UNIVERSITÉ LIBRE DE BRUXELLES ULB

FACULTÉ DES SCIENCES
DÉPARTEMENT D'INFORMATIQUE

Langages de programmation 2 : Projet en C++ (1re session)

Yves Roggeman*

INFO-F202 — Année académique 2020–2021

Résumé

Ceci est l'un des deux projets de programmation du cours ; sa réalisation est une condition nécessaire d'accès à l'examen oral en session (en janvier 2021). Il sert de base à cette épreuve orale ; la note finale reflète l'acquisition des concepts évaluée à cette occasion. Le but de cet exercice de programmation est donc de montrer une connaissance approfondie de la programmation orientée objet et un usage adéquat des constructions du langage C++, en particulier de l'usage des « **`template`** », du mécanisme d'héritage et du polymorphisme.

Le problème posé consiste à définir un ADT générique de conteneur de données particulier, le type de base des valeurs contenues étant paramétrique. Ce conteneur contient ses propres valeurs, donc pas une référence ou un pointeur vers des données extérieures. La capacité maximale du conteneur est un paramètre fixé à sa construction, modifiable exclusivement par copie ou transfert d'un conteneur du même type. Nous supposerons que le type des données de base permet au minimum les comparaisons d'égalité (==) et d'inégalité stricte (<) ; les données appartiennent ainsi à un ordre total.

Ce conteneur accepte deux types d'accès : un accès en lecture « aléatoire » (*random*) selon un indice fourni et un accès « associatif » sur base de la valeur, en lecture et écriture. L'un et l'autre accès peuvent être infructueux s'il n'existe pas d'élément dans le conteneur correspondant au critère spécifié. Ces deux accès doivent être efficaces : l'accès aléatoire est en temps constant, indépendant de la capacité et du taux d'occupation, tandis que l'accès associatif est, en moyenne, de complexité temporelle logarithmique selon le nombre d'éléments contenus. L'insertion et la suppression d'éléments ont cette même complexité moyenne logarithmique.

Il ne peut contenir au plus qu'une seule donnée par indice et une seule donnée par valeur ; il n'y a donc aucun doublon, pas d'éléments homonymes.

1 Type abstrait de conteneur

Le conteneur souhaité — que nous appellerons `Cont` — est défini à partir de types de données abstraits autonomes correspondant à ses différents aspects : un arbre de recherche et un vecteur.

1.1 Accès associatif par valeur

D'une part, le conteneur possède toutes les caractéristiques d'un arbre binaire de recherche. La recherche par valeur se fait au travers de cet arbre dont le conteneur hérite des méthodes de recherche, d'insertion et de suppression.

Un canevas d'arbre binaire de recherche vous est fourni. Vous pouvez (ou devez) le compléter ou le modifier, si nécessaire. Attention : ce canevas admet évidemment des doublons et cette caractéristique ne peut être modifiée à ce niveau. Dans cet arbre, l'insertion est donc toujours fructueuse, tandis que la suppression renvoie une valeur booléenne indiquant si elle a abouti. La recherche est

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

exclusivement en consultation et renvoie une valeur sentinelle si aucun élément correspondant n'a été trouvé.

Dans le contexte du conteneur, pour l'insertion, une position (un indice) **et** une valeur doivent être fournies ; par contre pour la recherche ou pour la suppression, la position peut être omise, mais si elle est fournie, elle doit correspondre à celle de l'élément visé. Une manière simple de réaliser cela est de définir un type de base pour l'arbre dont les comparaisons ne tiennent pas compte de l'indice, puis de le tester dans les redéfinitions des méthodes, s'il échet. Bien sûr, l'information contenue dans les sommets de l'arbre doit renseigner l'indice qui lui correspond dans le vecteur.

1.2 Accès aléatoire par indice

D'autre part, le conteneur possède toutes les caractéristiques d'un vecteur ou tableau à un indice de taille définie à la construction.

Pour des raisons d'efficacité spatiale et de cohérence d'implantation, les informations contenues logiquement dans le conteneur ne sont pas dupliquées. Elles sont placées exclusivement dans l'arbre — qui ne contient un sommet que pour les informations réellement présentes — et le vecteur ne contient qu'un pointeur vers l'information associée située dans l'arbre. Si une position est inoccupée, ce pointeur est nul (*i.e.* **`nullptr`**).

Toutefois, comme l'implantation de la mise à jour *via* ce tableau serait trop complexe, l'accès par indice ne doit être prévu qu'en consultation (fournissant une *R-value*). Ici aussi, il existe une façon simple de réaliser cela : définir un type de base du tableau associé à un pointeur vers une information **constante**. Toutefois, si la capacité du conteneur vaut n, un tel accès en consultation par opérateur $[i]$ doit être possible pour toute valeur satisfaisant $0 \leq i < n$ et fournir une valeur sentinelle particulière identifiant un emplacement vide.

Pour définir ce type de vecteur, vous pouvez (devriez) vous inspirer directement de l'exemple donné au cours (voir site sur l'UV).

1.3 Accès global

Outre les méthodes spéciales traditionnelles, le conteneur doit également maintenir une information indiquant le nombre de places occupées, ce qui correspond à la taille de l'arbre de recherche (son nombre de sommets).

Pour le reste, le conteneur doit pouvoir être utilisé comme un vecteur de type `Vect` (avec son opérateur `[]` constant) et comme un arbre de recherche de type `BST` (avec ses primitives « insert », « erase » et « find »). Il doit évidemment pouvoir être transmis à un flux de sortie (output) ; ses éléments sont parcourus selon l'ordre croissant de leurs valeurs.

2 Implantation

Pour garantir un comportement conforme à ce qui est demandé ci-dessus et respectant le principe d'encapsulation, les types de base respectifs du vecteur et de l'arbre doivent être des classes « emballant » leur véritable information, avec des conversions implicites depuis et vers le type de base du conteneur : ce sont des « *wrappers* ». Et bien sûr, ces types emballeurs doivent rester « locaux » à la classe du conteneur ; ils ne peuvent être définis de manière autonome, toujours pour des raisons d'encapsulation.

De telles situations sont facilement résolues en définissant une classe mère abstraite (générique) — que nous appellerons ici `Cont_base` — contenant les définitions de ces classes utilitaires sous forme imbriquée. Il suffit ensuite d'hériter, même de façon privée, de cette classe pour que tout fonctionne bien [1] ! L'accès aux informations internes de ces classes (pointeur, indice, etc.) peut ainsi rester discrétionnaire (privé ou protégé) pour les seules méthodes de la classe `Cont`. Il est parfois nécessaire de définir des méthodes protégées supplémentaires dans la classe mère pour fournir cet accès [2].

1. C'est une astuce classique en programmation orientée objet.
2. On ne peut évidemment pas définir **`friend`** une future classe-fille encore inconnue.

Un canevas de cette classe mère vous est également fourni. Vous devez en comprendre parfaitement le mécanisme et l'adapter ou le compléter au besoin.

3 Réalisation

Il vous est demandé d'écrire en C++ la définition de tous les *templates* des classes et de leurs méthodes décrites ci-dessus. Vous effectuerez des tests variés, avec différents types de base, y compris un type programmé (par vous), *i.e.* une classe, pas uniquement des types élémentaires.

Il vous est également demandé d'écrire des fonctions agissant sur un vecteur ou un arbre binaire de recherche auxquelles vous soumettrez une instance du conteneur pour en démontrer le parfait polymorphisme.

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 du langage C++ seront des critères essentiels d'appréciation. De brefs commentaires dans le code source sont souhaités pour éclairer les choix de codification.

Pour la compilation, vous respecterez les consignes (voir document général sur l'UV) : tous les avertissements doivent être activés à la compilation et, s'il en persiste pour votre programme, vous devez les justifier en commentaire dans le code.

Votre travail devrait être déposé pour le vendredi 18 décembre 2020 à 16 heures au plus tard ; **une remise tardive n'est pas acceptée**. Le format attendu est un seul fichier compressé reprenant vos différents fichiers constituant le code source de la solution. Un fichier contenant l'output de l'exécution de votre code doit y être joint. Par contre, aucun binaire ni code objet n'est demandé, mais si vous avez utilisé un *makefile* ou un script de compilation (ce qui est hautement probable), veuillez le joindre également.

Bon travail !