Оглавление

[Тема 1](#_Toc28388277)

[Преамбула 1](#_Toc28388278)

[Задачи 1](#_Toc28388279)

[Часть 1 1](#_Toc28388280)

[Часть 2 2](#_Toc28388281)

[Отчет 3](#_Toc28388282)

# Тема

Изучение свойств контейнеров STL. Применение умных указателей.

# Преамбула

В этом задании вам предстоит исследовать эффективность двух контейнеров с точки зрения затрат на добавление и удаление элементов. Для этого мы заполним контейнер N целыми числами. Будем выполнять M операций добавления и удаления первых элементов.

При этом важно чтобы измеряемое время было от нескольких микросекунд, до 50-60 микросекунд. Такое ограничение связано с тем, что Windows – многозадачная система. Она достаточно часто передает управление разным процессам. Если такая передача возникнет во время измерения, это сильно повлияет на результат.

Во второй части вы проделаете упражнения c “умными указателями” unique\_ptr и shared\_ptr.

Для выполнения задания необходимо изучить материал лекции 4.

# Задачи

## Часть 1

1. Начините новый консольный проект, назовите его ListOrVector. Подключите к нему потоки ввода-вывода.
2. Подключите модули list и vector. Объявите в main список (*list*) целых.  
   Убедитесь, что проект собирается и выполняется. Если необходимо, устраните проблемы.
3. Объявите 2 целочисленных переменных N и M. N будет определять количество элементов в контейнере, а M – количество операций вставки и удаления элемента. В качестве N можно для начала взять 200, и M = 2000.

В цикле от 0 до N заполните список числами i.

1. В цикле M раз выполните добавление элемента в начало списка и затем его удаление.
   1. Обратитесь к лекции 4, чтобы уточнить детали.
   2. Перед циклом объявите целочисленную переменную, например, k и увеличивайте ее как после вставки, так и после удаления элемента.
   3. По завершении цикла выведите значение k на консоль. Если все сделано правильно, то она должна быть равна 2\*M.

Переменная k важна, так как без нее компилятор может заметить, что цикл “ничего не делает” и просто игнорирует его в процессе оптимизации!

1. Настало время измерять производительность.
   1. Включите заголовочный файл Windows.h;
   2. Объявите перед измеряемым участком (циклом удаления-вставки) 3 переменные   
      LARGE\_INTEGER li1, li2, lif;
   3. Сразу за ними вызовите 2 метода, которые включат секундомер:  
      ::QueryPerformanceFrequency(&lif);

::QueryPerformanceCounter(&li1);

* 1. Сразу за циклом, еще до вывода k на консоль вызовите ::QueryPerformanceCounter(&li2);
  2. Теперь время в микросекундах можно вычислить следующим образом:  
     int mcSec = (int)((double)((li2.QuadPart-li1.QuadPart)\*1000000)/(double)lif.QuadPart);

1. Убедитесь, что вы компилируете программу в моде Release. Закройте все ненужные приложения. Подберите Nи M так, чтобы время было 15-20 микросекунд. Для ориентира, на моем компьютере это N=200 и M=200.
2. Ведите журнал ваших измерений. Постепенно увеличивая N – размер контейнера, засекайте время вычислений, пусть N изменится в 5-6 раз.

Постройте график для полученных данных в Excel и определите характер зависимости t от N для списка.

1. Используйте вместо списка динамический массив. Добавляйте и удаляйте элементы методами insert и erase, а для указания первого элемента методом <имя объекта>.begin().
   1. Снова подберите параметры N и M, такие, чтобы время вычисления было чуть меньше 10 микросекунд. Для примера, у меня это N=200 M=200.
   2. Увеличивайте N каждый на 200 элементов и ведите журнал. Получите 4-5 записей.
   3. Постройте график в Excel, используя логарифмическую шкалу для N. Выведите характер зависимости времени от N.

На этом программная часть 1 закончена.

Опишите эту часть задания в Word-документе. В отчете должны быть приведены результаты ваших исследований, а также объяснение результатов с точки зрения особенностей контейнеров.

## Часть 2

В лекции 4 были рассмотрены “умные указатели” *shared\_ptr* и фабрика класса *make\_shared*. Это удобный инструмент, но возникают сложности с владением указателя на массив данных.

Рассмотрим эту проблему, и познакомимся с 2-мя ее решениями.

1. Добавьте в решение новый консольный проект с именем SmartPointers. Подключите консольный ввод/вывод и модуль <memory>в котором определены классы “умных” указателей.
2. Добавьте в проект класс Test с “говорящими” конструктором и деструктором (конструктор и деструктор сообщают на консоль о своем выполнении).
   1. Добавьте в Test публичное поле int Val. Пусть оно зануляется в конструкторе.
   2. Включите заголовочный файл Test в основной модуль (там, где main).
3. В main добавьте строчку shared\_ptr<Test> sp (new Test[3]); и выполните проект. В строке создается умный указатель, который инициирован указателем на массив из 3-х объектов Test.   
   Выполните программу, изучите результат и осознайте проблему.
4. Решить проблему можно указывая тип-параметр как массив. shared\_ptr<Test[]> sp (new Test[3]);. Выполните проект снова и оцените произошедшие изменения!
5. Альтернативный вариант владения массивом - использовать другой конструктор для shared\_ptr.   
   shared\_ptr<Test> sp (new Test[3], default\_delete<Test[]>());  
   Вторым параметром здесь вызывается фабрика класса параметров оператора *delete*.

Выполните проект снова и оцените произошедшие изменения!

Часть проблемы решена.

1. Остается еще вопрос. Как использовать такой указатель для доступа к отдельному объекту в блоке. Оператора [] у shared\_ptr нет.   
   Для этого служит метод get(), который возвращает указатель, которым владеет shared\_ptr. А вот к этому указателю можно применять оператор [].  
     
   Воспользуйтесь этим методом, для того, чтобы задать различные значения Val для разных объектов.
2. Проверьте, что значения были успешно изменены. Выведите их в цикле на консоль.

Использование shared\_ptr не всегда оправдано. Например, если в классе имеется единственный указатель на блок данных. Других пользователей этого блока нет. Следить за количеством пользователей такого указателя нет смысла – он всегда 1. Такая ситуация часто складывается, когда речь идет о скрытых членах класса.

Вместо указателя shared\_ptr в таких случаях можно использовать умный указатель unique\_ptr. Такой указатель может быть только 1. Его нельзя копировать или присваивать.

1. Объявите unique\_ptr<Test[]> p(new Test[3]); и закомментируйте все другие команды в main, чтобы не загромождать консоль.  
   Выполните программу и убедитесь, что утечки памяти нет – создается и разрушается по 3 объекта.
2. Теперь попробуем обойтись без **new**. Вместо указателя *p(new Test[3])* используйте *p = make\_unique<Test[]>(3);* Убедитесь, что через **p** можно получать доступ к элементам массива, например, p[1].
3. Задайте различные значения Val для разных объектов блока используя оператор [] для умного указателя p. Выведите их в цикле на консоль, чтобы подтвердить изменения.
4. Посмотрим, как можно заменить текущее значение, охраняемое unique\_ptr:
   1. Поместите вывод “---------” на консоль после вывода значений массива.
   2. Выполните программу и убедитесь, что объекты разрушаются *после* строки вывода (когда заканчивается блок main).
   3. Поместите после вывода строку p.reset(new Test[2]); и добавьте еще одну строку вывода “---------”. *Здесь make\_unique применить не получится!*
   4. Выполните программу и убедитесь, что при выполнении команды reset были не только созданы новые объекты Test, но и разрушены предыдущие, которыми владел указатель. А новые объекты были разрушены по окончании блока main.

На этом программная часть задания завершена.

Вероятно, теперь вы лучше понимаете зачем и как действуют умные указатели.

# Отчет

Расширьте свой отчет (часть 1). Добавьте в него описание назначения и действия умных указателей shared\_ptr и unique\_ptr. Остановитесь на их особенностях. В каких случаях удобнее применять один тип, а в каких другой.

Общий размер отчета не должен превышать 1 стр.

Очистите решение от вспомогательных файлов и результатов компиляции и поместите каталог в архив.

Архив с решением и отчет в Word отправляйте как ответ на выполнение задания.