Kapitel 35

Fehlersuche und Debugging

In diesem Kapitel:

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Leider ist die Arbeit des Programmierers üblicherweise nicht mit dem erfolgreichen Kompilieren und Linken der Anwendung abgeschlossen. Danach beginnt das Austesten des Programms, verbunden mit dem Ausmerzen auftretender Laufzeitfehler (Bugs). Der Visual C++-Debugger ist dabei eine wertvolle Hilfe.

HINWEIS Selbst mit einem so leistungsfähigen Hilfsmittel wie dem Visual C++-Debugger kann das Debuggen von Laufzeitfehlern eine langwierige und frustrierende Aufgabe sein. Sie sollten daher – vor allem bei größeren Projekten – unbedingt von Anfang alles tun, um den Debugaufwand klein zu halten. Ihr wichtigster Verbündeter beim Kampf gegen Laufzeitfehler ist die Modularisierung: Wenn Sie Ihren Code in Methoden und Klassen organisieren, darauf achten, dass Methodenimplementierungen nicht ausufern, sondern möglichst kurz bleiben und sich auf eine konkrete Aufgabe konzentrieren, und Sie jede Methode gleich nach der Fertigstellung auf Herz und Nieren testen, sollten Sie mit dem Testen und Debuggen der fertigen Anwendung keine allzu großen Schwierigkeiten haben.

Fehlerkategorien

Grundsätzlich haben wir mit drei verschiedenen Arten von Fehlern zu kämpfen:

- syntaktische Fehler
- Laufzeitfehler
- logische Fehler

Syntaktische Fehler bedeuten, dass der Compiler auf Code gestoßen ist, den er nicht übersetzen kann. Üblicherweise gibt der Compiler dann eine entsprechende Meldung darüber aus, wo der Fehler aufgetreten und was die vermutete Ursache ist. Syntaktische Fehler nehmen also eine Sonderstellung ein, da sie vom Compiler automatisch entdeckt, lokalisiert und definiert werden.

Hässlicher verhalten sich dagegen die Laufzeitfehler und die logischen Fehler – die eigentlichen Bugs. (Der Name Bug (wörtlich: *Wanze*, *Insekt*) geht auf einen Vorfall an der Harvard University zurück, wo eine in die Schaltungen eingedrungene Motte den Computer lahmlegte.) Diese sind meist nur sehr schwer aufzuspüren, auch wenn sich häufig recht triviale Fehler dahinter verbergen. Eine saubere Programmiertechnik und der integrierte Visual C++-Debugger können Ihnen dabei helfen, diese Fehler zu lokalisieren und zu beseitigen. Bevor Sie jedoch daran gehen können, einen solchen Fehler in Ihrem Quellcode zu lokalisieren, muss der Fehler natürlich erst einmal auftreten. Dies ist nicht unbedingt bei jeder Ausführung des Programms der Fall, da manche Fehler nur für bestimmte Eingaben (unabhängig, ob von Tastatur, Maus, Datei etc.) auftreten. Es bleibt Ihnen daher nichts anderes übrig, als zu testen, testen, testen …

Syntaktische Fehler

- Zur Kategorie der Syntaxfehler zählen Verstöße gegen die Grammatik der Sprache oder der unsachgemäße Gebrauch eines Typs
- Werden von Compiler identifiziert und gemeldet
- Eine erfolgreiche Kompilierung bedeutet, dass das Programm von Syntaxfehlern frei ist

```
int n = 3.2;                // Erzeugt Warnung wegen möglichem Datenverlust
Console::WriteNumber(n);    // Erzeugt Fehler, weil Console keine statische Methode
                            // WriteNumber() enthält
```

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Laufzeitfehler

- Laufzeitfehler führen gewöhnlich zu Programmabstürzen, das heißt, das Programm wird vom Betriebssystem abgebrochen, oder das Programm wird im Gegenteil überhaupt nicht mehr beendet
- Nicht endende Programme entstehen meist durch Endlosschleifen ohne zu erfüllende Abbruchbedingung oder durch Rekursion
- Laufzeitfehler, die nicht auf direkte Fehler im Quelltext, sondern auf bestimmte, äußere Bedingungen (Benutzereingaben, verfügbarer Speicher) zurückgehen, sollten im Programm selbst durch eine entsprechende Fallunterscheidung abgefangen werden – gegebenenfalls mit Fehlerausgabe, Ausnahmebehandlung (Exception Handling) und/oder Beendigung des Programms.

```
int n, a;
for(a = 10; a >= -10; a--) { n = n/a;}      // Division durch Null im 11. Durchgang

for(int i = 10; i > 0; i--) { i++; }        // Endlosschleife

int func2();
int func1() {return func2();}               // wechselseitiger
int func2() {return func1();}               // Aufruf
```

Logische Fehler

- Logische Fehler äußern sich darin, dass ein einwandfrei ablaufendes Programm nicht zu dem erwarteten Ergebnis führt
- Es ist daher unerlässlich, auch die Ausgabe eines Programms zu überprüfen

```
double arg;

for(arg = 0.0; arg < Math::PI/2.0; arg+=0.05)
{
     Console::WriteLine(Math::Sinh(arg));    // Sinh() statt Sin()
}
```

Vorbeugende Maßnahmen

Je komplexer und umfangreicher ein Programm, umso aufwändiger ist das Debuggen desselbigen. Während man beispielsweise für einfache Programme noch theoretisch beweisen kann, dass sie stets korrekt arbeiten (allerdings sind diese Programme meist so simpel, dass sich der Nachweis nicht lohnt), sind umfangreichere Programme in der Regel so komplex, dass der Beweis der Korrektheit nicht einmal mehr mithilfe von Supercomputern erbracht werden kann. Daraus ergibt sich die Forderung, schon bei der Programmerstellung die Fehleranalyse zu berücksichtigen, beispielsweise durch Modularisierung und besondere Aufmerksamkeit beim Erstellen kritischen Codes.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Modularisierung

Die Modularisierung eines Programms geschieht auf verschiedenen Ebenen:

- Code in Schleifen zusammenfassen
- Teilaufgaben als Methoden implementieren
- Objekte als Klassen implementieren
- Bibliotheken verwenden
- Programme in mehrere Quelltextdateien aufteilen

Der Vorteil der Modularisierung für das Debuggen liegt darin, dass die einzelnen Module (z.B. eine implementierte Methode) für sich getestet und dem Debugging unterzogen werden können. Statt also ein komplexes Programm als Ganzes zu debuggen, prüft man zuerst die einzelnen, übersichtlicheren Module und danach das Zusammenspiel dieser Module im Programm – im Vertrauen darauf, dass die Module für sich genommen ja korrekt arbeiten (wovon man ja beispielsweise auch bei der Verwendung der Methoden und Klassen der Laufzeitbibliotheken ausgeht).

HINWEIS Ideal ist es, wenn Sie die Module, vor allem die Methoden, gleich nach der Implementierung ausgiebig testen und gegebenenfalls debuggen.

Kritischen Code überprüfen

Fehler sind selten gleichmäßig über den Code eines Programms verteilt, vielmehr sind bestimmte Stellen eines Programms besonders anfällig.

- Schleifen sind stets darauf zu überprüfen, ob ihre Abbruchbedingungen irgendwann erfüllt werden. Nicht abbrechende Programmläufe weisen meist auf Endlosschleifen hin.
- Ebenso muss für Rekursionen sichergestellt sein, dass irgendwann die gewünschte Rekursionstiefe erreicht und die Rekursion nicht endlos fortgesetzt wird. Endlosrekursionen enden meist damit, dass der Stack überläuft.
- Der indizierte Zugriff auf traditionelle Arrays im C-Stil ist besonders anfällig dafür, mit ungültigen Indizes auf Speicher außerhalb des vorgesehenen Speicherbereichs zuzugreifen. (C++/CLI-Arrays fangen solche fehlerhaften Zugriffe mittels einer `IndexOutOfRangeException`-Ausnahme ab.)
- Eingaben müssen stets auf ihre Korrektheit überprüft werden. Dies gilt für die Eingabe seitens des Benutzers, der vielleicht in einem für Zahlen vorgesehenen Textfeld irrtümlicherweise Buchstaben eingibt, ebenso wie für die Methodenaufrufe innerhalb des Programms, wo eine Methode beispielsweise »-1« oder »0« als Argument erhält, aber nur natürliche Zahlen verarbeiten kann. Meist erfolgt die Überprüfung von Eingaben mithilfe von `if`- oder `switch`-Anweisungen oder einer passenden Ausnahmebehandlung.
- Die Anforderung von externen Daten oder Ressourcen (zu öffnende Datei, herzustellende Internetverbindung, Benutzereingaben) kann scheitern. Entsprechender Code sollte daher möglichst durch eine Ausnahmebehandlung abgesichert werden.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Das assert-Makro

Eine sehr praktische Möglichkeit, um an einer gegebenen Stelle im Code zu prüfen, ob eine bestimmte Bedingung (Invariante) eingehalten wird, ist das zum Standardumfang von C++ gehörende `assert`-Makro.

```
#include <cassert>
// …

int n = 12;
for(int a = 10; a >= -10; a--)
{
   assert(a != 0);

   n = n/a;
   Console::WriteLine(n + " " + a);
}
```

Stellt das `assert`-Makro bei Ausführung des Codes fest, dass die angegebene Bedingung nicht erfüllt ist, wird eine Fehlermeldung ausgegeben und das Programm abgebrochen.

Mit dem `assert`-Makro können Sie prüfen, ob Invarianten, die Sie mehr oder weniger stillschweigend postulieren (im obigen Beispiel wird etwa vorausgesetzt, dass die Variable `a` niemals gleich 0 wird), auch wirklich eingehalten werden. Wegen seiner informellen und recht abrupten Reaktion auf nicht eingehaltene Bedingungen eignet sich das Makro allerdings nicht für das Abfangen von Ausnahmesituationen in ausgelieferten Programmen (beispielsweise als Reaktion auf eine nicht gefundene Datei). Hierfür sind `if`-Bedingungen und Ausnahmebehandlung besser geeignet.

Um zu verhindern, dass die in den Code eingefügten Assertionen im fertigen Programm ausgeführt werden, genügt es, wenn Sie das Programm abschließend in der Release-Konfiguration erstellen (was sowieso zu empfehlen ist). In der Release-Konfiguration definiert nämlich Visual C++ den Compilerschalter `NDEBUG`, der die Auswertung der `assert`-Makros unterbindet.

Wenn Sie ein `assert`-Makro während des Debuggens temporär deaktivieren möchten, ohne die Build-Konfiguration zu wechseln, kommentieren Sie das Makro einfach aus oder definieren Sie selbst den Schalter `NDEBUG`:

```
#define NDEBUG           // der Schalter muss vor der #include <cassert>-Direktive
#include <cassert>       // definiert werden
```

Der Visual C++-Debugger

Der Debugger ist mit Abstand das wichtigste Werkzeug zum Auffinden von Laufzeit- und logischen Fehlern. Die grundlegenden Schritte beim Debuggen sind stets die gleichen.

1. **Anwendung im Debugger ausführen** Um die Anwendung, die Sie gerade in der Visual C++-IDE bearbeiten, im Debugger auszuführen, rufen Sie den Menübefehl *Debuggen/Debugging starten* auf. (Nach Aktivierung des Debuggers wird dieser Menübefehl durch den Befehl *Weiter* ersetzt, mit dem Sie eine im Debugger angehaltene Anwendung weiter ausführen können.)

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

2. **Ausführung anhalten** An den Stellen der Anwendung, die Ihnen verdächtig vorkommen und in denen Sie einen Fehler vermuten, halten Sie die Ausführung der Anwendung an. Dazu setzen Sie Haltepunkte oder lassen den Programmcode schrittweise ausführen.
3. **Zustand prüfen** Bevor Sie die Anwendung weiter ausführen lassen, sehen Sie sich in den Anzeigefenstern des Debuggers (Aufruf über *Debuggen/Fenster*) Informationen über den aktuellen Zustand der Anwendung an, beispielsweise den Inhalt von Variablen, den Zustand des Aufrufstacks (*Aufrufliste*) oder der Register. Mithilfe dieser Informationen versuchen Sie, Rückschlüsse auf Ort und Art des Fehlers zu ziehen.

Vorbereitung für das Debuggen

Damit der Debugger korrekt arbeiten kann, ist er auf Debug-Informationen angewiesen, die ihn über die von der Anwendung definierten und verwendeten Bezeichner (Symbole) informieren und die eine Verbindung zwischen Programmbefehlen und Quelltext herstellen.

Der einfachste und sicherste Weg, diese Debug-Informationen erzeugen zu lassen, ist, das zu debuggende Projekt in der Debug-Konfiguration zu erstellen.

1. Rufen Sie im Menü *Erstellen* den Befehl *Konfigurations-Manager* auf.
2. Im Dialogfeld des Konfigurations-Managers wählen Sie in der Zeile für das Projekt als Konfiguration den Eintrag *Debug* aus.
3. Schließen Sie das Dialogfeld und lassen Sie das Projekt gegebenenfalls neu erstellen.

HINWEIS Unter Visual C++ können Anwendungen auch in der Release-Konfiguration debuggt werden. Grundsätzlich ist davon allerdings abzuraten. Zum einen werden in der angelegten *.pdb*-Symboldatei keine vollständigen Debug-Informationen abgelegt. Zum anderen erlaubt die Release-Konfiguration die Code-Optimierung, der dann womöglich im Quelltext stehende Anweisungen und Bezeichner zum Opfer fallen, was wiederum zur Verwirrung führen kann, wenn Sie versuchen, die Bezeichner zu überwachen, oder in den Zeilen mit den wegrationalisierten Anweisungen Haltepunkte zu setzen.

Anwendungen im Debugger ausführen

Nachdem Sie Ihre Anwendung mit den Debug-Einstellungen kompiliert haben, können Sie die Anwendung im Debugger ausführen lassen. Rufen Sie dazu einfach einen der folgenden Befehle aus dem Menü *Debuggen* auf:

- *Debuggen/Debugging starten* – F5. Führt die Anwendung bis zum nächsten Haltepunkt aus.

  Die Haltepunkte sollten Sie bei Aufruf dieses Befehls in der Regel vorab setzen (siehe Abschnitt »Haltepunkte und Einzelschrittausführung« auf Seite 840). Ansonsten erhalten Sie erst wieder in Phasen, in denen die Anwendung auf Benutzereingaben wartet, Gelegenheit, Ihre Haltepunkte zu setzen (oder müssen mit *Debuggen/Alle unterbrechen* die Notbremse ziehen).
- *Debuggen/Einzelschritt* – F11. Startet die Anwendung und stoppt sie zu Beginn der Eintrittsfunktion `main()`. Die nachfolgenden Aufrufe führen die Anwendung schrittweise, das heißt Anweisung für Anweisung aus. Bei Methodenaufrufen springt der Debugger in den Code der Methode.
- *Debuggen/Prozedurschritt* – F10. Startet die Anwendung und stoppt sie zu Beginn der Eintrittsfunktion `main()`. Die nachfolgenden Aufrufe führen die Anwendung schrittweise aus. Methodenaufrufe werden dabei als ein Schritt ausgeführt.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

An externe oder laufende Anwendungen anhängen

Zum Debuggen einer Anwendung ist es nicht unbedingt erforderlich, dass die Anwendung im Debugger gestartet wird. Mit dem Visual C++-Debugger können Sie sich ebenso gut an eine laufende Anwendung anhängen. Und dabei muss es sich nicht einmal um die Anwendung handeln, deren Projekt gerade in Visual C++ geöffnet ist.

Um den Visual C++-Debugger an eine laufende Anwendung anzuhängen, gehen Sie folgendermaßen vor:

1. Rufen Sie im Menü *Debuggen* den Befehl *An den Prozess anhängen* auf.
2. Wählen Sie im daraufhin angezeigten Dialogfeld den zu der Anwendung gehörenden Prozess aus und klicken Sie auf *Anfügen*.

TIPP Das Anhängen an laufende Prozesse ist beispielsweise interessant, wenn Sie beim Testen einer Anwendung oder eines Bibliotheksprojekts ein Fehlerverhalten feststellen und dieses debuggen möchten. Anstatt den Testlauf erst beenden, eine Debug-Sitzung zu starten und an die verdächtige Stelle hinführen zu müssen, brauchen Sie nur den Debugger anzuhängen und können direkt mit der Analyse beginnen (außer natürlich, der Fehler hat zur Beendigung der Anwendung geführt oder lässt keinen Rücksprung zum auslösenden Code zu).

Befehlszeilenargumente

Wenn Sie einer zu debuggenden Anwendung Befehlszeilenargumente mitgeben möchten, wechseln Sie in den Projekteigenschaften zur Seite *Debuggen* und geben die gewünschten Argumente in das *Befehlsargumente*-Feld ein.

DLLs debuggen

Der Debugger kann nur ausführbare Anwendungen starten. Trotzdem ist es auch möglich, Bibliotheksprojekte zu debuggen. Sie benötigen allerdings eine passende Anwendung, die die Bibliothek und die in ihr definierten Typen verwendet. Dies kann die Anwendung sein, zu deren Unterstützung die Bibliothek überhaupt entwickelt wurde, oder eine eigens für die Bibliothek geschriebene Testanwendung.

Zum Debuggen des Bibliothekscodes stehen Ihnen dann zwei Optionen zur Verfügung:

- Sie starten die zugehörige Anwendung im Debugger oder
- Sie legen das Bibliotheksprojekt als Startprojekt fest, starten den Debugger und wählen in dem dann angezeigten Dialogfeld die *.exe*-Datei der Testanwendung aus, welche die Bibliothek verwendet

Die Ausführung von Bibliothekscode im Debugger ist eine Sache, das Debuggen einer konkreten Bibliotheksmethode eine andere. Um gezielt den Code einer Bibliotheksmethode debuggen zu können, benötigen Sie nicht nur eine Anwendung, die diese Methode aufruft, sondern auch eine Möglichkeit, den Debugger in dieser Methode anzuhalten:

- Wenn Sie das Bibliotheksprojekt in Visual C++ geöffnet haben, laden Sie einfach die zugehörige Quelltextdatei in den Editor und setzen in der Methode einen Haltepunkt.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

- Wenn Sie nur das zugehörige Anwendungsprojekt geöffnet haben und Bibliotheks- und Anwendungsprojekt nicht in einer gemeinsamen Projektmappe liegen, müssen Sie nicht extra zum Bibliotheksprojekt wechseln oder eine zweite Visual C++-Instanz aufrufen. Laden Sie einfach die betreffende Quelltextdatei über den Menübefehl *Datei/Öffnen* und setzen Sie Ihren Haltepunkt.
- Alternativ können Sie auch im Quelltext der Anwendung einen Haltepunkt auf eine Zeile setzen, die die Bibliotheksmethode aufruft, und später, wenn der Debugger an der betreffenden Stelle gehalten hat, mit F11 in die Methode hineinspringen. Der Debugger lädt dann automatisch die zugehörige Quelltextdatei.

HINWEIS Wenn Sie über den Anwendungscode in eine Bibliotheksmethode springen, zu der keine Quelltextdatei verfügbar ist (etwa weil Ihnen nur die DLL-Datei vorliegt), bietet Ihnen Visual C++ die Option an, den disassemblierten Code zu debuggen.

Debug-Sitzung beenden

Eine Debug-Sitzung endet automatisch, wenn die ausgeführte Anwendung beendet wird. Dauert dies zu lange, können Sie die Sitzung aus dem Debugger heraus beenden. Voraussetzung ist, dass die Anwendung angehalten wurde und der Debugger die Kontrolle besitzt. Rufen Sie dann den Menübefehl *Debuggen/Debugging beenden* auf.

Haltepunkte und Einzelschrittausführung

Erkenntnisse über die internen Abläufe in einer Anwendung gewinnen Sie nicht dadurch, dass Sie die Anwendung im Debugger ausführen lassen, sondern immer erst, nachdem Sie die Anwendung im Debugger angehalten haben. Nur bei angehaltener Anwendung können Sie mithilfe des Debuggers prüfen, welche Werte aktuell in den Variablen stehen oder welche Methoden auf dem Aufrufstack liegen, um sich so ein Bild vom aktuellen Zustand der Anwendung zu machen. Anschließend können Sie die Anwendung schrittweise ausführen, um zu verfolgen, wie sich die Variablenwerte ändern, in welche `if…else`-Blöcke verzweigt oder welche Schleife wie oft ausgeführt wird.

Zwei ganz wesentliche Features des Debuggers sind daher das Setzen von Haltepunkten und die Einzelschrittausführung.

Haltepunkte setzen

Die einfachste Möglichkeit, einen Haltepunkt zu setzen, besteht darin, im Editor mit der Maus in die graue Spalte links neben der Quelltextzeile zu klicken. Wenn Sie die Arbeit mit der Tastatur bevorzugen, setzen Sie die Einfügemarke in die betreffende Zeile und drücken dann F9. Der gesetzte Haltepunkt wird durch einen roten Punkt gekennzeichnet.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

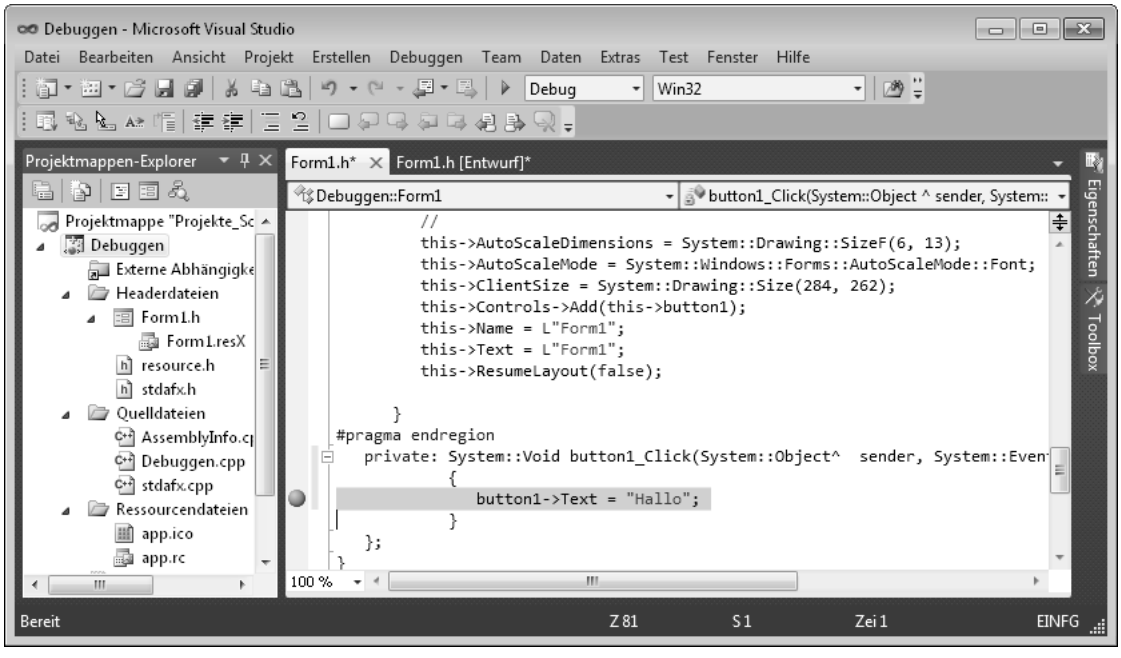

Abbildung 35.1 Haltepunkt im Editor

HINWEIS Nicht nur im Editor, auch im Fenster *Disassembly* können Haltepunkte gesetzt werden.

Haltepunkte deaktivieren oder löschen

Um einen Haltepunkt zu löschen, klicken Sie auf den Haltepunkt in der Randleiste oder setzen die Einfügemarke in die Zeile mit dem Haltepunkt und drücken erneut F9.

Haltepunkte, die zeitweilig nicht berücksichtigt, für spätere Verwendungen aber gesetzt bleiben sollen, können Sie im *Haltepunkte*-Fenster deaktivieren. Rufen Sie dazu dieses Fenster auf (Menübefehl *Debuggen/Fenster/Haltepunkte*) und klicken Sie auf das Häkchen neben dem gewünschten Haltepunkt, um diesen vorübergehend auszuschalten. Ein weiterer Klick aktiviert den Haltepunkt wieder.

Haltepunkte konfigurieren

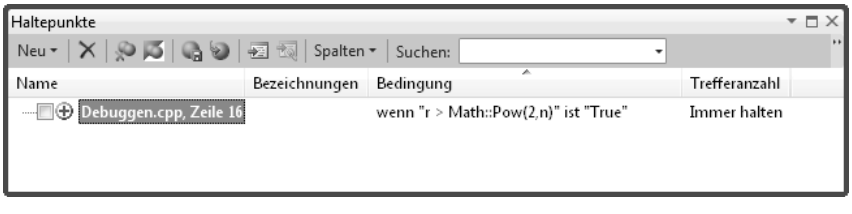

Abbildung 35.2 Im *Haltepunkte*-Fenster können Sie Haltepunkte verwalten und konfigurieren

Im Fenster *Haltepunkte* können Sie außerdem Haltepunkte konfigurieren und mit den verschiedensten Bedingungen und Aktionen verknüpfen. Zur Konfiguration öffnen Sie das Kontextmenü zu dem Haltepunkt und wählen einen der folgenden Befehle aus:

- **Speicherort** Zum Positionieren oder Verschieben des Haltepunkts.
- **Bedingung** Verknüpft den Haltepunkt mit einer booleschen Bedingung oder einem Ausdruck.

  Im ersten Fall wird die Anwendung nur dann angehalten, wenn die Bedingung `true` ergibt.

  Im zweiten Fall wird die Anwendung nur dann angehalten, wenn sich der Wert des Ausdrucks seit dem letzten Mal verändert hat (beim ersten Antreffen des Haltepunkts wird dieser immer ignoriert).

  Bedingung und Ausdruck werden in C++-Syntax formuliert und können Bezeichner aus dem aktuellen Gültigkeitsbereich verwenden. Eine ungültige Syntax erzeugt während des Debuggens eine Fehlermeldung und führt dazu, dass am Haltepunkt auf jeden Fall gehalten wird.
- **Trefferanzahl** Wird ein Haltepunkt im Verlauf einer Debug-Sitzung mehrere Male angesteuert, können Sie über die Trefferanzahl festlegen, für welche Treffer angehalten werden soll.

  Diese Einstellung kann beispielsweise für das Debuggen von Schleifen sehr hilfreich sein.
- **Filter** Diese Einstellung erlaubt Ihnen, Haltepunkte mit konkreten Computern, Prozessen und Threads zu verbinden.

  Haben Sie beispielsweise einen Haltepunkt in einen Anweisungsblock gesetzt, der von mehreren Threads ausgeführt wird, deren `Name`-Eigenschaften auf `"First"`, `"Second"` und `"Third"` gesetzt wurden, können Sie durch Angabe von `ThreadName = "Second"` festlegen, dass der Haltepunkt nur von diesem einen Thread beachtet wird.
- **Bei Treffer** Ruft ein Dialogfeld auf, in dem Sie festlegen können, ob bei Erreichen des Haltepunkts ein von Ihnen festgelegter Meldungstext in das *Ausgabe*-Fenster geschrieben, bestimmte Makros ausgeführt und die Ausführung angehalten oder automatisch fortgesetzt werden soll.
- **Bei Bezeichnungen bearbeiten** Ruft ein Dialogfeld auf, in dem Sie den Haltepunkt mit einem oder mehreren Schlüsselbegriffen verbinden können. Tippen Sie den Schlüsselbegriff ein und klicken Sie auf *Hinzufügen* oder wählen Sie einen oder mehrere Schlüsselbegriffe aus der Liste der vorhandenen Bezeichnungen aus.

  Der Sinn dieser Maßnahme ist, die Haltepunkte anhand ihrer Bezeichnungen zu gruppieren und gemeinsam zu verwalten. Haben Sie z.B. die Haltepunkte für zwei verschiedene Testläufe *TestA* bzw. *TestB* getauft, brauchen Sie in das *Suchen*-Feld des *Haltepunkte*-Fensters nur den Suchtext *TestA* einzugeben und abzuschicken und schon werden im *Haltepunkte*-Fenster nur noch die Haltepunkte mit der Bezeichnung *TestA* angezeigt. Weiterhin können Sie die Haltepunkte, auf die der Suchtext zutrifft, mit den Symbolschaltflächen des *Haltepunkte*-Fensters auf einen Schlag löschen, aktivieren/deaktivieren sowie exportieren.

  Es ist allerdings Vorsicht geboten. Die Suche findet nicht nur exakte Übereinstimmungen zwischen Suchtext und Bezeichnungen. Vielmehr wird die gesamte Textbeschreibung des Haltepunkts durchsucht (dies schließt z.B. den Inhalt der Felder *Bedingung* und *Trefferanzahl* mit ein) und jeder Haltepunkt, in dessen Text der Suchbegriff zu finden ist, wird ausgewählt. Der Suchbegriff *Test* würde also sowohl die Haltepunkte mit dem Bezeichner *TestA* als auch diejenigen mit dem Bezeichner *TestB* finden.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

HINWEIS Das *Haltepunkte*-Fenster zeigt standardmäßig nur die Bezeichnungen, die Bedingung und die Trefferanzahl an. Um weitere Einstellungen sichtbar zu machen, klicken Sie auf das Listenfeld *Spalten* und wählen die gewünschte Einstellung aus.

TIPP Mithilfe bedingter Haltepunkte lassen sich Schleifen effizient überwachen. Wenn Sie erst einmal wissen, in welcher Iteration der Fehler auftaucht, können Sie einen Haltepunkt in die Schleife setzen und als Bedingung angeben, dass dieser erst ab der betreffenden Iteration beachtet wird.

Zudem können Sie die Schleifenvariable über das Fenster *Überwachen* zurücksetzen, um den fehlerhaften Code mehrmals hintereinander austesten zu können (beispielsweise auch mit veränderten Variableninhalten).

Schrittweise Programmausführung

Eine der wertvollsten Eigenschaften des Debuggers ist die schrittweise Ausführung der debuggten Anwendung, die es erlaubt, genau mitzuverfolgen, welchen Weg die Programmausführung nimmt und wie sich die Werte der Variablen von Anweisung zu Anweisung verändern.

Befehl	Taste	Beschreibung
Weiter	F5	Führt die Anwendung bis zum Ende oder bis zum nächsten Haltepunkt aus
Einzelschritt	F11	Führt die jeweils nächste Anweisung aus. Handelt es sich um einen Methodenaufruf, verzweigt die Einzelschrittausführung in die Methode
Prozedurschritt	F10	Ähnlich wie Einzelschritt, verzweigt allerdings nicht in Methodenaufrufe
Ausführen bis Rücksprung	⇧ + F11	Führt die aktuelle Methode aus.

Tabelle 35.1 Befehle zur Programmausführung

Die Debug-Fenster

Zum integrierten Debugger gehören eine Reihe von Ansichtsfenstern, die im buchstäblichen wie übertragenen Sinne als Fenster in die Interna der Anwendung dienen.

Die Debug-Fenster werden über den Menübefehl *Debuggen/Fenster* aufgerufen.

Ausgabe

Das *Ausgabe*-Fenster ist für das Debuggen von eher untergeordneter Bedeutung. Der Debugger sendet verschiedene Statusmeldungen an das Fenster, anhand deren Sie beispielsweise kontrollieren können, ob die Symboldateien geladen wurden. Wenn Sie Haltepunkte mit Meldungen verbinden (siehe Abschnitt »Haltepunkte und Einzelschrittausführung«, Seite 840), werden diese ebenfalls zum *Ausgabe*-Fenster geschickt.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Überwachen

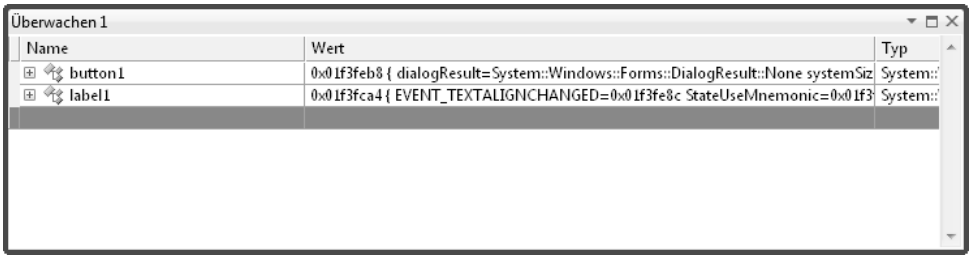

Abbildung 35.3 Das *Überwachen*-Fenster

In diesem Fenster können die Werte ausgewählter Variablen überwacht werden.

Um eine neue Variable zur Überwachung einzurichten, doppelklicken Sie einfach in der leeren Schablone in die Spalte *Name*, und geben Sie den Namen der zu überwachenden Variablen (oder einen Ausdruck) ein. Alternativ können Sie einen Namen auch per Drag & Drop aus Ihrem Quelltext in das Feld ziehen.

In der Spalte *Wert* wird der aktuelle Inhalt der Variablen angezeigt (für Ausdrücke der Wert des berechneten Ausdrucks). Wenn Sie hier einen Wert ändern, wird die Änderung an das Programm weitergereicht. Sie können auf diese Weise das Programm schnell für verschiedene Variablenwerte austesten.

Um eine Variable aus der Überwachung zu streichen, markieren Sie den Eintrag der Variablen und drücken Sie Entf.

Zur übersichtlicheren Verwaltung der zu überwachenden Ausdrücke stellt Ihnen das Fenster vier einzelne Seiten zur Verfügung (*Überwachen 1* bis *Überwachen 4*).

TIPP Um sich schnell über den Inhalt lokaler Variablen zu informieren, können Sie statt dieses Fensters auch die Fenster *Auto* und *Lokal* oder die *DataTips*-QuickInfos des Editors (siehe Abschnitt »DataTip (DatenInfo)«, Seite 846) nutzen.

HINWEIS Eine leistungsfähigere Alternative zur Auswertung und Zuweisung von Ausdrücken ist das Direktfenster, welches auch die Instanziierung von Objekten und die Ausführung sonstiger Anweisungen erlaubt.

Auto und Lokal

Diese beiden Fenster zeigen die Werte von Variablen aus dem aktuellen Kontext an. Die Liste wird automatisch erstellt, der Programmierer hat keinen Einfluss darauf.

Das Fenster *Auto* informiert über die Variablen der aktuellen und der vorangegangenen Anweisung.

Das Fenster *Lokal* informiert über die lokalen Variablen der aktuellen Methode.

TIPP Wenn Sie neben dem *Lokal*-Fenster das Fenster *Aufrufliste* einblenden, können Sie per Doppelklick auf die Methoden in der Aufrufliste zwischen den Stackrahmen hin und her springen. Im *Lokal*-Fenster werden dann jeweils die zugehörigen lokalen Variablen angezeigt.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Direktfenster

Im *Direktfenster* können Sie C++-Anweisungen und -Ausdrücke zur Auswertung und Ausführung an den Debugger schicken. In diesen Anweisungen und Ausdrücken lassen sich sämtliche Bezeichner verwenden, die im aktuellen Kontext (wo die zu debuggende Anwendung angehalten wurde) gültig sind.

Insbesondere können Sie (Ihre Eingabe schließen Sie jeweils mit ↵ ab):

- die Werte von Variablen kontrollieren (Variablennamen eingeben),
- die Werte von Ausdrücken berechnen (Ausdruck eingeben),
- die Rückgabewerte von Methoden ansehen (Methodenaufruf eingeben),
- beliebige Anweisungen ausführen lassen (Anweisung eingeben).

Sie können im *Direktfenster* auch Visual C++-Befehle ausführen. Bei der Auswahl der Befehle hilft Ihnen IntelliSense. Befehlsaufrufe im Direktfenster werden immer mit dem Zeichen »>« eingeleitet. Nachdem Sie den ersten Buchstaben eingetippt haben, beispielsweise den Anfangsbuchstaben eines Visual C++-Menüs, tritt IntelliSense in Erscheinung und hilft Ihnen bei der weiteren Auswahl.

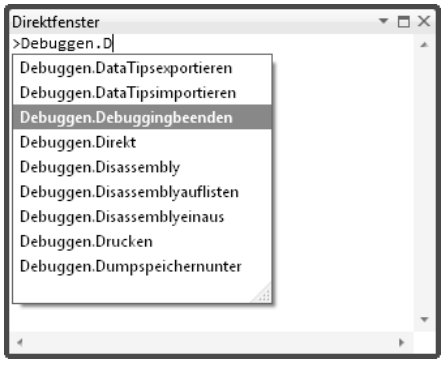

Abbildung 35.4 Auswahl eines Befehls im Direktfenster mit IntelliSense-Unterstützung

Befehle wie *Bearbeiten/Ersetzen*, die Dialogfelder öffnen, können mit Optionen (eingeleitet durch »/«) und Argumenten aufgerufen werden. Anstatt das Dialogfeld zu öffnen, wird der Befehl dann direkt ausgeführt. Die Optionen dieser Befehle korrespondieren mit den Einstellungen im Dialogfeld, die Argumente mit den Textfeldern. So ersetzt der folgende Befehl beispielsweise alle Vorkommen von `button1` durch `closeBtn`:

```
>Bearbeiten.Ersetzen /alles button1 closeBtn
```

Aufrufliste

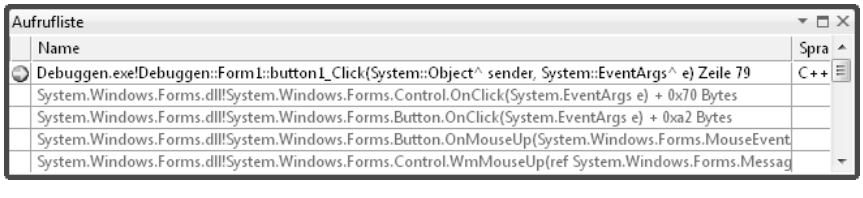

Abbildung 35.5 Die Aufrufliste zeigt den Zustand des Stacks

Dieses Fenster zeigt Ihnen an, welche Methoden bis zum Erreichen der aktuellen Ausführungsposition aufgerufen (und noch nicht beendet) wurden. Sie können in diesem Fenster also die Aufrufabfolge der Methoden (einschließlich der Parameterwerte) und den aktuellen Zustand des Programm-Stacks kontrollieren.

Weitere Fenster

- **Threads, Module und Prozesse** Über diese Fenster können Sie sich über die ausgeführten Prozesse, Module und Threads informieren
- **Arbeitsspeicher** Dieses Fenster lässt Sie den Zustand des Arbeitsspeichers kontrollieren
- **Disassembly** Dieses Fenster zeigt die disassemblierten Maschinencode-Anweisungen an (nicht den IL-Code). Die Darstellung kann über die Befehle im Kontextmenü angepasst werden.
- **Register** Dieses Fenster enthält die Namen und aktuellen Werte der plattformspezifischen Prozessorregister und Attribute

Debuggen im Editor

Der Code-Editor ist in die Schnittstelle des Debuggers voll integriert. Über die Randleiste des Editors können Sie wie bereits erwähnt Haltepunkte setzen und löschen, im Editor verfolgen Sie die schrittweise Programmausführung und selbst zur Dateninspektion können Sie den Editor verwenden.

DataTip (DatenInfo)

DataTip ist eine äußerst komfortable Möglichkeit, um sich während des Debuggens direkt im Editorfenster über die aktuellen Werte einzelner Variablen zu informieren.

Nachdem Sie die zu debuggende Anwendung an einer bestimmten Stelle angehalten haben, sehen Sie im Editor den zugehörigen Quelltext. Interessiert es Sie, welchen Wert die eine oder andere der im Quelltext verwendeten Variablen hat, bewegen Sie einfach den Mauszeiger für einen kurzen Moment über den Variablennamen. Daraufhin wird der DataTip eingeblendet und liefert Ihnen die gewünschte Information in einem Textrahmen.

In früheren Visual C++-Versionen konnten mit DataTip im Wesentlichen nur Variablen von primitiven Typen sinnvoll debuggt werden. Seit Visual C++ 2005 lassen sich auch die Inhalte von komplexen Typen, insbesondere Arrays, Strukturen und Klassen anzeigen. DataTip verwendet für diese Typen eine hierarchische Ansicht, die Sie per Klick auf die »+«-Symbole erweitern können. Der DataTip-Anzeigebereich bleibt geöffnet, solange der Mauszeiger im Bereich des Fensters bleibt.

TIPP Aufgeklappte *DataTip*-Anzeigen können schnell einen beachtlichen Teil des darunterliegenden Quelltextes verdecken. Wenn Sie einen Blick in diesen Code werfen möchten, ohne dazu den DataTip-Anzeigebereich zu schließen, drücken Sie einfach Strg.

Schnellüberwachung

Die Schnellüberwachung liefert in etwa dieselben Informationen wie die DataTip-Funktion, allerdings in einem eigenen Fenster und mit der Option, die überwachte Variable ins *Überwachen*-Fenster zu übernehmen.

Um die Schnellüberwachung für eine Variable zu aktivieren, klicken Sie mit der rechten Maustaste auf die Variable und wählen im Kontextmenü den Eintrag *Schnellüberwachung* aus.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Parallele Programme

Grundsätzlich verläuft das Debuggen parallelen Codes ebenso wie das Debuggen seriellen Codes. Das heißt, Sie können in den Code von Aufgaben, die Sie parallelisiert haben (siehe Kapitel 33), ganz normal Haltepunkte einfügen und sich dann, wenn die Anwendung am Haltepunkt angehalten wurde, über den Zustand der Anwendung informieren (Werte der Variablen, Aufrufstack etc.).

Eine Frage bleibt jedoch. Wenn Sie eine Anwendung per Haltepunkt an einer bestimmten Stelle innerhalb einer parallel ausgeführten Aufgabe anhalten, heißt dies in der Regel, dass der betreffende Code gerade von mehreren Threads gleichzeitig ausgeführt wird (zumindest auf Multicore-Architekturen). Welchen Thread haben wir also angehalten, mit welcher Aufgabe (Task) ist dieser Thread betraut und wie sieht es in den anderen Threads aus?

Zur Beantwortung dieser Fragen stellt uns der Debugger diverse spezielle Debug-Fenster zur Verfügung.

Das Threads-Fenster

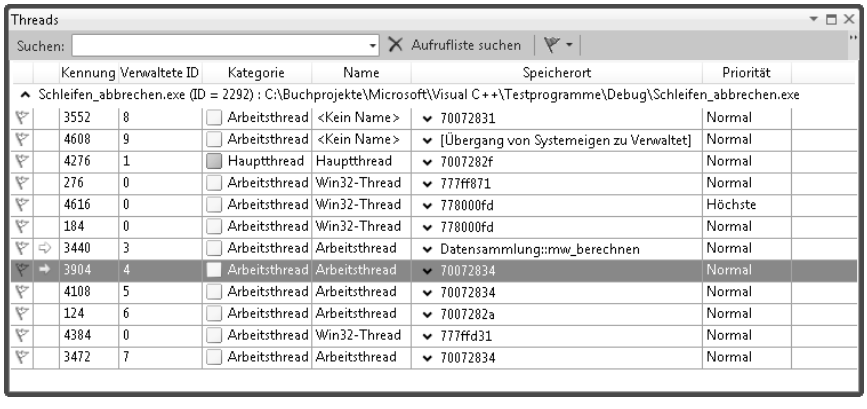

Abbildung 35.6 Das *Threads*-Fenster

Im *Threads*-Fenster werden die zur Anwendung (zum aktuellen Prozess) gehörenden Threads angezeigt. Aufgerufen wird das Fenster über den Befehl *Debuggen/Fenster/Threads*.

- Gelber Pfeil – Der aktuelle Thread, auf den sich die weiteren Debug-Fenster beziehen (also dessen Variableninhalte Sie z. B. im Fenster *Auto*, *Lokal* oder *Überwachen* kontrollieren), wird so gekennzeichnet.
- Weißer Pfeil – Der Thread, den der Debugger angehalten hat, wird auf diese Weise gekennzeichnet. (Die restlichen Threads werden anschließend ebenfalls angehalten.)

Anfangs, nachdem der Debugger die Anwendung angehalten hat, liegen weißer und gelber Pfeil übereinander. Sobald Sie aber im *Threads*-Fenster auf den Eintrag eines Threads doppelklicken, wird der angeklickte Thread zum aktuellen Thread, der gelbe Pfeil wechselt über und die weiteren Debug-Fenster passen gegebenenfalls ihren Inhalt an den aktuell ausgewählten Thread an.

Wie in Kapitel 33 erwähnt, wird paralleler Code auf Aufgaben verteilt, die wiederum mithilfe von Threads ausgeführt werden. Standardmäßig werden diese Threads in einem Threadpool verwaltet. Dabei kommt es häufig vor, dass es im Threadpool der Anwendung mehr Threads gibt als aktuell mit der Ausführung von Aufgaben beschäftigt sind. Wenn Sie im *Threads*-Fenster auf einen Thread doppelklicken, dem aktuell keine auszuführende Aufgabe zugeteilt ist oder der Code ausführt, zu dem keine Debuginformationen vorliegen, erscheint im Editor die Seite *Es ist keine Quelle verfügbar*.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Wenn Sie im *Threads*-Fenster auf einen Thread doppelklicken, der aktuell eine Aufgabe ausführt, zeigt ein grüner Pfeil links im Editorfenster an, welche Codezeile von diesem Thread als Nächstes ausgeführt werden wird.

HINWEIS Wenn Sie überprüfen möchten, ob ein Thread aus dem Threadpool stammt, doppelklicken Sie im *Threads*-Fenster auf den Thread und lassen Sie sich das Fenster *Aufrufliste* anzeigen. Hier sollte ein Eintrag *mscorlib.dll! System.Threading.ThreadPoolWorkQueue.Dispatch()* zu finden sein.

Das Fenster Parallele Aufgaben

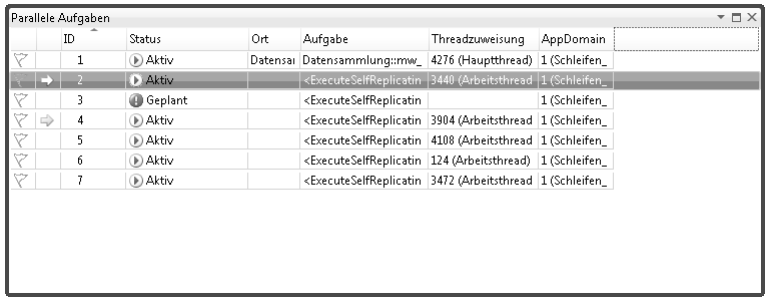

Abbildung 35.7 Auflistung der parallelen Aufgaben

Im *Parallele Aufgaben*-Fenster werden die aktuellen Aufgaben angezeigt. Es entspricht in Aufbau und Funktionsweise weitgehend dem Fenster *Threads*. Aufgerufen wird das Fenster über den Befehl *Debuggen/Fenster/Parallele Aufgaben*.

Das Fenster Parallele Stapel

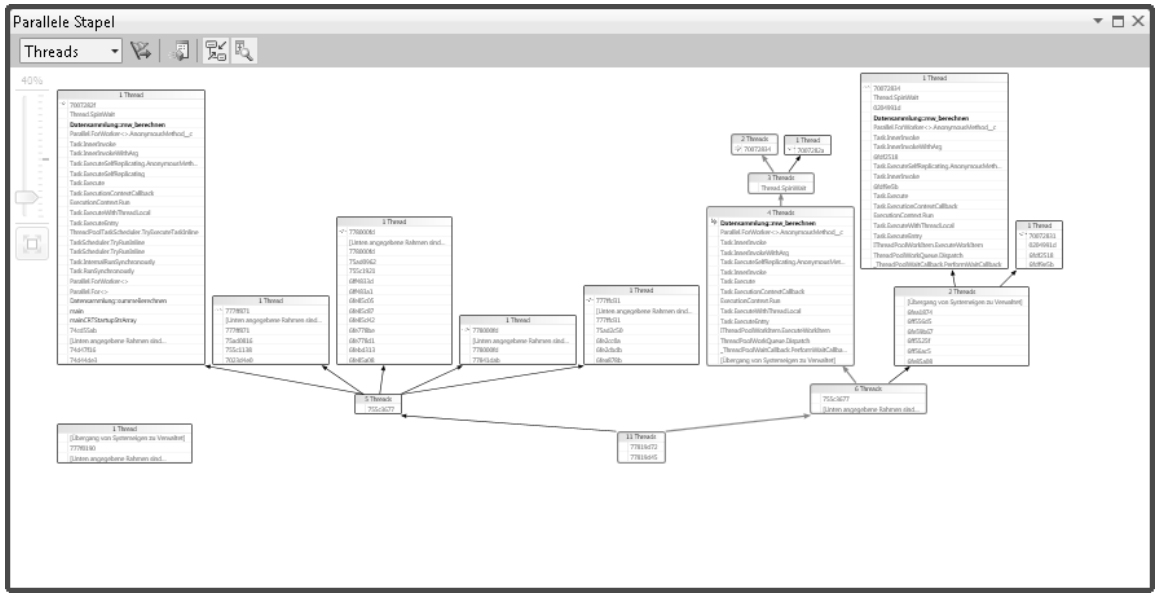

Abbildung 35.8 Hierarchische Darstellung der Threads (Aufgaben)

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

Im *Parallele Stapel*-Fenster werden die Stacks (Stapel) der Anwendung, aufgeteilt in gemeinsame Blöcke von Stackrahmen, grafisch dargestellt. Sie können die Stacks wahlweise geordnet nach Threads oder nach Aufgaben anzeigen lassen (Dropdown-Listenfeld in der Symbolleiste). Zudem lassen sich sowohl die Threads- als auch die Aufgaben-Darstellung in eine Methodenansicht umschalten (zweites Symbol neben dem Dropdown-Listenfeld), in welcher die Darstellung nach aktueller, aufrufenden und aufgerufenen Methoden geordnet wird. Eingeblendet wird das Fenster über den Befehl *Debuggen/Fenster/Parallele Stapel*.

TIPP Über das letzte Symbol in der Symbolleiste können Sie links oben im Fenster eine Zoomleiste zum Skalieren der Darstellung einblenden.

TIPP Vor allem die Thread-Ansicht kann wegen des Threadpools schnell unübersichtlich werden. Sie können die Ansicht aber filtern, sodass nur die Stapel bestimmter Threads angezeigt werden. Rufen Sie dazu das *Threads*-Fenster auf. Klicken Sie links auf die (weißen) Flaggensymbole des Hauptthreads und der Arbeitsthreads, denen bereits eine verwaltete ID zugewiesen wurde. (Die Win32-Threads und die Arbeitsthreads ohne ID lassen Sie außen vor). Wechseln Sie zurück zum Fenster *Parallele Stapel* und klicken Sie in der Symbolleiste auf die erste Schaltfläche mit dem Flaggensymbol.

Die Thread-Ansicht ist im Grunde eine alternative Darstellung für Aufrufstacks, wie Sie sie auch aus dem Debug-Fenster *Aufrufliste* her kennen. Es gibt jedoch Unterschiede: Erstens sehen Sie die Aufrufstacks aller Threads (bzw. der gefilterten Threads, siehe Tipp) gleichzeitig und zweitens sind die Aufrufstacks in Blöcke (die Kästchen) unterteilt, in denen die von einem oder mehreren Threads nacheinander ausgeführten Methoden (und damit Stackrahmen) zusammengefasst sind.

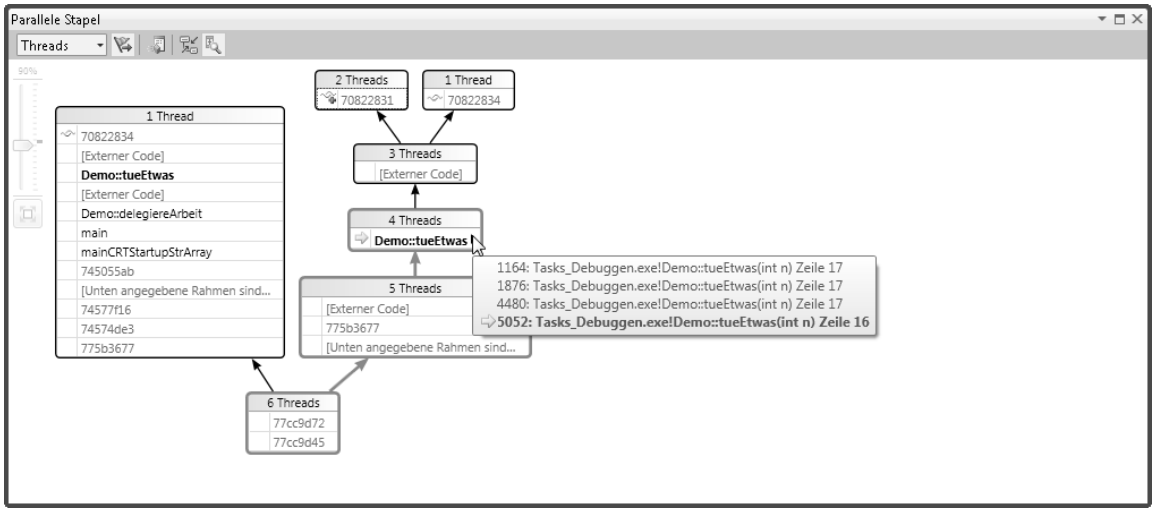

Abbildung 35.9 Thread-Ansicht für sechs ausgewählte Threads. Alle sechs Threads haben zuerst zwei Systemmethoden mit den Adressen 77cc9d45 und 77cc9d72 ausgeführt. Anschließend trennten sich die Wege des Hauptthreads und der fünf Arbeitsthreads. Vier der Arbeitsthreads haben auf ihrem Stack bereits den Stackrahmen für die Methode *tueEtwas()* liegen. Drei Threads führen sogar schon weitere Systemmethoden aus, die im Code von *tueEtwas()* aufgerufen werden.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)

In der Kopfzeile jedes Kästchens ist angegeben, wie viele Threads die im Kästchen aufgeführten Methoden durchlaufen haben (das heißt, sie haben – um ganz exakt zu sein – auf ihrem Stack Stackrahmen für die aufgeführten Methoden liegen.) Wenn Sie den Mauszeiger über die Kopfzeile bewegen, wird eine QuickInfo eingeblendet, der Sie entnehmen können, welche Threads dies im Einzelnen sind.

Ein gelber Pfeil markiert die aktuelle Ausführungsposition, an der der Debugger die Anwendung angehalten hat.

Die blau umrahmten Kästchen markieren zusammen den Stack des aktuell ausgewählten Threads. Die Abfolge der Stackrahmen in diesen Kästchen entspricht der Auflistung im Fenster *Aufrufliste*. So gibt es in Abbildung 35.9 z.B. vier Arbeitsthreads, die auf ihrem Stack bereits einen Stackrahmen für die Methode `tueEtwas()` liegen haben. Für den aktuellen Thread (mit der Kennung 5052) ist der Stackrahmen für die Methode `tueEtwas()` der oberste Stackrahmen (ansonsten gäbe es einen blauen Pfeil, der von dem Kästchen weiterführen würde). Die drei anderen Threads haben auf ihrem Stack bereits die Stackrahmen weiterer Methodenaufrufe liegen (siehe die darüber liegenden Kästchen).

Wenn Sie den aktuellen Thread wechseln, also z.B. im *Threads*-Fenster auf einen anderen Thread doppelklicken, ändert sich die Anzeige in der Thread-Ansicht, sodass die Abfolge der blauen Kästchen den Stack des neu ausgewählten Threads nachbildet.

HINWEIS Sie können den Thread auch direkt in der Thread-Ansicht des *Parallele Stapel*-Fensters wechseln. Klicken Sie dazu mit der rechten Maustaste auf den Stackrahmen einer Methode, die den betreffenden Thread ausführt, wählen Sie im erscheinenden Kontextmenü den Befehl *Zu Rahmen wechseln* und dann rechts in der eingeblendeten Liste auf den Thread.

Debugger konfigurieren

Die für den Debugger relevanten Einstellungen finden sich in den Projekteinstellungen (Aufruf über den Befehl *Eigenschaften* im Kontextmenü des Projektknotens) auf der Seite *Konfigurationseigenschaften/Debuggen*.

Walter Saumweber, Dirk Louis: Visual C++ - Das Entwicklerbuch, Microsoft Press 2011
(ISBN 978-3-86645-549-8)