Université de Lyon - Université Lyon 1 Master Informatique - M1
Programmation avancée C++ - M1if02
http ://liris.cnrs.fr/~ rchaine/COURS/M1if02_20_21/ 2020 - 2021

TP à rendre en fin de séance

1 Conditions de l'épreuve

Le travail est à faire en monome, sans interaction avec vos voisins, et sans explications de votre chargé de TP. Vous devez être placé dans la salle qui vous a été affectée dans Tomuss.

Vous aurez uniquement droit :

— à une page de votre navigateur ouverte sur Tomuss (AUCUNE AUTRE PAGE AUTORISEE),

— à un lecteur de pdf ouvert sur les slides du cours ainsi que sur les sujets de TP de l'UE (ces supports peuvent également être imprimés),

— à votre éditeur de code préféré ouvert sur un projet vide.

Cela signifie que vous n'aurez pas droit à internet ni à vos codes écrits précédemment (ni aux corrections fournies). Votre application de mail ou de tchat devra également être fermée, ainsi que votre téléphone portable et vos accès à des dépôts type git. Les pdf des transparents du cours et des sujets de TP comportent des exemple de code qui pourront vous aider pour la syntaxe.

Attention : Le chargé de TP pourra vérifier en cours de séance que vous respectez bien ces règles (liste des processus en train de tourner sur votre machine, fichiers précédemment ouverts dans votre éditeur : penser bien à tout purger au début de la séance). Votre écran doit rester visible tout le temps de la séance avec une luminosité suffisante.

A la fin de la séance vous devrez déposer une archive de votre travail sur Tomuss :

— Attention de ne mettre que les `.h` et `.cpp` ainsi qu'un `Makefile`.

— L'archive (`.zip` ou `.gz` ou `.tgz`) que vous déposez doit porter votre nom.

— **Attention, le dépôt ferme à 17h.**

Le sujet est proche des travaux faits pendant les séances de TP. Vous ne pourrez pas recourir aux `vector`, `deque`, `list`, `slist`, `set` et `map` de la STL, sauf si c'est spécifié dans le texte. Enfin, vous pourrez choisir de coder en utilisant ou non des éléments de la norme `C++11`, `C++14`, `C++17` ou `C++20`. Le but de ce travail est de tester votre capacité à aborder un problème en un temps limité et sans internet. Ce n'est pas inquiétant si vous ne terminez pas tout dans le temps imparti. Vous aurez ensuite du temps pour réfléchir au sujet pendant quelques semaines et vous reviendrez répondre à quelques questions supplémentaires sur le problème dans le cadre d'une interrogation sur papier.

2 Tourner en rond

L'objet de ce TP consiste à mettre en œuvre le type abstrait `Séquence Circulaire` en l'implémentant sous forme d'un `Tableau Dynamique` dont on va modifier l'interface usuelle.

Séquence circulaire

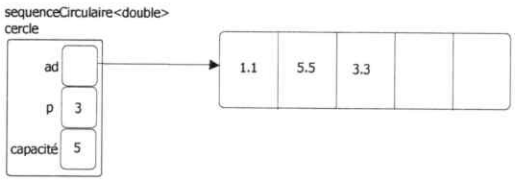
Une `Séquence Circulaire` correspond à une séquence périodique d'éléments. Une telle séquence est caractérisée par son nombre d'éléments `p` dans une période, ainsi que par la valeur de ses éléments. Bien entendu, il est suffisant de connaître la valeur des éléments dans une période pour les connaître tous. La longueur de la période pourra évoluer si jamais on insère un nouvel élément entre deux autres. Les éléments de la `Séquence Circulaire` sont associés à un indice qui pourra être utilisé pour accéder à leur emplacement et leur valeur. Ainsi un des élément de la `Séquence Circulaire` correspondra à l'indice `0`, mais il sera également accessible via les indices

`p, 2p, 3p, ...` ou via les indices `-p, -2p, -3p, ...`. Deux éléments successifs correspondent à des indices consécutifs. Vous pouvez vous représenter une `Séquence Circulaire` comme un cercle divisé en `p` secteurs angulaires de même taille, avec une valeur rangée dans chacun des secteurs. Si une origine des angles a été choisie sur ce cercle, il est possible de mesurer des angles et l'éléments d'indice 0 coincidera avec le secteur angulaire correspondant à l'angle 0.

Implémentation

Concernant l'implementation, nous vous proposons d'implémenter la `Séquence Circulaire` de manière similaire à celle d'un `Tableau Dynamique`, mais avec une interface différente. Le modèle de classe `Séquence Circulaire` que vous allez développer est *templaté* par le type `T` de ses éléments, pourvu que ce type `T` supporte les opérations de construction par défaut, de copie et d'affectation.

La structure de données `Séquence Circulaire` contiendra donc un champ contenant la taille de la période, ainsi qu'un champ contenant la capacité d'une période, c'est à dire la taille maximale de la période qu'il est possible d'atteindre sans recourir à une nouvelle allocation dynamique de mémoire et un déménagement de l'ensemble des éléments. Le dernier champ de la structure de données est un pointeur vers la séquence des éléments composant une période (voir figure représentant une `Séquence Circulaire` de période 3 contenant les valeurs 1.1, 5.5 et 3.3 de type `double`, l'implémentation a alors fixé la capacité à 5).



Fonctionnalités offertes

Le type `Séquence Circulaire` devra offrir les opérations adéquates pour être **initialisé** avec des éléments correspondant à des copies de la valeur v, la longueur de la période p et v étant passées en paramètre. Une `Séquence Circulaire` devra également permettre l'**insertion** d'un élément à une position d'indice donné (la longueur de la période sera alors incrémentée), ainsi qu'un **accès** à un élément à partir de son indice passé **entre crochets**. Vous commencerez par gérer les indices positifs uniquement. Votre module devra également proposer la **recherche** d'un élément de type `T` (en retournant un de ses indices positifs, -1 sinon), ainsi qu'une opération d'**affichage** des éléments d'une période. Vous devrez ajouter (ou bloquer) **toute autre fonctionnalité** qui vous paraîtra nécessaire pour une **gestion saine de l'espace mémoire** en gardant à l'esprit la "règle de 3" ou la "règle de 5".

Programme de test

Vous testerez vos `Séquences Circulaires` dans un programme principal contenant des instructions du type :

```
SequenceCirculaire<double> cercle(10,5.5); // sequence de période 10,
                                           // avec des éléments de valeur 5.5
SequenceCirculaire<double> cercle2(cercle);
std::cout << cercle << std::endl;    // Affichage

for(int i=0;i<30;i++)
    cercle.insere(i, i*0.5);
std::cout << cercle << std::endl;

cercle=cercle2;

for(int i=0;i<30;i++)
```

```
    cercle[i] = i*0.3;
std::cout << cercle << std::endl;

cercle=cercle2;

double elem=rand()%10+0.1; // #include<cstdlib>
while(cercle.find(elem)!=-1) {
   cercle.insere(0, elem);
   elem=rand()%10+0;1;
}
```

Les indices négatifs

S'il vous reste encore beaucoup de temps, vous pourrez gérer l'accès à un emplacement dans le cas où l'indice fourni entre crochets est négatif.

3 Circulez !

Circulateur

Définissez une classe d'itérateur permettant d'accéder aux éléments d'une `Séquence Circulaire` par déréférencement, et de parcourir séquentiellement les éléments de cette `Séquence Circulaire` grâce à l'opération d'incrémentation. D'un point de vue abstrait, une `Séquence Circulaire` contient un nombre infini d'éléments et il devra être possible d'incrémenter l'itérateur quel que soit sa position. Un tel itérateur porte le nom de `Circulateur`.

Implementation

Quelle structure de données proposez-vous pour les `Circulateurs`, sachant qu'un `Circulateur` devra être templaté par le type des éléments contenu dans la `Séquence Circulaire` sur laquelle il travaille. Quelle sont les données membres nécessaires au fonctionnement d'un `Circulateur` sur `Séquence Circulaire` ? Quelles sont les fonctions membres que vous souhaitez d'ores et déjà ajouter à votre classe ?

Opérations offertes

Le type `Circulateur` devra pouvoir être **initialisé** de manière à être positionné sur un emplacement d'une `Séquence Circulaire` passée en argument, ainsi que l'indice. Outre la possibilité d'**incrémentation** (version préfixée uniquement) et de **déréférencement**, vous munirez le type `Circulateur` d'un **test d'inégalité** et de toute autre fonctionnalité qui vous paraitra nécessaire pour exécuter le code suivant avec une gestion saine de l'espace mémoire. Dans la classe `Séquence Circulaire`, il faudra également que vous ajoutiez une fonction `makeCirculateur` qui retourne un `Circulateur` pointant sur l'emplacement d'indice i passé en paramètre. Là encore, l'indice i pourra être plus grand que la valeur de la période.

Moyennant l'ajout de toutes ces fonctionnalités, les itérateurs permettront de parcourir les éléments d'une `Séquence Circulaire` avec des instructions du type :

```
SequenceCirculaire<double> cercle(10,5.5);
std::cout << cercle << std::endl;

for(int i=0 ; i<10; i++)
   cercle[i] = i*0.1;
std::cout << cercle << std::endl;

Circulateur circ(cercle,5); //circ initialisé sur l'emplacement 5 de cercle
Circulateur circend(cercle,4);
```

```
for (  ;circ!= circend; ++circ)
  std::cout << *circ << std::endl;
std::cout << cercle << std::endl;

circ=cercle.makeCirculateur(0);
for (int i=0; i<20; i++) {
  *circ = i+0.2;
  ++circ;
 }
std::cout << cercle << std::endl;
```

Arithmétique des circulateurs

S'il vous reste encore beaucoup de temps, vous pourrez ajouter une opération d'addition d'un `int` i et d'un `Circulateur` c, sans effet de bord sur c. Une telle opération retourne une valeur de type `Circulateur` correspondant à l'état de c s'il était incrémenté i fois.

```
Circulateur circ(cercle,5);
Circulateur circ2(circ);
circ=circ2+4;
```

4 Travail à faire

Votre travail sera décomposé en 2 étapes relatives aux 2 parties de l'énoncé. Merci de placer le code résultant de chacune des étapes dans des répertoires différents, avec un programme `main.cpp` spécifique à chaque étape. Chacun de ces programme principaux devra illustrer le bon fonctionnement des éléments mis en place. Il devra en particulier permettre l'instanciation et l'appel de l'ensemble des classes et fonctions template, et montrer que l'ensemble forme un tout cohérent et sans faille. Vous détaillerez ce que vous avez eu le temps de faire dans un README. Attention de bien vérifier que votre code correspond à une gestion saine de l'espace mémoire.

4.1 Après dépot dans Tomuss

Maintenant on se repose.... Jusqu'à demain!!!!