UNIVERSITÉ DE MONTPELLIER
UNIVERSITÉ DE MONTPELLIER - FACULTÉ DES SCIENCES ET TECHNIQUES
PLACE EUGÈNE BATAILLON - 34095 MONTPELLIER CEDEX 5
Licence 2 informatique – 2019/2020

HLIN302 – Travaux Dirigés n° 7

Programmation impérative avancée
Alban MANCHERON et Pascal GIORGI

1 Fonction avec paramètre générique

La généricité paramétrique est un concept de la programmation objet permettant d'écrire du code sans connaître a priori le type de données d'un ou de plusieurs paramètres. L'idée est de pouvoir concevoir des codes communs à un ensemble de types qui partageraient des comportements similaires. Par *comportement*, il faudra comprendre appel de méthode/fonction sur un objet.

Par exemple, si l'on considère la fonction `max3(x,y,z)` qui renvoie le plus grand des éléments entre `x`, `y` et `z`. On peut considérer que le code de cette fonction est identique, quel que soit le type de `x`, `y` et `z` si celui-ci propose un opérateur de comparaison $<$.

```
1 if (x>y && x>z) return x;
2 if (y>x && y>z) return y;
3 return z;
```

1. Écrivez la signature d'une fonction générique `max3`.
2. Proposez un programme qui calcule le maximum de 3 entiers et 3 flottants par appel à la fonction `max3`.
3. Est-il possible d'appeler la fonction `max3` avec 2 entiers et un flottant ? Expliquez pourquoi.
4. Afin d'éviter toute confusion, nous souhaitons éviter que la fonction `max3` puisse fonctionner avec des types différents. En utilisant la surcharge des fonctions, proposez une solution qui affichera un message d'erreur et terminera le programme si cette fonction est appelée sur des types différents.

Une fonction générique ne compilera pas si vous l'utilisez sur un type de données qui ne respecte pas les pré-requis de cette dernière. Essayez par exemple de calculer le maximum entre trois `Cellule`s du jeu de la vie en appelant `max3`. Toutefois, il peut être utile de pouvoir appeler cette fonction sur des objets ne respectant pas les pré-requis du code générique. Dans ce cas précis, il est possible de surcharger la fonction générique pour un jeu de paramètres précis. On parle alors de spécialisation de code *template*.

5. Proposez la spécialisation de la fonction `max3` quand les paramètres sont de type `Cellule`.

2 Une classe de tableau extensible générique

En réponse au sujet de TD 5, vous avez écrit une classe de tableau extensible d'entiers puis un tableau extensible de `Cellule`s. À cette occasion, vous avez pu voir que les codes étaient identiques à l'exception des types de données manipulés.

1. En vous appuyant sur les codes de tableau extensible du TD 5, proposez une classe générique `MyVector` permettant de créer un tableau extensible sur n'importe quel type ayant un constructeur par défaut. Cette classe proposera :
   — un constructeur vide et un constructeur sur une taille de tableau ;
   — des méthodes `at` permettant d'accéder en lecture/écriture à une case du tableau ;

— une méthode `push_back` permettant d'ajouter un élément à la fin du tableau et une méthode `size` permettant de récupérer la taille du tableau ;
— des méthodes `find` et `erase` permettant respectivement de trouver un élément du tableau et de l'effacer.

2. Afin d'afficher un tableau de type `MyVector` sur un flux de sortie `ostream`, nous allons proposer une fonction générique `write`. Cette fonction doit permettre d'afficher n'importe quel conteneur (tableau, liste, ...) si ce dernier propose les méthodes `size()`, `at(i)` et que les éléments du tableau autorisent l'utilisation de l'opérateur d'insertion dans un flux d'affichage. Vous testerez votre classe `MyVector` et la fonction `write` sur le programme ci-dessous.

```
#include <iostream>
#include <vector>
#include "myvector.h"
#include "print-vector.h"
#include "cellule.h"
using namespace std;

template <typename Vect>
void write(ostream& os, const Vect& T){
  for(size_t i=0;i<T.size();i++)
    os<<T.at(i)<<" ";
  os<<endl;
}
template<>
void write<MyVector<Cellule>> (ostream& os, const MyVector<Cellule>& T){
  for(size_t i=0;i<T.size();i++){
    write_cell(os,T.at(i)); os<<" ";
  }
  os<<endl;
}
template<>
void write<vector<Cellule>> (ostream& os, const vector<Cellule>& T){
  for(size_t i=0;i<T.size();i++){
    write_cell(os,T.at(i)); os<<" ";
  }
  os<<endl;
}

int main(int argc,char** argv){
  //MyVector<int>     T1;
  vector<int> T1;
  MyVector<double>  T2;
  vector<Cellule> T3;

  for (size_t i=1;i<10;i++){
    T1.push_back(i);
    T3.push_back(Cellule(true,i,i));
    T2.push_back(1./i);
  }
  write(cout,T1);
  write(cout,T2);
  write(cout,T3);
  return 0;
}
```

3. La classe `std::vector` est une classe générique de tableau extensible proposé par la bibliothèque standard du C++. Pour utiliser cette classe, il suffit de rajouter `#include <vector>`. Modifiez le programme ci-dessus pour utiliser la classe générique `std::vector` à la place de votre classe `MyVector`.

4. Si l'on ajoute au programme un tableau générique de `Cellule`s (*e.g.* en utilisant `MyVector<Cellule>` ou `std::vector<Cellule>`) et qu'on essaie de l'afficher en utilisant la fonction `write`, que va-t-il se passer ? Proposez deux solutions permettant de contourner ce problème et implémentez l'une d'entre elles.

3 Des tris génériques et un jeu de la vie (presque terminé)

En réponse au sujet de TD 6, vous avez implanté trois tris différents (tri bulle, tri rapide et tri par tas) ainsi qu'une recherche dichotomique. Vos codes étaient dédiés au cas d'un `TableauCellule` mais en réalité, ces fonctions peuvent facilement être rendues génériques pour n'importe quel conteneur.

1. Trouvez les pré-requis sur les objets manipulés dans vos trois fonctions de tri ainsi que dans la fonction de recherche dichotomique ;
2. Proposez une version générique de ces codes permettant de fonctionner à la fois sur des tableaux de cellules et d'entiers. Attention, il faudra très certainement définir des fonctions génériques pour la relation d'ordre ;
3. La bibliothèque standard STL propose des conteneurs génériques (`std::vector`, `std::list`, ...) ainsi que des algorithmes génériques sur ces conteneurs (*e.g.* `std::sort`, `std::find`, ...). Comparez vos algorithmes de tris sur un `std::vector<int>` avec celui proposé par la STL. L'exemple ci-dessous vous montre l'utilisation de l'algorithme de tri de la STL :

Exemple de tri STL

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include "myvector.h"
#include "print-vector.h"
using namespace std;

#define MAX 100

bool compare(int a,int b){return a>b;}

int main(int argc, char** argv){

  if (argc !=2) {cerr<<"Usage: "<<argv[0]<<" [tab dim]"<<endl; return 1;}
  srand(123);

  std::vector<int> T(atoi(argv[1]));
  for(size_t i=0;i<T.size();++i)
    T.at(i)=rand() % MAX;

  write(cout,T);
  sort(T.begin(),T.end()); // tri selon l'ordre croissant (appel à <)
  write(cout,T);
  sort(T.begin(),T.end(),compare); // tri selon la fonction compare (ici > )
  write(cout,T);

  return 0;
}
```

4. Afin de finaliser le jeu de la vie (dans sa version ne stockant que les cellules vivantes dans un tableau trié), vous allez utiliser :
   — la classe `std::vector` pour stocker les cellules vivantes ;
   — la fonction `std::sort` pour trier les tableaux ;
   — votre recherche dichotomique pour trouver si une cellule donnée est vivante dans votre population du jeu de la vie.