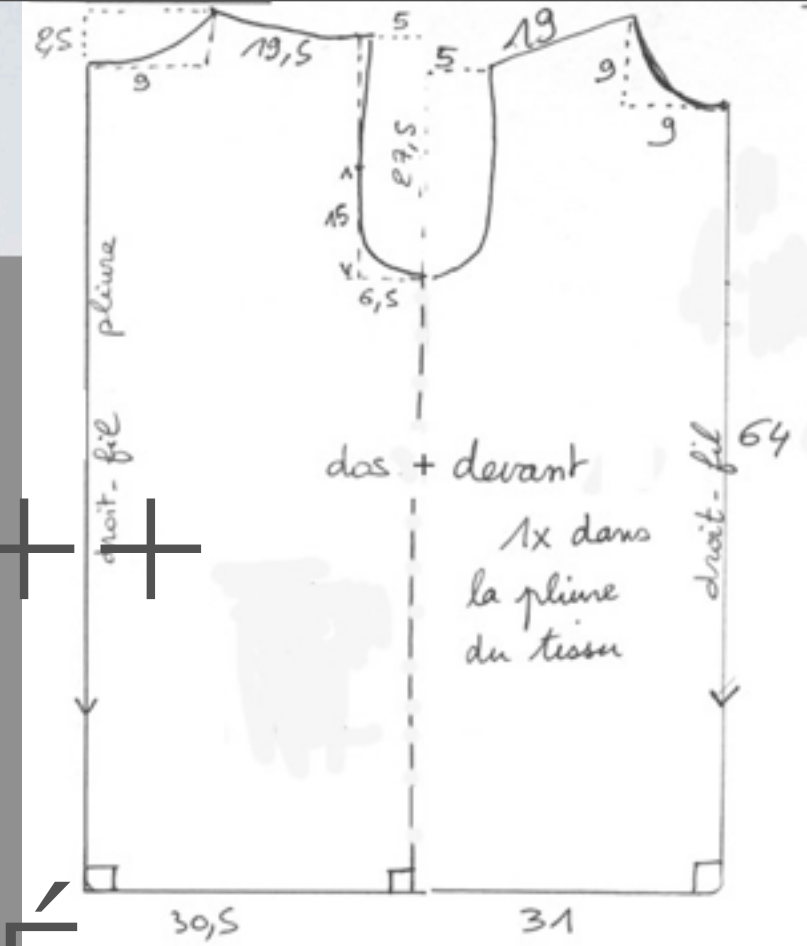


# LE LANGAGE C++ MASTER I LA GÉNÉRICITÉ

Jean-Baptiste.Yunes@univ-paris-diderot.fr  
U.F.R. d'Informatique  
Université Paris Diderot - Paris 7

10 nov. 2012



# LES MODÈLES DE CLASSES



- Le problème des containers...
  - En programmation orientée objet, un container est un objet fournissant des services de gestion d'une collection d'autres objets
- C'est une structure récurrente, comment éviter de la redéfinir à chaque fois ?

- Exemple: je veux représenter la liste des membres d'un club de hockey sur gazon

```
class HockeysurGazon {};  
class ClubDeHockey {  
private:  
    HockeysurGazon **membres;  
    int nombre;  
public:  
    void addMembre(const HockeysurGazon &h) {  
        membres[nombre++] = &h;  
    }  
    HockeysurGazon &membreAt(int i) const {  
        return *membres[i];  
    }  
};
```

- Exemple: je veux représenter la liste des membres d'un club de bridge

```
class JoueurDeBridge {};  
class ClubDeBridge {  
private:  
    JoueurDeBridge **membres;  
    int nombre;  
public:  
    void addMembre(const JoueurDeBridge &h) {  
        membres[nombre++] = &h;  
    }  
    JoueurDeBridge &membreAt(int i) const {  
        return *membres[i];  
    }  
};
```



- Exemple: je veux représenter la liste des étudiants du cours de C++

```
class EtudiantCPP {};  
class CoursCPP {  
private:  
    EtudiantCPP **membres;  
    int nombre;  
public:  
    void addMembre(const EtudiantCPP &h) {  
        membres[nombre++] = &h;  
    }  
    EtudiantCPP &membreAt(int i) const {  
        return *membres[i];  
    }  
};
```

- La modification est mineure... Mais manuellement elle présente des inconvénients (copie de bug, etc.)

```
class CollectionDeChoses {  
private:  
    Chose **listeDeChoses;  
    int nombreDeChoses;  
public:  
    void addElement(const Chose &c) {  
        listeDeChoses[nombreDeChoses++] = &c;  
    }  
    Chose &choseAt(int i) const {  
        return *listeDeChoses[i];  
    }  
};
```

- La déclaration C++ d'une collection de choses est :

```
template < class CHOSE > class Collection {  
private:  
    CHOSE **listeDeChoses;  
    int nombreDeChoses;  
public:  
    void addElement(const CHOSE &c) {  
        listeDeChoses[nombreDeChoses++] = &c;  
    }  
    CHOSE &choseAt(int i) const {  
        return *listeDeChoses[i];  
    }  
};
```

- La définition d'une collection de choses est :

```
Collection<JoueurDeHockey> clubDeHockey;  
Collection<Etudiant> coursCPP;  
Collection< Collection<Etudiant> > cursus;
```



- L'intrusion...
  - Comment permettre la définition d'un opérateur d'entrée/sortie ?
  - On réalise que parfois des contraintes existent sur le type utilisé comme paramètre... Par exemple



- La spécialisation d'un modèle de classe

```
template < class E > class Paire {  
private:  
    E e1, e2;  
public:  
    void addPremier(const E &e) { e1 = e; }  
    void addSecond(const E &e)  { e2 = e; }  
};
```



- Est-ce efficace avec des booléens ?
- On aimerait les stocker un booléen par bit...

- La spécialisation d'un modèle de classe

```
template <> class Paire<bool> {  
private:  
    int stock;  
    void stockBit(int i,bool value) {  
        stock = (stock & ~(1<<i)) | ((value?1:0)<<i);  
    }  
public:  
    void addPremier(const bool &e) { stockBit(0,e); }  
    void addSecond(const bool &e) { stockBit(1,e); }  
};
```

- La spécialisation intervient lorsqu'il est nécessaire d'adapter la représentation, le fonctionnement d'une méthode ou l'interface pour un type donné
- Attention : il faut **TOUT** redéfinir

- La spécialisation d'un modèle de classe

```
template <class T> class Paire {  
private:  
    T c1, c2;  
public:  
    bool egaux() { return c1==c2; }  
};
```

```
template <> class Paire<char> {  
private:  
    char c1, c2;  
public:  
    bool egaux() { return enMaj(c1)==enMaj(c2); }  
};
```

- Pour un type classe, on peut «!reporter!» le problème dans la classe choisie...

- La spécialisation d'un modèle de classe

```
template <class T> class Paire {  
private:  
    T c1, c2;  
public:  
    bool egaux() { return c1==c2; }  
};
```

```
template <> class Paire<Lunette> {  
private:  
    Lunette gauche, droite;  
public:  
    double correctionGauche() {  
        return gauche.getCorrection();  
    }  
    double correctionDroite() {  
        return droite.getCorrection();  
    }  
};
```





- La spécialisation partielle d'un modèle de classe

```
template <class T> class Paire {  
private:  
    T c1, c2;  
public:  
    T plusGrand() { return c1>c2 ? c1 : c2; }  
};
```

```
template <class T> class Paire<T *> {  
private:  
    T *c1, *c2;  
public:  
    T plusGrand() { return *c1>*c2 ? *c1 : *c2; }  
};
```

- Les modèles de classe et les valeurs par défaut
  - Comme pour les arguments de fonctions, il est possible de définir des valeurs par défaut pour les types...

```
template <class T=int> class Paire {  
private:  
    T c1, c2;  
public:  
    T plusGrand() { return c1>c2 ? c1 : c2; }  
};
```

```
...  
Paire<> unePaire;  
Paire< Claques > uneAutrePaire;  
Paire< Manche > encoreUnePaire;  
...
```

- Les modèles de classe et les paramètres valeurs

```
template <int N=1> class Famille {  
private:  
    Parent *parents[2];  
    Enfant *enfants[N];  
};
```

...

```
Famille<> uneFamilleAvecUnEnfant;
```

```
Famille<2> uneFamilleMoyenne;
```

```
Famille<10> uneFamilleNombreuse;
```

...

- Attention : Il s'agit bien de trois types différents!

- Attention :
  - La déclaration et la définition (le code) des templates doivent être disponibles à l'instanciation, c'est pourquoi le code des templates est en général contenu dans le fichier d'entête.
  - De plus, le compilateur ne vérifie que la correction syntaxique des templates, c'est uniquement à l'instanciation (lorsque le compilateur crée effectivement la classe correspondante) que la vérification du typage s'effectue... Par conséquent, c'est là que les erreurs se produisent généralement!!!

ET LES  
ENTRÉES/  
SORTIES ?



- un petit exemple...

```
template <class T> class A {
private:
    T *e;
public:
    A(T *e) { this->e = e; }

};

template <class X> ostream &operator << (ostream &os,const A<X> &a) {
    os << *(a.e); return os;
}

class B {};
ostream &operator<<(ostream &os,const B &b) { // intrusion
    os << "BBBB"; return os;
}

int main() {
    int i=12;
    A<int> unA(&i);
    cout << unA << endl;
    B unB;
    A<B> a2(&unB);
    cout << a2 << endl;
}
```

```
[Trotinette:~] yunes% test
12
BBBB
[Trotinette:~] yunes%
```

# DES MODÈLES DE CLASSES À LA GÉNÉRICITÉ

- Modèles de classe :
  - Inconvénient majeur :
    - Le modèle est instancié pour chaque nouveau type
      - multiplication du code généré...
      - mauvaise factorisation

```
template <class T> class Conteneur {  
private:  
    T *element;  
public:  
    T *getElement() { return element; }  
    void setElement(T &e) { element = &e; }  
};
```

```
class ConteneurGenerique {  
private:  
    void *element;  
public:  
    void *getElement() { return element; }  
    void setElement(void *e) { element = e; }  
};
```

Pfff! Pas de  
référence sur

- Problème : les types ont disparu...

```
class ConteneurGenerique {  
private:  
    void *element;  
protected:  
    void *getElement() { return element; }  
    void setElement(void *e) { element = e; }  
};
```

Notez le petit changement

- Ok on peut les réintroduire par héritage...

```
class ConteneurDeSchmilblick :  
{  
public:  
    Schmilblick *getElement() {  
  
    }  
    void setElement(Schmilblick *e) {  
        ConteneurGenerique::setElement(e);  
    }  
};
```



- Le mieux est encore d'en faire un modèle...

```
template <class T>
class Conteneur : private ConteneurGenerique {
public:
    T *getElement() {
        return (T *)ConteneurGenerique::getElement();
    }
    void setElement(T *e) {
        ConteneurGenerique::setElement(e);
    }
};

...
```

- Cette fois on bénéficie de l'héritage et donc on réutilise le code existant...



# LES MODÈLES DE FONCTIONS

- À l'image des modèles de classes, on peut avoir envie de d'abstraire la définition de certaines fonctions, c'est d'ailleurs ce que l'on fait en algorithmique où l'on travaille généralement sur des données abstraites
- Écrire enfin des algorithmes... (?!)

- Prenons le calcul du maximum de deux *choses*, en C++ cela s'écrit :

```
template <class T> T max(T &t1,T &t2) {  
    return t1>t2?t1:t2;  
}
```

- L'utilisation peut être :

```
void main() {  
    int i=4, j=5;  
    cout << max(i,j) << endl; // instancié avec int  
    float f=4.5, g=6.7;  
    cout << max(f,g) << endl; // instancié avec float  
}
```

- En cas de doute on peut forcer le compilateur :

```
cout << max<int>(4,5.2f) << endl;  
cout << max<float>(3,1.2f) << endl;
```

- La surcharge de modèles de fonctions est autorisée :

```
template <class T> T max(T *t1,unsigned int taille) {  
    T leMax = t1[0];  
    for (int i=1; i<taille; i++) {  
        if (t[i]>leMax) leMax=t[i];  
    }  
    return leMax;  
}
```



- On peut avoir besoin de surcharger un modèle de fonction :

```
// cas standard
template <class T> T add(T t1,T t2) {
    return t1+t2;
}

// le cas des pointeurs...
template <class T> T *add(const T *t1,const T *t2) {
    return new T(*t1 + *t2);
}
```

- On peut avoir besoin de spécialiser un modèle de fonction :
- Dans le cas de chaînes de caractères :

```
template <> char *add<char>(const char *c1,const char
*c2) {
    char *r = new char[strlen(c1)+strlen(c2)+1];
    strcpy(r,c1); strcat(r,c2);
    return r;
}
```

- **Attention :**

l'algorithme de  
sélection de la fonction  
adaptée à l'appel est  
particulier et les effets  
sont souvent  
inattendus...

- Dans le cas des fonctions :
  - ne pas confondre fonction de première classe et modèle de fonction
  - ne pas confondre modèle de fonction et spécialisation
  - ne pas confondre les surcharges dans ces différents *mondes*

- On cherche d'abord l'existence d'une fonction ordinaire correspondant à l'appel
- sinon on recherche parmi les modèles de fonction
- lorsqu'un modèle est choisi, on regarde s'il existe une spécialisation de ce modèle...

- Un exemple tordu (merci Dimov et Abrahams)

```
template <class T> void f(T);
```

```
template <class T> void f(T *);
```

```
template <> void f<int>(int *);
```

est la spécialisation de

```
int *pi;
```

```
f(pi);
```

Tout  
semble normal...

- Un exemple tordu (merci Dimov et Abrahams)



```
template <class T> void f(T);
```

```
template <class T> void f(T *);
```

```
template <> void f<int>(int *);
```

est la spécialisation de

```
int *pi;  
f(pi);
```

Tout  
semble normal...



- Un exemple tordu (merci Dimov et Abrahams)

```
template <class T> void f(T);
```

```
template <> void f<int *>(int *);
```

est la spécialisation de

```
template <class T> void f(T *);
```

```
int *pi;
```

```
f(pi);
```

Wharglll !!!

- Un exemple tordu (merci Dimov et Abrahams)



```
template <class T> void f(T);
```

```
template <> void f<int *>(int *);
```

est la spécialisation de

```
template <class T> void f(T *);
```

```
int *pi;  
f(pi);
```

Wharglll !!!

- Leçon de Morale :
  - si vous voulez modifier le comportement d'un modèle de fonction et que cette modification participe à l'algorithme de résolution de la surcharge ou qu'elle soit choisie en cas de correspondance exacte :
    - faites-en une fonction ordinaire pas une spécialisation
  - si vous créez de la surcharge, évitez la spécialisation