Template C++ par l'exemple

Template C++ par l'exemple

Un exemple classique de l'intérêt des classes Template C++ (dont le principe a été repris dans la dernière version de Java), serait la classe représentant une pile, dont on aurait besoin pour réaliser des piles d'entiers et des piles de chaînes.

On pourrait les implémenter comme ci-dessous. L'implémentation n'est pas très générique, pas très robuste, mais c'est pour l'exemple. class IntStack {
public:
IntStack() { top = -1; }
void push(int i)
{ st[++top] = i; }
int pop()
{ return st[top--]; }
private:
int top;
int st[100];
}; class StringStack {
public:
StringStack() { top = -1; }
void push(string i)
{ st[++top] = i; }
string pop()
{ return st[top--]; }
private:
int top;
string st[100];
};

Le programme principal (main) utiliserait ces classes comme ci-dessous: {mostip} IntStack ii;
StringStack ss;
ii.push(25);
ss.push("Hello"); {/mostip}

Si on note que la seule différence entre ces classes (outre le nom de la classe) est le type des données qui sont mises dans la pile.

C++ permet de définir une classe en fonction d'un type de générique, c'est à dire paramétrable, ce qui permet de représenter une pile de n'importe quel type possible (y compris défini par l'utilisateur). La déclaration ressemblerait à ceci : #ifndef STACK_H
#define STACK_H
template <class T> class Stack {
public:
Stack() { top = -1; }
void push(T i)
{ st[++top] = i; }
T pop()
{ return st[top--]; }
private:
int top;
T st[100];
};

#endif
Le T représente le type générique de la pile, à préciser à l'utilisation entre des symboles "". On voit que le code de la class diffère peu, elle dépend maintenant d'un type T indéfini. Le programme principal déclarerait et utiliserait les piles comme suit : {mostip}
Stack<int> ii;
Stack<string> ss;
ii.push(25);

ss.push("Hello");
{/mostip} Il n'y a jamais de fichier d'implémentation (*.cpp) pour une classe de template (on dit aussi "patron de classe"). Toutes les méthodes membres doivent être déclarées dans le fichier d'en-tête (*.h).

Conclusion
Les templates sont très pratiques, j'en parle car on les rencontre souvent, par exemple quand on utilise Corba.

Solutions alternatives:
- pré-procésseur (un #define paramétré) - pas très joli mais comparable.
- Utilisation d'une référence générique (void*, Object) + cast à tout va !
- polymorphisme associé à de l'héritage multiple !!!

De mon point de vue, c'est une facilité qui peut malheureusement amener un mauvais design d'application, dans le cas où il remplace un vrai polymorphisme associé à de l'héritage multiple. On ne devrait les utiliser que dans le cas de classes utilitaires pour lesquelles la performance maximale est requise.