UNIVERSITÉ LIBRE DE BRUXELLES ULB

FACULTÉ DES SCIENCES
DÉPARTEMENT D'INFORMATIQUE

Langages de programmation 2 : Projet C++

Yves Roggeman *

INFO-F202 — Année académique 2016–2017

Résumé

Ceci est un des deux énoncés de l'épreuve de première session qui se déroule en janvier. Il sert de base à l'épreuve orale ; l'évaluation finale porte sur l'acquisition des concepts démontrée à cette occasion. Le but de cet exercice de programmation est donc de démontrer une connaissance approfondie et un usage adéquat des constructions du langage de programmation C++, en particulier de l'usage des « **template** » et du mécanisme d'héritage.

Le problème posé consiste à définir une structure de données abstraite — que nous appellerons une *file min-max* — constituée de valeurs d'un type de base ordonnable. Il s'agira d'un *conteneur* particulier contenant effectivement ses valeurs (et pas des références vers des valeurs) sur lequel seules quelques opérations sont possibles. Cette structure sera définie de manière générique et implantée concrètement de deux façons différentes.

1 La file min-max

Nous définissions ici une « file min-max » comme un conteneur particulier dont les opérations d'extraction et de consultation sont limitées à sa valeur maximale et à sa valeur minimale. Par contre, l'insertion de toute valeur est permise et doit maintenir une structure qui rend ces accès efficaces. Les itérateurs ne permettent que de parcourir le conteneur, donc en simple consultation sans modification possible ni du conteneur ni de ses éléments.

1.1 Conteur

Un tel conteneur « `Min_Max` » est défini par les opérations de modification et consultation correspondant à ses méthodes suivantes.

- `front()` : valeur maximale (copie de l'élément) du conteneur (inchangé) ;
- `back()` : valeur minimale (copie de l'élément) du conteneur (inchangé) ;
- `pop_front()` : retire l'élément maximal du conteneur (à valeur **`void`**) ;
- `pop_back()` : retire l'élément minimal du conteneur (à valeur **`void`**) ;
- `clear()` : vide le conteneur de tout son contenu (à valeur **`void`**) ;
- `insert(t)` : insère l'élément « `t` » à la bonne place dans le conteneur (à valeur itérateur vers l'élément nouvellement inséré).

*Université libre de Bruxelles (ULB) <yves.roggeman@ulb.ac.be>

Bien sûr, les opérations de consultation ou d'extraction lèveront une exception si le conteneur est vide ; il en sera de même pour l'insertion si sa capacité maximale est déjà atteinte.

Des méthodes `max()`, `min()`, `pop_max()`, `pop_min()` seront également définies comme synonymes des quatre premières.

Par ailleurs, le conteneur devra répondre aux critères généraux de tout conteneur C++ repris dans la norme (version 2017), notamment ceux de la table 83. Vous devez donc définir tous les alias de type, toutes les méthodes et tous les opérateurs (membres de la classe ou extérieurs) qui y sont indiqués, ainsi bien sûr que toutes les méthodes spéciales standard.

1.2 Itérateur

Ce conteneur définit un itérateur spécifique. Celui-ci est nécessairement un *itérateur constant* qui préserve donc le conteneur : les types `Min_Max::iterator` et `Min_Max::const_iterator` sont des alias d'un même type et ses méthodes `begin()` et `cbegin()`, d'une part, `end()` et `cend()`, d'autre part, sont synonymes.

L'itérateur est de la catégorie « *input iterator* » de C++. Il respecte donc les critères d'un tel itérateur repris dans la norme (version 2017), notamment aux tables 94 et 95. On veillera à ce que les définitions soient adaptées pour que l'itérateur soit strictement un « itérateur constant » qui ne permette donc en aucun cas de modifier le contenu du conteneur. Pour assurer la cohérence, le type de cet itérateur est une classe imbriquée dans celle du conteneur.

2 Implantation

Le type du conteneur (que l'on appellera ici « **class** `Min_Max` ») et celui de son itérateur seront des modèles (**template**) de classes abstraites dont le paramètre désigne le type des éléments contenus. Comme indiqué, l'itérateur sera une classe imbriquée à celle du conteneur et inaccessible directement (elle n'est connue que par ses alias standard).

Afin de tester le bon fonctionnement de ces classes, deux fonctions génériques seront écrites : l'une pour afficher (sur un `std::ostream`) le nombre d'éléments d'un conteneur, puis son contenu (les valeurs de ses éléments successifs), l'autre pour remplir un conteneur à partir de données d'un fichier (de type `std::istream`).

De plus, les opérations de copie (par construction ou assignation) de tels conteneurs doivent être possibles, quel que soit le type concret de la source.

Pour réaliser concrètement un tel conteneur, on choisira une structure inspirée d'une file FIFO (une *queue*) : une suite d'éléments triés par ordre décroissant. Ainsi, le maximum correspond au premier et le minimum au dernier. Les opérations de consultation et d'extraction à ces deux extrémités sont donc possibles en temps constant. Par contre, l'insertion nécessite de maintenir le tri interne des éléments, donc en temps linéaire en la taille effective du conteneur.

Deux implantations de cette structure interne sont demandées :

- L'une sous forme de liste circulaire doublement liée : on y accède directement *via* un des éléments (un seul pointeur est nécessaire) ;
- L'autre sous forme d'un tableau de taille fixe où l'on connaît l'indice du premier élément (premier occupé) et du suivant du dernier élément (premier libre), ces indices étant gérés circulairement (*i.e.* incréments et décréments modulo la taille globale du tableau) : l'extraction ne doit que modifier ces indices, par contre l'insertion nécessite un glissement interne.

Dans l'un et l'autre cas, il est plus efficace de prévoir un élément « bidon » supplémentaire dans la structure interne, qu'il s'agisse d'une liste ou d'un tableau, afin qu'elle ne soit jamais vraiment vide (ce qui permet tout accès physique sans test de ce cas limite).

Dans le seul second cas, la taille maximale est un paramètre du modèle de classe concrète (qui possède une valeur par défaut). Le tableau interne possède donc un élément effectif de plus.

Veillez à l'efficacité de l'implantation de vos classes et méthodes ; évitez toute redondance et tout attribut inutile (on ne stocke pas une information que l'on peut calculer en temps constant). Et bien sûr, les structures, types et données internes de ces classes concrètes sont inaccessibles à tout code qui peut instancier et utiliser des objets de ces types.

3 Réalisation

Il vous est demandé d'écrire en C++ la définition de toutes les classes et fonctions nécessaires ainsi qu'un programme principal « `main` » servant de test de plusieurs instanciations d'objets de ces classes et d'appels à leurs méthodes pour des types de base différents (**`int`** et `std::string`, par exemple). Vous fournirez également un schéma, un diagramme « à la UML » de la hiérarchie des différentes classes définies.

L'évaluation portera essentiellement sur la pertinence des choix effectués dans l'écriture : la codification, la présentation et l'optimisation du programme justifiées par la maîtrise des mécanismes mis en œuvre lors de la compilation et l'exécution du code. D'une manière générale, le respect strict des directives, la concision, la précision, la lisibilité (clarté du texte source), l'efficacité (pas d'opérations inutiles ou inadéquates) et le juste choix des syntaxes typiques de C++ seront des critères essentiels d'appréciation. De brefs commentaires dans le code source sont souhaités pour éclairer les choix de codification.

Votre travail doit être réalisé pour le vendredi 22 décembre 2017 à 10 heures au plus tard. Vous remettrez tout votre travail empaqueté en un seul fichier compacté (« .zip » ou autre) *via* le site du cours (INFO-F202) sur l'Université Virtuelle (https ://uv.ulb.ac.be/). Ceux-ci devront contenir en commentaire vos matricule, nom et prénom. Le jour de l'examen, vous viendrez avec une version imprimée — un *listing* — de ces divers fichiers. Une impression du résultat d'une exécution du programme est également demandée.