H

Using the Visual Studio® Debugger

Objectives

In this appendix you'll:

- Set breakpoints and run a program in the debugger.
- Use the **Continue** command to continue execution.
- Use the **Locals** window to view and modify the values of variables.
- Use the **Watch** window to evaluate expressions.
- Use the **Step Into**, **Step Out** and **Step Over** commands to control execution.
- Use the **Autos** window to view variables that are used in the surrounding statements.
- Set breakpoints and run a program in the debugger.
- Use the **Continue** command to continue execution.

Outline

H.1 Introduction

In Chapter 2, you learned that there are two types of errors—compilation errors and logic errors—and you learned how to eliminate compilation errors from your code. Logic errors (also called **bugs**) do not prevent a program from compiling successfully, but can cause the program to produce erroneous results when it runs. Most C++ compiler vendors provide software called a **debugger**, which allows you to monitor the execution of your programs to locate and remove logic errors. The debugger will be one of your most important program development tools. This appendix demonstrates key features of the Visual Studio debugger. Appendix I discusses the features and capabilities of the GNU C++ debugger

H.2 Breakpoints and the Continue Command

We begin our study of the debugger by investigating **breakpoints**, which are markers that can be set at any executable line of code. When program execution reaches a breakpoint, execution pauses, allowing you to examine the values of variables to help determine whether a logic error exists. For example, you can examine the value of a variable that stores the result of a calculation to determine whether the calculation was performed correctly. Note that attempting to set a breakpoint at a line of code that is not executable (such as a comment) will actually set the breakpoint at the next executable line of code in that function.

To illustrate the features of the debugger, we use the program listed in Fig. H.3, which creates and manipulates an object of class `Account` (Figs. H.1–H.2). Execution begins in `main` (lines 10–27 of Fig. H.3). Line 12 creates an `Account` object with an initial balance of $50.00. `Account`'s constructor (lines 9–21 of Fig. H.2) accepts one argument, which specifies the `Account`'s initial `balance`. Line 15 of Fig. H.3 outputs the initial account balance using `Account` member function `getBalance`. Line 17 declares a local variable `withdrawalAmount`, which stores a withdrawal amount read from the user. Line 19 prompts the user for the withdrawal amount, and line 20 inputs the amount into `withdrawalAmount`. Line 23 subtracts the withdrawal from the `Account`'s `balance` using its `debit` member function. Finally, line 26 displays the new `balance`.

```
// Fig. H.1: Account.h
// Definition of Account class.
class Account
{
public:
   Account( int ); // constructor initializes balance
   void credit( int ); // add an amount to the account balance
```

Fig. H.1 | Header file for the `Account` class. (Part 1 of 2.)

```
   void debit( int ); // subtract an amount from the account balance
   int getBalance(); // return the account balance
private:
   int balance; // data member that stores the balance
}; // end class Account
```

Fig. H.1 | Header file for the Account class. (Part 2 of 2.)

```
// Fig. H.2: Account.cpp
// Member-function definitions for class Account.
#include <iostream>
using namespace std;

#include "Account.h" // include definition of class Account

// Account constructor initializes data member balance
Account::Account( int initialBalance )
{
   balance = 0; // assume that the balance begins at 0

   // if initialBalance is greater than 0, set this value as the
   // balance of the account; otherwise, balance remains 0
   if ( initialBalance > 0 )
      balance = initialBalance;

   // if initialBalance is negative, print error message
   if ( initialBalance < 0 )
      cout << "Error: Initial balance cannot be negative.\n" << endl;
} // end Account constructor

// credit (add) an amount to the account balance
void Account::credit( int amount )
{
   balance = balance + amount; // add amount to balance
} // end function credit

// debit (subtract) an amount from the account balance
void Account::debit( int amount )
{
   if ( amount <= balance ) // debit amount does not exceed balance
      balance = balance - amount;
   else // debit amount exceeds balance
      cout << "Debit amount exceeded account balance.\n" << endl;
} // end function debit

// return the account balance
int Account::getBalance()
{
   return balance; // gives the value of balance to the calling function
} // end function getBalance
```

Fig. H.2 | Definition for the Account class.

```
// Fig. H.3: figH_03.cpp
// Create and manipulate Account objects.
#include <iostream>
using namespace std;

// include definition of class Account from Account.h
#include "Account.h"

// function main begins program execution
int main()
{
   Account account1( 50 ); // create Account object

   // display initial balance of each object
   cout << "account1 balance: $" << account1.getBalance() << endl;

   int withdrawalAmount; // stores withdrawal amount read from user

   cout << "\nEnter withdrawal amount for account1: "; // prompt
   cin >> withdrawalAmount; // obtain user input
   cout << "\nattempting to subtract " << withdrawalAmount
      << " from account1 balance\n\n";
   account1.debit( withdrawalAmount ); // try to subtract from account1

   // display balances
   cout << "account1 balance: $" << account1.getBalance() << endl;
} // end main
```

Fig. H.3 | Test class for debugging.

Creating a Project in Visual Studio Express 2012 for Windows Desktop

In the following steps, you'll create a project that includes the code from Figs. H.1–H.3.

1. In Visual Studio 2012 Express for Windows Desktop select **File > New Project...** to display the **New Project** dialog.
2. In the **Installed Templates** list under **Visual C++**, select **Win32**, and in the center of the dialog, select **Win32 Console Application**.
3. In the **Name:** field, enter a name for your project and in the **Location:** field, specify where you'd like to save the project on your computer, then click **OK**.
4. In the **Win32 Application Wizard** dialog, click **Next >**.
5. Under **Application type:**, select **Console application**, and under **Additional options:** *uncheck* Precompiled header and Security Development Lifecycle (SDL) checks, select **Empty project** then click **Finish**.
6. In the **Solution Explorer**, right click your project's **Source Files** folder and select **Add > Existing Item...** to display the **Add Existing Item** dialog.
7. Locate the folder containing the Appendix H example code, select all three files and click **Add**.

Enabling Debug Mode, Inserting Breakpoints and Running in Debug Mode

In the following steps, you'll use breakpoints and various debugger commands to examine the value of the variable `withdrawalAmount` declared in Fig. H.3.

1. ***Enabling the debugger.*** The debugger is normally enabled by default. If it isn't, you can change the settings of the ***Solution Configurations* combo box** (Fig. H.4) in the toolbar. To do this, click the combo box's down arrow, then select **Debug**.

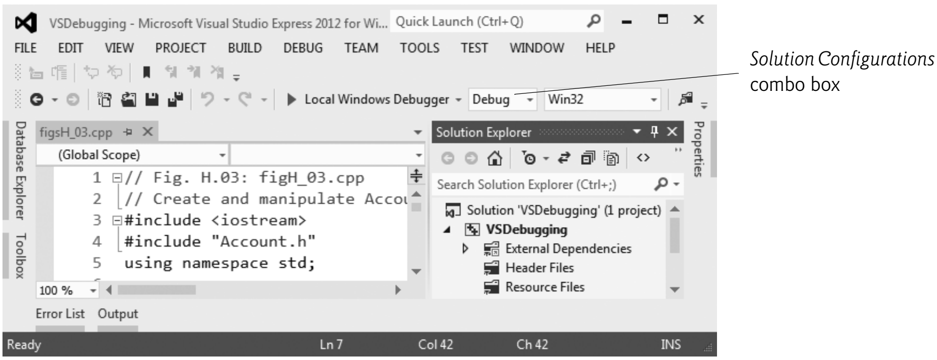

Fig. H.4 | Enabling the debugger.

2. ***Inserting breakpoints.*** Open `figH_03.cpp` by double-clicking it in the **Solution Explorer**. To insert a breakpoint, click inside the **margin indicator bar** (the gray margin at the left of the code window in Fig. H.5) next to the line of code at which you wish to break or right click that line of code and select **Breakpoint > Insert Breakpoint**. You can set as many breakpoints as necessary. Set breakpoints at lines 17 and 21 of your code. A red circle appears in the margin indicator bar where you clicked, indicating that a breakpoint has been set (Fig. H.5). When the program runs, the debugger pauses execution at any line that contains a breakpoint. The program is said to be in **break mode** when the debugger pauses the program. Breakpoints can be set before running a program, in break mode and while a program is running.

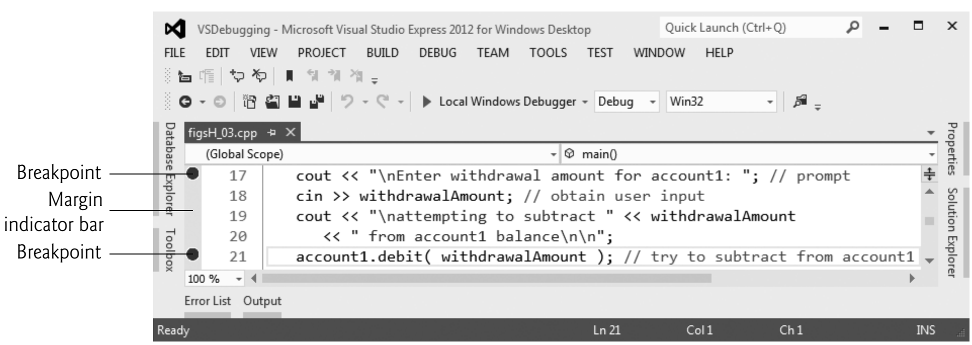

Fig. H.5 | Setting two breakpoints.

3. ***Starting to debug.*** After setting breakpoints in the code editor, select **Debug > Start Debugging** to build the program and begin the debugging process. When you debug a console application, a **Command Prompt** window appears (Fig. H.6) in which you can specify program input and view program output. The debugger enters break mode when execution reaches the breakpoint at line 17.

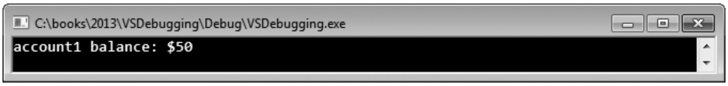

Fig. H.6 | **Inventory** program running.

4. ***Examining program execution.*** Upon entering break mode at the first breakpoint (line 17), the IDE becomes the active window (Fig. H.7). The **yellow arrow** to the left of line 17 indicates that this line contains the next statement to execute.

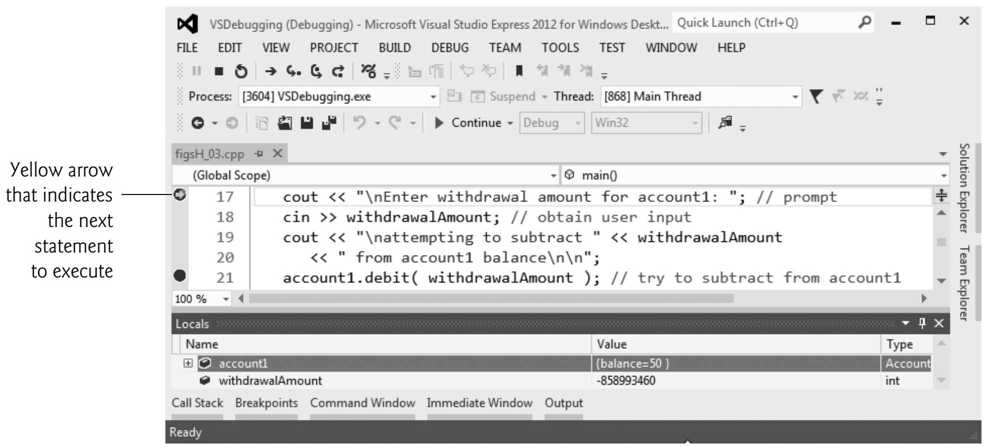

Fig. H.7 | Program execution suspended at the first breakpoint.

5. ***Using the* Continue *command to resume execution.*** To resume execution, select **Debug > Continue**. The **Continue command** resumes program execution until the next breakpoint or the end of `main` is encountered, whichever comes first. The program continues executing and pauses for input at line 18. Enter `13` as the withdrawal amount. The program executes until it stops at the next breakpoint (line 21). Notice that when you place your mouse pointer over the variable name `withdrawalAmount`, the value stored in the variable is displayed in a ***Quick Info*** **box** (Fig. H.8). As you'll see, this can help you spot logic errors in your programs.

6. ***Setting a breakpoint at `main`'s closing brace.*** Set a breakpoint at line 25 in the source code by clicking in the margin indicator bar to the left of line 25. This will prevent the program from closing immediately after displaying its result. When there are no more breakpoints at which to suspend execution, the program will execute to completion and the **Command Prompt** window will close. If you do not

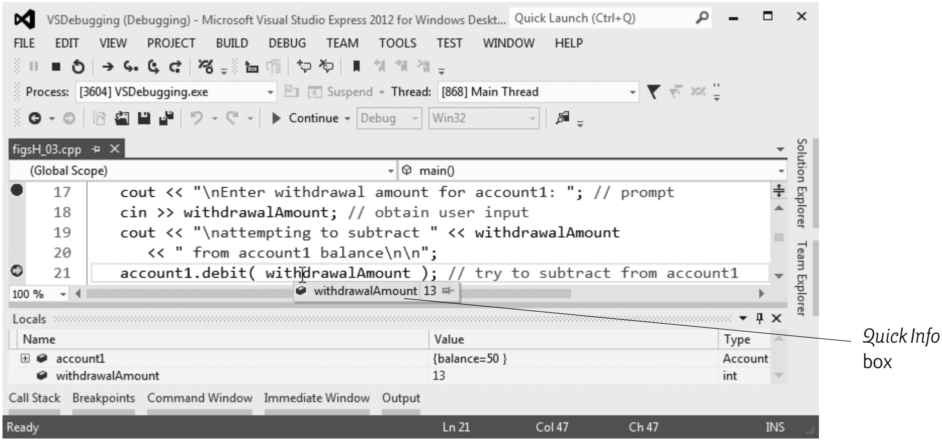

Fig. H.8 | *Quick Info* box showing the value of a variable.

set this breakpoint, you won't be able to view the program's output before the console window closes.

7. ***Continuing program execution.*** Use the **Debug > Continue** command to execute the code up to the next breakpoint. The program displays the result of its calculation (Fig. H.9).

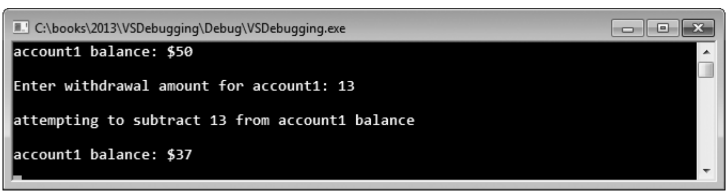

Fig. H.9 | Program output.

8. ***Removing a breakpoint.*** Click the breakpoint in the margin indicator bar.
9. ***Finishing program execution.*** Select **Debug > Continue** to execute the program to completion.

In this section, you learned how to enable the debugger and set breakpoints so that you can examine the results of code while a program is running. You also learned how to continue execution after a program suspends execution at a breakpoint and how to remove breakpoints.

H.3 Locals and Watch Windows

In the preceding section, you learned that the *Quick Info* feature allows you to examine a variable's value. In this section, you'll learn to use the **Locals window** to assign new values to variables while your program is running. You'll also use the **Watch window** to examine the value of more complex expressions.

1. ***Inserting breakpoints.*** Clear the existing breakpoints. Then, set a breakpoint at line 21 in the source code by clicking in the margin indicator bar to the left of line 21 (Fig. H.10). Set another breakpoint at line 24 by clicking in the margin indicator bar to the left of line 24.

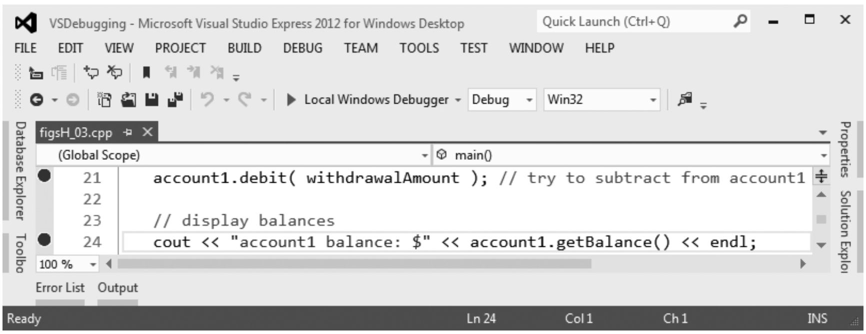

Fig. H.10 | Setting breakpoints at lines 21 and 24.

2. ***Starting debugging.*** Select **Debug > Start**. Type `13` at the **Enter withdrawal amount for account1:** prompt and press *Enter* so that your program reads the value you just entered. The program executes until the breakpoint at line 21.
3. ***Suspending program execution.*** The debugger enters break mode at line 21 (Fig. H.11). At this point, line 18 has input the `withdrawalAmount` that you entered (`13`), lines 19–20 have output that the program will attempt to withdraw money and line 21 is the next statement that will execute.

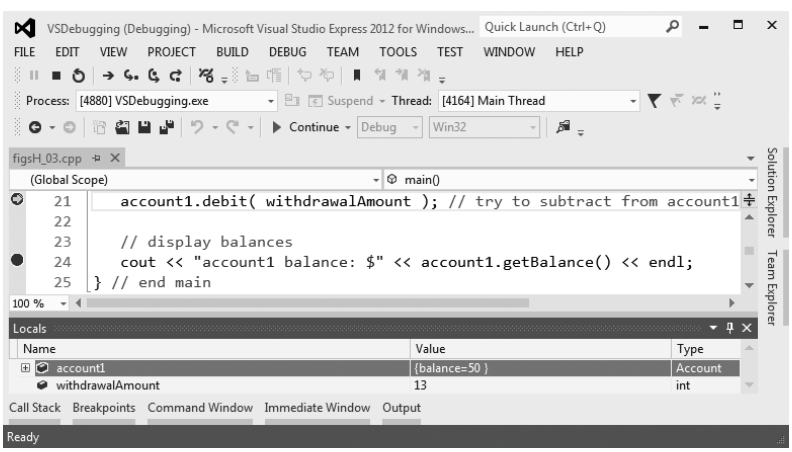

Fig. H.11 | Program execution suspended when debugger reaches the breakpoint at line 21.

4. ***Examining data.*** In break mode, you can explore the values of your local variables using the debugger's **Locals** window, which is normally displayed at the bottom left of the IDE when you are debugging. If it is not shown, you can view the **Locals** window, select **Debug > Windows > Locals**. Figure H.12 shows the values for `main`'s local variables `account1` and `withdrawalAmount` (`13`).

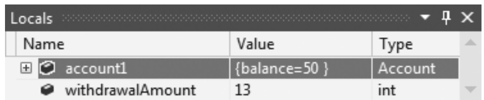

Fig. H.12 | Examining variable `withdrawalAmount`.

5. ***Evaluating arithmetic and boolean expressions.*** You can evaluate arithmetic and boolean expressions using the **Watch** window. You can display up to four **Watch** windows. Select **Debug > Windows > Watch > Watch 1**. In the first row of the **Name** column, type `(withdrawalAmount + 3) * 5`, then press *Enter*. The value of this expression (`80` in this case) is displayed in the **Value** column (Fig. H.13). In the next row of the **Name** column, type `withdrawalAmount == 3`, then press *Enter*. This expression determines whether the value of `withdrawalAmount` is `3`. Expressions containing the `==` operator (or any other relational or equality operator) are treated as `bool` expressions. The value of the expression in this case is `false` (Fig. H.13), because `withdrawalAmount` currently contains `13`, not `3`.

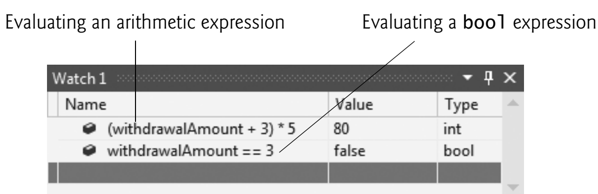

Fig. H.13 | Examining the values of expressions.

6. ***Resuming execution.*** Select **Debug > Continue** to resume execution. Line 21 debits the account by the withdrawal amount, and the debugger reenters break mode at line 24. Select **Debug > Windows > Locals** or click the **Locals** tab at the bottom of Visual Studio to redisplay the **Locals** window. The updated `balance` in `account1` is now displayed in red (Fig. H.14) to indicate that it has been modified since the last breakpoint. Click the plus box to the left of `account1` in the **Name** column of the **Locals** window. This allows you to view each of `account1`'s data member values individually—this is particularly useful for objects that have several data members.

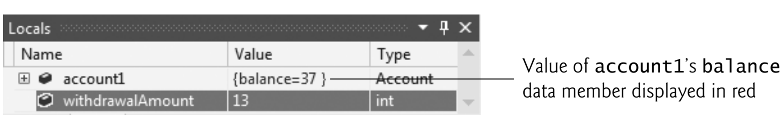

Fig. H.14 | Displaying the value of local variables.

7. ***Modifying values.*** Based on the value input by the user (`13`), the account balance output by the program should be `$37`. However, you can use the **Locals** window to change the values of variables during the program's execution. This can be valuable for experimenting with different values and for locating logic errors. In the **Locals** window, expand the `account1` node and double click the **Value** field in the `balance` row to select the value `37`. Type `33`, then press *Enter*. The debugger changes the value of `balance` and displays its new value in red (Fig. H.15).

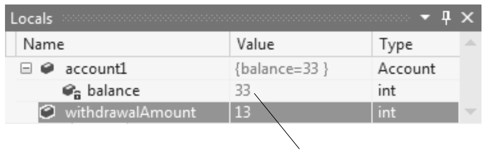

Fig. H.15 | Modifying the value of a variable.

8. ***Setting a breakpoint at at `main`'s closing brace.*** Set a breakpoint at line 25 in the source code to prevent the program from closing immediately after displaying its result. If you do not set this breakpoint, you won't be able to view the program's output before the console window closes.
9. ***Viewing the program result.*** Select **Debug > Continue** to continue program execution. Function `main` executes until the `return` statement in line 29 and displays the result. Notice that the result is `$33` (Fig. H.16). This shows that *Step 7* changed the value of `balance` from the calculated value (`37`) to `33`.

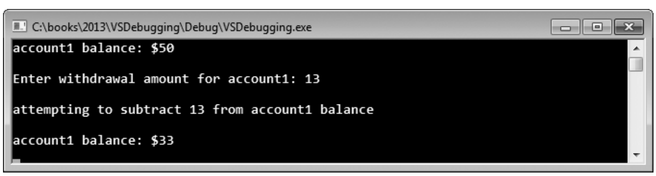

Fig. H.16 | Output displayed after modifying the `account1` variable.

10. ***Stopping the debugging session.*** Select **Debug > Stop Debugging**. This will close the **Command Prompt** window. Remove all remaining breakpoints.

In this section, you learned how to use the debugger's **Watch** and **Locals** windows to evaluate arithmetic and boolean expressions. You also learned how to modify the value of a variable during your program's execution.

H.4 Controlling Execution Using the Step Into, Step Over, Step Out and Continue Commands

Sometimes executing a program line by line can help you verify that a function's code executes correctly, and can help you find and fix logic errors. The commands you learn in this section allow you to execute a function line by line, execute all the statements of a function at once or execute only the remaining statements of a function (if you've already executed some statements within the function).

1. ***Setting breakpoints.*** Set breakpoints at lines 21 and 25 by clicking in the margin indicator bar to the left of the line.
2. ***Starting the debugger.*** Select **Debug > Start**. Enter the value `13` at the **Enter withdrawal amount for account1:** prompt. Execution will halt when the program reaches the breakpoint at line 21.
3. ***Using the Step Into command.*** The **Step Into command** executes the next statement in the program (line 21), then immediately halts. If that statement is a function call (as is the case here), control transfers into the called function. This enables you to execute each statement inside the function individually to confirm the function's execution. Select **Debug > Step Into** (or press *F11*) to enter the `debit` function. Then, Select **Debug > Step Into** again so the yellow arrow is positioned at line 31 of `Account.cpp`.
4. ***Using the Step Over command.*** Select **Debug > Step Over** to execute the current statement (line 31) and transfer control to line 32. The **Step Over command** behaves like the **Step Into** command when the next statement to execute does not contain a function call. You'll see how the **Step Over** command differs from the **Step Into** command in *Step 9*.
5. ***Using the Step Out command.*** Select **Debug > Step Out** to execute the remaining statements in the function and return control to the next executable statement (line 28 in Fig. H.3). Often, in lengthy functions, you'll want to look at a few key lines of code, then continue debugging the caller's code. The **Step Out command** enables you to continue program execution in the caller without having to step through the entire called function line by line.
6. ***Using the Continue command.*** Select **Debug > Continue** to execute until the next breakpoint is reached at line 25. Using the **Continue** command is useful when you wish to execute all the code up to the next breakpoint.
7. ***Stopping the debugger.*** Select **Debug > Stop Debugging** to end the debugging session. This will close the **Command Prompt** window.
8. ***Starting the debugger.*** Before we can demonstrate the next debugger feature, you must start the debugger again. Start it, as you did in *Step 2*, and enter `13` in response to the prompt. The debugger enters break mode at line 21.
9. ***Using the Step Over command.*** Select **Debug > Step Over**. This command behaves like the **Step Into** command when the next statement to execute *does not* contain a function call. If the next statement to execute *contains* a function call, the called function executes in its *entirety* (without pausing execution at any statement inside the function), and the yellow arrow advances to the next executable line (after the function call) in the current function. In this case, the debugger executes line 21, located in `main` (Fig. H.3). Line 21 calls the `debit` function. The debugger then pauses execution at line 24, the next executable line in the current function, `main`.

10. *Stopping the debugger*. Select **Debug > Stop Debugging**. This will close the **Command Prompt** window. Remove all remaining breakpoints.

In this section, you learned how to use the debugger's **Step Into** command to debug functions called during your program's execution. You saw how the **Step Over** command can be used to step over a function call. You used the **Step Out** command to continue execution until the end of the current function. You also learned that the **Continue** command continues execution until another breakpoint is found or the program exits.

H.5 Autos Window

The **Autos window** displays the variables used in the previous statement executed (including the return value of a function, if there is one) and the variables in the next statement to execute.

1. *Setting breakpoints*. Clear your prior breakpoints, then set breakpoints at lines 10 and 18 in `main`.
2. *Using the* **Autos** *window.* Start the debugger by selecting **Debug > Start**. When the debugger enters break mode at line 10, open the **Autos** window by selecting **Debug > Windows > Autos** (Fig. H.17). Since we are just beginning the program's execution, the **Autos** window lists only the variable(s) in the next statement that will execute—in this case, the `account1` object, its value and its type. Viewing the values stored in an object lets you verify that your program is manipulating these variables correctly. Notice that `account1` contains a large negative value. This value, which may be different each time the program executes, is `account1`'s uninitialized value. This unpredictable (and often undesirable) value demonstrates why it is important to initialize all C++ variables before they are used.

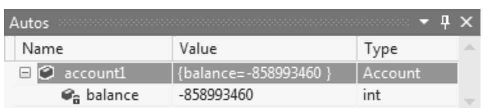

Fig. H.17 | **Autos** window displaying the state of `account1` object.

3. *Using the* **Step Over** *command.* Select **Debug > Step Over** to execute line 10. The **Autos** window updates the value of `account1`'s `balance` data member (Fig. H.18) after it is initialized.

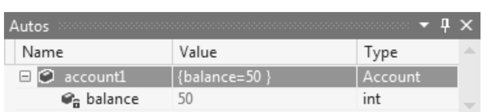

Fig. H.18 | **Autos** window displaying the state of `account1` object after initialization.

4. *Continuing execution.* Select **Debug > Continue** to execute the program until the second breakpoint at line 18. The **Autos** window displays uninitialized local variable `withdrawalAmount` (Fig. H.19), which has a large negative value.

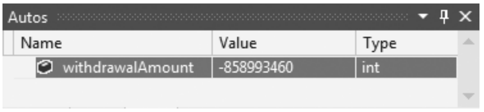

Fig. H.19 | **Autos** window displaying local variable `withdrawalAmount`.

5. *Entering data.* Select **Debug > Step Over** to execute line 18. At the program's input prompt, enter a value for the withdrawal amount. The **Autos** window updates the value of local variable `withdrawalAmount` with the value you entered (Fig. H.20).

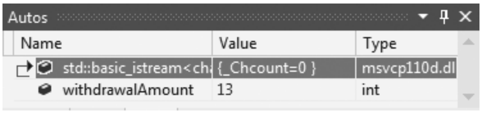

Fig. H.20 | **Autos** window displaying updated local variable `withdrawalAmount`.

6. *Stopping the debugger.* Select **Debug > Stop Debugging** to end the debugging session. Remove all remaining breakpoints.

H.6 Wrap-Up

In this appendix, you learned how to insert, disable and remove breakpoints in the Visual Studio debugger. Breakpoints allow you to pause program execution so you can examine variable values. This capability will help you locate and fix logic errors in your programs. You saw how to use the **Locals** and **Watch** windows to examine the value of an expression and how to change the value of a variable. You also learned debugger commands **Step Into**, **Step Over**, **Step Out** and **Continue** that can be used to determine whether a function is executing correctly. Finally, you learned how to use the **Autos** window to examine variables used specifically in the previous and next commands.