Programmation Orientée Objet : Structures de Données Abstraites & Fonctions et Classes Génériques

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

EPFL

État des lieux

Nous voici donc arrivés au terme des cours avec support MOOC.

Nous avons abordé jusqu'ici :

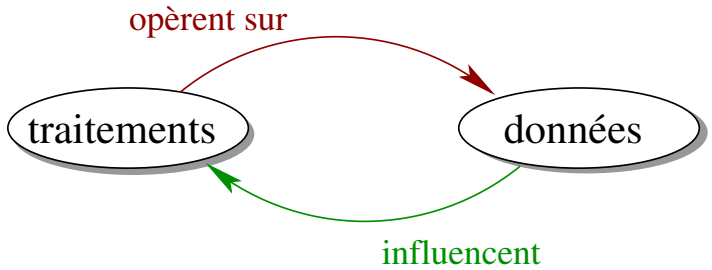
1. les bases de la programmation procédurale ;
2. les bases de la programmation orientée objets.

☞ Il nous reste à aborder quelques thèmes d'approfondissement : structures de données et «templates» ainsi qu'un survol de la librairie standard

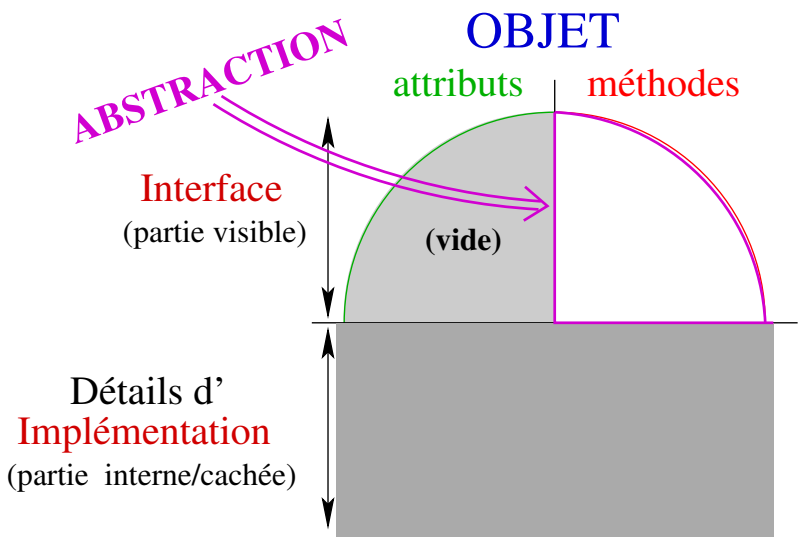
Jamila Sam
& Jean-Cédric Chappelier

Qu'avons nous vu en programmation ?

programmer c'est décomposer une **tâche** à automatiser en une séquence d'instructions (**traitements**) et des **données**



en programmation orientée objets, on regroupe dans le même objet **les traitements et les données** qui lui sont spécifiques (principe d'**encapsulation**)



Qu'avons nous vu en programmation ?

programmer c'est décomposer une **tâche** à automatiser en une séquence d'instructions (**traitements**) et des **données**

Algorithme	S.D.A.
Traitements	Données
Expressions & Opérateurs Structures de contrôle Fonctions	Variables Portée Chaînes de caractères Tableaux statiques Tableaux dynamiques Structures Pointeurs
Entrées/Sorties	

Jamila Sam
& Jean-Cédric Chappelier

Qu'avons nous vu en programmation ?

en programmation orientée objets, on regroupe dans le même objet **les traitements et les données** qui lui sont spécifiques (principe d'**encapsulation**)

Objet	
Encapsulation et Abstraction Classes Héritage simple/multiple Polymorphisme Classes abstraites/virtuelles Résolution des collisions de noms	
Traitements	**Données**
Méthodes Constructeurs & Destructeurs Const Virtuelles (pures) Surcharge d'opérateurs(interne/externe)	Attributs Appels aux constructeurs des attributs (hérités) Statiques
Privés/protégés/publiques Hérités/cachés (: :)	

Objectifs du cours d'aujourd'hui

- Introduction aux **structures de données abstraites** :
  - *Listes chaînées*
  - *Piles*
- Introduction à la programmation générique :
  - Exemples
  - Déclaration de modèles
  - Instanciation
  - Spécialisation
  - Compilation séparée

Jamila Sam
& Jean-Cédric Chappelier

Plan

- **Structures de données abstraites**
  - Listes
  - Piles
- Programmation générique
  - Programmation générique : introduction, exemples
  - Déclaration des modèles
  - Instanciation
  - Spécialisation
  - Compilation séparée

Pourquoi modéliser les données ?

L'élaboration d'un algorithme est grandement facilité par l'utilisation de **structures de données abstraites**, de plus haut niveau, et de fonctions de manipulations associées.

Une structure de données doit *modéliser au mieux les informations* à traiter pour en faciliter le traitement par l'algorithme considéré.

Choisir les bons modèles de données est aussi important que le choix de bons algorithmes

Algorithme et structure de données abstraite sont intimement liés :

Programme = algorithme + données

Jamila Sam
& Jean-Cédric Chappelier

C'est quoi une « structure de données abstraite » ?

La notion de **structure de données abstraite** (S.D.A.) est indépendante de tout langage de programmation
Une S.D.A. est un ensemble organisé d'informations (ou données) reliées logiquement et pouvant être manipulées non seulement individuellement mais aussi comme un tout.

Exemples généraux :

- tableau (au sens général du terme)
  contenu : divers éléments de types à préciser
  interactions : demander la taille du tableau, accéder (lecture/écriture) à chaque élément individuellement, ...
- vecteur (au sens général, pas C++) : formalisation mathématique d'espace vectoriel sur un corps $\mathscr{K}$
  contenu : n coordonnées (éléments de $\mathscr{K}$)
  interactions : les propriétés élémentaires définissant un espace vectoriel

Exemple informatique élémentaire :

Spécifications des structures de données abstraites

Une S.D.A. est caractérisée par :

- son contenu
- les interactions possibles (manipulation, accès, ...)

Du point de vue informatique, une structure de données abstraite peut être spécifiée à deux niveaux :

- niveau fonctionnel / logique : spécification formelle des données et des **algorithmes** de manipulation associés
- niveau physique (programmation) : comment est implémentée la structure de données abstraite dans la mémoire de la machine

  ☞ déterminant pour l'efficacité des **programmes** utilisant ces données.

Spécifications des S.D.A. [2]

Au niveau formel (modèle), on veut généraliser cette idée « d'objets » manipulables par des opérateurs propres, sans forcément en connaître la structure interne et encore moins l'implémentation.

Par exemple, vous ne pensez pas un `int` comme une suite de 32 bits, mais bien comme un « entier » (dans un certain intervalle) avec ses opérations propres : `+`, `-`, `*`, `/`

Une structure de données abstraite définit une abstraction des données et **cache les détails de leur implémentation**.

abstraction : identifier précisément les **caractéristiques** de l'entité (par rapport à ses applications), et en décrire les **propriétés**.

Jamila Sam
& Jean-Cédric Chappelier

Spécifications des S.D.A. [3]

Une structure de données abstraite modélise donc l'« **ensemble des services** » désirés plutôt que l'organisation intime des données (détails d'implémentation)

On identifie usuellement 4 types de « services » :

1. les modificateurs, qui modifient la S.D.A.
2. les sélecteurs, qui permettent « d'interroger » la S.D.A.
3. les itérateurs, qui permettent de parcourir la structure
4. les constructeurs

Exemple :
tableau dynamique
modifieur : affectation d'un élément (`t[i]=a`)
sélecteur : lecture d'un élément (`t[i]`)
sélecteur : le tableau est-il vide ? (`t.size() == 0`)
itérateur : index d'un élément (`[i]` ci-dessus)

Divers exemples de S.D.A.

Il y a beaucoup de structures de données abstraites en Informatique.

Dans ce cours, nous n'allons voir que les 2 plus fondamentales (après les tableaux) :

- les listes
- et les piles

Autres :

- files d'attente (avec ou sans priorité)
- multi-listes
- arbres (pleins de sorte...)
- graphes
- tables de hachage

Plan

- Structures de données abstraites
  - **Listes**
  - Piles
- Programmation générique
  - Programmation générique : introduction, exemples
  - Déclaration des modèles
  - Instanciation
  - Spécialisation
  - Compilation séparée

Listes

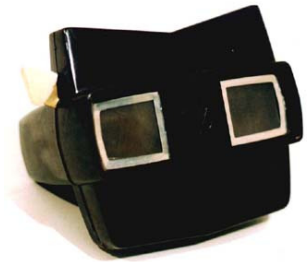
Spécification logique :
Ensemble d'éléments *successifs* (pas d'accès direct), ordonnés ou non

Interactions :

- accès au premier élément (sélecteur)
- accès à l'élément suivant d'un élément (sélecteur)
- modifier l'élément courant (modificateur)
- insérer/supprimer un élément après(/avant) l'élément courant (modificateur)
- tester si la liste est vide (sélecteur)
- parcourir la liste (itérateur)

Listes

Exemple concret :



visionneuse stéreo (essayez d'accéder à la 3e image directement, sans passer par la 2e !)

Exemple informatique :

(a (b (c (d))))

Une liste peut être vu comme une structure récursive :

liste = élément + liste OU liste = vide

Réalisations d'une liste

- réalisation statique :

  tableau

- réalisation dynamique (liste chaînée) :

  vector (mais inconvénient 1 ci-après)

ou

classe :

La bonne solution :

```
class Cellule;
typedef Cellule* PtrCell;
class ListeChaineeCellule {
  type_el donnee;
  ListeChainee*  PtrCell  suivant;
}
```

Pourquoi les listes dynamiques ?

Les tableaux sont un type de données très utile en programmation mais présentent **2 limitations** :

1. les données sont *contiguës* (les unes derrières les autres) et donc l'insertion d'un nouvel élément au milieu du tableau demande la recopie (le décalage) de tous les éléments suivants.
   $\Longrightarrow$ **insertion en** $\mathcal{O}(n)$
2. pour les tableaux statiques, augmenter la taille (par exemple si elle n'est pas connue *a priori*) nécessite la création d'un nouveau tableau
   $\Longrightarrow$ $\mathcal{O}(n)$

Complexité optimale des opérations élémentaires sur les listes

insérer un élément : $\mathcal{O}(1)$ (temps constant)

supprimer un élément : $\mathcal{O}(1)$ (temps constant)

calculer la longueur : $\mathcal{O}(n)$ (temps fonction linéaire en `n`)

(voire $\mathcal{O}(1)$ si le stockage de cette valeur est effectué, en particulier si « longueur » a été spécifiée dans les « services » de la SDA « liste »)

vider la liste : $\mathcal{O}(n)$

parcourir la liste : $\mathcal{O}(n)$

Exemples d'implémentation des opérations élémentaires sur les listes

À des fins pédagogiques, voici une implémentation simple des listes dynamiques de `double` sous forme de listes chaînées :

```
class Cellule;
typedef Cellule* PtrCell;
const PtrCell LISTE_VIDE(0);
// Une cellule de la liste
class Cellule{
public:
  Cellule(double un_double)
    :donnee(un_double), suite(LISTE_VIDE){}
  Cellule(double un_double, PtrCell suite)
    :donnee(un_double), suite(suite){}
  PtrCell getSuite(){return suite;};
  double getDonnee(){return donnee;};
  void setSuite(PtrCell une_suite){suite = une_suite;}
private:
        double donnee;    // la donnee
        PtrCell suite;    // le pointeur sur la cellule
                          // suivante
};
```

Exemples d'implémentation des opérations élémentaires sur les listes (2)

```
// Le type Liste chainee
class Liste{
public:
  Liste():queue(LISTE_VIDE){}
  bool est_vide();
  void insere(double un_double);
  void insere(Cellule& cell, double un_double);
  unsigned int taille();
private:
  PtrCell tete;  //un pointeur sur le premier element
  PtrCell queue; //un pointeur sur le dernier element
};
```

Exemples d'insertion d'un élément

- en queue de liste

```
void Liste::insere(double un_double)
{
  if (est_vide()) {
    tete = new Cellule(un_double) ;
    queue = tete;
  }
  else
    insere((*queue), un_double);
}
```

- après un élément donné de la liste

```
void Liste::insere(Cellule& existante,
                   double un_double)
{
  PtrCell suite(existante.getSuite());
  PtrCell c(new Cellule(un_double,suite));
  existante.setSuite(c);
  if (c->getSuite() == LISTE_VIDE) queue = c;
 }
```

Insertion d'un élément dans une liste chaînée

C++11 Pour les `forward_list` :

- l'insertion en tête de liste s'appelle `push_front`
- et l'insertion après un élément donné `insert_after`, mais nécessite la notion d'itérateur que nous verrons un peu plus loin.

Exemple de calcul de la longueur

```
unsigned int Liste::taille()
{
  unsigned int taille(0);
  PtrCell courant(tete);
  while(courant != LISTE_VIDE)
      {
        ++taille;
        courant = courant->getSuite();
      }
  return taille;
}
```

Exercice : quelle solution serait plus efficace ?

C++11 Note : Attention les `forward_list`, n'ont pas de fonction `size()` !

Jamila Sam
& Jean-Cédric Chappelier

Implémentations existantes des listes chaînées

Les listes (simplement) chainées existent depuis C++11 :

```
#include <forward_list>

forward_list<int> ma_liste({ 6, 1, 5, -23, 3 });

for (auto element : ma_liste) { cout << element << endl;

ma_liste.push_front(877);
```

Note : Les listes *doublement* chainées existent depuis C++98 :

```
#include <list>
```


Jamila Sam
& Jean-Cédric Chappelier

Plan

- Structures de données abstraites
  - Listes
  - **Piles**
- Programmation générique
  - Programmation générique : introduction, exemples
  - Déclaration des modèles
  - Instanciation
  - Spécialisation
  - Compilation séparée

Plan

Piles

Spécification :
Une pile est une structure de données abstraite dynamique contenant des éléments homogènes (de type non précisé) à 1 point d'accès et permettant

- d'ajouter une valeur à la pile (empiler ou push) ;
- de lire la **dernière** valeur ajoutée ;
- d'enlever la dernière valeur ajoutée (dépiler ou pop) ;
- de tester si la pile est vide.

On ne « connait » donc de la pile **que** le dernier élément empilé (son sommet).

Spécification physique :
liste chaînée
ou
tableau dynamique (vector)

Piles : exemples

Exemples concrets :

- une pile d'assiettes
- poupées russes

Piles : exemples (2)

Exemple d'utilisation (formelle) :

empiler x	x
empiler a	a x
dépiler	x
empiler b	b x
empiler y	y b x
dépiler	b x

Exemple d'utilisation des piles

Le problème des parenthèses : étant donnée une expression avec des parenthèses, est-elle bien ou mal parenthésée ?

$((a+b)*c-(d+4)*(5+(a+c)))*(c+(d+(e+5*g)*f)*a)$ (correct)

$(a+b)($ (incorrect)

Encore un peu plus complexe : différentes parenthèses
Exemple avec [et (

([])[()(()[])] ☞ correct
([)] ☞ incorrect

Autres exemples d'utilisation des piles (non traités ici) :

- tours de Hanoi
- notation postfixée (ou « polonaise inverse ») :

  $4\ 2\ +\ 5\ *$

  (☞ $5*(4+2)$)

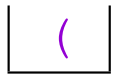
Vérification de parenthésage

Tant que lire caractère c
  Si c est (ou [
    empiler c
  Sinon
    Si c est) ou]
      Si pile vide
        ÉCHEC
      Sinon
        c' ← lire la pile
        Si c et c' correspondent
          dépiler
        Sinon
          ÉCHEC
Si pile vide
  OK
Sinon
  ÉCHEC

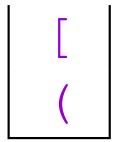
Exemple

Entrée : ([ )]

empile (



empile [

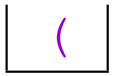


lu =), top = [
→ ne correspond pas
⟹ ERREUR

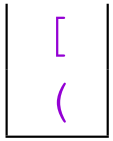
Deuxième Exemple

Entrée : ([( )])

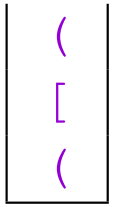
empile (



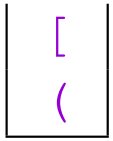
empile [



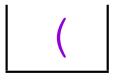
empile (



lu) → correspond ⟹ dépile



lu] → correspond ⟹ dépile



lu) → correspond ⟹ dépile

pile vide ⟹ **OK**

code C++

```
bool check(string s) {
  Pile p;
  for (unsigned int i(0); i < s.size(); ++i) {
    if ((s[i] == '(') || (s[i] == '['))
      p.empile(s[i]);
    else if (s[i] == ')') {
      if ((!p.est_vide()) && (p.top() == '('))
        p.depile();
      else
        return false;
    } else if (s[i] == ']') {
      if ((!p.est_vide()) && (p.top() == '['))
        p.depile();
      else
        return false;
    }
  }
  return p.est_vide();
}
```

code C++ (2)

Avec le standard `stack` :

`Pile` = `stack<Type>`, par exemple `stack<char>`
`empile` = `push`, par exemple `p.push(s[i])`
`est_vide` = `empty`, par exemple `p.empty()`
`depile` = `pop`, par exemple `p.pop()`
et `top` = `top`, par exemple `p.top()`

Jamila Sam
& Jean-Cédric Chappelier

Plan

- Structures de données abstraites
  - Listes
  - Piles
- Programmation générique
  - **Programmation générique : introduction, exemples**
  - Déclaration des modèles
  - Instanciation
  - Spécialisation
  - Compilation séparée

Programmation générique : introduction

Une cellule de notre liste chaînée de tout à l'heure se présentait comme suit :

```
// Une cellule de la liste
class Cellule {
public:
  //....
private:
        double donnee; // une donnee de type double
        PtrCell suite;
};
```

Si l'on veut une liste de `int` ?

- ☞ **c'est exactement le même code** pour `Liste` et `Cellule` sauf qu'il faut remplacer le type de la données pouvant être stockée dans une `Cellule`
- ☞ Duplication de code ! !

Programmation générique

L'idée de base est de **passer les types de données comme paramètres** pour décrire des traitements très généraux (« génériques »)
Il s'agit donc d'un *niveau d'abstraction supplémentaire*.

De tels modèles de classes/fonctions s'appellent aussi classes/fonctions génériques ou patrons (chablons), ou encore « ***template*** ».

Vous en connaissez déjà sans le savoir. Par exemple la « classe » `vector` n'est en fait pas une classe mais un modèle de classes : c'est le même modèle que l'on stocke des `char` (`vector<char>`), des `int` (`vector<int>`), ou tout autre objet.

Jamila Sam
& Jean-Cédric Chappelier

Un exemple

Prenons un exemple simple pour commencer :
une fonction échangeant la valeur de 2 variables.

Par exemple avec 2 entiers vous écririez une fonction comme :

```
// Echange la valeur de ses arguments
void echange(int& i, int& j) {
        int tmp(i);
        i = j;
        j = tmp;
}
```

Mais vous vous rendez bien compte que vous pourriez faire la même chose (le même **algorithme**) avec deux `double`, ou même deux objets quelconques, pour peu qu'ils aient un constructeur de copie (`Obj tmp(i);`) et un opérateur de copie (`operator=`).

Jamila Sam
& Jean-Cédric Chappelier

Exemple, suite...

L'écriture générale serait alors quelque chose comme :

```
// Echange la valeur de ses arguments
void echange(Type& i, Type& j) {
        Type tmp(i);
        i = j;
        j = tmp;
}
```

où `Type` est une représentation **générique** du type des objets à échanger.

La façon exacte de le faire en C++ est la suivante :

```
// Echange la valeur de ses arguments
template<typename Type>
void echange(Type& i, Type& j) {
        Type tmp(i);
        i = j;
        j = tmp;
}
```

...et fin

On pourra alors utiliser la fonction `echange` avec tout type/classe pour lequel le constructeur de copie et l'opérateur d'affectation (`=`) sont définis.

Par exemple :

```
int a(2), b(4);
echange(a,b);

double da(2.3), db(4.5);
echange(da,db);

vector<double> va, vb;
echange(va,vb);

string sa("ca marche"), sb("coucou");
echange(sa, sb);
```

Généralisation aux classes

Ce que l'on a fait ici avec une fonction, on peut le généraliser à n'importe quelle classe.

On pourrait
par exemple vouloir créer une classe qui réalise une paire d'objets :

```
template<typename T1, typename T2>
class Paire {
public:
    Paire(const T1& un, const T2& deux)
      : premier(un), second(deux) {}
    virtual ~Paire(){}
    T1 get1() const { return premier; }
    T2 get2() const { return second;  }
    void set1(const T1& val) { premier = val; }
    void set2(const T2& val) { second  = val; }
protected:
    T1 premier;
    T2 second;
};
```

Généralisation aux classes (2)

et par exemple créer la classe « paire `string`–`double` » :

```
Paire<string,double>
```

ou encore la classe « paire `char`–`unsigned int` » :

```
Paire<char,unsigned int>
```

Note : un tel modèle de classe existe dans la librairie standard : `pair` (défini dans `<utility>`).

Les modèles de classes sont donc **un moyen condensé d'écrire plein de classes potentielles à la fois**.

(de même que les modèles de fonctions/méthodes sont un moyen condensé d'écrire **plein** de fonctions/méthodes potentielles à la fois)

Jamila Sam
& Jean-Cédric Chappelier

Déclaration d'un modèle

Pour déclarer un modèle de classe ou de fonction, il suffit de faire précéder sa déclaration du mot clé `template` suivit de ses arguments (qui sont donc des noms génériques de type) suivant la syntaxe :

```
template<typename nom1, typename nom2, ...>
```

Exemple :

```
template<typename T1, typename T2>
class Paire {
...
```

Les types ainsi déclarés (paramètres du modèle) peuvent alors être utilisés dans la définition qui suit, exactement comme tout autre type.

Note : on peut aussi utiliser le mot `class` à la place de `typename`, par exemple :

```
template<class T1, class T2>
class Paire {
...
```

Déclaration d'un modèle (2)

Il est également possible de définir des types par défaut, avec la même contrainte que pour les paramètres de fonction : les valeurs par défaut doivent être placées en dernier.

Exemple :

```
template<typename T1, typename T2 = unsigned int>
class Paire {
...
```

qui permettrait de déclarer la classe « paire `char`–`unsigned int` » simplement par :

```
Paire<char>
```


Jamila Sam
& Jean-Cédric Chappelier

Définitions externes des méthodes de modèles de classes

Si les méthodes d'un modèle de classes sont définies en dehors de cette classe, elle devront alors aussi être définies comme modèle et être précédées du mot clé `template`, mais...

...il est **de plus absolument nécessaire** d'**ajouter les paramètres du modèle** (les types génériques) **au nom de la classe**

[pour bien spécifier que dans cette définition c'est la classe qui est en modèle et non la méthode.]

☞ exemple sur le transparent suivant

Jamila Sam
& Jean-Cédric Chappelier

Définitions externes des méthodes de modèles de classes

Exemple :

```
template<typename T1, typename T2> class Paire {
public:
  Paire(const T1&, const T2&);
...
};

// definition du constructeur
template<typename T1, typename T2>
// le constructeur du modele de classe Paire
// parametr'e par T1 et T2
Paire<T1,T2>::Paire(const T1& un, const T2& deux)
  : premier(un), second(deux) { }
```


Jamila Sam
& Jean-Cédric Chappelier

Instanciation des modèles

La définition des modèles ne génère en elle-même aucun code : c'est juste une **description de plein de codes potentiels**.

Le code n'est produit que lorsque tous les paramètres du modèle ont pris chacun un type spécifique.

Lors de l'utilisation d'un modèle, il faut donc fournir des valeurs pour tous les paramètres (au moins ceux qui n'ont pas de valeur par défaut). On appelle cette opération une **instanciation** du modèle.

L'instanciation peut être implicite lorsque le contexte permet au compilateur de décider de l'instance de modèle à choisir.

Par exemple, dans le code :

```
double da(2.3), db(4.5);
echange(da,db);
```

il est clair (par le contexte) qu'il s'agit de l'instance `echange<double>` du modèle `template<typename T> void echange(T&,T&);` qu'il faut utiliser.

Instanciation des modèles (2)

Mais dans la plupart des cas, on **explicite l'instanciation** lors de la déclaration d'un objet.

C'est ce qui vous faites lorsque vous déclarez par exemple `vector<double> tableau;`

Il suffit dans ce cas de spécifier le(s) type(s) désiré(s) après le nom du modèle de classe et entre `<>`.

L'instanciation explicite peut aussi être utile dans les cas où le contexte n'est pas suffisamment clair pour choisir.

Par exemple avec le modèle de fonction

```
template <typename Type>
Type monmax(const Type& x, const Type& y) {
   if (x < y) return y;
   else       return x;
 }
```

l'appel `monmax(3.14, 7);` est ambigu. Il faudra alors écrire `monmax<double>(3.14, 7);`

Modèles, surcharge et spécialisation

Les modèles de fonctions peuvent très bien être surchargés comme les fonctions usuelles (puisque, encore une fois, ce sont juste une façon condensée d'écrire plein de fonctions à la fois).

Par exemple :

```
template<typename Type>
void affiche(const Type& t) {
  cout << "J'affiche " << t << endl;
}
// surcharge pour les pointeurs : on prefere ici
//  ecrire  le contenu plutot que l'adresse.
template<typename Type> void affiche(Type* t) {
  cout << "J'affiche " << *t << endl;
}
```

Note : on aurait même pu faire mieux en écrivant :

```
template<typename Type> void affiche(Type* t) {
        affiche<Type>(*t);
}
```

qui fait appel au premier modèle.

Modèles, surcharge et spécialisation (2)

Mais les modèles (y compris les modèles de classes) offrent un mécanisme supplémentaire : la **spécialisation** qui permet de définir une version particulière d'une classe ou d'une fonction pour un choix spécifique des paramètres du modèle.

Par exemple, on pourrait spécialiser le second modèle ci-dessus dans le cas des pointeurs sur des entiers :

```
template<> void affiche<int>(int* t) {
    cout << "J'affiche le contenu d'un entier: ";
         << *t << endl;
}
```

La spécialisation d'un modèle (lorsqu'elle est totale) se fait en :

- ajoutant `template<>` devant la définition
- nommant explicitement la classe/fonction spécifiée
  C'est le `<int>` après `affiche` dans l'exemple ci-dessus.

Exemple de spécialisation de classe

```
template<typename T1, typename T2>
class Paire {
...
};
// specialisation pour les paires <string,int>
template<> class Paire<string,int> {
public:
  Paire(const string& un, int deux)
    : premier(un), second(deux) {}
  virtual ~Paire() {}
  string get1() const { return premier; }
  int get2()    const { return second;  }
  void set1(const string& val) { premier = val; }
  void set2(int val) { second  = val; }

  // une methode de plus
  void add(int i) { second += i; }

protected:
  string premier;
  int    second;
};
```


Jamila Sam
& Jean-Cédric Chappelier

Spécialisation : remarques

Note 1 : La spécialisation peut également s'appliquer uniquement à une méthode d'un modèle de classe sans que l'on soit obligé de spécialiser toute la classe.
Utilisée de la sorte, la spécialisation peut s'avérer particulièrement utile.

Note 2 : La spécialisation n'est pas une surcharge car il n'y a pas génération de plusieurs fonctions de même nom (de plus que signifie une surcharge dans le cas d'une classe ?) mais bien une **instance spécifique** du modèle.

Note 3 : il existe aussi des spécialisations partielles (de classe ou de fonctions), mais cela nous emmènerait trop loin dans ce cours.

Modèles de classes et compilation séparée

Les modèle de classes doivent nécessairement être définis au moment de leur instanciation afin que le compilateur puisse générer le code correspondant.

Ce qui implique, lors de compilation séparée, que les fichiers d'en-tête (`.h`) doivent contenir non seulement la déclaration, mais également la définition complète de ces modèles !!

On ne peut donc pas séparer la déclaration de la définition dans différents fichiers... Ce qui présente plusieurs inconvénients :

- Les mêmes instances de modèles peuvent être compilées plusieurs fois,
- et se retrouvent en de multiples exemplaires dans les fichiers exécutables.
- On ne peut plus cacher leurs définitions (par exemple pour des raisons de confidentialité, protection contre la concurrence, etc...)

Templates

Déclarer un modèle de classe ou de fonction :

```
template<typename nom1, typename nom2, ...>
```

Définition externe des méthodes de modèles de classes :

```
template<typename nom1, typename nom2, ...>
NomClasse<nom1, nom2, ...>::NomMethode(...
```

Instanciation : spécifier simplement les types voulus après le nom de la classe/fonction, entre `<>` (Exemple : `vector<double>`)

Spécialisation (totale) de modèle pour les types *type1*, *type2*... :

```
template<> NomModele<type1,type2,...> ...suite
de la declaration...
```

Compilation séparée : pour les templates, il faut tout mettre (déclarations et définitions) dans le fichier d'en-tête (`.h`).

Ce que j'ai appris aujourd'hui

- Les bases de la **formalisation des données** : les structures de données abstraites
- Les deux structures de données abstraites les plus utilisées en informatique (en plus des tableaux et des types élémentaires) : les **listes chaînées** et les **piles**
- A pouvoir faire des modèles génériques de traitements ou de classes, indépendamment du type, ce que l'on appelle de la **programmation générique**.
  - Comment déclarer de tels modèles
  - Comment en créer des instances
  - Comment spécialiser certains modèles

La suite

- Outils de la librairie standard