orientée objet (POO)
- Chaque membre d'une classe à un spécificateur d'accès
  - il indique si un programme externe à la classe a le droit de manipuler ce membre (lecture/écriture pour une donnée et appel pour une fonction/procédure)
  - si un membre est public, un programme externe peut le manipuler
  - si un membre est privé, il ne peut pas
    - si omis, privé est le spécificateur par défaut en C++
  - il existe un troisième spécificateur (*cf. LIFAPC & LIFAPOO*)
  - les membres sont toujours accédés grâce à l'opérateur . (point)
    - rappel du raccourci: (*ptrObjet).membre ⟺ ptrObjet->membre

Notations algorithmique et C++

- Déclaration d'une classe

```
Classe nom_classe
   spécificateur_accès: membre1
   spécificateur_accès: membre2
   ...
Fin classe
```

```
class nom_classe {
   spécificateur_accès: membre1;
   spécificateur_accès: membre2;
   ...
};
```

- Les spécificateurs d'accès peuvent former des blocs pour les membres de même accès

```
Classe nom_classe
   spécificateur_accès:
     membre1
     membre2
   ...
Fin classe
```

```
class nom_classe {
   spécificateur_accès:
      membre1;
      membre2;
   ...
};
```

- Par convention, on regroupe les membres par spécificateur d'accès (d'abord public ensuite privé) mais en séparant données et fonctions/procédures

Notations algorithmique et C++

- Donc conventionnellement on obtient

```
Classe nom_classe
   public:
     donneeMembre1
     donneeMembre2
     ...

     fonctionMembre1
     fonctionMembre2
     ...

   privé:
     donneeMembre3
     donneeMembre4
     ...

     fonctionMembre3
     fonctionMembre4
    ...

Fin classe
```

```
class nom_classe {
   public:
     donneeMembre1
     donneeMembre2
     ...

     fonctionMembre1
     fonctionMembre2
     ...

   private:
     donneeMembre3
     donneeMembre4
     ...

     fonctionMembre3
     fonctionMembre4
    ...
};
```

Exemple pour la classe Date

```
Classe Date
   public:
     jour : entier
     mois : entier
     annee : entier

     Procédure saisir ()
     Debut
       afficher("Saisir le jour:")
       saisir(jour)
       afficher("Saisir le mois:")
       saisir(mois)
       afficher("Saisir l'année:")
       saisir(annee)
     Fin saisir

     Procédure afficher ()
     Debut
       afficher(jour,"/",mois,"/",annee)
     Fin afficher

   privé:
     Fonction convertirEnJours () : entier
     ...

Fin classe
```

Programme principal

```
Variables
   uneDate : Date
Debut
   uneDate.saisir()
   uneDate.afficher()
   ...
Fin
```

pas le droit d'appeler uneDate.convertirEnJours()

Accès à un membre fonction

- On voit bien que nous n'avons plus besoin de spécifier que le jour saisi correspond à une variable en paramètre
  - L'objet sur lequel la fonction est appelée indique de manière unique quel objet est manipulé

```
Procédure saisir ()
Debut
  afficher("Saisir le jour:")
  saisir(jour)
  ...
Fin saisir
```

```
uneDate.saisir()
```

Mise en œuvre en C++

```
class Date {
   public:
      int jour, mois, annee;

      void saisir() {
          cout << "Saisir le jour:";
          cin >> jour;
          cout << "Saisir le mois:";
          cin >> mois;
          cout << "Saisir l'annee:";
          cin >> annee;
      }

      void afficher() {
          cout << jour << "/" << mois << "/" << annee;
      }

   private:
     int convertirEnJours() { ... }

};
```

Mise en œuvre en C++

```
class Date {
   public:
      int jour, mois, annee;

      void saisir() {
          cout << "Saisir le jour:";
          cin >> jour;
          cout << "Saisir le mois:";
          cin >> mois;
          cout << "Saisir l'annee:";
          cin >> annee;
      }

      void afficher() {
          cout << jour << "/" << mois << "/" << annee;
      }

   private:
     int convertirEnJours() { ... }

};
```

Les membres du même type et même spécificateur peuvent être mis sous forme de liste

Mise en œuvre en C++

```
class Date {
   public:
      int jour, mois, annee;

      void saisir() {
          cout << "Saisir le jour:";
          cin >> jour;
          cout << "Saisir le mois:";
          cin >> mois;
          cout << "Saisir l'annee:";
          cin >> annee;
      }

      void afficher() {
          cout << jour << "/" << mois << "/" << annee;
      }

   private:
     int convertirEnJours() { ... }

};
```

Comme prévu, on peut accéder directement aux membres de la classe (instance sur laquelle la fonction est appelée)

Mise en œuvre en C++

- Le programme principal

```
int main () {
   Date uneDate;

   uneDate.saisir();
   uneDate.afficher();

   return 0;
}
```

Mise en œuvre en C++

- Le programme principal

```
int main () {
   Date uneDate;

   uneDate.saisir();
   uneDate.afficher();

   return 0;
}
```

Création d'une instance de la classe Date (réservation mémoire sur la pile)

Mise en œuvre en C++

- Le programme principal

```
int main () {
   Date uneDate;

   uneDate.saisir();
   uneDate.afficher();

   return 0;
}
```

Accès à des membres publics, ici appel à deux fonctions membres, on pourrait aussi faire uneDate.jour = ... puisque données membres publiques

Mise en œuvre en C++

- On peut aussi créer l'instance de classe sur le tas par allocation dynamique de la mémoire

```
int main () {
   Date * ptrUneDate = new Date;

   ptrUneDate->saisir();
   ptrUneDate->afficher();

   delete ptrUneDate;

   return 0;
}
```

Mise en œuvre en C++

- On peut aussi créer l'instance de classe sur le tas par allocation dynamique de la mémoire

```
int main () {
   Date * ptrUneDate = new Date;

   ptrUneDate->saisir();
   ptrUneDate->afficher();

   delete ptrUneDate;

   return 0;
}
```

Remarquez la notation avec l'opérateur -> équivalent à (*ptrUneDate).saisir() et (*ptrUneDate).afficher()

Mise en œuvre en C++

- On peut aussi créer l'instance de classe sur le tas par allocation dynamique de la mémoire

```
int main () {
   Date * ptrUneDate = new Date;

   ptrUneDate->saisir();
   ptrUneDate->afficher();

   delete ptrUneDate;

   return 0;
}
```

- On peut bien sur créer un tableau de Date

```
// SUR LA PILE
Date tabDates [taille_constante];
```

```
// SUR LE TAS
Date * tabDates = new Date [taille];
delete [] tabDates;
```

Constructeur

- Les instructions `Date uneDate;` et `Date * ptrUneDate = new Date;`

  réserve de la mémoire pour stocker l'instance de la classe, c.-à-d. ses données membres

- Elles appellent aussi le constructeur de la classe
  - membre spécial qui peut être spécifié par le programmeur pour exécuter des instructions (en général d'initialisation de l'instance)
  - il est appelé automatiquement à la création de l'instance
  - il a le même nom que celui de la classe
  - il n'a pas de type de retour (c'est une procédure)
  - il existe un constructeur par défaut vide si le programmeur n'en spécifie pas
  - il peut exister plusieurs constructeurs avec des paramètres différents (mécanisme de surcharge)

Destructeur

- Lorsqu'une instance de classe est dépilée ou lorsque l'on exécute l'instruction **delete**, la mémoire stockant l'instance est libérée

```
delete ptrUneDate;
```

- Le destructeur de la classe est alors appelé
  - membre spécial qui peut être spécifié par le programmeur pour exécuter des instructions (de terminaison de l'instance)
  - il est appelé automatiquement à la destruction de l'instance
  - il a le même nom que celui de la classe précédé du symbole ~ (tilde)
  - il n'a pas de type de retour (c'est une procédure)
  - il existe un destructeur par défaut vide si le programmeur n'en spécifie pas
  - il n'existe qu'un seul destructeur (celui sans paramètre)

Constructeur et destructeur

```
Classe Date
   public:
     jour : entier
     mois : entier
     annee : entier

     Date ()
     Début
       jour ← 1
       mois ← 1
       annee ← 1990
     Fin Date

     Date (j: entier, m: entier, a: entier)
     Début
       jour ← j
       mois ← m
       annee ← a
     Fin Date

     ~Date ()
     Début
       afficher("Date détruite")
     Fin ~Date

     Procédure saisir () ...

Fin classe
```

Programme principal

```
Variables
   uneDate : Date
   autreDate : Date(2,2,2000)
Debut
   uneDate.afficher()
   autreDate.afficher()
Fin
```

Constructeur et destructeur en C++

```
class Date {
   public:
      int jour, mois, annee;

      Date() {
         jour = 1; mois = 1; annee = 1990;
      }

      Date(int j, int m, int a) {
         jour = j; mois = m; annee = a;
      }

      ~Date() {
         cout << "Date détruite";
      }

      void saisir() {...}

      void afficher() {...}

   private:
     int convertirEnJours() {...}

};
```

Programme principal

```
int main () {
   Date uneDate;
   Date autreDate(2,2,2000);
   return 0;
}
```

Constructeur et destructeur en C++

Ne pas écrire `Date uneDate ();` en pensant appeler le constructeur sans paramètre

- cette instruction déclare une fonction s'appelant uneDate qui ne prend pas de paramètre et qui rend une instance de la classe Date

Constructeur et destructeur en C++

- Constructeur par défaut
  - S'il n'y a pas de constructeur spécifié dans la classe, le compilateur considère que la classe a un constructeur par défaut (sans argument et vide)
  - Si au moins un constructeur est spécifié, le compilateur ne fournit plus de constructeur par défaut et donc toutes les créations d'objets doivent utiliser ce(s) constructeur(s)

```
class Date {
   public:
      int jour, mois, annee;
      Date(int j, int m, int a) { ... }
};

int main () {
   Date uneDate;              // IMPOSSIBLE
   Date autreDate(2,2,2000); // OK
   return 0;
}
```

Constructeur et destructeur en C++

- A la création d'un objet, on peut vouloir affecter directement l'objet avec le contenu d'un autre objet de la même classe

```
Date autreDate1 = uneDate; // appel au constructeur par copie
Date autreDate2 (uneDate); // appel au constructeur par copie
```

- Par défaut, cette création copie les données membres d'un objet à l'autre, donc les membres qui sont des pointeurs sont aussi recopiés tels quels, mais pas les objets pointés

```
class Date {
   public:
      int jour, mois, annee;
      int * siecle;
   ...
};
```

```
Date autreDate = uneDate;
```

L'adresse dans `autreDate.siecle` est la même que dans `uneDate.siecle`

L'entier pointé n'est pas recopié.

Constructeur et destructeur en C++

- Constructeur par copie
  - le programmeur peut spécifier lui-même ce constructeur qui est utilisé pour copier un objet lors de la création d'un autre

```
nom_classe (const nom_classe & obj_a_copier);
```

  - le paramètre est une référence sur l'objet à copier
  - le programmeur donne les instructions pour faire cette copie
- Exemple sur Date

```
class Date {
   public:
      int jour, mois, annee;
      int * siecle;
      Date(int j, int m, int a) { ... }
      Date(const Date & d) {
         jour = d.jour;
         mois = d.mois;
         annee = d.annee;
         siecle = new int (*(d.siecle));
      }
    ...
};
```

Programme principal

```
int main () {
   Date uneDate(2,2,2000);
   Date autreDate (uneDate);
   return 0;
}
```

Constructeur et destructeur en C++

- Initialisation des membres dans le constructeur
  - La majorité du temps dans les constructeurs à paramètres, les valeurs des paramètres sont simplement recopiés dans les membres correspondant
  - On peut les initialiser directement avant même d'exécuter le code du constructeur
  - Permet d'appeler des constructeurs à paramètres pour les membres objets

```
class Date {
   public:
      int jour, mois, annee;
      int * siecle;
      SystemeTemporel sys;
      Date (int j, int m, int a) : jour(j), mois(m), annee(a), sys(j+m+a) {
         siecle = new int ((annee/100)+1);
      }
   ...
};
```

Récupérer l'instance manipulée

- Il est parfois utile de connaitre l'adresse mémoire de l'objet manipulé
  - ex. pour qu'une fonction membre puisse le rendre en résultat
  - ex. pour accéder à un membre sans ambiguïté
- En C++, le mot-clé **this** peut être utilisé pour récupérer le pointeur sur l'objet manipulé

```
Date* ajouteJours(int jour) {
   this->jour += jour; // jour += jour aurait modifié le paramètre jour
   return this;        // retourne un pointeur sur l'objet
}
```

Fonction membre constante

- Vous connaissez le mot-clé **const** pour indiquer le mode de passage d'un paramètre par référence

Norme algorithmique	Mise en œuvre en C++	Utilisation
Mode donnée	*type* x	Petits objets
	const *type* & x	Gros objets
Mode donnée-résultat ou résultat	*type* & x	Tous objets

- Il sert aussi à indiquer si une fonction membre a le droit de modifier les données membres d'une classe (vérifié à la compilation)

```
class Date {
   public:
      int jour, mois, annee;

      void saisir() {...}   // A besoin de modifier les données membres
      void afficher() const { ... } // Ne doit pas les modifier
};
```

Paramètres par défaut

- Il est possible de donner des valeurs par défaut à des paramètres de fonctions membres (en fait toute fonction, même les globales ou celles associées aux structures)
- On utilise la notation *= valeur* dans l'entête

```
class Date {
   public:
      int jour, mois, annee;
      void incrementer(int nbJour = 1, int nbMois = 1, int nbAnnee = 1) {
         jour += nbJour; mois += nbMois; annee += nbAnnee;
         // + vérifications: j < 28/29/30/31  et m < 12
      }
   ...
};
```

```
Date uneDate (1,1,2000);
uneDate.incrementer();       // utilise les trois valeurs par défaut (1,1,1)
uneDate.incrementer(2,5,3);  // utilise les paramètres (2,5,3)
uneDate.incrementer(2);      // utilise nbMois et nbAnnee par défaut (2,1,1)
uneDate.incrementer(2,5);    // utilise nbAnnee par défaut (2,5,1)
// impossible d'utiliser le nbJour par défaut et un nbMois ou nbAnnee en paramètre
// uneDate.incrementer(,5,3) n'est pas valide
```

Paramètres par défaut

Attention:

- Les paramètres par défaut doivent être les derniers paramètres dans l'entête

```
void incrementer(int nbJour = 1, int nbMois, int nbAnnee) { ... }
void incrementer(int nbJour = 1, int nbMois = 1, int nbAnnee) { ... }
// Ceci n'est pas valide (erreur à la compilation)
```

- Le premier des paramètres par défaut doit être celui que l'utilisateur a le plus de chance de spécifier
  - pour minimiser les risques qu'il ait à spécifier des paramètres avec leur valeur par défaut

```
void incrementer(int nbJour, int nbMois = 1, int nbAnnee = 1) { ... }
```

```
Date uneDate (1,1,2000);
uneDate.incrementer();       // Erreur compilation: parametre nbJour manquant
uneDate.incrementer(2,5,3);  // OK, utilise (2,5,3)
uneDate.incrementer(2);      // OK, utilise (2,1,1)
uneDate.incrementer(2,1,5);  // OK, mais dommage d'avoir a donner nbMois
```

Surcharge des fonctions membres

- Il est possible d'avoir plusieurs fonctions membres avec le même nom et même type de retour mais avec des paramètres différents
- On l'a déjà vu pour le constructeur, cela nous permet d'avoir
  - un constructeur par défaut sans paramètre
  - un constructeur pour initialiser les données membres avec les valeurs des paramètres
  - un constructeur pour recopier le contenu d'un autre objet
  - un constructeur pour initialiser les données membres avec le contenu d'un fichier dont le nom est en paramètre
  - etc.

Une classe ne peut par contre n'avoir qu'un seul destructeur

Surcharge des fonctions membres

- Ce mécanisme de surcharge peut être utilisé pour n'importe quelle fonction membre d'une classe

```
class Date {
   public:
      int jour, mois, annee;

      bool estAnterieur(const Date & d) const { ... }
      bool estAnterieur(const Date * d) const { ... }
      bool estAnterieur(int nbJours) const { ... }

};
```

```
Date uneDate (1,1,2000);
Date autreDate (2,2,2016);
bool b;
b = uneDate.estAnterieur(autreDate);  // appelle estAnterieur(Date & d)
b = uneDate.estAnterieur(&autreDate); // appelle estAnterieur(Date * d)
b = uneDate.estAnterieur(25100);      // appelle estAnterieur(int nbJours)
```

Surcharge des opérateurs

- Ce mécanisme de surcharge s'applique aussi aux opérateurs sur les objets
- Rappel: on peut appliquer par exemple l'opérateur d'addition sur des variables de type primitif (ex. **int**), puis l'opérateur d'affectation

```
int a,b,c;
...
a = b + c;
```

- Pour des types primitifs, le sens de ce genre d'opérations est évident et non ambigu
- Pour des types complexes, ça n'est pas forcément si clair

```
class Date {
   public:
      int jour, mois, annee;
};
```

```
Date date1, date2, date3;
...
date1 = date2 + date3;
```

  - Quel est le résultat de l'opération d'addition entre les deux dates? Doit-on sommer les jours, mois, années séparément? Quelles vérifications faire? Comment le compilateur peut-il savoir ce que l'on a en tête?

Surcharge des opérateurs

- En fait dans l'exemple précédent, le compilateur produirait une erreur car l'opération d'addition n'a pas été spécifiée par le programmeur
- En C++, la plupart des opérateurs peuvent être redéfinis pour une classe, c'est la surcharge des opérateurs

Quelques opérateurs surchargeables									
+	-	*	/	=	<	>	+=	-=	*=
/=	<<	>>	==	!=	<=	>=	++	--	%
&&	!	||	[ ]	( )	->	.			

Surcharge des opérateurs

- Afin de spécifier un opérateur pour une classe, on l'ajoute dans la liste des fonctions membres de la classe
- En langage algorithmique, on utilise le mot-clé opérateur suivi du symbole de l'opérateur

```
Procédure/Fonction opérateur symbole (paramètres) : type
Debut
   ...
Fin opérateur
```

- En C++, on utilise le mot-clé **operator**

```
type operator symbole (paramètres) { ... }
```

  - chaque opérateur à un ensemble d'entêtes précises qu'il faut respecter pour pouvoir les utiliser
  - certains peuvent aussi être définis globalement (en dehors d'une classe) et prendre les objets sur lesquels opérer en paramètres

Surcharge des opérateurs

- Exemples avec Date
  - On peut définir que additionner deux dates, produit un décalage de la première date du nombre de jours, mois et années de la seconde
    - On aurait par exemple (25,12,2000) + (10,0,0) vaut (4,1,2001)
  - On peut définir que mesurer l'infériorité consiste à regarder, dans cet ordre, si l'année est plus petite, puis le mois, puis le jour
    - On aurait (25,12,2000) < (10,11,2001) est vrai
  - On peut définir que l'affectation recopie les 3 données membres
  - On peut définir que l'opérateur d'accès [ ] retourne le jour, mois, année en fonction de la valeur de l'indice (0, 1 ou 2)
    - On aurait uneDate[1] vaut 12 si uneDate est (25,12,2000)
  - On peut définir l'opérateur d'extraction << afin d'afficher la date sous le format jj / mm / aaaa
    - On aurait cout << uneDate qui afficherait 25 / 12 / 2000 si uneDate est (25,12,2000)

Surcharge des opérateurs

```
class Date {
   public:
      int jour, mois, annee;

      const Date operator + (const Date& operandeDroite) { ... }
      bool operator < (const Date& operandeDroite) { ... }
      Date& operator = (const Date& operandeDroite) { ... }
      int& operator [] (int indice) { ... }
      friend std::ostream& operator << (std::ostream& flux, const Date& d) { ... }
}
```

Surcharge des opérateurs

```
class Date {
   public:
      int jour, mois, annee;

      const Date operator + (const Date& operandeDroite) { ... }
      bool operator < (const Date& operandeDroite) { ... }
      Date& operator = (const Date& operandeDroite) { ... }
      int& operator [] (int indice) { ... }
      friend std::ostream& operator << (std::ostream& flux, const Date& d) { ... }
}
```

L'objet de la classe sur lequel est appelé l'opérateur est l'opérande gauche de l'opérateur, l'objet en paramètre est l'opérande droite

Surcharge des opérateurs

```
class Date {
   public:
      int jour, mois, annee;

      const Date operator + (const Date& operandeDroite) { ... }
      bool operator < (const Date& operandeDroite) { ... }
      Date& operator = (const Date& operandeDroite) { ... }
      int& operator [] (int indice) { ... }
      friend std::ostream& operator << (std::ostream& flux, const Date& d) { ... }
}
```

Les types de retour sont spécifiques à chaque opérateur. Attention il peut exister plusieurs entêtes pour un même opérateur (ex. si un paramètre est **const** ou non).

Classes amies

- Une fonction ou une classe amie à une autre classe a accès à tous les membres de cette classe
  - quelque soit le spécificateur d'accès
  - est définie en tant que membre de la classe que l'on veut accéder

```
Fonction amie nom_fonction (parametres) : type_retour
```

```
friend type nom_fonction (parametres) { ... }
```

```
classe amie nom_classe
```

```
friend class nom_classe;
```

Surcharge des opérateurs

```
class Date {
   public:
      int jour, mois, annee;

      const Date operator + (const Date& operandeDroite) { ... }
      bool operator < (const Date& operandeDroite) { ... }
      Date& operator = (const Date& operandeDroite) { ... }
      int& operator [] (int indice) { ... }
      friend std::ostream& operator << (std::ostream& flux, const Date& d) { ... }
}
```

Attention, fonctionnement particulier pour >> et << ! Ceci n'est pas une surcharge de << par la classe Date car il ne s'applique pas sur l'objet de la classe. Ce n'est donc pas une fonction membre de la classe. Il faut donc définir cette fonction globale comme amie de la classe pour pouvoir accéder aux données membres (ici toutes les trois publiques, mais le faire tout de même en général). L'objet à afficher est passé en paramètre (puisque pas une fonction membre).
Il faut rendre le flux afin de pouvoir enchaîner les opérateurs (ex. cout << uneDate << uneAutreDate << endl;)

Surcharge des opérateurs

```
const Date operator + (const Date& operandeDroite) {
   Date resultatAddition;
   resultatAddition.jour = jour + operandeDroite.jour;
   resultatAddition.mois = mois + operandeDroite.mois;
   resultatAddition.annee = annee + operandeDroite.annee;
   // + vérification j < 28,29,30,31 et m < 12 et mise à jour
   return resultatAddition;
}

bool operator < (const Date& operandeDroite) {
    if (annee < operandeDroite.annee) return true;
    if (annee > operandeDroite.annee) return false;
    if (mois < operandeDroite.mois) return true;
    if (mois > operandeDroite.mois) return false;
    if (jour < operandeDroite.jour) return true;
    return false;
}
```

Surcharge des opérateurs

```
const Date operator + (const Date& operandeDroite) {
   Date resultatAddition;
   resultatAddition.jour = jour + operandeDroite.jour;
   resultatAddition.mois = mois + operandeDroite.mois;
   resultatAddition.annee = annee + operandeDroite.annee;
   // + vérification j < 28,29,30,31 et m < 12 et mise à jour
   return resultatAddition;
}

bool operator < (const Date& operandeDroite) {
    if (annee < operandeDroite.annee) return true;
    if (annee > operandeDroite.annee) return false;
    if (mois < operandeDroite.mois) return true;
    if (mois > operandeDroite.mois) return false;
    if (jour < operandeDroite.jour) return true;
    return false;
}
```

Ces données sont les données membres de l'opérande gauche.
Par exemple la date uneDate dans les instructions:

```
troisiemeDate = uneDate + autreDate;
if (uneDate < autreDate) { ... }
```

Surcharge des opérateurs

```
const Date operator + (const Date& operandeDroite) {
   Date resultatAddition;
   resultatAddition.jour = jour + operandeDroite.jour;
   resultatAddition.mois = mois + operandeDroite.mois;
   resultatAddition.annee = annee + operandeDroite.annee;
   // + vérification j < 28,29,30,31 et m < 12 et mise à jour
   return resultatAddition;
}

bool operator < (const Date& operandeDroite) {
    if (annee < operandeDroite.annee) return true;
    if (annee > operandeDroite.annee) return false;
    if (mois < operandeDroite.mois) return true;
    if (mois > operandeDroite.mois) return false;
    if (jour < operandeDroite.jour) return true;
    return false;
}
```

Ce paramètre formel est l'opérande de droite dans l'opération appelée.
Par exemple la date autreDate dans les instructions:

```
troisiemeDate = uneDate + autreDate;
if (uneDate <  autreDate) { ... }
```

Surcharge des opérateurs

```
const Date operator + (const Date& operandeDroite) {
   Date resultatAddition;
   resultatAddition.jour = jour + operandeDroite.jour;
   resultatAddition.mois = mois + operandeDroite.mois;
   resultatAddition.annee = annee + operandeDroite.annee;
   // + vérification j < 28,29,30,31 et m < 12 et mise à jour
   return resultatAddition;
}

bool operator < (const Date& operandeDroite) {
    if (annee < operandeDroite.annee) return true;
    if (annee > operandeDroite.annee) return false;
    if (mois < operandeDroite.mois) return true;
    if (mois > operandeDroite.mois) return false;
    if (jour < operandeDroite.jour) return true;
    return false;
}
```

En C++, les accès sont sur la base des classes et non des instances de classes. Donc dans ces fonctions membres de la classe Date, on peut accéder directement aux données membres de tout objet du type Date, y compris celui en paramètre, et ce quelque soit le spécificateur d'accès (public ou private).

Surcharge des opérateurs

```
const Date operator + (const Date& operandeDroite) {
   Date resultatAddition;
   resultatAddition.jour = jour + operandeDroite.jour;
   resultatAddition.mois = mois + operandeDroite.mois;
   resultatAddition.annee = annee + operandeDroite.annee;
   // + vérification j < 28,29,30,31 et m < 12 et mise à jour
   return resultatAddition;
}

bool operator < (const Date& operandeDroite) {
    if (annee < operandeDroite.annee) return true;
    if (annee > operandeDroite.annee) return false;
    if (mois < operandeDroite.mois) return true;
    if (mois > operandeDroite.mois) return false;
    if (jour < operandeDroite.jour) return true;
    return false;
}
```

Ces opérateurs sont ceux définis (surchargés) dans la classe **int**.

Surcharge des opérateurs

```
const Date operator + (const Date& operandeDroite) {
    Date resultatAddition;
    resultatAddition.jour = jour + operandeDroite.jour;
    resultatAddition.mois = mois + operandeDroite.mois;
    resultatAddition.annee = annee + operandeDroite.annee;
    // + vérification j < 28,29,30,31 et m < 12 et mise à jour
    return resultatAddition;
}

bool operator < (const Date& operandeDroite) {
    if (annee < operandeDroite.annee) return true;
    if (annee > operandeDroite.annee) return false;
    if (mois < operandeDroite.mois) return true;
    if (mois > operandeDroite.mois) return false;
    if (jour < operandeDroite.jour) return true;
    return false;
}
```

Remarque: l'opérateur + retourne une copie de la Date créée localement. Le résultat de l'addition n'est donc dans aucune des deux opérandes mais dans une troisième adresse mémoire. Ici la copie de l'objet Date local vers la valeur de retour est inévitable (attention au coût pour les gros objets), le résultat de date1+date2 doit bien être stocké quelque part.

Surcharge des opérateurs

```
Date& operator = (const Date& operandeDroite) {
   jour = operandeDroite.jour;
   mois = operandeDroite.mois;
   annee = operandeDroite.annee;
   return *this;
}

int& operator [] (int indice) {
   switch (indice) {
      case 0 : return jour;
      case 1 : return mois;
      case 2 : return annee;
      default : return jour;
   }
}

friend std::ostream&  operator << (std::ostream& flux, const Date& d) {
   flux << d.jour << " / " << d.mois << " / " << d.annee;
   return flux;
}
```

Surcharge des opérateurs

```
Date& operator = (const Date& operandeDroite) {
   jour = operandeDroite.jour;
   mois = operandeDroite.mois;
   annee = operandeDroite.annee;
   return *this;
}

int& operator [] (int indice) {
   switch (indice) {
      case 0 : return jour;
      case 1 : return mois;
      case 2 : return annee;
      default : return jour;
   }
}

friend std::ostream&  operator << (std::ostream& flux, const Date& d) {
   flux << d.jour << " / " << d.mois << " / " << d.annee;
   return flux;
}
```

Ceci doit vous faire penser au code du constructeur par copie.

Surcharge des opérateurs

```
Date& operator = (const Date& operandeDroite) {
   jour = operandeDroite.jour;
   mois = operandeDroite.mois;
   annee = operandeDroite.annee;
   return *this;
}

int& operator [] (int indice) {
   switch (indice) {
      case 0 : return jour;
      case 1 : return mois;
      case 2 : return annee;
      default : return jour;
   }
}

friend std::ostream&  operator << (std::ostream& flux, const Date& d) {
   flux << d.jour << " / " << d.mois << " / " << d.annee;
   return flux;
}
```

Attention:

```
Date date1;              // constructeur par défaut
Date date2 = date1;      // constructeur par copie
Date date3 (date1);      // constructeur par copie
Date date4;              // constructeur par défaut
date4 = date1;           // affectation
```

Ceci doit vous faire penser au code du constructeur par copie.

Surcharge des opérateurs

```
Date& operator = (const Date& operandeDroite) {
   jour = operandeDroite.jour;
   mois = operandeDroite.mois;
   annee = operandeDroite.annee;
   return *this;
}

int& operator [] (int indice) {
   switch (indice) {
      case 0 : return jour;
      case 1 : return mois;
      case 2 : return annee;
      default : return jour;
   }
}

friend std::ostream&  operator << (std::ostream& flux, const Date& d) {
   flux << d.jour << " / " << d.mois << " / " << d.annee;
   return flux;
}
```

L'affectation rend une référence sur la date après mise à jour des données membres. On utilise un déréférencement du pointeur sur l'objet (**this**) pour accéder à l'objet et on déclare le type de retour Date& pour rendre une référence sur cet objet.

Surcharge des opérateurs

```
Date& operator = (const Date& operandeDroite) {
   jour = operandeDroite.jour;
   mois = operandeDroite.mois;
   annee = operandeDroite.annee;
   return *this;
}

int& operator [] (int indice) {
   switch (indice) {
      case 0 : return jour;
      case 1 : return mois;
      case 2 : return annee;
      default : return jour;
   }
}

friend std::ostream& operator << (std::ostream& flux, const Date& d) {
   flux << d.jour << " / " << d.mois << " / " << d.annee;
   return flux;
}
```

En fonction de l'indice en paramètre, on rend une référence sur une donnée membre. On choisit ici ce que l'on fait si l'indice n'est pas « valide ».

Surcharge des opérateurs

```
Date& operator = (const Date& operandeDroite) {
   jour = operandeDroite.jour;
   mois = operandeDroite.mois;
   annee = operandeDroite.annee;
   return *this;
}

int& operator [] (int indice) {
   switch (indice) {
      case 0 : return jour;
      case 1 : return mois;
      case 2 : return annee;
      default : return jour;
   }
}

friend std::ostream& operator << (std::ostream& flux, const Date& d) {
   flux << d.jour << " / " << d.mois << " / " << d.annee;
   return flux;
}
```

La date à afficher est en paramètre puisque ce n'est pas une fonction membre mais une fonction amie. Le flux sur lequel écrire est aussi donné en paramètre (ex. cout ou fichier).

Surcharge des opérateurs

```
Date& operator = (const Date& operandeDroite) {
   jour = operandeDroite.jour;
   mois = operandeDroite.mois;
   annee = operandeDroite.annee;
   return *this;
}

int& operator [] (int indice) {
   switch (indice) {
      case 0 : return jour;
      case 1 : return mois;
      case 2 : return annee;
      default : return jour;
   }
}

friend std::ostream& operator << (std::ostream& flux, const Date& d) {
   flux << d.jour << " / " << d.mois << " / " << d.annee;
   return flux;
}
```

Même si la fonction n'est pas membre, on peut accéder aux données membres puisque c'est une amie (bon ici les données sont aussi publiques).

Surcharge des opérateurs

```
Date& operator = (const Date& operandeDroite) {
   jour = operandeDroite.jour;
   mois = operandeDroite.mois;
   annee = operandeDroite.annee;
   return *this;
}

int& operator [] (int indice) {
   switch (indice) {
      case 0 : return jour;
      case 1 : return mois;
      case 2 : return annee;
      default : return jour;
   }
}

friend std::ostream& operator << (std::ostream& flux, const Date& d) {
   flux << d.jour << " / " << d.mois << " / " << d.annee;
   return flux;
}
```

On retourne le flux pour pouvoir enchaîner les appels.

Surcharge des opérateurs

Programme principal

```
int main () {
   Date dateDebut(7,9,2016);      // début de semestre
   Date dateFin(10,1,2017);       // fin de semestre
   Date tempsTP1(0,1,0);          // prévision temps travail pour TP1
   Date tempsTP2(15,0,0);         // prévision temps travail pour TP2
   Date tempsTP3(15,1,0);         // prévision temps travail pour TP3

   Date tempsNecessaire;          // constructeur par défaut
   tempsNecessaire = tempsTP1 + tempsTP2 + tempsTP3;

   if (dateDebut + tempsNecessaire < dateFin)
      cout << "Estimation OK" ;
   else
      cout << "Mauvaise estimation, date de fin prévue: " << dateDebut +
      tempsNecessaire << "dépasse la fin du semestre: " << dateFin;

   return 0;
}
```

Surcharge des opérateurs

Programme principal

```
int main () {
   Date dateDebut(7,9,2016);      // début de semestre
   Date dateFin(10,1,2017);       // fin de semestre
   Date tempsTP1(0,1,0);          // prévision temps travail pour TP1
   Date tempsTP2(15,0,0);         // prévision temps travail pour TP2
   Date tempsTP3(15,1,0);         // prévision temps travail pour TP3

   Date tempsNecessaire;          // constructeur par défaut
   tempsNecessaire = tempsTP1 + tempsTP2 + tempsTP3;

   if (dateDebut + tempsNecessaire < dateFin)
      cout << "Estimation OK" ;
   else
      cout << "Mauvaise estimation, date de fin prévue: " << dateDebut +
      tempsNecessaire << "dépasse la fin du semestre: " << dateFin;

   return 0;
}
```

Appel à l'opérateur d'affectation =

Surcharge des opérateurs

Programme principal

```
int main () {
   Date dateDebut(7,9,2016);      // début de semestre
   Date dateFin(10,1,2017);       // fin de semestre
   Date tempsTP1(0,1,0);          // prévision temps travail pour TP1
   Date tempsTP2(15,0,0);         // prévision temps travail pour TP2
   Date tempsTP3(15,1,0);         // prévision temps travail pour TP3

   Date tempsNecessaire;          // constructeur par défaut
   tempsNecessaire = tempsTP1 + tempsTP2 + tempsTP3;

   if (dateDebut + tempsNecessaire < dateFin)
      cout << "Estimation OK" ;
   else
      cout << "Mauvaise estimation, date de fin prévue: " << dateDebut +
      tempsNecessaire << "dépasse la fin du semestre: " << dateFin;

   return 0;
}
```

Appels à l'opérateur d'addition + entre deux dates

Surcharge des opérateurs

Programme principal

```
int main () {
   Date dateDebut(7,9,2016);      // début de semestre
   Date dateFin(10,1,2017);       // fin de semestre
   Date tempsTP1(0,1,0);          // prévision temps travail pour TP1
   Date tempsTP2(15,0,0);         // prévision temps travail pour TP2
   Date tempsTP3(15,1,0);         // prévision temps travail pour TP3

   Date tempsNecessaire;          // constructeur par défaut
   tempsNecessaire = tempsTP1 + tempsTP2 + tempsTP3;

   if (dateDebut + tempsNecessaire < dateFin)
      cout << "Estimation OK" ;
   else
      cout << "Mauvaise estimation, date de fin prévue: " << dateDebut +
      tempsNecessaire << "dépasse la fin du semestre: " << dateFin;

   return 0;
}
```

Appel à l'opérateur d'infériorité stricte < entre deux dates

Surcharge des opérateurs

Programme principal

```
int main () {
   Date dateDebut(7,9,2016);      // début de semestre
   Date dateFin(10,1,2017);       // fin de semestre
   Date tempsTP1(0,1,0);          // prévision temps travail pour TP1
   Date tempsTP2(15,0,0);         // prévision temps travail pour TP2
   Date tempsTP3(15,1,0);         // prévision temps travail pour TP3

   Date tempsNecessaire;          // constructeur par défaut
   tempsNecessaire = tempsTP1 + tempsTP2 + tempsTP3;

   if (dateDebut + tempsNecessaire < dateFin)
      cout << "Estimation OK" ;
   else
      cout << "Mauvaise estimation, date de fin prévue: " << dateDebut +
      tempsNecessaire << "dépasse la fin du semestre: " << dateFin;

   return 0;
}
```

Appels à l'opérateur d'affichage << d'une date

Surcharge des opérateurs

Programme principal

```
int main () {
   Date dateDebut(7,9,2016);      // début de semestre
   Date dateFin(10,1,2017);       // fin de semestre
   Date tempsTP1(0,1,0);          // prévision temps travail pour TP1
   Date tempsTP2(15,0,0);         // prévision temps travail pour TP2
   Date tempsTP3(15,1,0);         // prévision temps travail pour TP3

   Date tempsNecessaire;          // constructeur par défaut
   tempsNecessaire = tempsTP1 + tempsTP2 + tempsTP3;

   if (dateDebut + tempsNecessaire < dateFin)
      cout << "Estimation OK" ;
   else
      cout << "Mauvaise estimation, date de fin prévue: " << dateDebut +
      tempsNecessaire << "dépasse la fin du semestre: " << dateFin;

   return 0;
}
```

Un appel à l'opérateur d'accès [ ] pourrait être: tempsTP3[0], rendant 15

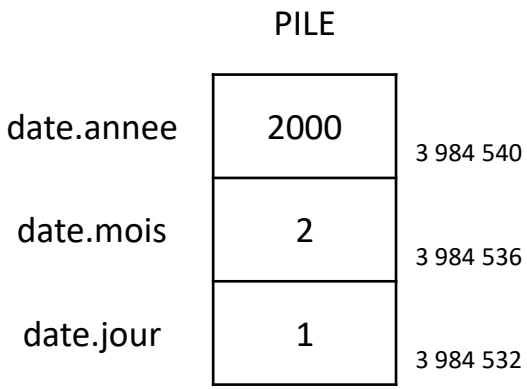
Organisation en mémoire

- En LIFBAP
  - organisation des données (variables) déclarées sur la pile
  - organisation des données (paramètres, variable de retour) lors des appels à fonction/procédure
    - mode donnée et mode donnée-résultat (référence)
- Au cours 2 de LIFAPSD
  - organisation des données (variables) allouées sur le tas
  - organisation des données (paramètres, variable de retour) lors des appels à fonction/procédure
    - mode donnée et mode donnée-résultat (pointeur)
- Maintenant
  - organisation des données (objets) sur la pile et le tas
  - organisation des données (objets) lors des appels à fonction/procédure (globale et membre)

Objets en mémoire

- Création d'un objet sur la pile

```
Date date(1,2,2000);
```

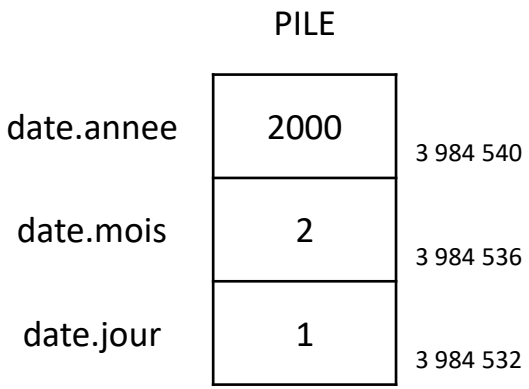


- Attention, notez l'ordre dans lequel les données membres sont empilés (les premières données ont les adresses les plus basses, comme les éléments d'un tableau)
  - l'adresse d'un objet vaut l'adresse de sa première donnée membre
- La mémoire est libérée lorsque l'objet n'est plus visible (comme n'importe quelle variable)

Objets en mémoire

- Création d'un objet sur la pile

```
Date date(1,2,2000);
```

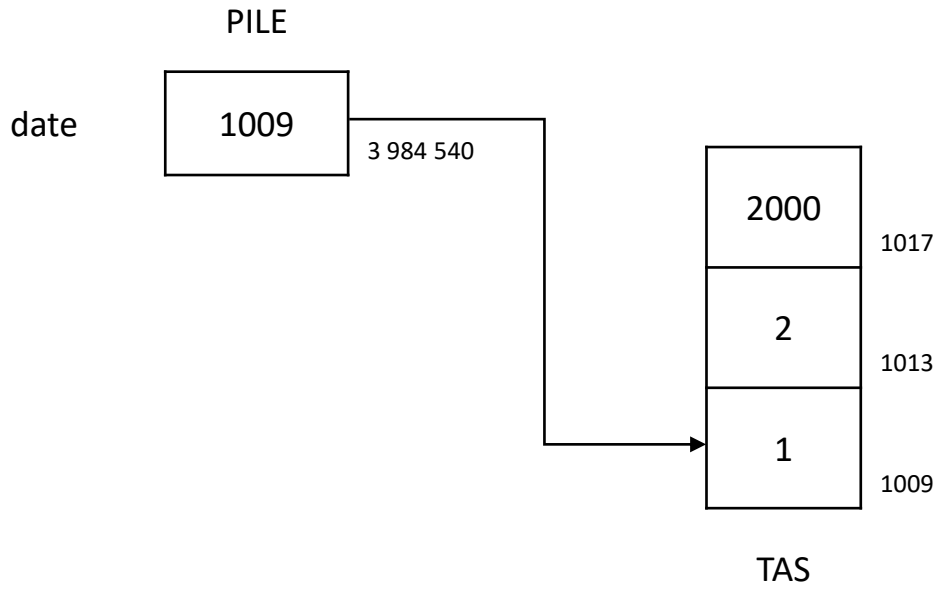


- On a donc
  - &date vaut 3 984 532 de type Date*
  - &(date.jour) vaut aussi 3 984 532 mais de type **int** *
  - la taille de l'objet en mémoire **sizeof**(Date) est 3 * **sizeof**(**int**)

Objets en mémoire

- Création d'un objet sur le tas

```
Date * date = new Date(1,2,2000);
```

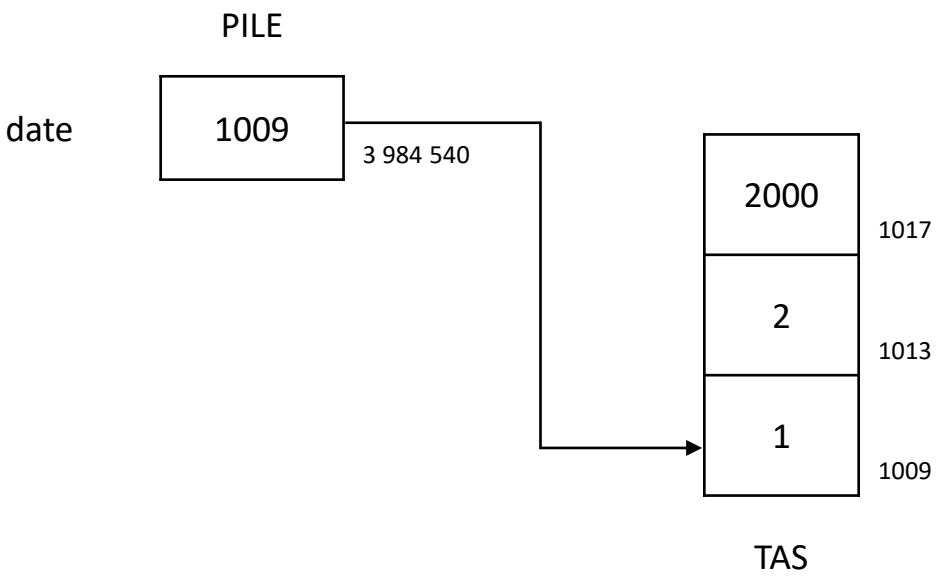


- Les données membres sont également stockées dans l'ordre des adresses croissantes sur le tas

Objets en mémoire

- A la libération (manuelle) de la mémoire, les données membres sont libérées

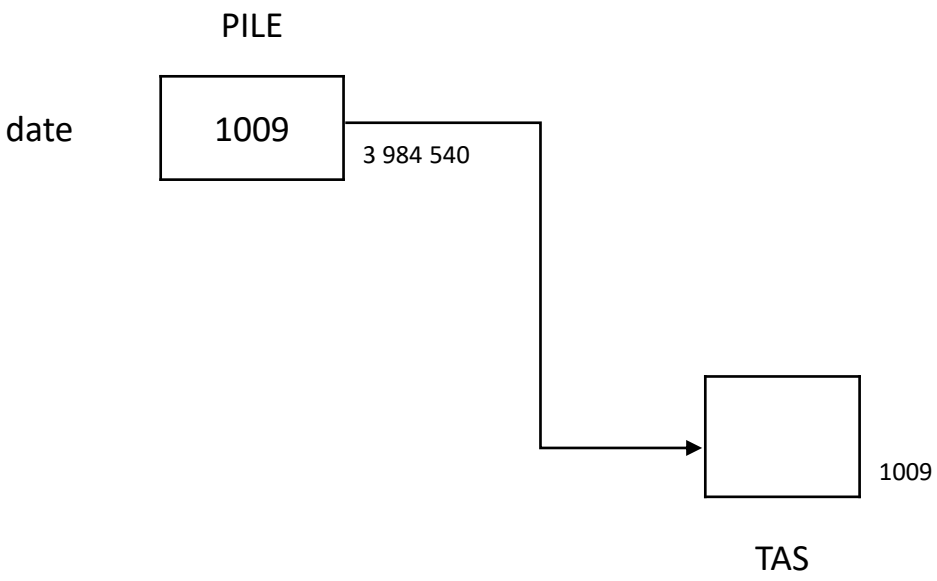
```
delete date;
date = nullptr;
```



Objets en mémoire

- A la libération (manuelle) de la mémoire, les données membres sont libérées

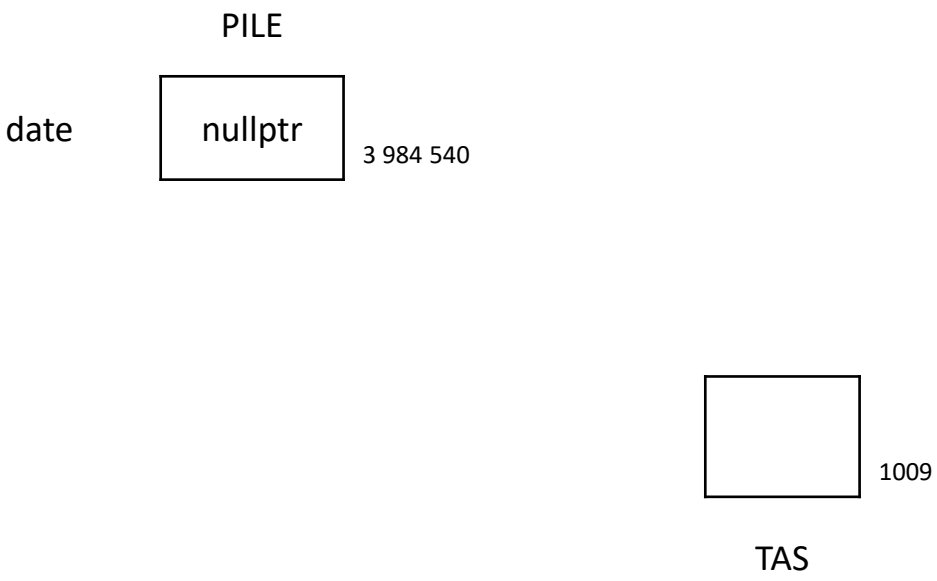
```
delete date;
date = nullptr;
```



Objets en mémoire

- A la libération (manuelle) de la mémoire, les données membres sont libérées

```
delete date;
date = nullptr;
```

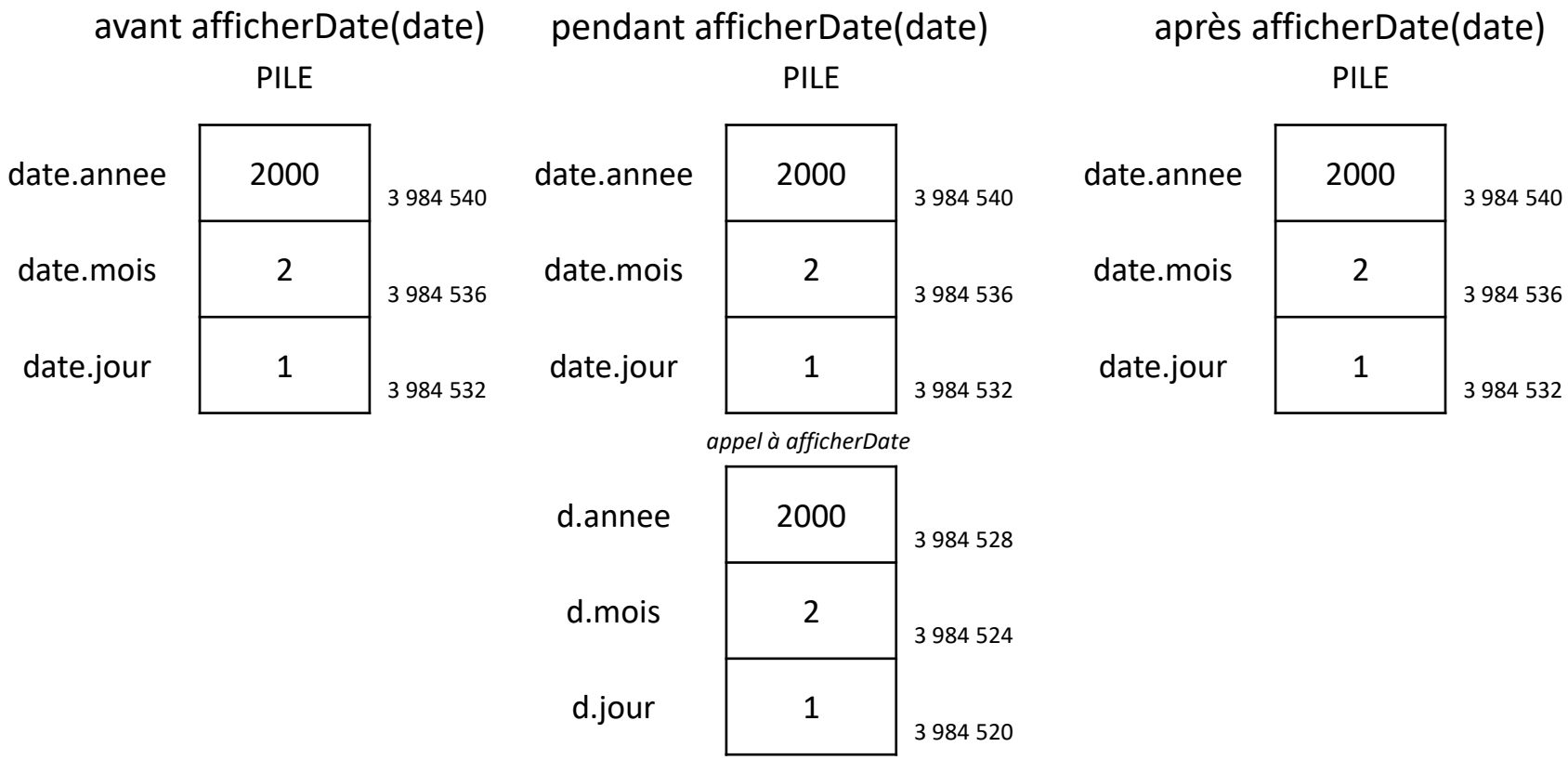


Objet en paramètre

- En mode donnée par recopie (à éviter pour les gros objets)

```
Date date(1,2,2000);
afficherDate(date);
```

```
void afficherDate(Date d) {...}
```

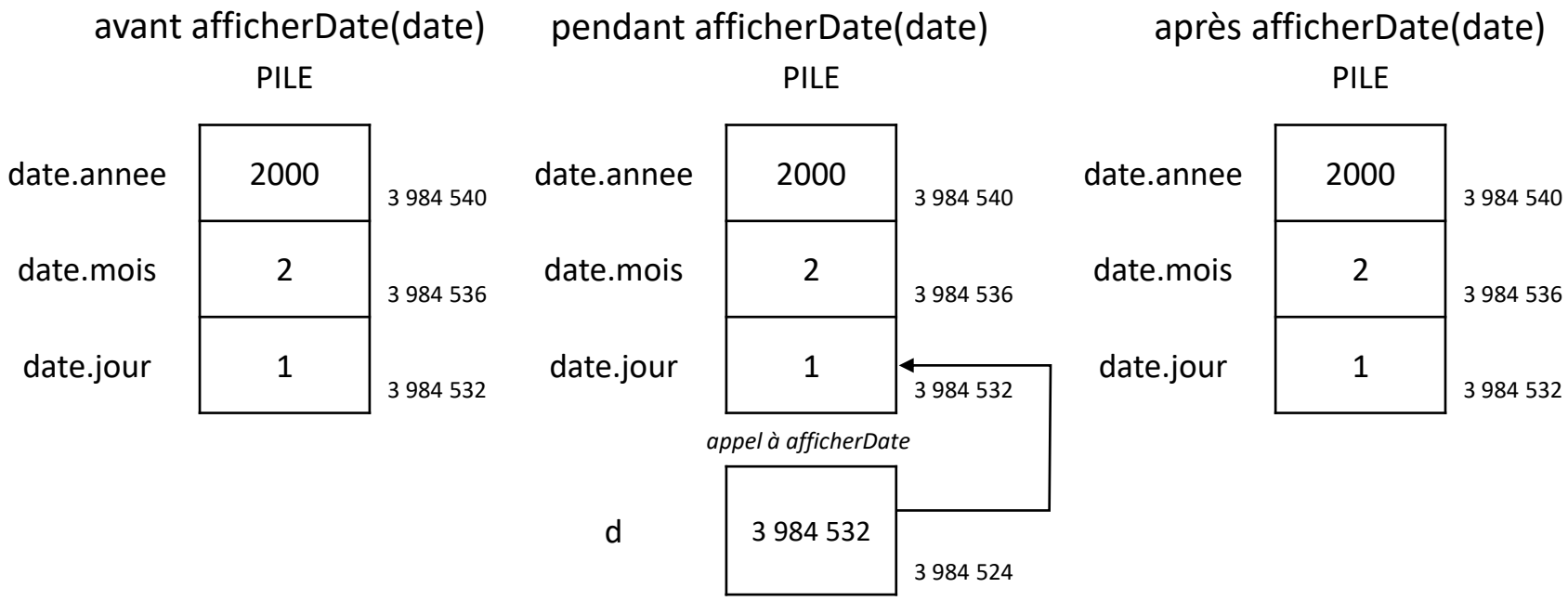


Objet en paramètre

- En mode donnée par référence (bien pour les gros objets)

```
Date date(1,2,2000);
afficherDate(date);
```

```
void afficherDate(const Date& d) {...}
```

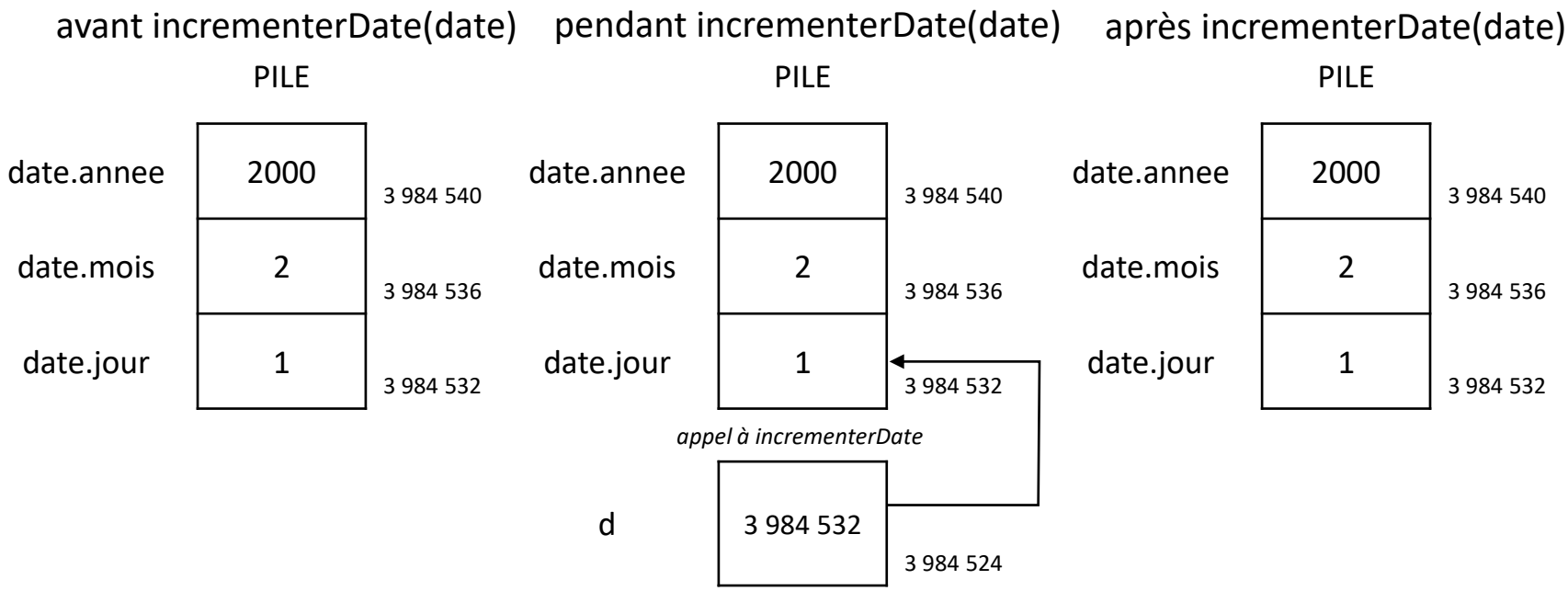


Objet en paramètre

- En mode donnée-résultat (i.e. sans le **const**)

```
Date date(1,2,2000);
incrementerDate(date);
```

```
void incrementerDate(Date& d) {...}
```



Donc la même organisation que par mode donnée par référence! La vérification d'autorisation de modification n'est faite qu'à la compilation, pas à l'exécution.

Appel à des fonctions membres

- Comment la fonction sait où récupérer les données membres de l'objet?
  - Un pointeur sur l'objet est automatiquement ajouté aux paramètres (le premier empilé)
  - Donc en fait, une fonction membre est juste une fonction normale (globale) où un paramètre est ajouté automatiquement
  - Le compilateur a prévu un appel à **this** et l'ajout du pointeur résultant aux paramètres
  - Le compilateur change les accès directs aux membres par des accès par le pointeur: **this**->membre

Appel à des fonctions membres

- Exemple avec une référence à un autre objet en paramètre

```
bool egal(const Date& d) const {
    if (annee != d.annee) return false;
    if (mois != d.mois) return false;
    return jour == d.jour;
}
```

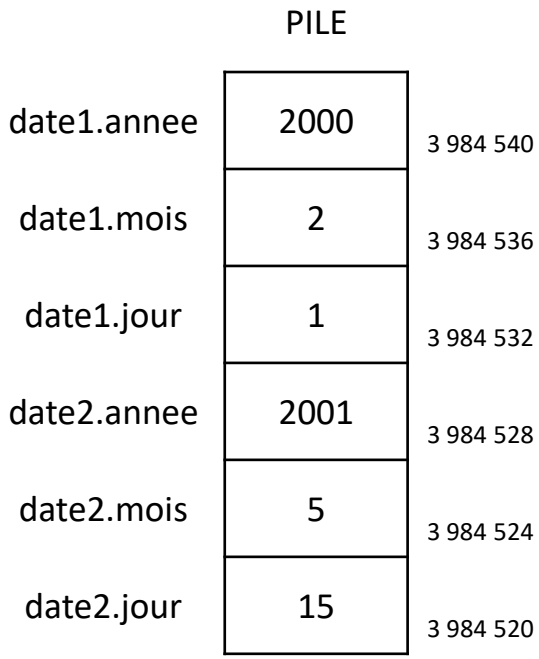
```
Date date1(1,2,2000);
Date date2(15,5,2001);
if (date1.egal(date2)) ...
```

Appel à des fonctions membres

- Exemple avec une référence à un autre objet en paramètre

```
bool egal(const Date& d) const {
    if (annee != d.annee) return false;
    if (mois != d.mois) return false;
    return jour == d.jour;
}
```

```
Date date1(1,2,2000);
Date date2(15,5,2001);
if (date1.egal(date2)) ...
```



Appel à des fonctions membres

- Exemple avec une référence à un autre objet en paramètre

```
bool egal(const Date& d) const {
    if (annee != d.annee) return false;
    if (mois != d.mois) return false;
    return jour == d.jour;
}
```

```
Date date1(1,2,2000);
Date date2(15,5,2001);
if (date1.egal(date2)) ...
```