Задание 3 для первого семестра 2 курса

Требуется разработать и реализовать параметризованный класс списочного типа. Класс должен содержать конструктор без параметров, конструктор копирования, другие конструкторы по смыслу решаемых задач.

Набор методов класса должен соответствовать идеологии работы с соответствующей схемой хранения данных, т.е. обеспечивать требуемую дисциплину доступа к данным (возможно, с некоторыми модификациями). В частности, должны быть методы для добавления, извлечения и удаления элементов, методы для прямого доступа к значениям элементов (в соответствии с заданной дисциплиной), методы опроса состояний, если это необходимо. Для контроля за некорректными ситуациями следует использовать механизм исключений (для фатальных отказов) или в некоторых случаях логические возвращаемые значения (для нефатальных).

Структура, задающая узел списка, реализуется как внутренний класс этого списка, члены этой структуры могут быть public. Все объекты самого списка помещаются в private секцию, а в public секции описываются только интерфейсные методы, определящие дисциплину работы с элементами данных.

Для перемещения по спискам и указания текущих позиций нужно реализовать различные итераторы (прямой, обратный, если он возможен, константный и т.п.). Итераторы реализуются как внутренние классы по отношению к основному классу.

Для класса должен быть переопределен оператор вывода <<, который в наглядной форме изображает состояние данного класса.

Для класса должен быть реализован оператор присваивания и также могут быть переопределены некоторые операции, например, сложение как объединение множеств элементов.

Как правило, для класса надо будет реализовать процедуры сортировки содержимого и поиска требуемого элемента в соответствии с некоторым внешне задаваемым критерием сравнения. Поиск возвращает итератор на найденный элемент.

После реализации собственного списочного класса, его работу надо сравнить с STL реализациями списочных контейнеров (std::list и/или std::forward_list или других подобных). Это означает, что надо придумать некоторую "задачу" для тестирования своей реализации и решить эту задачу с использованием своих классов и классов STL. Интересно сравнить время работы программы в случае вашей реализации и при использовании классов STL.

Подобная тестирующая задача может заключаться в многократном добавлении и удалении элементов в списочную структуру с использованием перемещения текущих позиций и поиска.

Например, заполнить список последовательными значениями $1, \ldots, n$, проходом по списку удалить все нечетные значения, а четные поделить пополам, выполнить присваивание этого списка другому списку, переставить элементы списка в обратном порядке, проверить, что получившийся список содержит последовательные числа $n/2, \ldots, 1$ а исходный содержит числа $1, \ldots, n/2$. Отладку для малых n можно проводить визуально просматривая распечатку состояния списков. Для больших $n \sim 1000000$ для отладки надо придумать способ проверить ответ автоматически. Скажем, в описанном выше примере в цикле значения из списка явно сравниваются с известными $1, \ldots, n/2$.

Дополнительные требования к формулировкам и реализациям конкретного задания обсуждаются в рабочем порядке.

Типы объектов

Релизация тестируется на нескольких различных типов хранимых объектов — "простых" и "сложных". Простые объекты — это примитивные числовые типы (int, double, char). Для них можно построить тесты на большое количество элементов с явной проверкой состояния списков после выполнения серии операций.

Более сложные — структурные типы вроде массивов или числовых классов из предыдущего задания. Тут можно ограничиться небольшой размерностью и визуальной проверкой. Т.е. придется определить операторы вывода и для этих классов.

Наконец, реализация должна работать для "рекурсивного" типа, т.е. список списков векторов и т.п. (например, List<List<int> > и т.п.). Здесь проверяется возможность добавления или извлечения нетривиального класса в подобную списочную структуру.

График выполнения задания

Работа над заданием предполагает еженедельный контроль по следущему графику (естественно, его можно выполнять с опережением)

неделя 1: разработка и согласование описания (интерфейса) класса. Нужно представить описание класса с указанием прототипов требуемых методов (функций) и внутренних объектов класса.

неделя 2: реализация некоторых методов и первичный тест на простых типах данных, проверка работы механизма исключений Нужно представить тест с проверкой работы отдельных функций класса: по принципу "операция – распечатка состояния". В частности, должна быть готова функция распечатки (оператор «). Проверка на разных простых типах. Проверка на некорректных обращениях.

неделя 3. реализация более полного набора методов, тест совместной работы этих методов, тест на сложных типах данных. Тест с "нагрузкой" - выполняются массовые операции с автоматической проверкой результата. Например, добавили 1000000 последовательных целых чисел. Прошлись итератором и удалили четные. Прошлись итератором и проверили, что осталисть только нечетные и при этом ни одно нечетное число не пропало. Другие подобные тесты, смысл которых в многократном выполнении операций в середине и по краям списка. Присваивание списков друг другу. Некоторые ошибки (по памяти) могут не проявиться в однократной операции, а массовая проверка их обнаружит. Тест на "рекурсивных" данных типа список списков векторов :))) Т.е. элементами списка являются тоже списки. Тестирование правильности копирования и доступа. При доступе к элементу мы также получаем возможность работать с этим элементом как с контейнером. Как будет работать процедура распечатки?

неделя 4: Сравнение с STL библиотекой. Получить из указанного начального состояния вашего набора данных другое состояние, определяемое некоторыми правилами, с помошью ваших реализаций и с помощью классов библиотеки STL. Например, в STL есть класс list — список с некоторым набором операций. Вы берете свой набор данных (например, последовательные числа от 1 до N) и преобразуете его по заданным правилам к другому финальному состоянию с помощью ваших реализаций списков и с помощью класса std::list. Сравниваете время, затраченное на обработку данных. Более подробно все детали задачи обсуждается индивидуально в рабочем порядке.

Таким образом, отчетность по этому заданию будет содержать 4 контрольные точки, соответственно данным этапам.

В реальности реализация всех заявленных возможностей и требований может вылиться в весьма большой объем работы. Поэтому не стоит пытаться сделать их все сразу. В первую очередь следует реализовывать то, без чего нельзя обойтись в случае каждого конкретного теста. Далее, расширяя логику и задачи тестирования, можно дописывать недостающие фрагменты кода. Поэтому после понимания постановки задачи и разработки базового интерфейса вашего класса надо сразу формулировать задачи тестов и под эти задачи начинать реализовывать методы вашего класса и сам тестовый код.

Варианты структур данных (т.е. задания)

Далее описаны варианты списков и их внутренней структуры. Возможность и смысл выполнения различных операций со списками зависит от этой структры. Некоторые операции не могут быть выполнены принципиально, для других придется придумывать специальные подходы или вводить дополнительные переменные для идентификации возможных состояний. В каждом конкретном случае можно обсуждать возможность и необходимость таких операций.

1. Динамический массив как список блоков. Элементы последовательно и непрерывно размещаются в блоке памяти некоторого ограниченного размера. Если место в текущем блоке заканчивается, то выделяется новый такой же блок, и новые элементы размещаются в нем, а сами блоки связаны друг с другом указателями как в списке. Доступ к элементам по индексам и по итератору произвольного доступа. При этом придется определять номер блока и номер позиции в блоке, перемещение к требуемому блоку — медленная операция (последовательный проход), к позиции в блоке — быстрая (непосредственное индексирование).

2. Однонаправленный список. Структура с дополнительным элементом перед началом содержательной части списка (позиция before_begin) и нулевым указателем в конце списка. Однонаправленный итератор с одним указателем на текущий элемент.

3. Однонаправленный список. Структура со списком, закольцованным через дополнительный элемент и итератором с парой указателей на соседние элементы. Текущая позиция определяется

по второму указателю из пары. Возможны операции удаления текущего элемента и вставки перед текущим.

4. Двунаправленный список. Структура с нулевыми указателями от крайних элементов. Двунаправленный итератор с одним указателем на текущую позицию. Булевские флаги положения итератора в перед началом (rend) или за концом (end) списка.

5. Двунаправленный список. Структура с дополнительными элементами по краям списка (end и rend). Двунаправленный итератор с одним указателем на текущую позицию.

6. Двунаправленный список. Структура со списком, закольцованным через дополнительный элемент и итератором с одним указателем на текущий элемент (end и rend фактически совпадают).

7. Кольцевой однонаправленный список. Все элементы завязаны в кольцо без какого-либо выделенного элемента. begin() возвращает позицию какого-то (произвольного) элемента списка и итератор запоминает ее как конечную для контроля полного прохода по всем элементам. Тут придется немного подумать как обеспечить полный проход типа `for(i=a.begin(); i!=a.end();++i)`. Удаляется текущий элемент. При этом используется трюк с присваиванием значения от следующего элемента и фактическим удалением следующего.

8. Кольцевой однонаправленный список. Все элементы завязаны в кольцо с дополнительным фиктивным элементом, который и определяет собой позиции начала и конца для итераторов. Список может переставить этот выделенный элемент в другое место кольца (т.е. убрать его со старого места и вставить на новое). При этом все итераторы продолжают на него ориентироваться, т.е. останавливаются именно на позиции текущего положения этого элемента. Таким образом можно пройти по циклу несколько раз, если соответственно изменять позицию начала списа между сдвигами итератора.

9. Дек на базе двунаправленного списка. Полный доступ только по началу и концу дека. Итератор константный двунаправленный только на чтение по всем элементам.

10. Хеш-контейнер на базе однонаправленных списков. Это массив, в котором каждый элемент является списком. Соответственно можно обратиться к i-тому списку и выполнить с ним операции добавления, удаления, доступа к его элементам по его итератору (который действует только в пределах этого списка). Такой итератор также хранит в себе номер своего списка и позволяет его узнать при необходимости. Кроме этого есть итератор по всему контейнеру, который последовательно проходит по элементам всех списков от первого и до последнего. Он также позволяет узнать в каком списке в данный момент находится текущая позиция. Эта текущая позиция может быть преобразована в итератор по этому конкретному списку.