Programmation orientée objet

Fonctions génériques

Samuel Kadoury & Claudy Picard

Polymorphisme de compilation

- Une fonction générique (ou modèle de fonction, ou patron de fonction) est une fonction qui accepte plusieurs types de paramètre
- Quel que soit le type de paramètre qui lui est passé, elle fait exactement le même traitement
- Cela signifie qu'à la compilation, le **compilateur crée autant de versions de la fonction que nécessaire**

Définition de fonction générique

- On précède la fonction d'une déclaration ayant la forme suivante:

```
template< typename T1, typename T2, ..., typename T3 >
```

où T1, T2, .. T3 sont des paramètres représentant des types

Exemple de fonction générique

- Soit une fonction qui affiche toutes les valeurs d'un vecteur:

Il faudrait spécifier un type ici.

```
void print(ostream& out, const vector<???>& data){
    out << "[";
    for (??? item : data)
        cout << item << " ";

    out << "]";
}
```

Exemple de fonction générique (suite)

```
void print(ostream& out, const vector<int>& data){
    out << "[";
    for (int item : data)
        cout << item << " ";

    out << "]";
}
void print(ostream& out, const vector<Employee>& data){
    out << "[";
    for (Employee item : data)
        cout << item << " ";

    out << "]";
}
```

Exemple de fonction générique (suite)

- Notre exemple sera donc écrit de la manière suivante:

```
template< typename T>
void print(ostream& out, const vector<T>& data){
    out << "[";
    for (T item : data)
        cout << item << " ";

    out << "]";
}
```

Exemple de fonction générique (suite)

- Exemple d'utilisation de notre fonction générique:

```
int main() {
    vector<int> vecInt = { 0,1,2 };
    vector<Employee> vecEmployee = { Employee(),
       Employee(), Employee() };

    print(cout, vecInt);
    print(cout, vecEmployee);
}
```

Le compilateur créera deux versions de la fonction: une pour les *int* et une autre pour les *Employee.*

Le compilateur trouve le type correct “automatiquement”

- Si le type générique est un paramètre de la fonction, le compilateur sera en mesure de trouver le type automatiquement lors de son utilisation
- Par contre, si le type générique est caché dans l’implémentation de la fonction, il faut l’expliciter
- **NOTE**: On peut toujours expliciter le type générique, même si le compilateur est en mesure de le déduire

Exemple 1 - Expliciter le type générique

```
template<typename T>
void f() {
   T item;
}
```

Le type générique est caché dans l'implémentation de la fonction.

```
int main() {
   f();
   f<int>();
}
```

Il y a une erreur de compilation puisque le compilateur ne peut pas déduire le type générique, **il faut absolument l'expliciter**

Exemple 2 - Expliciter le type générique

```
template<typename T>
T f() {
   T item;
    return item;
}

int main() {
   int item = f();
   int item = f<int>();
}
```

Exemple 3 - Expliciter le type générique

```
template<typename T, typename S>
T f2(S itemS) {
    cout << itemS << endl;
    T itemT;
    return itemT;
}

int main() {
    int item = f("allo");
    int item = f<int>("allo");
}
```

Contrainte câchée sur les types

- Attention, il y a une condition pour qu'on puisse appeler une fonction générique
- **Dans la fonction, toutes les opérations effectuées sur un paramètre de type générique doivent être valides pour le type en question**

Contrainte câchée sur les types (exemple 1)

```
template< typename T>
void print(ostream& out, const vector<T>& data){
    out << "[";
    for (T item : data)
        cout << item << " ";

    out << "]";
}
```

Si item est un objet, il faut que l'opérateur << soit défini pour sa classe

Contrainte câchée sur les types (exemple 2)

```
template< typename T >
T maximum(const T& left, const T& right){
    if (left < right)
        return right;
    return left;
}
```

Si ces deux paramètres sont des objets, il faut que l'opérateur < soit défini pour leur classe

Programmation orientée objet

Classes génériques

Samuel Kadoury & Claudy Picard

Classes génériques

- Il arrive que des classes soient identiques, à l'exception des types des attributs, ou encore des types des paramètres des méthodes
- Les classes de la librairie standard comme les vecteurs ou les pointeurs intelligents sont génériques!

Définition d'une classe générique

- Une classe générique est définie de la même manière qu'une fonction générique:

```
template<typename T>
class GestionnaireCollection {
public:
    bool trouverElement(const T& element);
    void insererElement(const T& element);
private:
    vector<T> collection_;
};
```

Implémentation d'une classe générique

```
template<typename T>
bool GestionnaireCollection<T>::trouverElement(const T& element) {
    for (const T& i : collection_) {
        if (i == element) {
            return true;
        }
    }
    return false
}

template<typename T>
void GestionnaireCollection<T>::insererElement(const T& element) {
    conteneur_.push_back(element);
}
```

il faut le remettre au début de chaque implementation.

L'implémentation d'une classe générique doit être dans le .h!

Il n'y a pas de .cpp

Classe générique avec méthode générique

```
template<typename T>
class GestionnaireCollection {
private:
    template<typename Comparateur>
    bool trouverElement(const T& element, Comparateur comp);
    void insererElement(const T& element);
public:
    vector<T> collection_;
};
```

Classe générique avec méthode générique

```
template<typename T>
template<typename Comparateur>
bool GestionnaireCollection<T>::trouverElement(const T& element,
                                               Comparateur comp) {
    for (const T& i : collection_) {
        if (comp(i, element)) {
            return true;
    }
}

    return false
}
```

Utilisation d'une classe générique

Contrairement aux fonctions génériques!

- **Chaque fois qu'on déclare un objet d'une classe générique, il faut toujours spécifier tous les types qui sont paramétrisés dans le modèle**
- Par exemple, on déclare de la manière suivante une paire de deux entiers:

Le compilateur produit une version spécifique de la classe pour chaque déclaration de celle-ci.

```
pair< int, int > unePaire(3,4);
```

- Pour déclarer une paire composée d'un entier et d'une chaîne de caractères :

```
pair< int, string > unePaire(3,"Michel");
```

Argument non typé

- Dans certains cas, on veut spécifier non pas un type variable, mais une **valeur** variable
- Exemple d'une classe matrice:

```
template< typename T, int ROWS, int COLUMNS >
class Matrix{
    ...
private:
    T data_[ROWS][COLUMNS];
};
```

Argument non type (suite)

- Ainsi, on pourra déclarer des matrices de types différents et de tailles différentes:

Le compilateur créera une version de la classe générique pour chaque combinaison de T, ROWS et COLUMNS différentes

```
int main() {
    Matrix< int, 3, 4 > a;
    Matrix< double, 3, 4 > b;
    Matrix< string, 2, 3 > c;
}
```

Programmation orientée objet

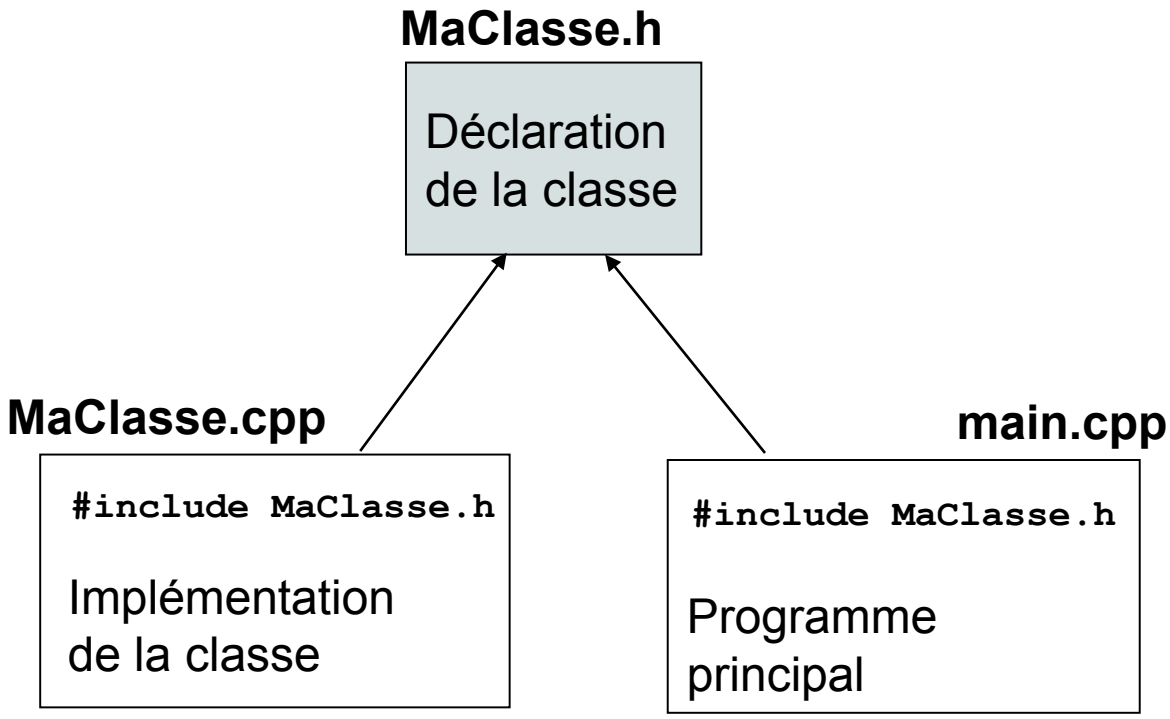
Compilation séparée

Michel Gagnon & Claudy Picard

Compilation séparée

- Pour compiler un programme principal, il suffit de connaître les déclarations de classes
- Les implémentations des classes seront compilées séparément
- Une fois compilés le programme principal et les implémentations de classe, les fichiers ainsi obtenus sont combinés pour produire l'application exécutable (étape de *liaison*)

Compilation séparée (suite)



Compilation séparée et classes génériques

- Liste.h:

```
template<typename T>
class Liste {
public:
    bool trouver(const T& t);
};
```

- main.cpp:

```
int main() {
    Liste<Point> listeDePoints;
    Point p;
    listeDePoints.trouver(p);
}
```

- Liste.cpp:

```
template<typename T>
bool Liste<T>::trouver(const T& t) {
    ...
    t.uneMethode();
    ...
}
```

Cette méthode doit exister pour la classe T.

La déclaration suivante devrait etre refusée par le compilateur, si la classe Point ne possède pas la méthode uneMethode()

Supposons que l'implémentation est dans le .cpp

liste.h

```
template<typename T>
class Liste {
public:
      bool trouver(const T& t);
};
```

point.h

```
class Point{
public:
      Point();
      double getX();
      double getY();
      ...
};
```

Lorsqu'on compile main.cpp, on sait que la méthode trouver() existe, mais on ne sait pas qu'elle exige que l'objet de la classe T contienne la méthode uneMethode().

La classe Point ne contient pas la méthode uneMethode().

main.cpp

```
#include "Liste.h"
#include "Point.h"

int main() {
      Liste<Point> listeDePoints;
      Point p;
      listeDePoints.trouver(p);
}
```

liste.cpp

```
template<typename T>
bool Liste<T>::trouver(const T& t) {
      ...
      t.uneMethode();
      ...
}
```

Compilation séparée et classes génériques (suite)

- Si la classe est générique, il n'est donc pas suffisant d'inclure seulement la déclaration de la classe
- Le compilateur **doit** savoir si toutes les méthodes implémentées sont valides pour le type spécifié

Supposons que l'implémentation est dans le .h

liste.h

```
template<typename T>
class Liste {
public:
      bool trouver(const T& t);
};

template<typename T>
bool Liste<T>::trouver(const T& t) {
      ...
      t.uneMethode();
      ...
}
```

Lorsqu'on compile main.cpp, on sait maintenant que la méthode trouver() exige que l'objet de la classe T contienne la méthode uneMethode().

point.h

```
class Point{
public:
      Point();
      double getX();
      double getY();
      ...
};
```

main.cpp

```
#include "Liste.h"
#include "Point.h"

int main() {
      Liste<Point> listeDePoints;
      Point p;
      listeDePoints.trouver(p);
}
```

Compilation séparée et classes génériques (suite)