Chapitre IV Exercice 00 : Easy find

Exercice : 00

Easy find
Répertoire de rendu : ex00/

Fichiers à rendre : Makefile, main.cpp, easyfind.{h, hpp}
et fichier optionnel : easyfind.tpp
Fonctions interdites : Aucune

Un premier exercice facile est la meilleure manière de bien commencer.

Écris une fonction template easyfind qui accepte un type T. Elle prend deux paramètres : le premier est de type T, et le second est un entier.

En supposant que T est un conteneur d’entiers, cette fonction doit trouver la première occurrence du second paramètre dans le premier.

Si aucune occurrence n’est trouvée, tu peux soit lever une exception, soit retourner une valeur d’erreur de ton choix. Si tu as besoin d’inspiration, analyse le comportement des conteneurs standards.

Bien sûr, implémente et rends tes propres tests pour t’assurer que tout fonctionne comme prévu.

Tu n’as pas à gérer les conteneurs associatifs.

Chapitre V Exercice 01 : Span

Exercice : 01

Span
Répertoire de rendu : ex01/

Fichiers à rendre : Makefile, main.cpp, Span.{h, hpp}, Span.cpp
Fonctions interdites : Aucune

Développe une classe Span qui peut stocker un maximum de N entiers. N est une variable de type unsigned int et sera le seul paramètre passé au constructeur.

Cette classe aura une fonction membre appelée addNumber() pour ajouter un seul nombre dans le Span. Elle sera utilisée pour le remplir. Toute tentative d'ajouter un élément alors que N éléments sont déjà stockés devra lever une exception.

Ensuite, implémente deux fonctions membres : shortestSpan() et longestSpan()

Elles devront respectivement déterminer l'intervalle le plus court ou le plus long (ou la distance, si tu préfères) entre tous les nombres stockés, et le retourner. S'il n'y a aucun nombre stocké, ou un seul, aucun intervalle ne peut être trouvé. Donc, lève une exception.

Bien sûr, tu écriras tes propres tests, et ils seront bien plus complets que ceux ci-dessous. Teste ton Span avec au moins 10 000 nombres. Plus serait encore mieux.

Exécuter ce code :
```
int main()
{
    Span sp = Span(5);

    sp.addNumber(6);
    sp.addNumber(3);
    sp.addNumber(17);
    sp.addNumber(9);
    sp.addNumber(11);

    std::cout << sp.shortestSpan() << std::endl;
    std::cout << sp.longestSpan() << std::endl;

    return 0;
}
```

Devrait produire :
```
$> ./ex01
2
14
$>
```

Dernière chose mais non des moindres, ce serait formidable de remplir ton Span avec une plage d'itérateurs.
Faire des milliers d'appels à addNumber() est tellement pénible. Implémente une fonction membre pour ajouter plusieurs nombres à ton Span en un seul appel.

Si tu n'as aucune idée, étudie les conteneurs. Certaines fonctions membres acceptent une plage d'itérateurs pour ajouter une séquence d'éléments au conteneur.

Chapitre VI Exercice 02 : Mutated abomination

Exercice : 02

Mutated abomination
Répertoire de rendu : ex02/

Fichiers à rendre : Makefile, main.cpp, MutantStack.{h, hpp}
et fichier optionnel : MutantStack.tpp
Fonctions interdites : Aucune

Il est temps de passer à des choses plus sérieuses. Développons quelque chose d'étrange.

Le conteneur std::stack est très bien. Malheureusement, c'est l'un des seuls conteneurs STL qui N'est PAS itérable. Dommage.

Mais pourquoi l'accepter ? Surtout si on peut se permettre de charcuter la stack originale pour lui ajouter les fonctionnalités manquantes.

Pour réparer cette injustice, tu dois rendre le conteneur std::stack itérable.

Écris une classe MutantStack. Elle sera implémentée à partir de std::stack.
Elle offrira toutes ses fonctions membres, plus une fonctionnalité supplémentaire :
des itérateurs.

Bien sûr, tu écriras et rendras tes propres tests pour t'assurer que tout fonctionne comme prévu.

Exemple de test :

```
int main()
{
    MutantStack<int> mstack;

    mstack.push(5);
    mstack.push(17);

    std::cout << mstack.top() << std::endl;

    mstack.pop();

    std::cout << mstack.size() << std::endl;

    mstack.push(3);
    mstack.push(5);
    mstack.push(737);
    //[...]
    mstack.push(0);

    MutantStack<int>::iterator it = mstack.begin();
    MutantStack<int>::iterator ite = mstack.end();

    ++it;
    --it;
    while (it != ite)
    {
        std::cout << *it << std::endl;
        ++it;
    }
    std::stack<int> s(mstack);
    return 0;
}
```

Si tu l'exécutes une première fois avec ta MutantStack, puis une seconde fois en remplaçant MutantStack par, par exemple, une std::list, les deux sorties devraient être

les mêmes. Bien sûr, lorsque tu testes un autre conteneur, adapte le code ci-dessus avec les fonctions membres correspondantes (push() devient push_back(), etc.)