Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Заведующий  методическим кабинетом  \_\_\_\_\_\_\_\_\_\_\_\_\_Е.В.Фалей  « \_\_\_\_ » \_\_\_\_\_\_\_\_\_ 2016 |

|  |  |
| --- | --- |
| Специальность: 2-40 01 01 «Программное обеспечение информационных технологий» | Учебная практика по программированию |
| Составлена в соответствии с учебной программой, утвержденной директором Колледжа бизнеса и права 31.05.2016 | |

**Практическая работа №11**

**Инструкционно-технологическая карта**

Тема:Обработка исключительных ситуаций.

* Обработка исключительных ситуаций (2 часа).
* Перехват исключений (2 часа).
* Отладка с помощью Visual Studio: конфигурирование отладочного режима, точки останова, исследование состояния приложения (2 часа).

Цель: Сформировать умения обработки исключительных ситуаций. Развить умение перехватывать исключения. Закрепить навык отладки с помощью Visual Studio

Время выполнения: 6 часов

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическими сведениями.
2. Получить вариант индивидуального задания и выполнить его.
3. Провести отладку полученного кода с помощью Visual Studio.
4. Оформить решенные задания в отчет.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

**Обработка исключительных ситуаций и перехват исключений**

Оператор try указывает блок кода, предназначенный для обработки ошибок или очистки. За блоком try должен следовать блок catch или блок finally или оба. Блок catch выполняется, когда возникает ошибка в блоке try. Блок finally выполняется после выполнения блока try (или блока catch, если он присутствует), обеспечивая очистку, независимо от того, возникала ошибка или нет.

Блок catch имеет доступ к объекту Exception, который содержит информацию об ошибке. Блок catch используется либо для компенсации последствий ошибки, либо для повторной генерации исключения. Повторная генерация исключения применяется, если нужно просто зарегистрировать факт возникновения проблемы в журнале или если необходимо сгенерировать новый тип исключения более высокого уровня.

Блок finally добавляет в программу детерминизм: среда CLR старается выполнять его всегда. Он полезен для проведения очистки, такой как закрытие сетевых подключений.

Оператор try выглядит следующим образом:

try

{

... // во время выполнения этого блока может возникнуть исключение

}

catch (ExceptionA ex)

{

... // обработать исключение типа ExceptionA

}

catch (ExceptionB ex)

{

... // обработать исключение типа ExceptionB

}

finally

{

... II код очистки

}

Взгляните на показанную ниже программу:

class Test

{

static int Calc(int x) { return 10 / x; }

static void Main()

{

int у = Calc(0);

Console.WriteLine(y);

}

}

Поскольку x имеет нулевое значение, исполняющая среда генерирует исключение DivideByZeroException и программа завершается. Чтобы предотвратить такое поведение, перехватываем исключение следующим образом:

class Test

{

static int Calc(int x) { return 10 / x; }

static void Main()

{

try

{

int у = Calc(0);

Console.WriteLine(y);

}

catch (DivideByZeroException ex)

{

Console.WriteLine("x cannot be zero"); // x не может быть равно О

}

Console.WriteLine("program completed"); // программа завершена

}

}

ВЫВОД: x cannot be zero program completed

Этот простой пример был предназначен для иллюстрации обработки исключений. На практике вместо реализации такого сценария лучше явно проверять делитель на равенство нулю перед вызовом Calc.

Исключения являются относительно дорогостоящими в плане ресурсов, требуя на свою обработку немало процессорного времени.

Когда возникло исключение, среда CLR выполняет следующую проверку.

Находится ли в текущий момент поток выполнения внутри оператора try, который может перехватить исключение ?

* Если да, поток выполнения переходит к совместимому блоку catch. Если этот блок catch завершился успешно, поток выполнения перемещается на следующий оператор после try (сначала выполнив блок finally, если он предусмотрен).
* Если нет, поток выполнения возвращается обратно в вызывающий компонент и проверка повторяется (после выполнения любых блоков finally, внутри кото­рых находится оператор).

Если ни одна функция не взяла на себя ответственность за исключение, пользова­телю отображается диалоговое окно с сообщением об ошибке и программа заверша­ется.

Конструкция catch

Конструкция catch указывает тип исключения для перехвата. Типом может быть либо System.Exception, либо какой-то подкласс System. Exception. Указание типа System.Exception приводит к перехвату всех возможных ошибок. Это удобно в следующих ситуациях:

* программа потенциально может восстановиться независимо от специфического типа исключения;
* планируется повторная генерация исключения (возможно, после его регистра­ции в журнале);
* обработчик ошибок является последним средством перед тем, как завершить программу.

Однако более обычной является ситуация, когда перехватываются исключения специфических типов, чтобы не иметь дела с исключениями, для которых обработчик не был предназначен (например, OutOfMemoryException).

Перехватывать исключения нескольких типов можно с помощью множества конструкций catch (опять-таки, это проще реализовать посредством явной проверки ар­гумента, а не за счет обработки исключений):

class Test

{

static void Main(string[] args)

{

try

{

byte b = byte.Parse(args[0]);

Console.WriteLine(b);

}

catch (IndexOutOfRangeException ex)

{

// Должен быть предоставлен хотя бы один аргумент Console.WriteLine ("Please provide at least one argument");

}

catch (FormatException ex)

{

// Аргумент должен быть числовым

Console.WriteLine("That's not a number!");

}

catch (OverflowException ex)

{

// Возникло переполнение

Console.WriteLine("You've given me more than a byte!");

}

}

}

Для заданного исключения выполняется только одна конструкция catch. Если вы хотите поймать в страховочную сеть большее число общих исключений (вроде System.Exception), размещайте более специфичные обработчики первыми.

Исключение может быть перехвачено без указания переменной, если доступ к свойствам исключения не нужен:

catch (StackOverflowException) // переменная не указана

{

}

Более того, можно опустить и переменную, и тип (это значит, что будут перехватываться все исключения): catch {…}

**Блок** finally

Блок finally выполняется всегда — независимо от того, возникало ли исключе­ние, и полностью ли был выполнен блок try. Блоки finally обычно используются для размещения кода очистки.

Блок finally выполняется в любом из следующих случаев:

* после завершения блока catch;
* после того, как поток управления покидает блок try из-за оператора перехода (например, return или goto);
* после завершения блока try.

Единственное, что может воспрепятствовать выполнению блока finally — это бесконечный цикл или неожиданное завершение потока.

Блок finally помогает добавлять в программу детерминизм. В приведенном при­мере файл, который был открыт, всегда закрывается независимо от следующих обсто­ятельств:

* блок try завершается нормально;
* происходит преждевременный возврат из-за того, что файл пуст (EndOf Stream);
* во время чтения файла возникает исключение IOException.

static void ReadFileO {

StreamReader reader = null; // В пространстве имен System.10 try {

reader = File.OpenText ("file.txt");

if (reader.EndOfStream) return;

Console.WriteLine (reader.ReadToEnd());

}

finally

{

if (reader != null) reader.Dispose ();

}

}

В этом примере закрываем файл вызовом Dispose на StreamReader. Вызов Dispose на объекте внутри блока finally — это стандартное соглашение, соблюда­емое повсеместно в .NET Framework, и оно явно поддерживается в C# посредством оператора using.

Оператор using

Многие классы инкапсулируют неуправляемые ресурсы, такие как файловые и графические дескрипторы и подключения к базе данных. Такие классы реализуют интерфейс System.IDisposable, который определяет единственный метод без пара­метров по имени Dispose, предназначенный для очистки этих ресурсов. Оператор using предлагает элегантный синтаксис для вызова Dispose на объекте IDisposable внутри блока finally. Показанный ниже код:

using (StreamReader reader = File.OpenText ("file.txt"))

{

}

эквивалентен следующему коду:

StreamReader reader = File.OpenText ("file.txt"); try {

}

finally

{

if (reader != null)

((IDisposable)reader).Dispose();

}

Генерация исключений

Исключения могут генерироваться либо исполняющей средой, либо пользовательским кодом. В показанном далее примере Display генерирует исключение

System.ArgumentNullException:

class Test {

static void Display (string name)

{

if (name == null)

throw new ArgumentNullException ("name") ;

Console.WriteLine (name);

}

static void Main()

{

try { Display (null); }

catch (ArgumentNullException ex)

{

Console.WriteLine ("Caught the exception");

}

}

}

Повторная генерация исключения

Исключение можно захватить и сгенерировать повторно следующим образом:

try { . . . }

catch (Exception ex)

{

/ / Записать в журнал информацию об ошибке

throw; // Повторно сгенерировать то же самое исключение

}

Если throw заменить throw ex, пример сохранит работоспособность, но свойство StackTrace нового распространяемого события не будет отображать исходную ошибку.

Повторная генерация в такой манере позволяет записать в журнал информацию об ошибке, не подавляя ее. Она также дает возможность отказаться от обработки ис­ключения, если обстоятельства сложились не так, как ожидалось:

using System.Net; // (См. главу 16) string s = null;

using (WebClient wc = new WebClientO)

try { s = wc.DownloadString ("http://www.albahari.com/nutshell/"); } catch (WebException ex)

{

if (ex.Status == WebExceptionStatus.NameResolutionFailure)

Console.WriteLine ("Bad domain name"); else

throw; // Нет возможности обработать другие виды WebException,

// поэтому повторная генерация

}

Другой распространенный сценарий предусматривает повторную генерацию ис­ключения более специфичного типа. Например:

try

{

... // Получить DateTime из данных XML-элемента

}

catch (FormatException ex)

{

throw new XmlException ("Invalid DateTime", ex); // Недопустимый формат DateTime

}

Повторная генерация менее специфичного исключения может осуществляться при пересечении доверительных границ, чтобы не допустить утечки технической инфор­мации потенциальным взломщикам.

При повторной генерации отличающегося исключения можно установить в свойстве InnerException исходное исключение, чтобы помочь в отладке. Почти все типы исключений обеспечивают конструктор для этой цели (такой как в нашем примере).

Ключевые свойства System.Exception

Ниже описаны наиболее важные свойства System.Exception.

* StackTrace. Строка, представляющая все методы, которые были вызваны, на­чиная с источника исключения и заканчивая блоком catch.
* Message. Строка с описанием ошибки.
* InnerException. Внутреннее исключение (если есть), которое вызвало внешнее исключение. Это свойство само может иметь другое свойство InnerException.

Все исключения в C# происходят во время выполнения. В C# нет эквива­лента проверяемых исключений этапа компиляции Java.

Общие типы исключений

Перечисленные ниже типы исключений используются повсеместно в CLR и .NET Framework. Их можно инициировать самостоятельно или применять в качестве базо­вых классов для порождения специальных типов исключений.

* System.ArgumentException. Генерируется, когда функция вызывается с невер­ным аргументом. В общем случае это означает программную ошибку.
* System.ArgumentNullException. Подкласс ArgumentException, который генерируется, когда аргумент функции неожиданно равен null.
* System.ArgumentOutOfRangeException. Подкласс Argument Exception, который генерируется, когда (обычно числовой) аргумент имеет слишком большое или слишком малое значение. Например, это исключение возникает при пере­даче отрицательного числа в функцию, принимающую только положительные значения.
* System.InvalidOperationException. Генерируется, когда состояние объекта оказывается неподходящим для успешного выполнения метода, независимо от лю­бых заданных значений аргументов. В качестве примеров можно назвать чтение неоткрытого файла или получение следующего элемента из перечислителя, когда лежащий в основе список был изменен на полпути к завершению итерации.
* System.NotSupportedException. Генерируется для указания, что конкретная функциональность не поддерживается. Хорошим примером может служить вызов метода Add на коллекции, для которой IsReadOnly возвращает true.
* System.NotImplementedException. Генерируется для указания, что функция пока еще не реализована.
* System. Obj ectDisposedException . Генерируется, когда объект, на котором вызывается функция, был освобожден.

Еще одним часто встречающимся типом исключения является NullReferenceException. Среда CLR генерирует это исключение, когда вы пытаетесь получить доступ к члену объекта, значение которого равно null (указывая на ошибку в коде). Исключение NullReferenceException можно инициировать напрямую (в тестовых целях) следующим образом: throw null;

Шаблон методов Тгуххх

В реализации метода при учете ситуации, когда что-то идет не так, вы имеете возможность выбора между возвратом некоторого вида кода неудачи и генерацией ис­ключения. В общем случае исключение генерируется, когда ошибка находится за пре­делами нормального рабочего потока, или же когда ожидается, что непосредственно вызвавший код не имеет возможности справиться с ней. Однако иногда лучше предло­жить пользователю оба варианта. Примером может служить тип int, в котором опре­делены две версии метода разбора Parse:

public int Parse (string input);

public bool TryParse (string input, out int returnValue) ;

Если разбор оказывается неудачным, Parse генерирует исключение, тогда как TryParse возвращает false. Такой шаблон можно реализовать за счет вызова в мето­де XXX метода ТгуХХХ, как показано ниже:

public возвращаемый-тип XXX (входной-тип input)

{

возвращаемый-тип returnValue;

if (!TryXXX (input, out returnValue))

throw new ITYException (...)

return returnValue;

}

Альтернативы исключениям

Как и int.Try Parse, функция может сообщать о сбое путем возврата кода ошибки вызывающей функции через возвращаемый тип или параметр. Несмотря на то что такой подход хорошо работает с простыми и предсказуемыми ошибками, он становится громоздким при необходимости охвата всех ошибок, засоряя сигнатуры методов и привнося ненужную сложность и беспорядок. Кроме того, его нельзя распространить на функции, не являющиеся методами, такие как операции (например, деление) или свойства. В качестве альтернативы информация об ошибке может храниться в общем местоположении, доступном для просмотра всеми функциями в стеке вызовов (к при­меру, в статическом методе, хранящем состояние ошибки в потоке). Тем не менее, это требует от каждой функции участия в шаблоне распространения ошибок, который яв­ляется громоздким и, по иронии судьбы, подверженным ошибкам.

**Отладка с помощью Visual Studio**

Отладчик Visual Studio позволяет вести наблюдение за поведением программы во время выполнения и выявлять проблемы. Отладчик работает со всеми языками программирования Visual Studio и библиотеками. С помощью отладчика можно прерывать или приостанавливать выполнение программы с целью проверки кода, вычислять и редактировать значения переменных программы, отслеживать состояние регистров, просматривать инструкции, созданные из исходного кода, а также просматривать область памяти, используемую приложением.

**Окна отладчика**

В процессе отладки программы можно открывать большинство окон отладчика. Чтобы просмотреть список окон отладчика, установите точку останова и начните отладку. Когда точка останова будет достигнута и выполнение остановится, выберите пункт **Отладка / Окна**.

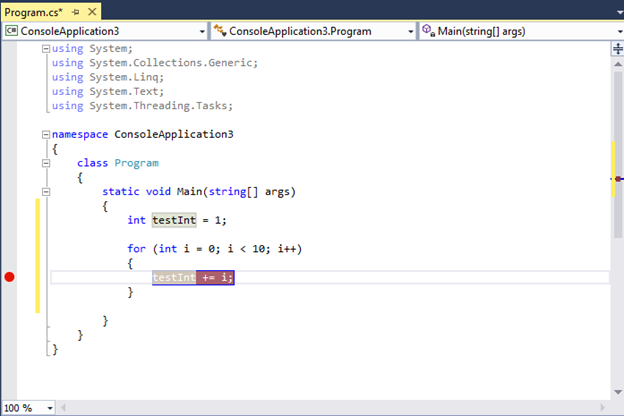
|  |  |  |
| --- | --- | --- |
| **Окно** | **Сочетание клавиш** | **Раздел** |
| Точки останова | CTRL+ALT+B | [Использование точек останова](https://msdn.microsoft.com/ru-ru/library/5557y8b4.aspx) |
| Параметры исключений | CTRL+ALT+E | [Управление исключениями с помощью отладчика](https://msdn.microsoft.com/ru-ru/library/x85tt0dd.aspx) |
| Вывод | CTRL+ALT+O | [Окно выходных данных](https://msdn.microsoft.com/ru-ru/library/3hk6fby3.aspx) |
| Задачи | CTRL+SHIFT+D, K | [Использование окна задач](https://msdn.microsoft.com/ru-ru/library/dd998369.aspx) |
| Параллельные стеки | CTRL+SHIFT+D, S | [Использование окна "Параллельные стеки"](https://msdn.microsoft.com/ru-ru/library/dd998398.aspx) |
| Контроль параллельных данных | CTRL+SHIFT+D, (1, 2, 3, 4) | [Использование окна "Параллельные стеки"](https://msdn.microsoft.com/ru-ru/library/dd998398.aspx) |
| Контрольное значение | CTRL+ALT+W, (1, 2, 3, 4) | [Окна "Контрольные значения" и "Быстрая проверка"](https://msdn.microsoft.com/ru-ru/library/0taedcee.aspx) |
| Контрольное значение | SHIFT+F9 | [Окна "Контрольные значения" и "Быстрая проверка"](https://msdn.microsoft.com/ru-ru/library/0taedcee.aspx) |
| Авто | CTRL+ALT+V, A | [Окна переменных](https://msdn.microsoft.com/ru-ru/library/4dt5w8ta.aspx) |
| Локальные | CTRL+ALT+V, L | [Окна переменных](https://msdn.microsoft.com/ru-ru/library/4dt5w8ta.aspx) |
| Интерпретация | CTRL+ALT+I | [Окно интерпретации](https://msdn.microsoft.com/ru-ru/library/f177hahy.aspx) |
| Консоль JavaScript | CTRL+ALT+V, C | [Краткое руководство. Отладка JavaScript](https://msdn.microsoft.com/ru-ru/library/dn688631.aspx) |
| Обозреватель DOM | CTRL+ALT+V, D | [Отладка макета с использованием проводника DOM](https://msdn.microsoft.com/ru-ru/library/hh696638.aspx) |
| Динамическое визуальное дерево | — | [Просмотр свойств XAML во время отладки](https://msdn.microsoft.com/ru-ru/library/mt270227.aspx) |
| Динамический обозреватель свойств | — | [Просмотр свойств XAML во время отладки](https://msdn.microsoft.com/ru-ru/library/mt270227.aspx) |
| Стеки вызовов | CTRL+ALT+C | [Практическое руководство. Использование окна стека вызова](https://msdn.microsoft.com/ru-ru/library/a3694ts5.aspx) |
| Потоки | CTRL+ALT+H | [Практическое руководство. Использование окна потоков](https://msdn.microsoft.com/ru-ru/library/w15yf86f.aspx) |
| Модули | CTRL+ALT+U | [Практическое руководство. Использование окна модулей](https://msdn.microsoft.com/ru-ru/library/4c8f14c9.aspx) |
| Процессы | CTRL+ALT+Z | [Отладка потоков и процессов](https://msdn.microsoft.com/ru-ru/library/ms164739.aspx) |
| Память | CTRL+ALT+M, (1, 2, 3, 4) | [Окно памяти](https://msdn.microsoft.com/ru-ru/library/s3aw423e.aspx) |

**Точки останова**

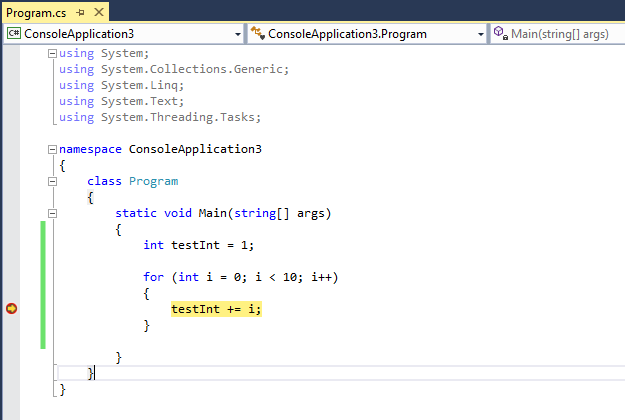
Если необходимо остановить выполнение отладчика, например, чтобы увидеть состояние переменных кода или просмотреть стек вызовов, можно установить точки останова. Точка останова — один из важнейших инструментов разработчика для выполнения отладки.

## **Задание точки останова в функции в исходном коде**

Можно установить точку останова функции в исходном коде, щелкнув в левом поле файла исходного кода или поместив курсор на строку кода и нажав клавишу F9. Точка останова отображается в виде красной точки в левом поле. Строка кода также окрашивается цветом:



При выполнении этого кода в отладчике выполнение останавливается при достижении точки останова перед выполнением кода в этой строке. Строка исходного кода имеет желтый цвет:



На этом этапе значение testInt равно 1.

Можно просмотреть текущее состояние приложения, включая значения переменных и стека вызова. Можно установить точку останова в любой строке исполняемого кода. Например, в коде C# выше можно установить точку останова для объявления переменной, цикла for или всего кода внутри цикла for, но нельзя задать точку останова для объявления пространства имен или класса или сигнатуры метода.

## **Задание точки останова в окне стека вызовов**

Можно прервать выполнение на инструкции или строке, к которой возвращается вызывающая функция, установив соответствующую точку останова в окне **Стек вызовов**. Отладчик должен был остановить выполнение.

1. Начните отладку приложения и ожидайте остановки выполнения, например в точке останова. Откройте окно **Стек вызовов** (выберите **Отладка > Окна > Стек вызовов** или нажмите клавиши **CTRL+ALT+C**).
2. Щелкните правой кнопкой мыши вызывающую функцию, затем выберите **Точка останова / Вставить точку останова**, или просто нажмите клавишу **F9**.
3. В левом поле рядом с именем вызова функции появится символ точки останова.

В окне **Точки останова** точка останова стека вызова будет представлена как адрес с областью памяти, который соответствует следующей исполняемой инструкции в функции. Отладчик приостанавливает выполнение на этой инструкции.

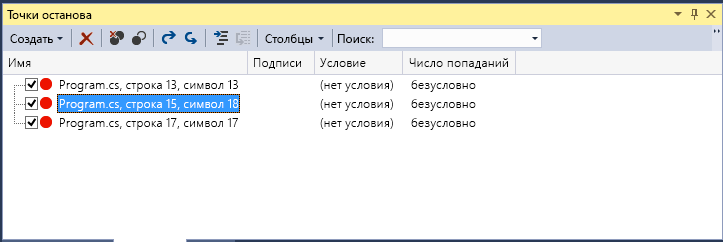
## **Задание точки останова в окне дизассемблирования**

Чтобы установить точку останова на инструкции ассемблера, отладчик должен находиться в режиме приостановки выполнения.

1. Начните отладку приложения и ожидайте остановки выполнения, например, в точке останова. Откройте окно **Дизассемблирование** (выберите **Отладка > Окна > Дизассемблирование** или нажмите клавиши **CTRL+ALT+D**).
2. Щелкните в левом поле инструкцию, на которой необходимо прервать выполнение, или установите курсор в инструкции и нажмите клавишу **F9**.

## **Управление точками останова**

Окно **Точки останова** (**Отладка > Окна > Точки останова** или **CTRL+ALT+B**) можно использовать для просмотра всех точек останова, заданных в решении:

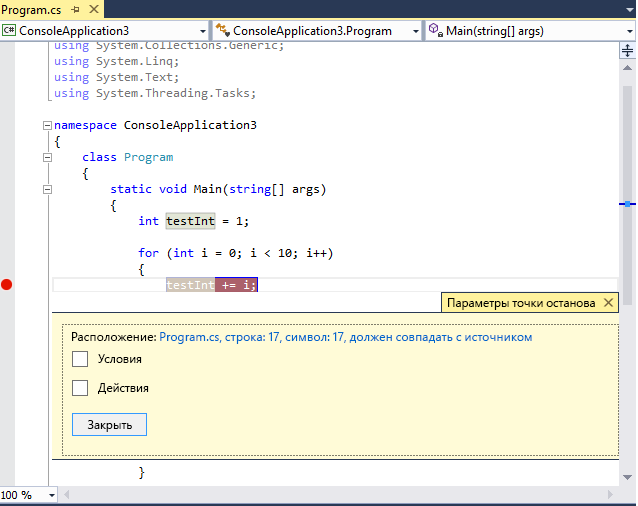


Окно **Точки останова** дает возможность централизованно управлять сразу всеми точками останова, что особенно удобно при работе с программой большого объема или в сложных сценариях отладки, где точки останова имеют принципиально важное значение. Кроме того, если требуется сохранить или предоставить другим пользователям состояние и расположение набора точек останова, функция экспорта точек останова и их импорта доступна только в окне **Точки останова**.

## **Условия точки останова**

Можно управлять тем, где и когда выполняется точка останова, задавая условия.

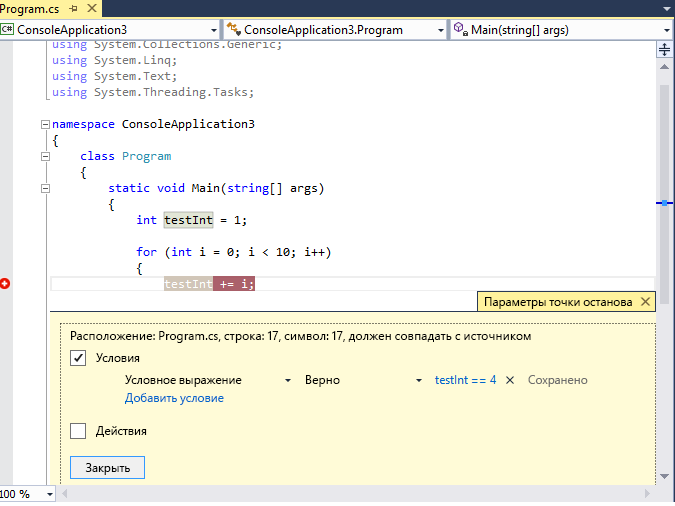
1. Щелкните правой кнопкой мыши точку останова или наведите указатель мыши на точку останова и выберите значок "Параметры".
2. В контекстном меню выберите **Условия**. Откроется окно **Параметры точки останова**:



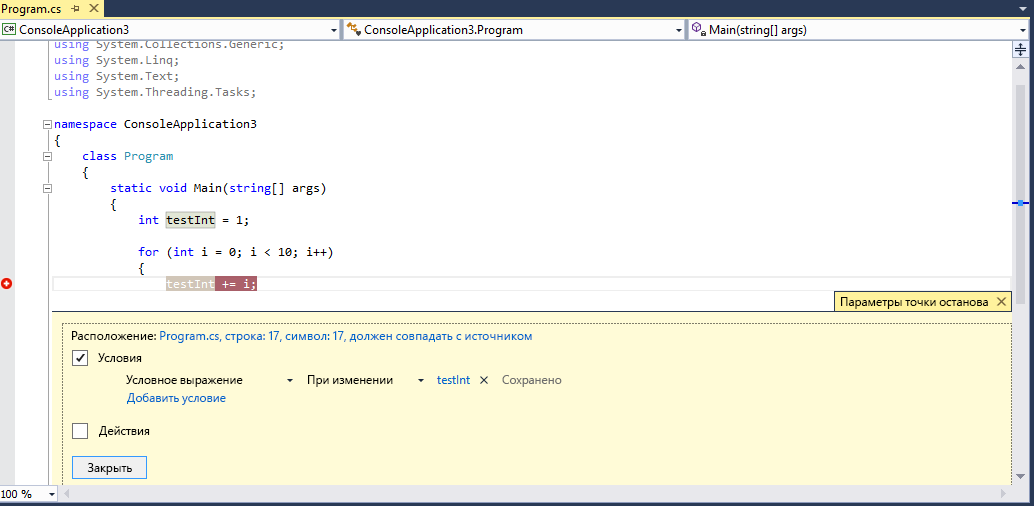
Если установить флажок **Условия**, откроется окно с различными видами условий.

**Условные выражения.** При выборе условного выражения можно выбрать два условия: **Является true** и **При изменении**. Выберите значение **Является true**, если требуется прервать выполнение при истинности выражения, или значение **При изменении**, если требуется прервать выполнение при изменении значения выражения.

В следующем примере задается применение точки останова, только если значение testInt равно **4**:



В следующем примере задается применение точки останова, только если значение testInt изменено:



Поведение поля "При изменении" отличается для различных языков программирования. Если значение **При изменении** выбрано для машинного кода, отладчик не рассматривает первое вычисление условия как изменение, поэтому при первом вычислении выражения точка останова не сработает. Если значение **При изменении** выбрано для управляемого кода, точка останова срабатывает при первом вычислении после выбора условия **При изменении**.

Если условие точки останова имеет недопустимый синтаксис, появится предупреждающее сообщение. Если указать условие для точки останова с недопустимой семантикой, но допустимым синтаксисом, предупреждающее сообщение появится при достижении точки останова в первый раз. В любом случае, отладчик прерывает выполнение при попадании на недопустимую точку останова. Точка останова пропускается, только если условие допустимо и принимает значение false.

Условие может быть любым допустимым выражением, которое распознает отладчик.

## **Использование идентификаторов объектов в условиях точек останова**

Бывают случаи, когда необходимо посмотреть поведение конкретного объекта; например, может потребоваться узнать, почему объект был вставлен в коллекцию больше одного раза. В C# можно создавать идентификаторы объектов для определенных экземпляров ссылочных типов и использовать их в условиях точек останова. Идентификатор объекта создается службами отладки среды CLR и связан с объектом. Чтобы создать идентификатор объекта, выполните следующее.

1. Установите точку останова в коде после создания объекта.
2. Начните отладку и, когда выполнение остановится в точке останова, найдите точку останова в окне **Локальные**, щелкните ее правой кнопкой мыши и выберите пункт **Создать идентификатор объекта**.

В окне **Локальные** вы должны увидеть символ **$** и число. Это и есть идентификатор объекта.

1. Добавьте новую точку останова в точке, которую нужно исследовать, например в точке добавления объекта в коллекцию.
2. Используйте идентификатор объекта в поле "Условное выражение". Например, если есть переменная item, ссылающаяся на объект, который должен быть добавлен в коллекцию, задайте **item == $n**, где **n** — номер идентификатора объекта.

Выполнение прервется в точке добавления объекта в коллекцию.

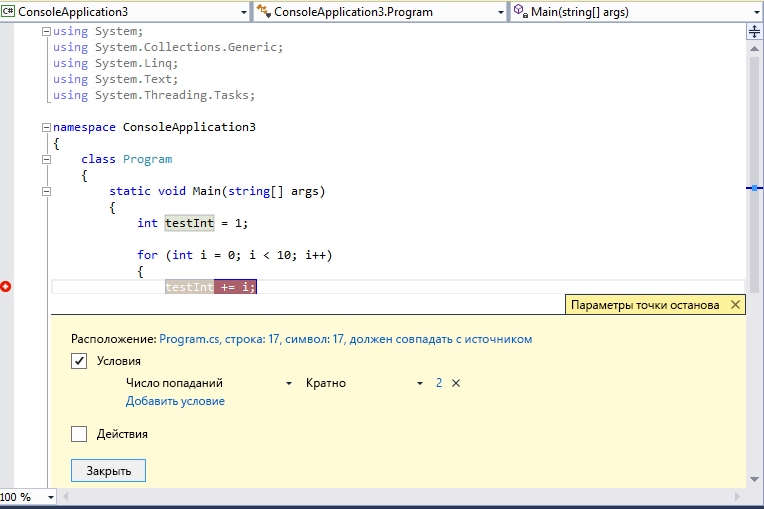
Если позже потребуется удалить идентификатор объекта, щелкните правой кнопкой мыши переменную в окне **Локальные** и выберите пункт **Удалить идентификатор объекта**.

Обратите внимание, что идентификаторы объектов создают слабые ссылки и не предотвращают сборку мусора для объекта. Они действительны только в рамках текущего сеанса отладки.

## **Число попаданий**

Если есть подозрение, что цикл в коде начинает неправильно вести себя после определенного числа итераций, можно установить точку останова для остановки выполнения после указанного числа попаданий в соответствующую строку кода, вместо того чтобы многократно нажимать клавишу **F5** для достижения нужного уровня итерации.

В окне **Параметры точки останова** задайте в качестве условия **Количество обращений**. Затем можно указать число итераций. В следующем примере задается выполнение точки останова при каждой итерации:



## **Фильтр**

Вы можете ограничить точку останова, сделав ее срабатывание возможным лишь на определенных устройствах или в определенных процессах или потоках.

В окне **Параметры точки останова** задайте в качестве условия **Фильтр**. Задайте одно или несколько выражений, приведенных ниже.

* MachineName = "имя"
* ProcessId = значение
* ProcessName = "имя"
* ThreadId = значение
* ThreadName = "имя"

Значения строк следует заключить в двойные кавычки. Для комбинации условий можно использовать знаки & (И), || (ИЛИ), ! (НЕ) и скобки.

## **Действия точки останова и точки отслеживания**

Точка трассировки — это точка останова, которая выводит сообщение в окно вывода. Точка трассировки может играть роль временного оператора трассировки в языке программирования.

В окне **Параметры точки останова** установите флажок **Действия**. В группе **Действие** выберите пункт **Запись сообщения в окне вывода**. Можно вывести универсальную строку, например **это тест**. Чтобы включить в сообщение значение переменной или выражение, используйте фигурные скобки.

Чтобы приостановить выполнение при достижении точки трассировки, снимите флажок **Продолжить выполнение**. Если флажок **Продолжить выполнение** установлен, выполнение не останавливается. В обоих случаях выводится сообщение.

## **Метки точки останова**

Метки точек останова используются только в окне **Точки останова**. Они служат для сортировки и фильтрации списка точек останова. Для добавления метки к точке останова выберите строку точки останова, затем в контекстном меню выберите команду **Метка**.

## **Экспорт и импорт точек останова**

Чтобы экспортировать точки останова в XML-файл, щелкните правой кнопкой мыши на точке останова и выберите **Экспорт**. По умолчанию файл сохраняется в каталоге решения. Чтобы импортировать точки останова, откройте окно **Точки останова** (**CTRL+ALT+B**) и на панели инструментов щелкните стрелку вправо (подсказка — **Импорт точек останова из файла**).

## **Устранение неполадок точек останова**

### **После удаления точки останова она по-прежнему применяется при запуске отладки**

Если удалить точку останова во время отладки, в некоторых случаях точка останова может снова применяться при очередном запуске отладки. Чтобы прекратить применение этой точки останова, убедитесь, что все ее экземпляры удалены из окна **Точки останова**.

### **Отладчик не может найти правильную версию исходного файла для точки останова**

Если исходный файл изменился и исходный код больше не соответствует отлаживаемому коду, отладчик может найти исходный файл, который соответствует точке останова, даже если исходный файл существует.

1. Если требуется, чтобы в Visual Studio отображался исходный код, не соответствующий отлаживаемой версии, выберите **Отладка / Параметры и настройки**. На странице **Отладка / Общие** снимите флажок **Требовать точного соответствия исходной версии файлов**.
2. Также можно выполнять привязку точки останова к исходному файлу. Выделите точку останова и в контекстном меню выберите пункт **Условия**. Установите флажок **Разрешить наличие отличий в исходном коде от первоначальной версии** в окне **Параметры точки останова**.

### **Точки останова не работают в библиотеке DLL**

Невозможно установить точку останова в исходном файле, если отладчик не загрузил отладочную информацию для модуля, в котором находится код. Признаком такой ситуации может быть, например, сообщение **Точка останова не будет задана**. На месте точки останова появляется глиф предупреждения о точке останова. Эти глифы предупреждений о точке останова, однако, становятся фактическими точками останова после загрузки кода.

# Управление исключениями с помощью отладчика

Исключение указывает на состояние ошибки, возникающее при выполнении программы. Вы можете и должны предоставить обработчики, реагирующие на наиболее важные исключения. Важно знать, как настроить отладчик для останова исключений, которые требуется просмотреть.

При возникновении исключения отладчик записывает его сообщение в окно «Вывод». Он может прервать выполнение в следующих случаях:

* если исключение возникает и не обрабатывается;
* если отладчик настроен на прерывание выполнения сразу же после возникновения исключения, до вызова любого обработчика;
* если задан параметр [Только мой код](https://msdn.microsoft.com/ru-ru/library/dn457346.aspx) и отладчик настроен на прерывание по любому исключению, не обрабатываемому в коде пользователя.

## **Управление исключениями с помощью окна «Параметры исключений»**

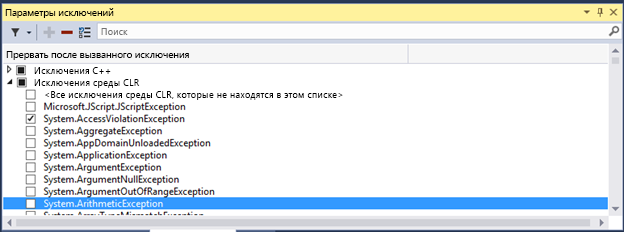
В окне **Параметры исключений** можно указать, какие исключения (или наборы исключений) приведут к прерыванию работы отладчика, и на этом этапе нужно сделать этот останов. Можно добавлять или удалять исключения или задавать исключения, при возникновении которых выполняется прерывание. Откройте это окно в открытом решении, последовательно выбрав **Отладка/Окна/Параметры исключений**.

Для поиска конкретных исключений можно воспользоваться окном **Поиск** на панели инструментов **Параметры исключений** или применить функцию поиска для фильтрации определенных пространств имен (например, **System.IO**).

### **Настройка отладчика для прерывания выполнения при создании исключения**

Отладчик может прервать выполнение приложения в точке возникновения исключения, предоставляя, таким образом, шанс проверки исключения еще до вызова обработчика.

В окне **Параметры исключений** разверните узел для категории исключений (например, **Исключения среды CLR**, то есть исключения .NET) и установите флажок для конкретного исключения в этой категории (например, **System.AccessViolationException**). Можно также выбрать всю категорию исключений.



При проверке данного исключения выполнение отладчика будет прерываться везде, где возникает это исключение независимо от того, обрабатывается оно или нет. На этом этапе исключение называется первым экземпляром исключения.

В следующем консольном приложении C# метод Main создает исключение **AccessViolationException** внутри блока Try/Catch.

static void Main(string[] args)

{

try{

throw new AccessViolationException();

Console.WriteLine("here");

}

catch (Exception e){

Console.WriteLine("caught exception");

}

Console.WriteLine("goodbye");

}

Если исключение **AccessViolationException** отмечено в окне **Параметры исключений**, при выполнении этого кода в режиме отладчика произойдет останов на строке Throw. После этого выполнение можно продолжить. В консоли должны отображаться обе строки.

caught exception

goodbuy

Но в ней не отображается строка here.

# Окно выходных данных

Окно **Вывод** может отображать сообщения о состоянии различных компонентов интегрированной среды разработки (IDE). Открыть окно **Вывод**, в строке меню выберите пункт **Представления/выходных данных** (или нажмите сочетание клавиш CTRL + ALT + O).

## **Панель выходных данных**

Панель **Выходные данные** можно выбрать в списке **Показать выходные данные от**, на этой панели будут отображены данные из выбранного источника.

## **Перенаправление сообщений в окно выходных данных**

Чтобы открыть окно **Вывод** при построении проекта, в диалоговом окне **Страница "Общие", страница " проекты и решения ", параметры** выберите **Показывать окно вывода при запуске построения**. Затем, с файлом кода открытым для редактирования, нажмите кнопки **К следующему сообщению** и **К предыдущему сообщению** на панели инструментов окна **Вывод**, чтобы выбрать записи в области **Вывод**. При этом курсор в редакторе кода перейдет в строку, где находится выбранная ошибка.

Некоторые средства IDE, такие как внешние служебные программы или команды, вызываемые через окно команд Окно "Команда", направляют выходные данные в окно **Вывод**. Выходные данные из внешних средств, таких как bat-файлы и .com, обычно отображается в окне командной строки, перенаправляется в области **Вывод** при выделении параметр **Использовать окно вывода** в Управление внешними инструментами. В окне **Вывод** могут отображаться и другие виды сообщений. Например, когда синтаксис Transact-SQL в хранимой процедуре проверяется по целевой базе данных, результаты отображаются в окне **Вывод**.

# Исследование состояния приложения

Начнем с простого консольного приложения C# (последовательно щелкните **Файл | Создать | Проект**, выберите **Visual C#**, а затем — **Консольное приложение**). Метод Main просто добавляет 1 к целочисленной переменной 10 раз и отображает результат в консоли:

static void Main(string[] args)

{

int testInt = 0;

for (int i = 1; i < 10; i++)

{

testInt += 1;

}

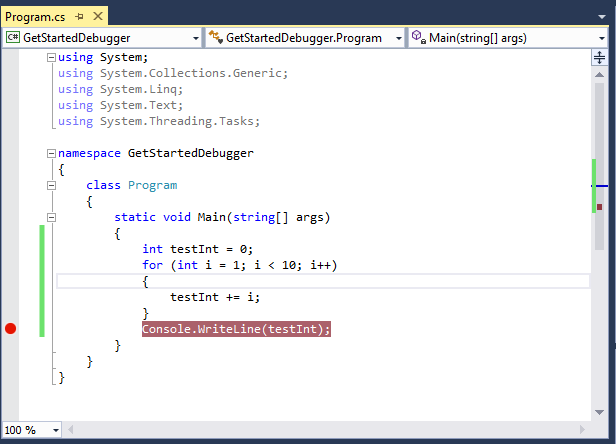
Console.WriteLine(testInt);

}

Выполните сборку этого кода в конфигурации **Отладка**. Эта конфигурация задается по умолчанию.

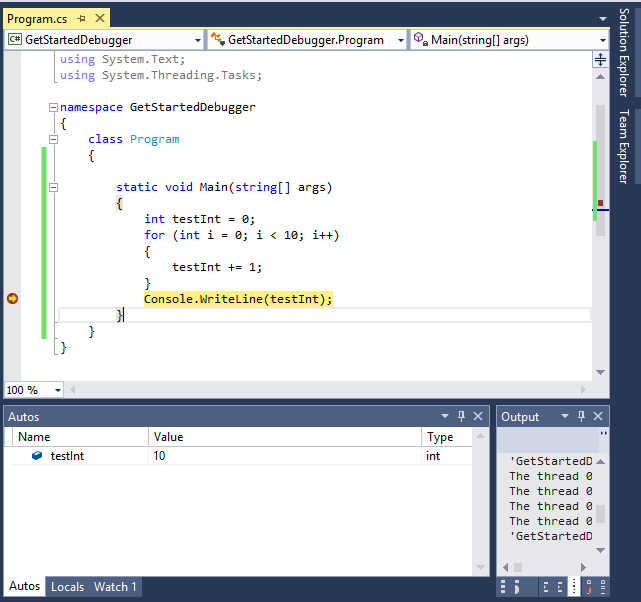
Выполните этот код в отладчике, последовательно щелкнув **Отладка | Начать отладку** (или **Запустить** на панели инструментов либо нажмите клавишу F5). Приложение должно практически сразу завершить работу, поэтому невозможно определить, было ли что-то выведено в окне консоли.

Вы можете задать точку останова и остановить выполнение на достаточно длительный интервал времени, чтобы увидеть окно консоли, а затем перейти к следующему шагу. Чтобы задать точку останова, поместите курсор в строку Console.WriteLine и последовательно выберите **Отладка | Создать точку останова | Точка останова в функции** или просто щелкните левое поле в той же строке. Точка останова должна выглядеть следующим образом:



Снова начните отладку. Выполнение остановится перед выполнением кода Console.WriteLine. Вы можете вызвать выполнение, перейдя к следующему шагу (щелкните **Отладка | Шаг с обходом** или нажмите клавишу **F10**). В этом случае можно выбрать **Шаг с заходом** (**F11**) и получить тот же результат. Их различия будут описаны позже. Строка с последней фигурной скобкой метода должна окраситься желтым цветом. Взгляните на окно консоли. Вы должны увидеть **10**. Остановите отладку (щелкните **Отладка | Остановить отладку** или **Остановить отладку** на панели инструментов либо нажмите сочетание клавиш **SHIFT + F5**).

Теперь рассмотрим значения переменных. Под окном кода должны отображаться окна **Видимые**, **Локальные** и **Контрольные значения**. В этих окнах отображаются текущие значения переменных во время выполнения. Как в окне **Видимые**, так и в окне **Локальные** отображается testInt со значением **10**.



Рассмотрим, как значения переменных меняются при работе с программой. Задайте точку останова в строке testInt += 1; и начните отладку. Должно отобразиться, что значение **testInt** в окнах **Локальные** и **Видимые** равно **0**, а **i** равно **1**. При продолжении отладки (щелкните **Отладка | Продолжить** или **Продолжить** на панели инструментов либо нажмите клавишу **F5**) можно увидеть, что значение testInt меняется на **1**, затем на **2** и т. д. Если вам не хочется больше смотреть на эти изменения, удалите точку останова (щелкните **Отладка | Точка останова** или щелкните ее в поле) и продолжите отладку. Если вы хотите удалить все точки останова, щелкните **Отладка | Удалить все точки останова** или нажмите сочетание клавиш **CTRL + SHIFT + F9** и нажмите кнопку **Да** в диалоговом окне с запросом **Удалить все точки останова?**.

## **Шаги с заходом и обходом вызовов функций**

Чтобы понять различия между **шагом с заходом** и **шагом с обходом**, необходимо добавить метод, вызываемый другим методом. Добавьте метод в приложение C# и вызовите его из метода Main. Код должен выглядеть следующим образом:

static void Main(string[] args)

{

Method1();

Console.WriteLine("end");

}

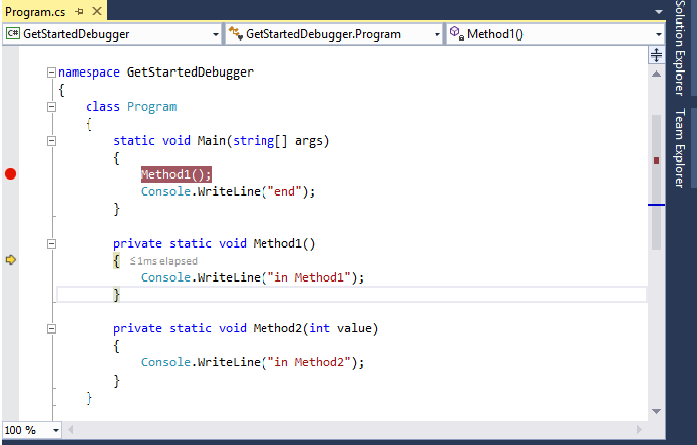
private static void Method1()

{

Console.WriteLine("in Method1");

}

Задайте точку останова в вызове Method1(); в методе Main и начните отладку. При остановке выполнения щелкните **Отладка | Шаг с заходом** (или **Шаг с заходом** на панели инструментов либо нажмите клавишу **F11**). Выполнение снова остановится у первой фигурной скобки в Method1():



Остановите отладку и снова запустите ее, а при остановке выполнения на точке останова щелкните **Отладка | Шаг с обходом** (или **Шаг с обходом** на панели инструментов либо нажмите клавишу **F10**). Выполнение снова остановится в строке Console.WriteLine("end");.

# Навигация по коду с помощью отладчика

Существует множество способов для перемещения по коду в отладчике: методы шага с обходом и выполнения по шагам, выполнение до точки останова или указанного расположения, а также указание того, нужно ли ограничить отладку своим собственным кодом или включить символы для отладки внешнего кода.

## **Выполнение шагов с заходом в код, с обходом кода или выходом из него**

Одной из наиболее распространенных процедур отладки является пошаговое выполнение. При пошаговом выполнении код выполняется по одной строке за раз. Во время остановки выполнения, например при достижении отладчиком точки останова, можно осуществлять пошаговое выполнение кода с помощью трех команд меню **Отладка**.

* **Шаг с заходом (F11) -** Если строка содержит вызов функции, команда **Шаг с заходом** выполняет только сам вызов, а затем останавливает выполнение в первой строке кода внутри функции. В противном случае команда **Шаг с заходом** выполняет следующий оператор;
* **Шаг с обходом (F10) -** Если строка содержит вызов функции, команда **Шаг с обходом** выполняет вызываемую функцию, а затем останавливает выполнение в первой строке кода внутри вызывающей функции. В противном случае команда **Шаг с заходом** выполняет следующий оператор;
* **Шаг с выходом (Shift+F11) -** Команду **Шаг с выходом** возобновляет выполнение кода до возврата функции, а затем прерывает выполнение в точке возврата вызывающей функции.

При вызове вложенных функций команда **Шаг с заходом** позволяет попасть в самую глубокую вложенную функцию. Если использовать **Шаг с заходом** на вызове Func1(Func2()), отладчик заходит в функцию Func2.

Отладчик фактически осуществляет пошаговое выполнение операторов кода, а не физических строк. Например, предложение if может быть записано в одной строке:

int x = 42;

string s = "Not answered";

if( int x == 42) s = "Answered!";

При пошаговом выполнении этой строки отладчик обрабатывает условие как один шаг, а следствие как другой (в этом примере условие выполняется).

## **Приостановка выполнения кода с помощью точек остановка или команды "Прервать все"**

Во время отладки приложения с помощью отладчика VS приложение может либо выполняться, либо находиться в режиме приостановки выполнения.

Отладчик прерывает выполнение приложения, когда достигается точка останова или возникает исключение. Можно также прервать выполнение вручную в любой момент времени.

Точка останова — это сигнал, который указывает отладчику временно остановить выполнение приложения в определенной точке. Приостановка выполнения программы в точке останова называется режимом приостановки. Вход в режим приостановки выполнения не приводит к прекращению или завершению работы программы, поэтому выполнение программы может быть продолжено в любое время.

Многие возможности отладчика, такие как просмотр значений переменных в окне локальных переменных или вычисление выражений в окне контрольных значений, доступны только в режиме приостановки. В этом режиме все элементы приложения, например функции, переменные и объекты, сохраняются в памяти, но их перемещения и действия приостанавливаются. Во время режима приостановки выполнения можно выполнить поиск ошибок и нарушений целостности данных, проверив положения элементов и их состояние. В режиме приостановки выполнения можно также внести корректировки в приложение.

**Задание точек останова в коде**

Для установки простой точки останова в коде откройте исходный файл в редакторе Visual Studio. Поместите курсор в строку кода, на которой требуется приостановить выполнение, и выберите пункты **Точка останова**, **Вставить точку останова** в контекстном меню (на клавиатуре: **F9**). Отладчик приостанавливает выполнение непосредственно перед выполнением указанной строки.

Задание точки останова

## **Выполнение до указанного расположения или функции**

Иногда необходимо осуществить выполнение до определенного места в коде, а затем сделать остановку. Если вы установили точку останова в место, где требуется приостановить выполнение, выберите **Отладка**, **Начать отладку**, если отладка еще не запущена, или **Отладка**, **Продолжить** (в обоих случаях можно нажать клавишу **F5**). При выполнении кода отладчик останавливается в следующей точке останова. Выбирайте команду **Отладка**, **Продолжить**, пока не будет достигнута требуемая точка останова. Можно также осуществлять выполнение до места, где установлен курсор в редакторе кода, или до указанной функции.

**Выполнение до расположения курсора**

Чтобы осуществить выполнение до расположения курсора, поместите курсор в исполняемую строку кода в окне исходного кода. В контекстном меню редактора выберите команду **Выполнить до текущей позиции**.

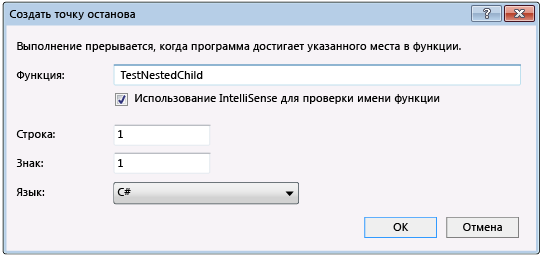
**Выполнение до функции в стеке вызовов**

Выберите функцию в окне **Стек вызовов** и щелкните команду **Выполнить до текущей позиции** в контекстном меню.

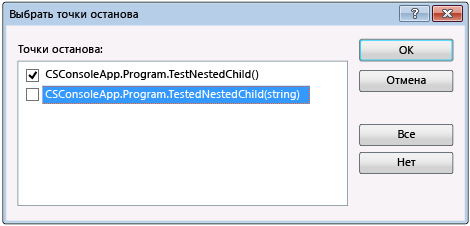
**Выполнение до функции, указанной по имени**

Можно дать команду отладчику на выполнение приложения до тех пор, пока не будет достигнута заданная функция. Можно задать функцию по имени или выбрать ее из стека вызовов.

Чтобы указать функцию по имени, последовательно выберите **Отладка**, **Создать точку останова**, **Прервать в функции**, а затем введите имя функции и другие идентификационные сведения.

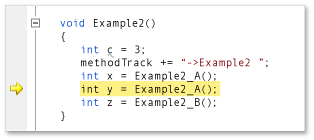


Если функция перегружается или находится в нескольких пространствах имен, необходимые функции можно выбрать в диалоговом окне **Выбор точек останова**.



**Задание следующего оператора для выполнения**

Приостановив выполнение в отладчике, можно переместить точку выполнения, чтобы задать следующий оператор кода для выполнения. Желтая стрелка на границе исходного текста или в окне "Дизассемблированный код" отмечает текущее расположение оператора, который должен быть выполнен следующим. Посредством перемещения этой стрелки можно пропустить часть кода или вернуться к строке, выполненной ранее. Это можно использовать при возникновении таких ситуаций, как пропуск раздела кода, содержащего известную ошибку.



Чтобы задать следующий оператор для выполнения, воспользуйтесь одной из следующих процедур.

В окне исходного кода перетащите желтую стрелку в то место этого же исходного файла, где необходимо задать следующий оператор.

В окне исходного кода установите курсор в строке, которую требуется выполнить следующей, и выберите в контекстном меню команду **Задать следующий оператор**.

В окне "Дизассемблирование" установите курсор на инструкцию ассемблера, которую требуется выполнить следующей, и выберите в контекстном меню команду **Задать следующий оператор**.

Нельзя устанавливать следующий оператор, если приложение выполняется. Чтобы задать следующий оператор, отладчик должен находиться в режиме приостановки.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ

Задания, предоставленные в ИТК 4 и 10, дополнить обработкой исключительных ситуаций: при возникновении ошибок должны выбрасываться исключения.

ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Албахари, Дж. C#5.0. Справочник. Полное описание языка / Дж. Албахари, Б. Албахари. – 5-е изд. – М.: ООО «И.Д. Вильямс», 2014. – 1008 с.: ил.
2. Дейтел, П. Как программировать на Visual C# 2012 / П. Дейтел, Х. Дейтел. – 5-е изд. – СПб.: Питер, 2014. – 864 с.: ил.
3. Зиборов, В.В. Visual C# 2012 на примерах / В.В Зиборов. – СПб.: БХВ-Петербург, 2013. – 480 с.: ил.
4. Майо, Дж. Самоучитель Microsoft Visual Studio 2010 / Дж. Майо. – СПб.: БХВ-Петербург, 2011. – 464 с.: ил.
5. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж. Рихтер. – 4-е изд. – СПб.: Питер, 2013. – 896 с.: ил.
6. Стиллмен, Э. Изучаем C# / Э. Стиллмен, Дж. Грин. – 3-е изд. – СПб.: Питер, 2014. – 816 с.: ил.
7. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен. – 6-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1312 с.: ил.
8. Шилдт, Г. C#4.0: Полное руководство / Г. Шилдт. – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.: ил.

Преподаватель Е.В. Багласова

|  |
| --- |
| Рассмотрено на заседании цикловой комиссии программного обеспечения информационных технологий №10  Протокол №\_\_\_\_\_от «\_\_\_»\_\_\_\_2016  Председатель ЦК *( )* Т.Г. Багласова |