Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

**Факультет прикладной математики и информатики**

ФУРС КОНСТАНТИН ВИКТОРОВИЧ

**ОТЛАДКА ПРОГРАММ:**

Реферат  
студента 1 курса 2 группы

Минск 2017

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ……………………………………………………………………………………….……3

1 Отладочная С-библиотека времени выполнения DCRT…………………………...............4

1.1 Включение обнаружения утечек памяти……………………………………….…….4

1.2 Интерпретация отчета об утечке памяти…………………………………………….5

1.3 Задание точек останова для номера выделения памяти…………………………6

1.4 Сравнение состояний памяти………………………………………………………….7

1.5 Ложные срабатывания…………………………………………………………………..8

2 Отладка в Visual Studio 15………………………………………………………………………...9

2.1 Настройка сборки…………………………………………………………………………9

2.2 Сборка кода……………………………………………………………………………….10

2.3 Просмотр списка ошибок………………………………………………………………..10

2.4 Просмотр подробных сведений об ошибках…………………………………………11

2.5 Выполнение статического анализа кода……………………………………………..11

2.6 Использование значка лампочки для исправления или рефакторинга кода…..11

2.7 Отладка выполняемого кода……………………………………………………………12

2.8 Задание простых точек останова………………………………………………………13

2.9 Задание условных точек останова……………………………………………………..13

2.10 Проверка кода во время исполнения…………………………………………………14

ЛИТЕРАТУРА…………………………………………………………………………………………...14

**РЕФЕРАТ**

Реферат, 14с., 1 источник.

**Ключевые слова:** CRT, Visual Studio, \_CrtDumpMemoryLeaks, malloc, отладка, ошибка, точка останова, лампочка, рефакторинг.

**Объект исследования** – отладка программ.

**Цель работы** – изучение отладки программ.

**Методы исследования** - использование отладочной С-библиотеки времени выполнения DCRT и отладки в Visual Studio 15.

**Область применения** – разработка программ.

**ВВЕДЕНИЕ**

Отладка программ – этап разработки программы, на котором обнаруживают и устраняют ошибки. Так как без отладки программы сама программа не всегда работает, то эта тема весьма актуальна. На сегодняшний день имеются различные методы отладки программ. В этом реферате исследуются отладочная С-библиотека времени выполнения DCRT и отладка в Visual Studio 15. В данном реферате изучены и получены методы отладки программ.

1. **Отладочная С-библиотека времени выполнения DCRT**

Утечки памяти, определяемые как сбой при освобождении ранее выделенной памяти, — это одна из наиболее трудно обнаруживаемых ошибок в приложениях C/C++. Небольшая утечка памяти сначала может остаться незамеченной, но постепенно нарастающая утечка памяти может приводить к различным симптомам, от снижения производительности до аварийного завершения приложения из-за нехватки памяти. Более того, приложение, в котором происходит утечка памяти, может использовать всю доступную память и привести к аварийному завершению другого приложения, в результате чего может быть непонятно, какое приложение отвечает за сбой. Даже безобидная на первый взгляд утечка памяти может быть признаком других проблем, требующих устранения.

Отладчик Visual Studio и библиотеки времени выполнения C (CRT) предоставляют средства для обнаружения утечек памяти.

* 1. Включение обнаружения утечек памяти

Основным средством для обнаружения утечек памяти является отладчик и отладочные функции кучи библиотеки времени выполнения C (CRT).

Для включения отладочных функций кучи нужно вставить в программу следующие операторы:

#define \_CRTDBG\_MAP\_ALLOC #include <stdlib.h> #include <crtdbg.h>

Для правильной работы функций CRT операторы #include должны следовать в приведенном здесь порядке.

Включение заголовочного файла crtdbg.h сопоставляет функции malloc и free с их отладочными версиями, \_malloc\_dbg и free, которые отслеживают выделение и освобождение памяти. Это сопоставление используется только в отладочных построениях, в которых определен \_DEBUG. В окончательных построениях используются первоначальные функции malloc и free.

Оператор #define сопоставляет базовые версии функций кучи CRT соответствующим отладочным версиям. Если оператор #define не используется, дамп утечки памяти будет менее подробным.

После того как с помощью этих операторов будут включены отладочные функции кучи, можно поместить вызов \_CrtDumpMemoryLeaks перед точкой выхода приложения для отображения отчета об утечке памяти перед завершением работы приложения:

\_CrtDumpMemoryLeaks();

Если у приложения несколько точек выхода, не требуется вручную размещать вызовы функции \_CrtDumpMemoryLeaks в каждой точке выхода. Вызов функции \_CrtSetDbgFlag в начале приложения приведет к автоматическому вызову функции \_CrtDumpMemoryLeaks в каждой точке выхода. Необходимо установить два показанных здесь битовых поля:

\_CrtSetDbgFlag ( \_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF );

По умолчанию \_CrtDumpMemoryLeaks выводит отчет об утечке памяти в область **Отладка** окна **Вывод**. \_CrtSetReportMode можно использовать для перенаправления отчета в другое расположение.

Если используется библиотека, она может переустановить вывод в другое расположение. В этом случае можно вернуть вывод обратно в окно **Вывод**, как показано ниже:

\_CrtSetReportMode( \_CRT\_ERROR, \_CRTDBG\_MODE\_DEBUG );

* 1. Интерпретация отчета об утечке памяти

Если приложение не определяет \_CRTDBG\_MAP\_ALLOC, \_CrtDumpMemoryLeaks отображает отчет об утечке памяти, выглядящий следующим образом:

Detected memory leaks! Dumping objects -> {18} normal block at 0x00780E80, 64 bytes long. Data: < > CD CD CD CD CD CD CD CD CD CD CD

CD CD CD CD CD Object dump complete.

Если приложение определяет \_CRTDBG\_MAP\_ALLOC, отчет об утечке памяти выглядит следующим образом:

Detected memory leaks! Dumping objects -> C:\PROGRAM FILES\VISUAL STUDIO\MyProjects\leaktest\leaktest.cpp(20) : {18} normal block at 0x00780E80, 64 bytes long. Data: < > CD CD CD CD CD CD CD CD CD CD CD CD

CD CD CD CD Object dump complete.

Разница заключается в том, что во втором отчете отображается имя файла и номер строки, в которой впервые было произведено выделение утекающей памяти.

Независимо от определения \_CRTDBG\_MAP\_ALLOC, в отчете об утечке памяти отображаются следующие сведения:

* Номер выделения памяти, в этом примере — 18.
* Тип блока, в этом примере — normal.
* Расположение памяти в шестнадцатеричном формате, в этом примере — 0x00780E80.
* Размер блока, в этом примере — 64 bytes.
* Первые 16 байт данных в блоке, в шестнадцатеричном формате.

В отчете об утечке памяти блок памяти может определяться как обычный, клиентский или CRT. *Обычный блок* — это обыкновенная память, выделенная программой. *Клиентский блок* — особый тип блока памяти, используемой программами MFC для объектов, для которых требуется деструктор. Оператор new в MFC создает либо обычный, либо клиентский блок, в соответствии с создаваемым объектом. *Блок CRT* — это блок памяти, выделенной библиотекой CRT для внутреннего использования. Освобождение этих блоков производится библиотекой CRT. Поэтому маловероятно увидеть их в отчете об утечке памяти — разумеется, если не возникнет серьезный сбой, например, повреждение библиотеки CRT.

Существуют два других типа блоков памяти, которые никогда не отображаются в отчетах об утечке памяти. *Свободный блок* — это блок памяти, которая была освобождена. Это по определению означает, что она не имеет отношения к утечкам. *Пропускаемый блок* — это память, специально помеченная для исключения из отчета об утечке памяти.

Эти способы работают для памяти, выделенной с помощью стандартной функции malloc библиотеки CRT. Однако если программа выделяет память с использованием оператора new C++, необходимо переопределить оператор new, если требуется, чтобы в отчете об утечке памяти отображались имя файла и номера строк. Это можно сделать с помощью блока кода, аналогичного приведенному ниже:

#ifdef \_DEBUG #ifndef DBG\_NEW #define DBG\_NEW new ( \_NORMAL\_BLOCK , \_\_FILE\_\_ , \_\_LINE\_\_ ) #define new DBG\_NEW #endif #endif // \_DEBUG

* 1. Задание точек останова для номера выделения памяти

Номер выделения памяти сообщает, когда был выделен утекающий блок памяти. Например, блок с номером выделения памяти 18 — это 18-й блок памяти, выделенный во время выполнения программы. В отчете CRT учитываются все выделения блоков памяти во время выполнения. Сюда входят выделения, произведенные библиотекой CRT и другими библиотеками, такими как MFC. Поэтому блок с номером выделения памяти 18 может не быть 18-м блоком памяти, выделенным кодом. Как правило, не будет.

Номер выделения можно использовать для того, чтобы задать точку останова в том месте, где выделяется память.

Установка точки останова для выделения памяти с помощью окна контрольных значений

1. Задать точку останова недалеко от начала приложения, затем запустить приложение.
2. Когда выполнение приложения остановится в точке останова, открыть окно **Контрольные значения**.
3. В окне **Контрольные значения** введите \_crtBreakAlloc в столбце **Имя**.

Если используется многопоточная версия DLL библиотеки CRT (параметр /MD), добавить контекстный оператор: {,,ucrtbased.dll}\_crtBreakAlloc.

1. Нажать клавишу **ВВОД**.

Отладчик выполнит оценку вызова и поместит результат в столбец **Значение**. Это значение будет равно -1, если в местах выделения памяти не задано ни одной точки останова.

1. В столбце **Значение** заменить отображаемое значение номером выделения памяти, на котором нужно приостановить выполнение.

После задания точки останова для номера выделения памяти можно продолжить отладку. Нужно проследить за тем, чтобы программа была запущена в таких же условиях, как и в предыдущий раз, чтобы порядок выделения памяти не изменился. Когда выполнение программы будет приостановлено на заданном выделении памяти, с помощью окна **Стек вызовов** и других окон отладчика нужно определить условия выделения памяти. Затем можно продолжить выполнение программы и проследить, что происходит с этим объектом и почему выделенная ему память освобождается неправильно.

Иногда может быть полезно задать точку останова по данным на самом объекте.

Точки останова для выделения памяти можно также задать в коде. Это можно сделать двумя способами.

\_crtBreakAlloc = 18;

Или

\_CrtSetBreakAlloc(18);

* 1. Сравнение состояний памяти

Другая технология для обнаружения утечек памяти включает получение "снимков" состояния памяти приложения в ключевых точках. Чтобы получить снимок состояния памяти в заданной точке приложения, нужно создать структуру **\_CrtMemState** и передать ее функции \_CrtMemCheckpoint. Функция поместит в структуру снимок текущего состояния памяти:

\_CrtMemState s1; \_CrtMemCheckpoint( &s1 );

Функция \_CrtMemCheckpoint поместит в структуру снимок текущего состояния памяти.

Чтобы вывести содержимое структуры **\_CrtMemState**, нужно передать ее функции \_CrtMemDumpStatistics:

\_CrtMemDumpStatistics( &s1 );

Функция \_ CrtMemDumpStatistics выводит дамп состояния памяти, который выглядит примерно таким образом:

0 bytes in 0 Free Blocks. 0 bytes in 0 Normal Blocks. 3071 bytes in 16 CRT Blocks. 0 bytes in 0 Ignore Blocks. 0 bytes in 0 Client Blocks. Largest number used: 3071 bytes. Total allocations: 3764 bytes.

Чтобы определить, произошла ли утечка памяти на отрезке кода, можно сделать снимок состояния памяти перед ним и после него, а затем сравнить оба состояния с помощью функции \_ CrtMemDifference:

\_CrtMemCheckpoint( &s1 ); // memory allocations take place here \_CrtMemCheckpoint( &s2 ); if ( \_CrtMemDifference( &s3, &s1, &s2) ) \_CrtMemDumpStatistics( &s3 );

Функция \_CrtMemDifference сравнивает состояния памяти s1 и s2 и возвращает результат в (s3), представляющий собой разницу s1 и s2.

Еще один способ поиска утечек памяти заключается в размещении вызовов \_CrtMemCheckpoint в начале и конце программы с последующим использованием \_CrtMemDifference для сравнения результатов. Если \_CrtMemDifference показывает утечку памяти, можно добавить дополнительные вызовы функции \_CrtMemCheckpoint, чтобы разделить программу с помощью двоичного поиска, пока не будет найден источник утечки.

1.5 Ложные срабатывания

В некоторых случаях \_CrtDumpMemoryLeaks может ошибочно диагностировать утечку памяти. Это может произойти в случае использования библиотеки, в которой внутренние выделения отмечены как \_NORMAL\_BLOCK вместо \_CRT\_BLOCK или \_CLIENT\_BLOCK. В таком случае функция \_CrtDumpMemoryLeaks не может различать пользовательские выделения и внутренние выделения библиотеки. Если глобальные деструкторы для выделений библиотеки выполняются после точки вызова функции \_CrtDumpMemoryLeaks, каждое внутреннее выделение библиотеки принимается за утечку памяти. Предыдущие версии библиотеки стандартных шаблонов, предшествовавшие Visual Studio .NET, приводили к тому, что функция \_CrtDumpMemoryLeaks сообщала о таких ложных утечках, но в последних выпусках это было исправлено.

**2 Отладка в Visual Studio 15**

Отладка в Visual Studio 2015, как и в большинстве интегрированных сред разработки (IDE), осуществляется в два этапа: построение кода для обнаружения и устранения ошибок проекта и компилятора и выполнение этого кода в среде для обнаружения и устранения ошибок времени выполнения и динамических ошибок.

### **2.1** **Настройка сборки**

Существует два основных типа конфигурации сборки: **Отладка** и **Выпуск**. При использовании первой конфигурации создается более крупный и медленный исполняемый файл, обеспечивающий более широкие интерактивные возможности отладки во время выполнения, однако отправлять этот файл не следует. Вторая конфигурация позволяет создать более быстрый оптимизированный исполняемый файл, подходящий для отправки (по крайней мере с точки зрения компилятора).

По умолчанию используется конфигурация **Отладка**.

Можно также указать конкретную целевую платформу сборки, например, **x86** (32-разрядные процессоры Intel), **x64** (64-разрядные процессоры Intel) и **ARM** (процессоры ARM, поддерживается только для определенных типов приложений). Значение по умолчанию — **x86** для проектов управляемого и машинного кода. Чтобы изменить его, нужно щелкнуть раскрывающийся список платформ сборки и выбрать другую платформу или выбрать **Configuration Manager**.

Указать целевую конфигурацию сборки можно с помощью **Configuration Manager**. Чтобы запустить этот компонент, нужно щелкнуть раскрывающийся список **Конфигурация** или **ЦП** и выбрать **Создать...** для создания новой сборки или платформы.

В начале работы нужно просто использовать варианты **Отладка** и **x86** в качестве конфигурации сборки и платформы соответственно. После завершения создания кода и отладки нужно изменить конфигурацию на **Выпуск** и выбрать определенную целевую платформу. (В предыдущих версиях Visual Studio для проектов кода .NET предоставлялась платформа по умолчанию **AnyCPU**).

Примечание. При построении проекта значения конфигурации и платформы также используются для определения создаваемого пути к каталогу проекта для хранения исполняемого файла. Как правило, это **<path-to-project>\<project-name>\<configuration>\<platform>**. Например, проект с конфигурацией Отладка и платформой x86 будет размещаться по пути Projects\MyProjectNameHere\MyProjectNameHere\bin\Debug\x86. Это может быть полезно при наличии собственных средств или сценариев, управляющих исполняемыми файлами сборки.

### **2.2** **Сборка кода**

Настроив среду сборки, можно приступить к фактической сборке проекта. Самый просто способ — нажать клавишу F7, однако также можно начать сборку, выбрав в главном меню **Построение -> Построить решение**.

Процесс сборки можно наблюдать в окне состояния **Вывод** в нижней части пользовательского интерфейса Visual Studio. Здесь отображаются ошибки, предупреждения и операции сборки. При наличии ошибок (или предупреждений выше заданного уровня) сборка завершится ошибкой. Можно щелкнуть ошибку и предупреждение, чтобы перейти к строке, где они возникли. Для перестроения проекта можно нажать клавишу **F7** (чтобы перекомпилировать только файлы с ошибками) или **Ctrl + Alt + F7** (для чистого полного перестроения).

В окне результатов содержатся два окна сборки с вкладками под областью редактора: окно **Вывод**, в котором содержатся необработанные выходные данные компилятора (включая сообщения об ошибках), и окно **Список ошибок**, содержащее список всех ошибок и предупреждений, к которому можно применить сортировку и фильтры.

В случае успешного выполнения вы увидите примерно следующие результаты в окне **Вывод**.

### **2.3 Просмотр списка ошибок**

Если внесены какие-либо изменения в код, который был ранее и успешно скомпилирован, возможно, возникнет ошибка. Ошибки иногда очевидны, например, простая синтаксическая ошибка или неправильное имя переменной, а иногда их причину трудно выяснить, имея в распоряжении только зашифрованный код. Чтобы получить более четкое представление о проблеме, нужно перейти вниз окна **Вывод** сборки и щелкнуть вкладку **Список ошибок**. При этом произойдет переход к более организованному представлению ошибок и предупреждений для проекта и будет получен доступ к некоторым дополнительным параметрам.

Нужно щелкнуть строку ошибки в окне **Список ошибок** и перейти в строку кода, в которой возникла ошибка. (Номера строк также можно включить, щелкнув панель **быстрого запуска** в верхнем правом углу: нужно ввести «номера строк» в поле и нажать клавишу ВВОД.) Это самый быстрый способ перейти в окно **Параметры**, где можно включить номера строк.

Можно применить CTRL+G для быстрого перехода к номеру строки, в которой возникла ошибка.

Ошибку можно узнать по подчеркиванию красной волнистой линией. Чтобы получить дополнительные сведения, нужно навести на нее указатель мыши. Требуется внести исправления, и подчеркивание исчезнет, хотя в результате исправления может возникнуть новая ошибка (это называется «регрессия»). Требуется пройти список ошибок и устранить все ошибки в коде.

### **2.4 Просмотр подробных сведений об ошибках**

Многие ошибки трудны для восприятия, будучи представленными в терминах компилятора. В этом случае могут потребоваться дополнительные сведения. В окне **Список ошибок** можно выполнить автоматический поиск Bing, чтобы получить дополнительные сведения об ошибке (или предупреждении): нужно щелкнуть правой кнопкой мыши в соответствующей строке записи и выбрать в меню **Показать справочные сведения об ошибке**.

Внутри Visual Studio 2015 откроется вкладка, на которой будут отображаться результаты поиска Bing для кода ошибки и текста. Представленные результаты — из различных источников в Интернете, и, возможно, не все они будут полезными.

Кроме того, можно щелкнуть значение кода ошибки с гиперссылкой в столбце **Код списка ошибок**. Это приведет к запуску поиска Bing только по коду ошибки.

### **2.5 Выполнение статического анализа кода**

Под статическим анализом кода, как правило, понимается автоматическая проверка кода на наличие распространенных проблем, которые могут привести к ошибкам во время выполнения или проблемам управления кодом. Запустить этот компонент после устранения всех очевидных ошибок, препятствующих сборке, и потратить некоторое время, чтобы устранить создаваемые им предупреждения, — занятие, повышающее качество написания кода. Так можно избавиться от определенных будущих проблем, а также можно научиться некоторым полезным приемам написания кода.

Нужно нажать Alt+F11 (или выберите **Анализ -> Выполнить анализ кода в решении** в верхнем меню) для запуска статического анализа кода. При наличии большого объема кода это может занять некоторое время.

Все новые или обновленные предупреждения будут отображаться на вкладке **Список ошибок** в нижней части интегрированной среды разработки. Для перехода к предупреждению нужно щелкнуть его.

Предупреждения выделяются яркой желто-зеленой волнистой линией вместо красной. Нужно навести указатель мыши на предупреждение, чтобы получить дополнительные сведения. При щелчке правой мыши выводится контекстное меню, в котором можно выбрать пункты для исправления или рефакторинга.

### **2.6 Использование значка лампочки для исправления или рефакторинга кода**

Лампочки — это новая функция Visual Studio 2015, позволяющая выполнить рефакторинг кода в окне решения. Это простой способ быстрого и эффективного устранения распространенных предупреждений. Для доступа к ним нужно щелкнуть правой кнопкой мыши волнистую линию предупреждения или нажать сочетание клавиш CTRL+. (указатель мыши должен быть наведен на волнистую линию), а затем выбрать пункт меню **Быстрые действия**.

Отобразится список возможных исправлений или операций рефакторинга, которые можно применить к соответствующей строке кода.

Лампочки можно использовать в любом случае, когда средство анализа кода определяет возможность исправления, рефакторинга или улучшения кода. Можно щелкнуть любую строку кода, вызвать контекстное меню и выбрать **Быстрые параметры** (или нажать сочетание клавиш CTRL+.). Если для этой области есть доступные параметры рефакторинга или улучшения, они будут выведены на экран; в противном случае на панели в левом нижнем углу окна IDE будет выведено сообщение Быстрые параметры недоступны.

Можно использовать клавиши со стрелками и сочетание клавиш CTRL+. для проверки возможностей оптимизации с помощью быстрых параметров и очистки кода.

### **2.7 Отладка выполняемого кода**

Успешно завершив построение кода и его очистку, нужно запустить код, нажав клавишу F5 или выбрав **Отладка -> Начать отладку**. Приложение будет запущено в среде отладки, и можно будет пронаблюдать его поведение. Среда IDE Visual Studio 2015 изменяется во время выполнения приложения: окно **Вывод** заменяется двумя новыми окнами (в конфигурации окна по умолчанию), окном с вкладками **Видимые/Локальные/Модули/Контрольные значения** и окном с вкладками **Стеки вызовов/Точки останова/Параметры исключений/Вывод**. Эти окна имеют несколько вкладок, которые позволяют просмотреть и проверить переменные, потоки, стеки вызовов приложения и другие характеристики поведения во время выполнения приложения.

Нужно выполнять разные действия с приложением и наблюдать за изменениями. Если приложение ведет себя ненормально, нужно приостановить его с помощью сочетания клавиш Ctrl+Alt+Break (или нажать кнопку **Пауза**).

Нажать F5 для продолжения выполнения приложения (или нажать кнопку **Продолжить**).

Остановить приложение можно, нажав сочетание клавиш SHIFT+F5 или кнопку **Остановить**. Кроме того, можно просто закрыть главное окно приложения (или диалоговое окно командной строки). Если код выполняется полностью и точно так, как ожидалось, то задача выполнена. Нужно изменить конфигурацию сборки на **Выпуск** и перестроить его для развертывания. Однако в случае зависания, сбоя или непредвиденных результатов может потребоваться найти источник проблем и исправить ошибки.

### **2.8 Задание простых точек останова**

Точки останова — это один из самых простых и важных компонентов надежной отладки. Точка останова указывает, где Visual Studio следует приостановить выполнение кода, чтобы можно было проверить значения переменных или поведение памяти либо выполнение ветви кода. После установки или удаления точек останова перестраивать проект не нужно.

Можно установить точку останова, щелкнув дальнее поле строки, где требуется приостановить выполнение, или выбрав строку кода и нажав клавишу F9. При выполнении кода оно останавливается перед выполнением инструкций для этой строки кода.

В момент прерывания кода отмеченная строка кода еще не выполнена. На этом этапе можно выполнить инструкции для строки кода, отмеченной точкой останова, и проверить измененные значения. Это называется «пошаговым выполнением» кода. Если отмеченный код является вызовом метода, можно выполнить его, нажав клавишу F11. Кроме того, можно «перешагнуть» через строку кода, нажав клавишу F10.

Чаще всего точки останова используются для решения следующих задач:

1. Для сужения области поиска источника сбоя или зависания, когда точки останова устанавливаются в разных местах участка кода вызова метода, который, возможно, является источником проблемы, и за его пределами. При пошаговом выполнении кода нужно удалять, а затем снова устанавливать точки останова ближе друг к другу, пока не найдется строка кода, вызывающая ошибку.
2. При вводе нового кода можно установить точку останова в начале кода и пройти по коду, чтобы убедиться, что он работает правильно.
3. При реализации сложного поведения можно задать точки останова для алгоритмического кода, чтобы можно было проверить значения переменных и данные при прерывании программы.
4. При написании кода C или C++ можно использовать точки останова для остановки кода, чтобы можно было проверить значения адреса (искать значение NULL) и просмотреть значения счетчиков при отладке ошибок, связанных с памятью.

### **2.9 Задание условных точек останова**

При наличии точки останова в цикле или рекурсии, или большого числа точек останова, которые будут просматриваться пошагово, можно использовать условную точку останова, которая позволяет приостанавливать код только при выполнении определенных условий. В противном случае придется много раз нажимать клавишу F11.

Чтобы задать условную точку останова и приостанавливать код при установке определенного значения переменной (или достижения определенного порогового значения), нужно щелкнуть поле для задания точки остановка и выбрать в меню, появившемся при наведении указателя мыши, значок шестеренки.

Появится диалоговое окно, где можно задать определенные условия для останова.

### **2.10 Проверка кода во время выполнения**

Когда выполняемый код достигает точки останова и останавливается, можно проверить переменные и стеки вызовов, чтобы разобраться в происходящем. Находятся ли значения в тех диапазонах, которые вы ожидали увидеть? Выполняются ли вызовы в правильном порядке?

Нужно навести указатель мыши на переменную, чтобы просмотреть значения и ссылки, которые она содержит в данный момент. Если отображается значение, которое не соответствует желаемому, возможно, в предыдущей или вызывающей строке кода имеется ошибка. Нужно переместить точки остановки выше или добавить условия в существующие точки останова, чтобы сузить область поиска.

Кроме того, Visual Studio 2015 выводит на экран окно средств диагностики, где можно наблюдать за загрузкой ЦП и использованием памяти приложением в динамике по времени. Их рекомендуется использовать в случае непредвиденно больших показателей загрузки ЦП или выделения памяти. Это окно можно использовать в сочетании с окном **Контрольные значения** и точками останова, чтобы определить причину непредвиденно интенсивного использования или неосвобожденных ресурсов.

**ЛИТЕРАТУРА**

1. **MSDN**