Exercices : Les bases du langage

# Partie 1 : connexion avec le C

Les questions suivantes peuvent être faites dans un même fichier de code d’extension .cpp

1. Ecrire une fonction main retournant la valeur 0.
2. Y appeler une nouvelle fonction qui affiche « hello world » en utilisant uniquement des fonctions et éléments de syntaxe reconnus par le langage C
3. Y appeler également une nouvelle fonction qui affiche « hello C++ » en utilisant des instructions propres au C++
4. Faire une surcharge de la fonction qui affiche « hello C++ » pour qu’elle reçoive un entier en paramètre et qu’elle l’affiche également sur la console. Mettre à jour la fonction main pour qu’elle prenne en compte cette évolution sans demander à l’utilisateur de saisir quoi que ce soit au clavier.
5. Définir une constante globale au fichier de type double nommée Pi et valant 3,1416 (ou plus précis si vous préférez, cela n’aura pas d’importance).
6. Faire une fonction PrintSin contenant un affichage sur la console (façon C++) pour annoncer le calcul puis une boucle for allant de 0 à 10 affichant chaque sinus de Pi + compteur de boucle. Utiliser l’instruction #include<cmath> pour référencer la fonction sinus. Appeler cette fonction dans la fonction main.
7. Faites une fonction recevant un pointeur constant sur flottants mais dont la valeur pointée est modifiable. Cette fonction recevra également un autre paramètre entier, correspondant au nombre d’éléments du tableau. Faire une boucle allant de zéro à ce paramètre et ranger dans chaque case du tableau le cosinus du compteur de boucle.
8. Dans la fonction main créer une constante entière de taille 60. L’utiliser pour créer un pointeur sur 60 flottants, puis pour appeler la fonction créée à la question précédente.
9. Faire une nouvelle fonction qui reçoive un tableau de flottants en paramètre avec sa taille et qui affiche son minimum, son maximum et sa moyenne.
10. Appeler cette fonction avec le tableau créé et rempli à la question 8.

# Partie 2 : programmation orientée objet

1. Créer un fichier Point.cpp.
2. Y faire une classe Point représentée par deux entiers et fournissant une fonction Distance renvoyant la distance (en flottant) entre le Point et un autre Point reçu en paramètre.
3. Extraire la déclaration de la classe Point dans un fichier Point.h et ne laisser que l’implémentation dans Point.cpp
4. Depuis la fonction main créer deux points et afficher sur la console la distance les séparant. L’afficher en appelant la fonction depuis le premier point puis depuis le deuxième.
5. Créer deux fichiers StackInt.h et StackInt.cpp. Y placer l’interface et l’implémentation de la classe Stack attendue à la question suivante.
6. Créer une classe StackInt décrivant une pile d’entiers. Elle doit exposer trois méthodes : push, pop et peek. Push doit recevoir un entier et l’empiler, pop ne reçoit rien et dépile l’entier le plus haut placé dans la pile et peek renvoie la valeur du prochain entier qui sera renvoyé par pop mais sans le dépiler.
7. Instancier une StackInt dans la fonction main et y empiler une dizaine de valeurs avant de les afficher sur la console.
8. Créer deux fichiers LinkedList.h et LinkedList.cpp. Y placer l’interface et l’implémentation de la classe LinkedList attendue à la question suivante.
9. Créer une classe LinkedList décrivant une liste chaînée d’entiers. Elle doit exposer une méthode GetSize() renvoyant le nombre total de maillons, Add(int) ajoutant un entier à la chaine et GetAt(int position) renvoyant la valeur trouvée à la position demandée dans la chaine.
10. Créer deux fichiers Student.h et Student.cpp. Y placer l’interface et l’implémentation de la classe Student attendue dans la question suivante.
11. Créer une classe Student décrivant un étudiant à partir de son nom, son prénom et d’un tableau de notes fait à partir de la LinkedList des questions 18 et 19, et exposant les méthodes Average() renvoyant la moyenne des notes d’un étudiant, WillGraduate() renvoyant un booléen indiquant si l’étudiant à au moins 10 de moyenne et Print ne renvoyant rien mais affichant sur la console les nom et prénoms de l’étudiant ainsi que toutes ses notes et sa moyenne. La classe présentera un constructeur prenant un nom et un prénom en paramètre et un destructeur libérant toute ressource allouée.
12. Instancier quelques étudiants dans une fonction main, leur affecter quelques notes et vérifier si les moyennes et appels de fonctions sont cohérents.

Dans tous les questions suivantes il faudra utiliser des références vers les paramètres avec le degré de sécurité qui convient (sauf si précisé autrement bien entendu).

1. Soit une classe vecteur3d définie par trois données flottantes x, y et z.

Ecrire son interface dans un .h et son implémentation dans un .cpp

Introduire une fonction membre nommée Egal permettant de savoir si deux vecteurs ont les mêmes composantes x, y et z. L’écrire de trois façons :

1. En utilisant le passage par valeur du deuxième vecteur
2. En recevant un pointeur
3. En recevant une référence

Ecrire le code nécessaire dans la fonction main pour tester toutes les méthodes de cette classe avec tous les retours possibles (vrai, faux).

1. Ajouter à la classe de la question précédente une fonction NormeMax qui reçoit un vecteur3D et en renvoie celui qui a la norme la plus élevée entre celui reçu en paramètre et celui courant.

Ajouter dans la fonction main plusieurs instructions permettant de tester le bon comportement de NormeMax, notamment en s’assurant que A.NormeMax(B) renvoie la même valeur que B.NormeMax(A) (avec A et B deux vecteurs 3d).

1. Ajouter à la classe de la question précédente une fonction Somme qui reçoit un vecteur3D et l’ajoute au vecteur courant.

Ajouter dans la fonction main plusieurs instructions permettant de tester le bon fonctionnement de Somme, notamment en s’assurant que A.Somme(B) remplisse les champs du vecteur courant avec les mêmes valeurs que B.Somme(A).

Exercices : L’héritage en C++

1. Créer une classe Point contenant une abscisse et une ordonnée double. Définir une méthode double Distance(Point) qui renvoie la distance entre le point instancié et le point reçu en paramètre.
2. Vérifier le bon fonctionnement de cette classe et de cette méthode en créant quelques points dans la fonction main et en affichant la distance les séparant sur la console.
3. Créer une classe abstraite Forme ayant une méthode abstraite double Perimetre() et une méthode string Nom(). Dans toutes les questions suivantes la méthode Nom devra renvoyer le nom de chaque classe héritant de Forme.
4. Créer une classe Cercle héritant de Forme. Cette classe doit avoir un constructeur recevant un point qu’elle doit stocker dans une donnée membre privée, ce sera son centre. Le cercle comme le constructeur doivent également disposer d’un rayon. Implémenter la méthode Perimetre pour le cercle.
5. Créer une classe Carre héritant de Forme. Lui définir deux points qui formeront la diagonale du carré. Implémenter la méthode Perimetre pour le carré.
6. Vérifier le bon fonctionnement de ces classes dans la fonction main. Ne travailler que sur des références de Forme instanciées à partir d’un cercle et d’un carré.
7. Définir une classe Position ayant comme donnée protégée un Point et comme méthode publique void Déplacer(const Point &)
8. Créer une classe Rectangle ayant une longueur et une largeur privées et héritant de Forme et de Position. La position du rectangle correspondra à son coin supérieur gauche. Ajouter au rectangle une méthode Centre renvoyant le milieu de ses diagonales.
9. Vérifier le bon fonctionnement du Rectangle dans la fonction main.
10. Faire du Carré et du Cercle des Positions en plus de Formes.
11. Vérifier leur bon fonctionnement dans la fonction main.
12. Dans la fonction main créer un tableau de Formes. Y ranger toutes les instances créées précédemment pour les tests. Dans une boucle appeler la méthode Perimetre de chaque élément du tableau et en afficher les résultats sur la console. Vérifier ainsi le bon fonctionnement du polymorphisme.
13. Modifier la classe Position pour qu’elle hérite de Point. Vérifier si le programme compile et fonctionne toujours, éventuellement corriger.
14. Créer une classe Vecteur héritant de Position, incluant une norme (double) et un sens (booléen).
15. Créer une classe Segment héritant d’un Point et d’un Vecteur. Corriger tout problème de diamant que cela pourrait provoquer.
16. Vérifier le bon fonctionnement de tout le programme.
17. Permettre à l’utilisateur de créer une forme au choix parmi celles écrites précédemment. Stocker toute forme créée dans le tableau de formes.
18. Permettre à l’utilisateur d’afficher le contenu du tableau
19. Permettre à l’utilisateur de supprimer n’importe quelle forme du tableau via son index.
20. Permettre à l’utilisateur de supprimer toutes les formes d’un certain type.
21. Vérifier le bon fonctionnement de toutes ces fonctions.

Problème : application médiathèque

L’objectif de ce problème est double : réaliser une application de type console réutilisable et réaliser un logiciel de gestion de médiathèque prêt à évoluer à l’avenir.

# Partie 1 : console

Une commande est identifiée par son nom et correspond à une méthode à exécuter. Toute commande recevra une liste de paramètres de type string.

Une console permet la saisie d’une commande avec ses paramètres, séparés par des espaces.

On souhaite disposer d’une console affichant la liste de ses commandes en leur associant un numéro. L’utilisateur n’aura plus qu’à saisir celui associé à la commande de son choix pour la lancer.

Une fois la commande terminée l’utilisateur reviendra sur la console et pourra relancer tout programme de son choix.

Parmi les programmes disponibles il doit y avoir « quitter », qui met fin à tout cela après avoir affiché le bilan d’utilisation de la console.

Pour faire ce bilan il faudra d’abord réaliser une classe Compteur disposant de trois méthodes, Incrémenter, Réinitialiser et Lire, permettant de manipuler un petit compteur entier ne pouvant s’incrémenter que de 1 et repartant forcément à 0 quand on appelle Réinitialiser.

Lorsque l’utilisateur lancera la commande « quitter » il verra combien de fois il a utilisé chaque commande par le biais d’une instance de Compteur pour chaque commande…

# Partie 2 : médiathèque

Une médiathèque est un ensemble de livres, de musiques, de films et de jeux permettant d’en retrouver tout ou partie ainsi que d’en ajouter ou supprimer.

Créer un jeu de classes traduisant ces différents média en sachant que :

* Tous ont un nom, un type (livre, musique, film ou jeu), une année de parution et un éditeur
* Musiques et films ont une durée
* Livres et musiques ont un auteur
* Les films ont un réalisateur et un scénariste
* Les jeux ont une équipe de développeurs

Une médiathèque doit permettre de retrouver tous ses éléments ayant un certain nom, parus à une certaine date, ou toute autre combinaison de critères. Prévoir une méthode Search recevant un objet SearchRequest et renvoyant un ensemble de Media. Prévoir également une méthode d’ajout d’un Media ainsi que des méthodes de suppression aussi bien unitaire que de masse.

Toutes ces fonctions devront in fine être accessibles via une console…

Exercices : gestion des flux d’IO

Note : pensez à tester vos fonctions *au fur et à mesure* que vous les écrivez. Ce n’est pas parce qu’il n’y a pas une question « tester si le code produit à la question précédente fonctionne » qu’il ne faut pas le faire… Ecrire dans une fonction main une petite interface console permettant de tester chacune des fonctions à écrire.

1. Ecrire une classe intitulée *CustomStream* ayant pour donnée membre privée une chaine de caractères nommée *path*. Faire deux méthodes publiques pour encapsuler son utilisation. Ce membre privé contiendra le chemin vers un fichier, qui sera utilisé dans plusieurs des questions suivantes.

Note : dans les questions 2 à 8 il est inutile de se préoccuper de l’existence ou non d’un fichier à l’emplacement indiqué par *path*. Bien sûr il faut que ce fichier existe pour que les tests fonctionnent mais nous ne nous occuperons d’ajouter le nécessaire pour sécuriser l’utilisation de path ultérieurement.

1. Ajouter à cette classe une méthode *void Read()* qui affiche sur la sortie standard le contenu du fichier désigné dans la donnée path sous forme texte.
2. Ajouter à cette classe une méthode *void Write(string const &)* qui remplace le contenu du fichier par le texte reçu en paramètre.
3. Ajouter à cette classe une méthode *void* *Append(string const &)* qui ajoute à la fin du fichier le texte reçu en paramètre.
4. Ajouter à cette classe une méthode *void Copy(string const &)* qui remplace le contenu du fichier désigné par *path* par celui dont le chemin est reçu en paramètre, à travers une copie binaire.
5. Ajouter à cette classe une méthode *int Count (char)* qui compte le nombre d’occurrences d’un caractère reçu en paramètre dans le fichier.
6. Ajouter à cette classe une méthode *void CountAll()* qui affiche sur la sortie standard le nombre d’occurrence de chacun des 128 caractères du code ASCII sur la sortie standard sous le format suivant : <code ASCII> <caractère correspondant> : <quantité> (Ex : 50 a : 12).

On affichera uniquement les caractères pour lesquels le nombre trouvé est supérieur à 0.

1. Ajouter un paramètre booléen facultatif (ayant la valeur false par défaut) à la méthode *CountAll* qui, s’il prend la valeur true, envoie sur la sortie standard « le bilan se trouve là : [insérer ici la valeur de path à laquelle on ajoute l’extension .SAV] ». Bien sûr, il faut effectivement enregistrer dans ledit fichier le résultat trouvé pour chaque caractère.
2. Ajouter une fonction privée à cette classe, *void CheckFileExistence()* qui lève une exception si la donnée membre *path* ne pointe sur aucun fichier. L’exception levée doit contenir un petit texte indiquant *« aucun fichier spécifié »* si path est vide ou *« fichier introuvable »* sinon.
3. Ajouter un appel à cette fonction dans chacune des méthodes écrites aux questions 2 à 8.
4. Dans la fonction main s’assurer que les exceptions levées sont bien attrapées.
5. Créer maintenant une classe d’exception FileNotFoundException et l’appeler dans le cas où path est non nulle mais ne pointe pas sur un fichier. Traiter les FileNotFoundException de façon appropriée dans la fonction main.

Exercices : programmation générique

1. Ecrire une fonction générique affichant un paramètre reçu sur la sortie standard accompagné d’un entier. Le type du paramètre et l’entier devront être des paramètres génériques.
2. Ecrire la déclaration d’une classe générique LinkedList<T> ayant comme données membres privées un élément T et un pointeur sur LinkedList<T> et des getters et setters pour les manipuler.
3. Ecrire une classe générique BaseCollection<T> n’ayant pas de données membres et dont toutes les méthodes sont virtuelles pures et publiques. Les méthodes sont les suivantes :
   1. void Add(T)
   2. void Remove(T)
   3. T GetAt(int)
   4. int Count()
   5. void PrintAllElements()
4. Ecrire une classe générique Collection<T> héritant de BaseCollection<T> et ayant comme donnée membre un tableau de T et un entier MaxIndex.
5. Ajouter à la méthode GetAt une levée d’exception si l’indice reçu en paramètre est en dehors des bornes du tableau.
6. Dans un projet de tests ou dans une fonction main, tester le bon fonctionnement des réponses aux questions précédentes. En particulier, penser à tester que GetAt doit lever une exception pour un paramètre hors bornes mais ne doit pas en lever sinon.
7. Ecrire une classe LinkedCollection<T> héritant de BaseCollection<T> mais utilisant un LinkedList<T> à la place du tableau de T.
8. Ecrire les tests correspondant à la LinkedCollection<T>.
9. Ecrire une classe KeyValuePair<T, U> ayant deux données, une de type T appelée Key et une de type U appelée Value.
10. Ecrire une classe ListKVP<T, U> héritant de BaseCollection<KeyValuePair<T, U> > en manipulant un ensemble d’instances de KeyValuePair<T, U>.
11. Tester la classe ListKVP<T, U>
12. Ecrire une classe BasePairCollection<T, U> n’ayant pas de données membres et dont toutes les méthdoes sont virtuelles pures. Les méthodes sont les suivantes :
    1. void Add(T, U)
    2. void Remove(T)
    3. U Get(T)
    4. T[] GetKeys()
    5. U[] GetValues()
    6. int Count()
    7. void PrintAllElements()
    8. U operator[](T) // ceci est une surcharge de l’opérateur []
13. Ecrire une classe Dictionary<T, U> héritant de BasePairCollection<T, U> ayant comme donnée membre un tableau de KeyValuePair<T, U>.
14. Tester la classe Dictionary<T, U>
15. Comparer la taille de l’exécutable produit par la compilation en fonction du nombre d’implémentations différentes de chaque classe générique.