**1 Ошибки.**  
  
**1.1 Планируемые ошибки**  
  
Обычной практикой должна стать практика максимального упрощения кода.  
  
Ошибки - которые мы можем проверить и запланировать их компенсацию и/или устранение   
являются в нашем случае предсказуемыми и/или планируемыми, т.е. не являются исключительными ситуациями, они скорее являются вполне запланированными событиями. Эти ошибки мы можем устранить либо компенсировать, ещё ДО их вероятного появления путём изменения алгоритмов и стиля кодирования таким образом, что ошибки никак возникнуть, либо если возникнут не смогут никак себя проявить, хотя быть может очень будут этого хотеть.   
  
В основе компенсации ошибок можно положить принцип наименьшего количества ветвлений.  
иными словами если можно обойтись без ветвления, необходимо обойтись. Циклы являются одной из разновидностей ветвлений, поэтому избыточность циклов так-же необходимо устранять.   
И уж совсем по возможности необходимо избавляться от избыточных ветвлений в теле циклов, вложенных циклов с вложенными ветвлениями. Необходимо сокращать тела циклов на сколько это возможно, как можно чаще применять операторы break и continue, они придают наглядность коду.  
Избыточные циклы и ветвления, это настоящее удобрение для ошибок.  
  
Простейший пример сокращения кода. Возможно после компиляции всё будет не так очевидно и   
если мы посмотрим на результирующий код через дизассемблер, всё будет не сильно проще, но мы всегда должны беспокоиться о двух главных читателях нашего кода,1 – компилятор , 2 – сопровождающий другой программист, который так-же возможно станет сопровождать наш код.  
  
У медиков есть такое понятие как скомпенсированная болезнь. Вот пока организм компенсирует  
повреждение того, или иного органа болезнь никак не проявляется и симптомы не обнаруживаются, как только происходит декомпенсация, болезнь манифестирует и иногда бывает слишком поздно. Наша с вами друзья задача, организовать компенсацию возможной болезни самой организацией кода, таким образом, чтобы декомпенсация не наступала. Возможно, я не всегда знаю, как делать необходимо и правильно. Но я понял, как делать точно нет никакой необходимости.   
Для манифестации безвыходной ситуации, т.е. наступления декомпенсации системы.существует множество способов. Однако нам следует начать с компенсации.  
  
Вот пример простейшей Хэш функции. Её главная задача по двум аргументам сделать   
уникальный хэш. Предположим у нас есть строка, и наша задача вытащить из неё все элементы удовлетворяющие /некоторому условию и количество их повторений.

Есть некоторая структура:

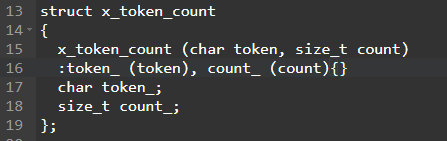
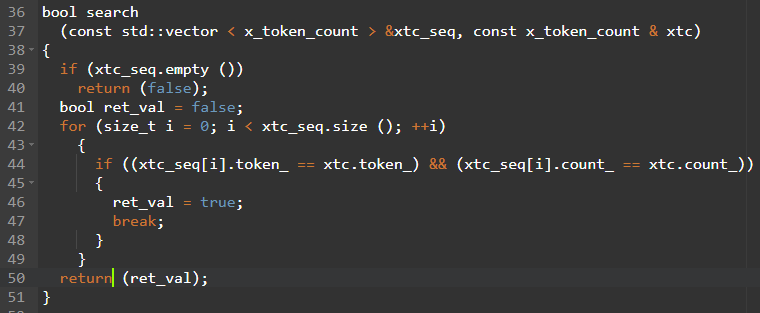
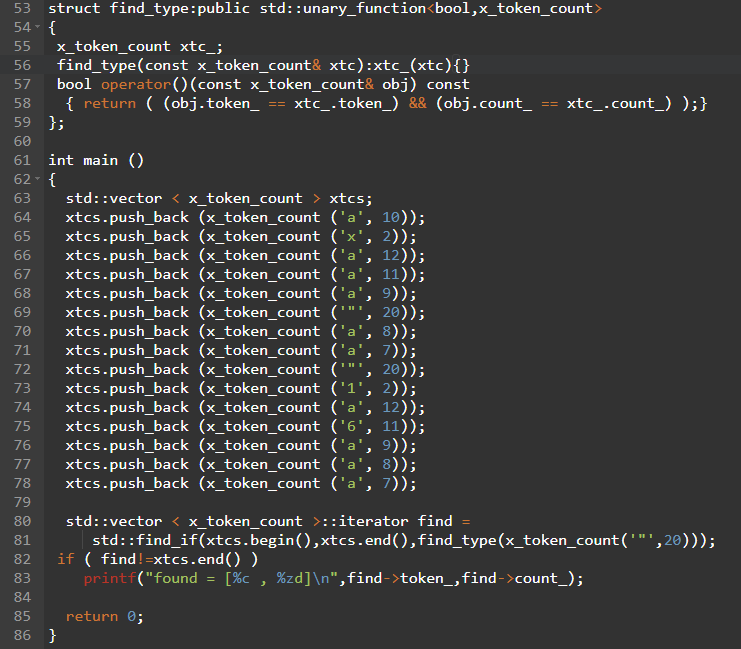


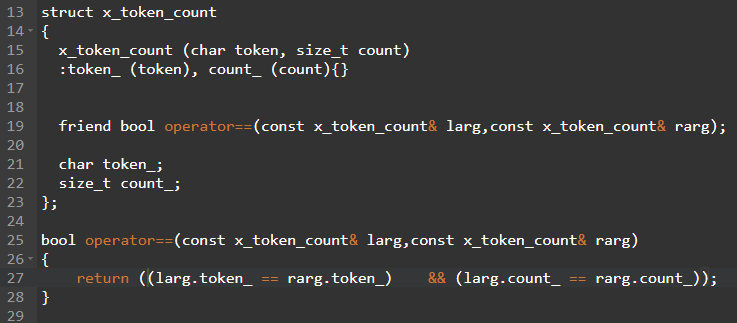
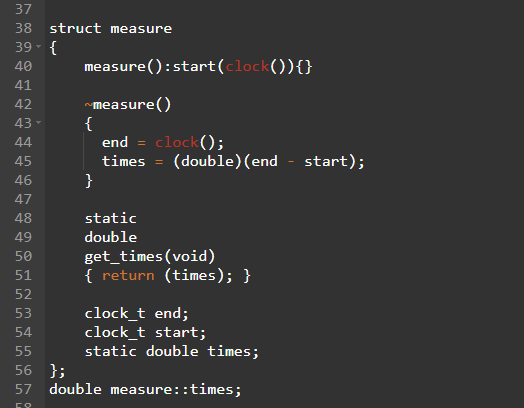
Рис 1.1.1 экспериментальная структура данных.

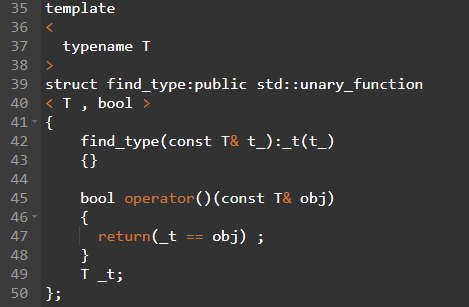
Есть также некоторая последовательность этих структур, она может быть выражена любым

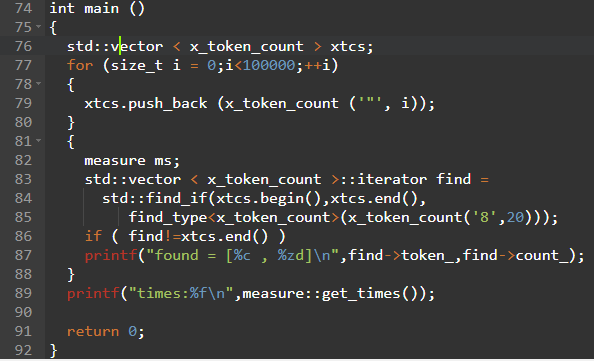
stl контейнером. Например: std::vector< x\_token\_count> xtc\_seq;

Нам нужно найти объект удовлетворяющий условию.

X\_token\_count xtc(‘”’,20);  
вот попытка реализации алгоритма поиска:  
  
Рис 1.1.2 простейшая реализация поиска вхождение объекта типа x\_token\_count в вектор.  
На первый взгляд, выглядит метод search вполне пристойно. Но, как всегда есть ньюанс.  
1. Слишком много кода в методе search, при том, что полезной информации метод предоставляет слишком мало. Вероятно, обычно недостаточно знать, что, искомый элемент обнаружен. Иногда хотелось-бы знать его позицию в векторе, а лучше сколько их всего, и позиции. каждого В дальнейшем метод скорее всего подвергнут рефакторингу , чтобы получить больше информации.  
Какие есть положительные стороны в search()? Ну с точки зрения вероятности возникновения ошибок, код достаточно неплох. Сразу происходит проверка на пустоту вектора, после которой нет смысла код смотреть дальше. Предположение о том, что вектор может быть пуст, мы отметаем сходу, это даёт нам возможность не рассматривать код, после строки 40, просто потому , что он не выполняется там.  
Иными словами нажимаем step over в отладчике и нас вытолкнет из метода search.   
Все параметры передаются по константным ссылкам, это даёт нам возможность не беспокоиться о конструкторах и деструкторах вектора xtc\_seq и объекта xtc, создание и уничтожение этих параметров метода search не входит в его обязанности. Единственная локальная переменная метода search объявлена и инициализирована в строке 41 , не должна вызывать беспокойства потому что её уничтожением займётся сам компилятор, как только произойдёт выход из области видимости метода search , значение этой переменной и вернёт метод.  
Конструкция for которую мы применяем для реализации цикла, помогает нам избежать логических ошибок. Связанных с выходом за пределы вектора, и даже если мы случайно поставим точку с запятой после скобки условия цикла, компилятор нам сообщит о том, что не может найти переменную цикла i. Это сразу даст нам подсказку где искать концы.  
в случае стандартной библиотеки , всё гораздо проще. Код сокращается значительно.   
Соответственно можно предположить, что вероятность возникновения ошибок так-же сильно снижается. Алгоритмы библиотеки STL благополучно используются десятки лет, они оптимизированы и проверены. Множеством программистов. И если вы , достаточно хорошо ими владеете это существенно сократит вероятность ошибок.  
Листинг 1.1.3  
Обобщение метода search.Можно немного переписать метод, и убедиться в том, что эффективность может быть увеличена без ущерба качеству кода.  
Пример:  
Мы можем добавить оператор сравнения в структуру x\_token\_count, стандарт не запрещает нам этого делать. Внимательный читатель мог заметить что мы и конструктор с параметрами использовали ранее. Вот как выглядит структура сейчас.

  
  
Сравнение элементов теперь выполняется в соответствующем перегруженном операторе структуры. обобщим метод search, и постараемся сделать его несколько эффективней за счёт сокращения количества итераций в цикле.  
Для анализа скорости выполнения метода, напишем простейший код.Измеряющий код работает просто, в момент создания объекта в конструкторе инициализируется время старта .  
В деструкторе время конца выполнения, после чего расчитывается переменная times, которая и содержит в себе время выполнения кода.  


Листинг 1.1.4  
Поскольку я использую online компилятор, мне приходится профайлингом заниматься, вот както так, по простому.  
Измерим скорость выполнения примера кода с листинга 1.1.3.  
Для этого изменим немного код, таким образом , чтобы std::find\_if, совершил полный проход по  
всем 100000 элементам вектора.  
Функтор модифицируем таким образом:  
  
Листинг 1.1.5  
Функтор стал значительно понятнее. После реализации оператора operator==().

  
Листинг 1.1.6  
Результат измерения 100000 элементов алгоритм std::find\_if прошёл за   
times:1570 .  
Попробуем применить наш, самодельный алгоритм.  
 Если std::vector заменить на std::list times:2204