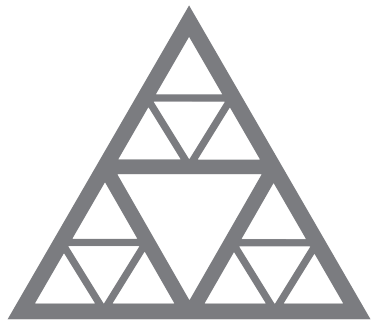
PRALG: Templates, STL, itérateurs, foncteurs

Pascal Monasse
pascal.monasse@enpc.fr



ÉCOLE NATIONALE DES
PONTS
ET CHAUSSÉES

IP PARIS

Macros

Le pré-processeur (qui gère toutes les lignes en # : #if #ifdef #ifndef #else #endif #include #pragma) propose des macros

```
#define MIN(a,b) (a<b? a:b)
int i = MIN(-1, 3);
float f = MIN(-1.0f, 3.0f);
```

$\Leftrightarrow$

```
int i = (-1<3? -1:3);
float f = (-1.0f<3.0f? -1.0f:3.0f);
```

OK pour les types “maison” s'ils supportent $<$:

```
class Entier {
public:
  int i_;
  Entier(int i) { i_=i; }
};
bool operator<(Entier a, Entier b) { return a.i_<b.i_; }
Entier i = MIN(Entier(1), Entier(3));
```

Mais :

```
int i=-1, j=3;
cout << MIN(i++, j) << endl;
```

$\Leftrightarrow$

```
int i=-1, j=3;
cout << (i++<j? i++:j) << endl;
```

i incrémenté 1 ou 2 fois ? Indéfini par la norme...

Les macros sont à éviter !

Solution : fonction template

```
template <typename T>
T min(T a, T b) { return (a<b? a:b); }
int i=-1, j=3;
cout << min(i++, j) << endl; //Aucun probleme
```

- La fonction s'appelle en fait min<int>, mais le C++ le devine tout seul !
- Il a néanmoins une intelligence limitée :

```
float f=3.0f;
f = min(-1.0f, f); //OK
f = min(-1, f); //Non, min<float> ou min<int>?
f = MIN(-1, f); //OK avec macro
f = min<float>(-1, f); //OK
```

- Ne réimplémentez pas vous-même, std :: min<T> et std :: max<T> dans #include <algorithm>

Classes templates

```
template <typename T> class Matrice {
  int m,n; T* tab;
public:
  Matrice(int nlig, int ncol);
  ~Matrice();
  Matrice(const Matrice<T>& B);
  Matrice<T>& operator=(const Matrice<T>& M);
};
template <typename T> Matrice<T>::Matrice(int nlig, int ncol) {
  m=nlig; n=ncol; tab = new T[m*n];
}
template <typename T> Matrice<T>::~Matrice() { delete [] tab; }
template <typename T> Matrice<T>::Matrice(const Matrice<T>& B) {...}
template <typename T> Matrice<T>& Matrice<T>::operator=(const Matrice<T>& M) {...}
```

On est obligé de préciser le type à l'usage :

```
Matrice A(4,3); //Non, Matrice n'est pas un type
Matrice<float> A(4,3); //OK
Matrice<float> B=A; //Constructeur par copie
```

La définition des templates doit être dans le .h, pas de .cpp

On peut avoir plusieurs paramètres template, comme std :: pair<T,U> (#include < utility >) :

```
pair<int,float> p = pair(3,1.5f); //Non
pair<int,float> p = pair<int,float>(3,1.5f); //OK
pair<int,float> p(3,1.5f); //Mieux
f( pair<int,float>(3,1.5f) ); //Appel fonction
f( make_pair(3,1.5f) ); //Moins lourd
```

La fonction template std :: make_pair<T,U> permet d'alléger.

STL : Standard Template Library

- Conteneurs de données : std :: vector<T>, std :: list <T>, std :: stack<T>, std :: queue<T>, std :: priority_queue <T>, std :: set<T>, std :: map<T,U>, std :: multi_map<T,U>.
- Des “algorithmes” : std :: min<T>, std :: max<T>, std :: swap<T>, std :: sort <T>, std :: find <T>...
- Ceux qui agissent sur un conteneur le font par des itérateurs.
- Un itérateur se comporte comme un pointeur : ∗it, ++it, etc.

```
template <class T>
void init(vector<T>& V) {
  vector<T>::iterator it=V.begin();
  for(; it!=V.end(); ++it)
    *it = 0;
} // it se comporte comme T*
```

```
template <class T>
void print(const vector<T>& V) {
  vector<T>::const_iterator it=V.begin();
  for(; it!=V.end(); ++it)
    cout << *it << endl;
} // it se comporte comme const T*
```

- Les conteneurs de la STL ont des méthodes begin() (1er élément) et end() (juste après dernier élément)

Interface conteneur/algo par itérateur

```
int T[2] = {3, 1};
std::sort(T, T+2); //Pointeurs
std::vector<int> V;
V.push_back(3); V.push_back(1);
std::sort(V.begin(), V.end());
std::list<int> L;
L.push_back(3); L.push_back(1);
L.sort(); // Pas std::sort(L.begin(), L.end())
```

L'exception pour std :: list <T> vient de son itérateur qui est seulement *bidirectionnel* mais pas *random*.

Pour désigner un emplacement dans un conteneur, c'est un itérateur qui est utilisé :

```
std::vector<int> V;
V.push_back(3); V.push_back(1); // V: (3 1)
V.insert(V.begin()+1, 2); // V: (3 2 1)
V.insert(V.end(), 0); // V: (3 2 1 0), pareil que V.push_back(0)
V.erase(V.begin()); // V: (2 1 0)
V.erase(V.end()-1); // V: (2 1), pareil que V.pop_back()
V.erase(V.end()); // BUG
```

Foncteurs

Équivalent de pointeur sur fonction, toute classe possédant un operator() :

```
class CompareInt {
  const int* tab;
public:
  CompareInt(const int* t) { tab=t; }
  bool operator()(int i, int j) const { return tab[i]<tab[j]; }
};
const int T[3] = {8, 1, 5};
int ordre[3] = {0, 1, 2};
std::sort(ordre, ordre+3, CompareInt(T));
// T non modifie, mais ordre trie ses elements par indice
int rang[3]; //Permutation inverse de ordre
for(int i=0; i<3; i++)
  rang[ordre[i]] = i;
for(int i=0; i<3; i++)
  cout << rang[i] << '-';
//Affiche 2 0 1
```

En effet, on a T[ordre [0]] $\leq$T[ordre [1]] $\leq$T[ordre [2]] , donc rang[ordre[i]]=i.
Il y a une version de la fonction std :: sort<T> qui prend un foncteur en argument :

```
class CompareInt { public:
  bool operator()(int i, int j) const { return (i>j); }
};
int T[3] = {8, 1, 5};                  // T: (8 1 5)
std::sort(T, T+3);                     // T: (1 5 8)
std::sort(T, T+3, CompareInt());       // T: (8 5 1)
```