La programmation "objet"

algorithmique classique ("propriétaire")+
encapsulation+
héritage+
généricité => algorithmique objet.

La généricité

- permet de ***réutiliser*** un algorithme avec des types de données différents ***sans*** utiliser le Copier - Coller
- garantit la "qualité" d'un algorithme indépendamment de la complexité des objets qu'il gère

La généricité de type

- tous les exemples sont compilés sans erreur par g++

```
// Version non générique
#include <iostream.h>

typedef int Info;
typedef int Clef;
Clef donneClef(const Info & uneInfo){return uneInfo;}

const Info & recherche(const Info * unTabInfo, const Clef & laClef) throw (const char *) {
    for (int i = 0; i < 10; i++)
        if (donneClef(unTabInfo[i]) == laClef) return unTabInfo[i];
    throw "Information absente";
    }

void main(){
    Info unTab [10];
    for (int i = 0; i < 10; i++) unTab[i] = 2*i;
    cout << "Je cherche 4 et je trouve : " << recherche(unTab, 4) << endl;
    }
```

Version générique sur le type


```
#include <iostream.h>

template <class Info, class Clef>
    Clef donneClef(const Info & uneInfo){return uneInfo;}

template <class Info, class Clef>
const Info & recherche(const Info * unTabInfo, const Clef & laClef) throw (const char *) {
    for (int i = 0; i < 10; i++)
        if (donneClef<Info, Clef>(unTabInfo[i]) == laClef) return unTabInfo[i];
    throw "Information absente";
    }

void main(){
    int unTab [10];
    for (int i = 0; i < 10; i++) unTab[i] = 2*i;
    cout << "Je cherche 4 et je trouve : " << recherche <int, int> (unTab, 4) << endl;
    cout << "Je cherche 4 et je trouve : " << recherche(unTab, 4) << endl;
    }
```

La généricité en C++

- Les spécifications et les corps des sous-programmes génériques sont précédés de :
  `template <class unNomDeClasse, …>`
- Lors de l'utilisation du sous-programme, son nom est :
  `sonNom <unNomDeClasseExistant, …>`

Une classe banale (spécifs)


```
#ifndef LISTE_H
#define LISTE_H

#include <iostream.h>

 template <class Info>

class Liste {
private :
   Info valeur;
   bool vide;
   Liste * suivante;
public :
   Liste() {vide = true; suivante = NULL;}
   ~Liste() {if (suivante) delete suivante;}

   void ajoute (const Info &);
   };
#endif
```

Dans le programme utilisateur :

Liste <int> maListe;

maListe.ajoute(4);

Une classe banale (corps)


```
#include "liste.h"

template <class Info>
void Liste::ajoute(const Info & arg) {
   if (vide) {vide = false; valeur = arg; return;}
   if (arg<=valeur){     // insertion en tête de liste
       Liste & laNouvelle = *new Liste();
       laNouvelle.valeur = valeur;
       laNouvelle.vide = false;
       laNouvelle.suivante = suivante;
       valeur = arg; suivante = &laNouvelle;
       return;
       }
   if (!suivante)  suivante=new Liste();
   suivante->ajoute(arg);
   }
```

En Java

- toutes les classes dérivent de la classe `Object`.

```
import java.util.ArrayList;

class gene{
  static void main(String [] args){
    ArrayList unTableau = new ArrayList();

    for (int i = 0; i < 10; i++)
        unTableau.add(new Integer(i*2));

     unTableau.add(2, new
   Integer(((Integer)(unTableau.get(2))).intValue()+4));
    System.out.println(unTableau);
    }
```

[0, 2, **8**, 4, 6, 8, 10, 12, 14, 16, 18]

La généricité de structure

- la ***généricité de type*** permet de réutiliser l'algorithme sur des ***types*** de données différents : tableau d'entiers, tableau de réels, tableau d'objets…
- la *généricité de structure* permet de réutiliser l'algorithme sur des ***structures de données*** différentes : **tableau** d'entiers, **liste** d'entiers, liste d'objets…

Un exemple : le tri

- Pour un tableau

```
void tri(Tableau & unTableau){
  int fin = unTableau.nbElements();
   while (fin > 1){
        for (int j = 0; j < fin - 1; j++)
            if (unTableau[j] > unTableau[j+1])
              permute(unTableau, j, j+1);
        fin --;
        }
    }
```

Nécessaires :

int Tableau::nbElements()const,
const Objet & Tableau::operator [] (int) const;
void permute(Tableau &, int, int);

Pourrions nous écrire ?

- Pour une liste

```
void tri(Liste & uneListe){
int fin = uneListe.nbElements();
while (fin > 1){
    for (int j = 0; j < fin - 1; j++)
            if (uneListe[j] > uneListe[j+1])
                    permute(uneListe, j, j+1);
    fin --;
    }
  }
```

Nécessaires :

int Liste::nbElements()const,
const Objet & Liste::operator [] (int) const;
void permute(Liste &, int, int);

Nous obtiendrions :

```
void tri(Conteneur & unConteneur){
  int fin = unConteneur.nbElements();
   while (fin > 1){
        for (int j = 0; j < fin - 1; j++)
            if (unConteneur[j] > unConteneur[j+1])
              permute(unConteneur, j, j+1);
        fin --;
        }
    }
```

Nécessaires :

int Conteneur::nbElements()const,

const Objet & Conteneur::operator [] (int) const;

void permute(Conteneur &, int, int);

Par exemple:

```
template <class Conteneur>
void tri(Conteneur & unConteneur){
 int fin = unConteneur.nbElements();
    while (fin > 1){
          for (int j = 0; j < fin - 1; j++)
                    if (unConteneur[j] > unConteneur[j+1])
                              permute(unConteneur, j, j+1);
          fin --;
          }
    }

template <class Objet>
class Tableau{
public :
    int nbElements();
    const Objet & operator[] (int)const;
    };

template <class Conteneur>
void permute(Conteneur &, int, int);

void main(){
    Tableau <int> unTableau;
    tri(unTableau);
    }
```

Ce n'est pas suffisamment générique

- nous avons supposé que le conteneur était numéroté à partir de 0
- nous avons supposé qu'il était numéroté par des nombres entiers
- nous avons supposé qu'il était numéroté par des nombres consécutifs
- l'algorithme est trop lié au fonctionnement d'un tableau, ***donc probablement inefficace sur autre chose***

L'itérateur

- est un objet particulier
- il est relié et adapté à une structure de données (c'est elle qui le fournit)
- il permet de progresser à l'intérieur
- méthodes usuelles (Cf Interface `Iterator` de Java) :
  - hasNext()
  - next()
  - previous()

Une solution plus générique

```
void tri(Conteneur & unConteneur){
  int fin = unConteneur.nbElements();
  while (fin > 1){
      int cpt=0;
      for (Iter j = unConteneur.debut();
            cpt++ < fin-1;
            j = j.suivant()){
        if (unConteneur[j] > unConteneur[j.suivant()])
            permute(unConteneur, j, j.suivant());
      }
    fin --;
    }
  }
```