**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:

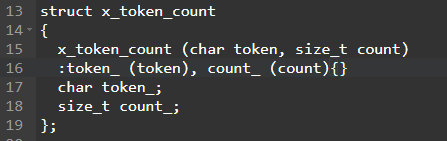
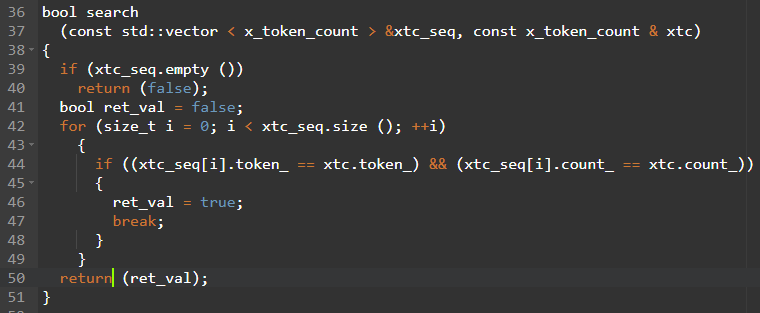


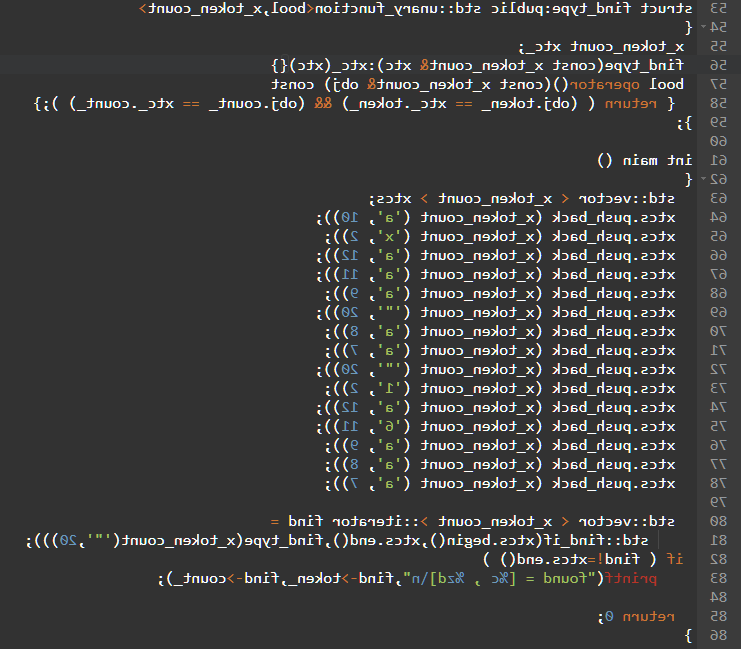
Рис 1.1.1 экспериментальная структура данных.

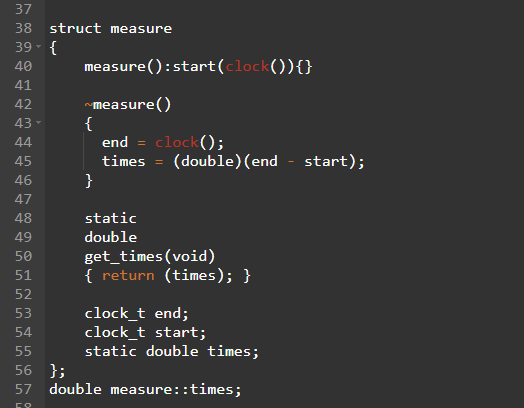
Есть также некоторая последовательность этих структур, она может быть выражена любым

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

Нам нужно найти объект удовлетворяющий условию.

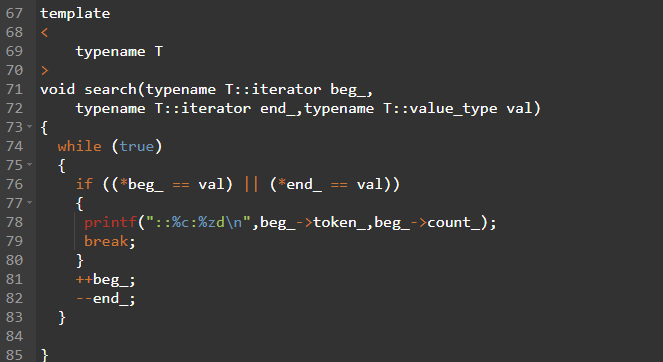
X\_token\_count xtc(‘”’,20);  
вот попытка реализации алгоритма поиска:  
  
Рис 1.1.2 простейшая реализация поиска вхождение объекта типа x\_token\_count в вектор.  
На первый взгляд, выглядит метод search вполне пристойно. Но, как всегда есть ньюанс.  
1. Слишком много кода в методе search, при том, 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.

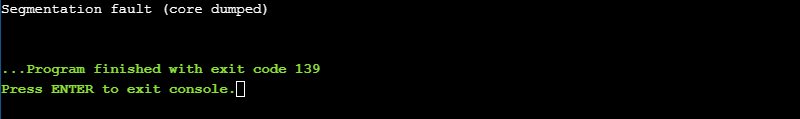


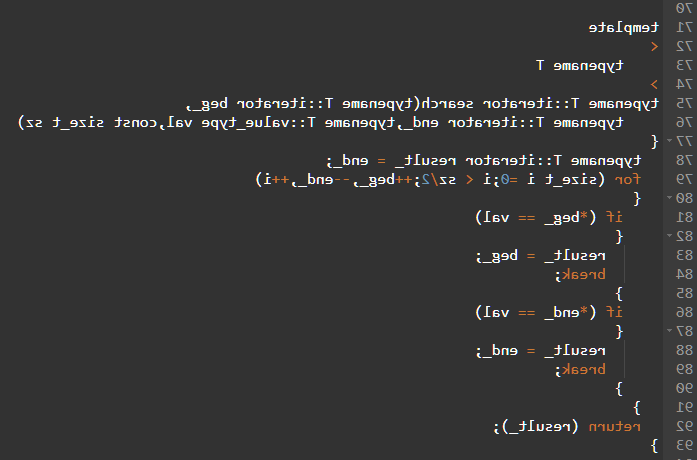
Сравнение элементов теперь выполняется в соответствующем перегруженном операторе структуры. обобщим метод search, и постараемся сделать его несколько эффективней за счёт сокращения количества итераций в цикле.  
Для анализа скорости выполнения метода, напишем простейший код.Измеряющий код работает просто, 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.  


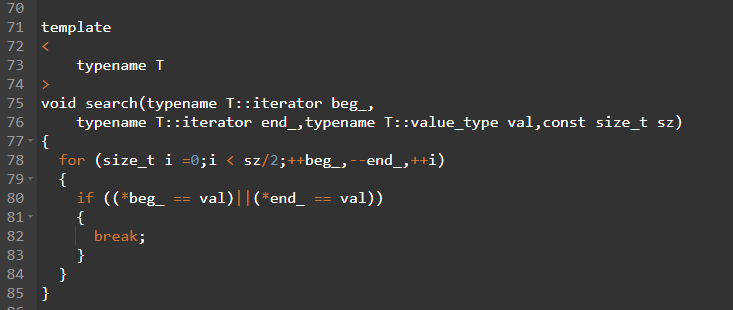
Листинг 1.1.4 Поскольку я использую online компилятор, мне приходится профайлингом заниматься, вот как-то так, по простому.  
Измерим скорость выполнения примера кода с листинга 1.1.3.  
Для этого изменим немного код, таким образом , чтобы std::find\_if, совершил полный проход по  
всем 100000 элементам вектора.  
Создадим блок(строка 81-88, именно в этом блоке отработает конструктор и деструктор объекта ms):  
Функтор модифицируем таким образом:  
Листинг 1.1.5  
Функтор стал значительно понятнее. После реализации оператора operator==().

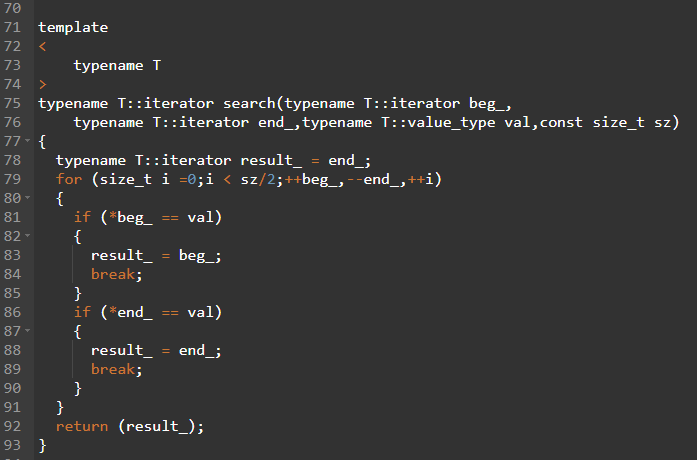
Листинг 1.1.6  
Результат измерения 100000 элементов алгоритм std::find\_if прошёл за   
times:1570 .  
Попробуем применить наш, самодельный алгоритм.  
 Если std::vector заменить на std::list times:2204

Изменим немного наш алгоритм search, сократим кол-во итераций примерно в 2-а раза, этот способ взят из одной из реализаций алгоритма пузырьковой сортировки:  
  


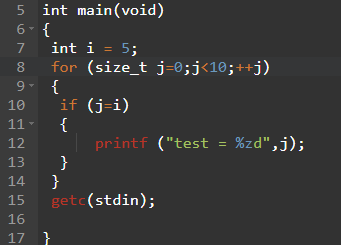
Листинг 1.1.7 Модифицированный и несколько оптимизированный алгоритм линейного поиска в контейнере.  
  
Алгоритм стал примерно в 2-а раза быстрее, просто потому, что теперь кол-во итераций станет значительно меньше, поиск идёт практически одновременно и с хвоста и с начала.  
  
Рис. 1.1.8 Критическая ошибка, вероятно попытка разыменовать недействительный итератор.



Но мы видим как минимум один серьёзный недостаток, который неминуемо   
приведёт к крайне печальному результату Рис 1.1.8 : Если аргумент val не будет присутствовать в контейнере итераторы которого используются, всё рано или поздно будет как на Рис 1.1.8..  
После того как это случится, возобновить работу программы будет крайне сложно.   
Именно поэтому я предлагаю только один выход. Всегда передавать в цикл определённое кол-во итераций. как можно реже использовать бесконечные циклы с одним условием, вероятность наступления которого P(A) != 1. Лично я сторонник циклов for, с ясным описанием условий начала и конца. И да, я всегда настаиваю на ещё хотя одном условии завершения цикла , вероятность наступления которого P(A)==1, в таймфрейме не приводящем к критическим ошибкам.  
Как исправить такой код(Листинг 1.1.7)? Вероятный выход из положения выглядит как показано на (Листинг 1.1.8) необходимо просто добавить ещё один параметр – в котором хранится кол-во элементов в исследуемом контейнере.   
  
Листинг 1.1.8 Модификация листинга 1.1.7 предупреждения бесконечного цикла и выхода за пределы диапазона действительных итераторов контейнера.   
  
Добавление дополнительного параметра size\_t sz – позволяет нам задать конец выполнения цикла.  
Поскольку из-за использования метода прохода двумя итераторами, кол-во итераций будет в два раза меньше, чем количество элементов в контейнере(именно поэтому sz/2).

Листинг 1.1.9 Алгоритм линейного поиска с возвращаемым итератором и дополнительным параметром для контроля итераций цикла.

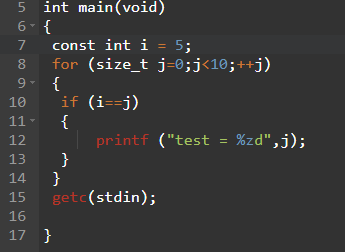
Обратите внимание насколько удобней стало читать условия начала и завершения цикла.   
Всё в строке инициализации цикла for всё сразу понятно, не надо нигде искать дополнительных условий.Код цикла стал короче хотя и проверок больше.  
Вот примерно конечный вариант цикла(Листинг 1.1.9) с контролем количества итераций независимым от значения искомого параметра и наличия его в контейнере. Примерно так выглядит конечная версия нашего кода для поиска элемента в контейнере, сложность почти линейная, хотя есть конечно некоторые преимущества. Алгоритм возвращает итератор контейнера в котором содержится искомый элемент.  
Мы можем собрать все искомые итераторы контейнера, в контейнер с итераторами.  
  
2. Ветвления.  
2.1 Как всегда lvalue.   
  
Иногда нам кажется что существуют участки кода, в которых мы просто никогда не сможем допустить ошибку, даже будучи сильно пьяными, не попадая в нужные кнопки пальцами, мы всё равно, не ошибёмся в некоторых образцах простого кода. Поверьте друзья именно там как правило и следует искать ошибки. Даже если у вас гениальный стиль программирования, вы гениальны и внимательны как сам дьявол, но вы не один на земле и ваш когд может сопровождать такой-же человек, который может проснулся слишком рано, может устал из-за дедлайна а может и попросту выпил слишком много алкоголя, возможно он и совершит такую мелкую незаметную ошибку. Речь идёт об ошибках невнимательности, искать их можно стуками. И чем больше их ищешь, тем больше вероятность допустить ещё такую-же и ещё и ещё. Дальше следует лавинный пробой, метание клавиатурой в стену и истерика. Но ведь ситуацию почти всегда можно предотвратить.

Листинг 2.1.1 Простейшая иллюстрация ошибки начинающих.  
Код в Листинге 2.1.1 прекрасен во всём. А именно в том , что он хотя-бы не приведёт к бесконечному циклу, цикл завершится сразу.  
Однако есть и более неприятный вариант.

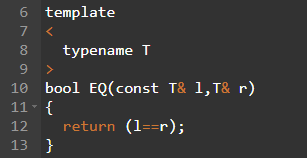
Листинг 2.1.2 Тот-же код что и в Листинг 2.1.1 только без оператора break.

Если в данном примере ветвления не будет оператора break(Листинг 2.1.2); цикл станет бесконечным как вселенная, ну или как человеческая глупость. Что-же делать? Можно задаться вопросом. Как быть , как выйти из положения? Всё просто. Главное правило не пытайтесь, нигде и никогда ставить переменную цикла как lvalue с лева, дело в том , что оператор присваивания ожидает lvalue всегда с левой стороны.

Листинг 2.1.3 Вот именно так должен выглядеть код, даже если оператор сравнения будет каким-то образом случайно обозначен как оператор присваивания, это точно не приведёт к бесконечному циклу. Как защититься от оператора присваивания в ситуации когда нужен оператор сравнения? Очень просто.



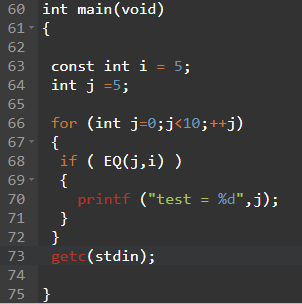
Листинг 2.1.4 Использование константы.

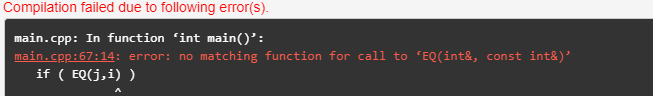
 Теперь (Листинг 2.1.4 )заменить оператор сравнения на присваивание будет сложно, переменная I — константа. Компилятор сразу даст по рукам, сообщив о том, что кто-то что-то присваивает константе.  
Какой можно придумать ещё способ спастись от такой неприятной ситуации? Самый простой, написать метод использующий вывод аргументов шаблона например как в листинге(Листинг 2.1.5)

Листинг 2.1.5 вывод аргументов шаблона в функции EQ

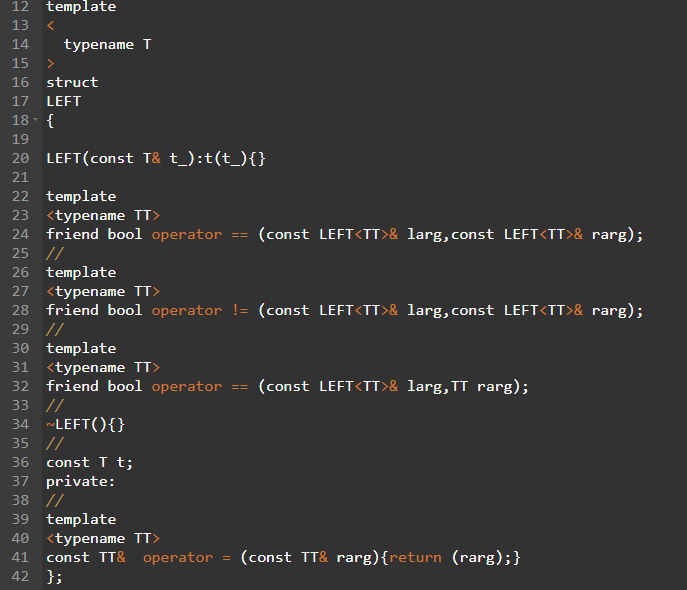
Если использовать метод EQ для контроля аргументов результат будет аналогичным, но придётся выполнять контроль типа аргументов при сравнении в рантайме.

Это не приятно. Но гораздо лучше чем бесконечный или обрывающийся цикл.

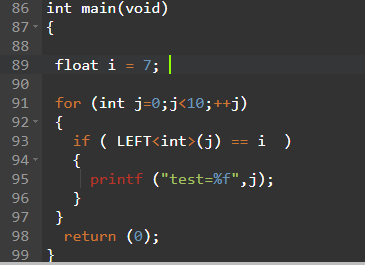


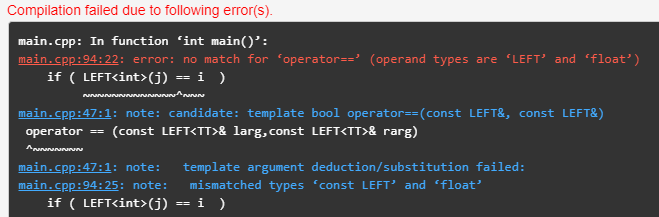
Листинг 2.1.6 Использование метода EQ.   
Теперь если попробуем использовать переменную цикла как lvalue получим сообщение от компилятора:

Листинг 2.1.7 Ошибка при попытке использовать иной порядок аргументов, кроме того, который задан в EQ. Какой минус такого параноидального подхода. Минус в том что метод придётся делать для всех операторов сравнения где есть символ «=». это не хорошо, вызывает дополнительные затраты и путаницу. Не каждый человек даже с комментариями поймёт зачем именно в операторе сравнения нужен вызов метода EQ(..).  
Чтобы избежать подобных проблем ,можно написать простейший класс, реализующий операторы ==, != , <= , => и т. д. И сравнивать не непосредственно переменные, а их копии в объектах класса.  
Объект такого класса , живёт ровно до конца условия , оператора сравнения. После выполнения своей функции, благополучно ликвидируется. Реализация такого крайне проста, не требует особых знаний, и выполняется любым новичком. Пример. назовём наш небольшой класс(обычно я использую структуры, разница между классом и структурой лишь в способе инкапсуляции данных, в классе по умолчанию все поля private: и надо указывать что именно public:, в структуре всё наоборот.) Создадим структуру с именем LEFT<T>(Листинг 2.1.8) чтобы точно показать её место среди аргументов оператора сравнения ( LEFT<int>(z)==i ) например. Мало того, что мы обозначаем что переменная z,в нашем случае с лева, мы так-же довольно однозначно сообщаем что она типа int и сравнивать её с переменной другого типа мы никак не хотим.  
К тому-же наличие подобного класса позволит нам перегрузить все операторы сравнения, чтобы потом не путаться в том какой метод вызывать и в каком случае. Это-же позволяет нам, поместить оператор присваивания в поле private: что даёт нам возможность запретить присваивание нашему объекту.

Листинг 2.1.7 Класс реализующий простейшую страховку от невнимательности сопровождающих код.  
  
Пример применения класса LEFT<T>.

Как видим оператор присваивания в классе LEFT<T> находится в секции private, на случай если кто-то решит что-то присвоить объекту.   
Реализованы операторы сравнения в виде друзей LEFT<T> реализация их тривиальна.



Листинг 2.1.8 Демонстрационный код объекта предохранителя от случайного присваивания.  
Как мы видим защита работает, даже на этапе компиляции. В данном случае проявился ещё один плюс такого подхода, контроль типа аргумента при сравнении.