

# Using the Xcode® Debugger

## Objectives

In this appendix you'll:

- Set breakpoints and run a program in the debugger.
- Use the **Continue program execution** command to continue execution.
- Use the **Auto** window to view and modify the values of variables and watch expression values.
- Use the **Step Into**, **Step Out** and **Step Over** commands to control execution.



<b>J.1</b> Introduction	<b>J.4</b> Controlling Execution Using the <b>Step Into</b> , <b>Step Over</b> and <b>Step Out</b> Commands
<b>J.2</b> Breakpoints and the <b>Continue program execution</b> Command	<b>J.5</b> Wrap-Up
<b>J.3</b> Auto Window	

## J.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 Xcode debugger.

## J.2 Breakpoints and the Continue program execution 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. J.3, which creates and manipulates an object of class `Account` (Figs. J.1–J.2). Execution begins in `main` (lines 10–27 of Fig. J.3). Line 9 creates an `Account` object with an initial balance of \$50.00. `Account`'s constructor (lines 8–19 of Fig. J.2) accepts one argument, which specifies the `Account`'s initial balance. Line 12 of Fig. J.3 outputs the initial account balance using `Account` member function `getBalance`. Line 14 declares a local variable `withdrawalAmount`, which stores a withdrawal amount read from the user. Line 16 prompts the user for the withdrawal amount, and line 17 inputs the amount into `withdrawalAmount`. Line 20 subtracts the withdrawal from the `Account`'s balance using its `withdraw` member function. Finally, line 23 displays the new balance.

```

1 // Fig. J.1: Account.h
2 // Definition of Account class.
3 class Account {
4 public:
5     Account(int); // constructor initializes balance
6     void deposit(int); // add an amount to the account balance
7     void withdraw(int); // subtract an amount from the account balance
8     int getBalance(); // return the account balance

```

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

---

```
9 private:
10     int balance{0}; // data member that stores the balance
11 }; // end class Account
```

---

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

---

```
1 // Fig. J.2: Account.cpp
2 // Member-function definitions for class Account.
3 #include <iostream>
4 #include "Account.h" // include definition of class Account
5 using namespace std;
6
7 // Account constructor initializes data member balance
8 Account::Account(int initialBalance) {
9     // if initialBalance is greater than 0, set this value as the
10    // balance of the Account; otherwise, balance remains 0
11    if (initialBalance > 0) {
12        balance = initialBalance;
13    }
14
15    // if initialBalance is negative, print error message
16    if (initialBalance < 0) {
17        cout << "Error: Initial balance cannot be negative.\n" << endl;
18    }
19 }
20
21 // deposit (add) an amount to the account balance
22 void Account::deposit(int amount) {
23     balance = balance + amount; // add amount to balance
24 }
25
26 // withdraw (subtract) an amount from the account balance
27 void Account::withdraw(int amount) {
28     if (amount <= balance) { // withdrawal amount OK
29         balance = balance - amount;
30     }
31     else { // withdraw amount exceeds balance
32         cout << "Withdrawal amount exceeded balance.\n" << endl;
33     }
34 }
35
36 // return the account balance
37 int Account::getBalance() {
38     return balance; // gives the value of balance to the calling function
39 }
```

---

**Fig. J.2** | Definition for the Account class.

---

```

1 // Fig. J.3: figJ_03.cpp
2 // Create and manipulate Account objects.
3 #include <iostream>
4 #include "Account.h"
5 using namespace std;
6
7 // function main begins program execution
8 int main() {
9     Account account1{50}; // create Account object
10
11     // display initial balance of each object
12     cout << "account1 balance: $" << account1.getBalance() << endl;
13
14     int withdrawalAmount; // stores withdrawal amount read from user
15
16     cout << "\nEnter withdrawal amount for account1: "; // prompt
17     cin >> withdrawalAmount; // obtain user input
18     cout << "\nattempting to subtract " << withdrawalAmount
19         << " from account1 balance\n\n";
20     account1.withdraw(withdrawalAmount); // try to subtract from account1
21
22     // display balances
23     cout << "account1 balance: $" << account1.getBalance() << endl;
24 }

```

---

**Fig. J.3** | Test class for debugging.

### *Opening the Xcode Project for This Appendix*

We've included an Xcode project (Debugging.xcodeproj) for the code from Figs. J.1–J.3 in the appJ/Debugging folder with this book's examples. In the Mac OS X Finder, navigate to

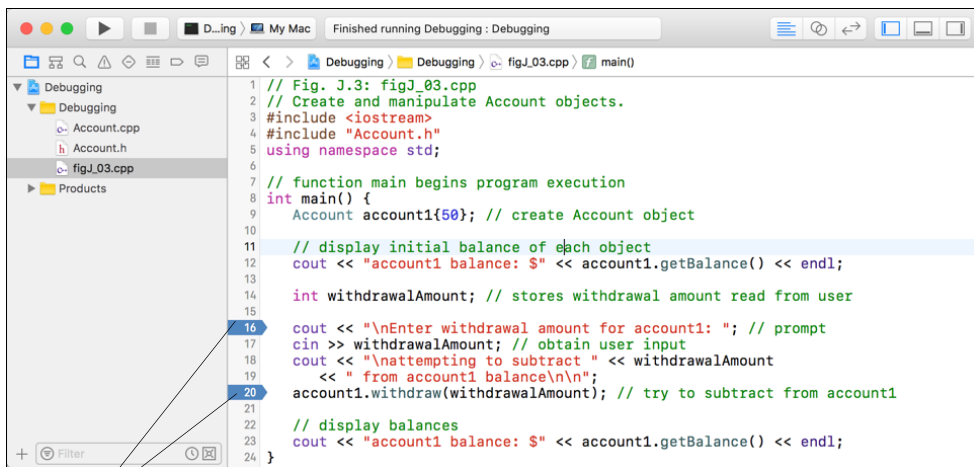
Documents/examples/appJ/Debugging

and double click the Debugging.xcodeproj to open the project's workspace window in Xcode.

### *Inserting Breakpoints*

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

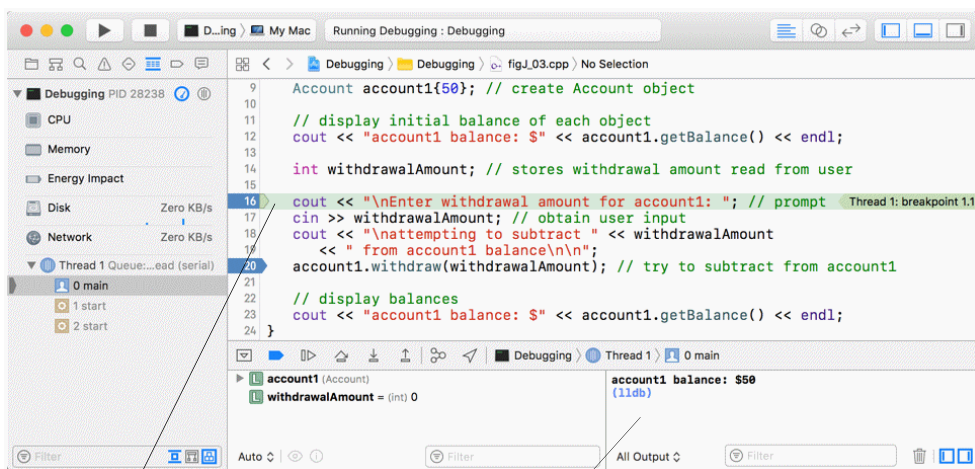
1. **Inserting breakpoints.** Click `figJ_03.cpp` in the **Project** navigator to display the file in the Xcode **Editor** area. To insert a breakpoint, click inside the gray bar to the left of the line of code at which you wish to break. You can set as many breakpoints as necessary. Set breakpoints at lines 16 and 20. A blue arrow appears to the left of the line where you clicked, indicating that a breakpoint has been set (Fig. J.4). 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.



Breakpoints

**Fig. J.4** | Setting two breakpoints.

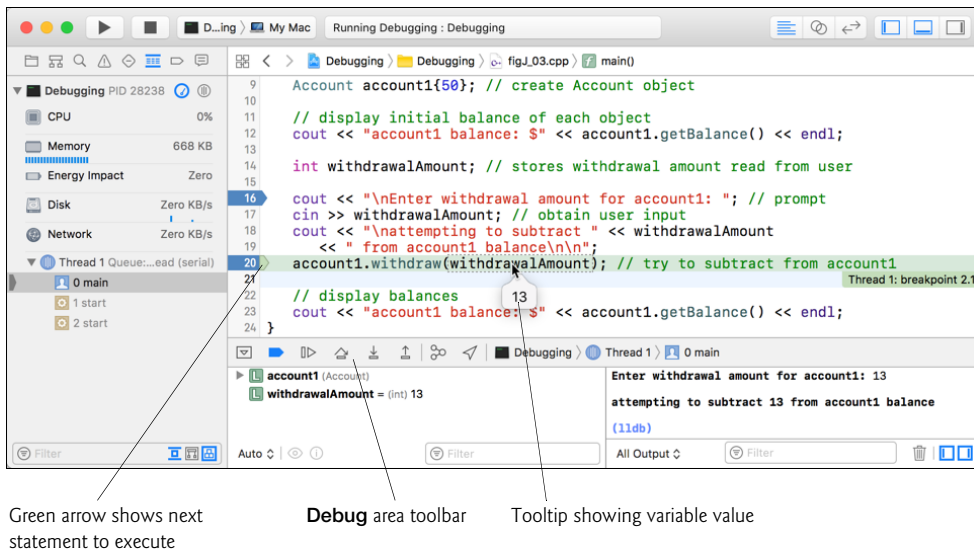
2. **Starting to debug.** After setting breakpoints in the code editor, click the **Run** (▶) button on the workspace window's tool bar to build the program and begin the debugging process. When you debug an application that normally runs in a **Terminal** window, you can see the program's output and provide input in the **Debug** area below the **Editor** area in the workspace window (Fig. J.5). The debugger enters break mode when execution reaches the breakpoint at line 16.



Next line of code to execute is highlighted in green      You interact with the application in this part of the **Debug** area

**Fig. J.5** | Inventory program suspended at first breakpoint.

3. *Examining program execution.* Upon entering break mode at the first breakpoint (line 16), the IDE becomes the active window (Fig. J.5). The **green arrow** to the left of line 16 indicates that this line contains the next statement to execute.
4. *Using the Continue program execution button to resume execution.* To resume execution, click the **Continue program execution** (⏮) button in the **Debug** area's toolbar, which resumes program execution until the next breakpoint or until the program terminates, whichever comes first. The program continues executing and pauses for input at line 17. Enter 13 as the withdrawal amount. The program executes until it stops at the next breakpoint (line 20). Notice that when you place your mouse pointer over the variable name `withdrawalAmount`, the value stored in the variable is displayed in a yellow tooltip below the mouse cursor (Fig. J.6). As you'll see, this can help you spot logic errors in your programs.



**Fig. J.6** | Quick Info box showing the value of a variable.

5. *Completing the program's execution.* Click the **Continue program execution** (⏮) button to complete the program's execution. The result of the program's calculation is shown in the **Debug** area (Fig. J.7).

```


account1 balance: $50
Enter withdrawal amount for account1: 13
attempting to subtract 13 from account1 balance
account1 balance: $37
Program ended with exit code: 0
  
```

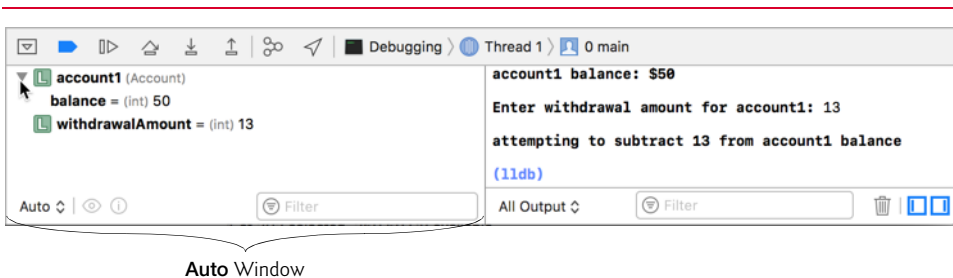
**Fig. J.7** | Program output.

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.

## J.3 Auto Window


In this section, you'll learn to use the **Auto window** to watch variable and expression values and to assign new values to variables while your program is running.

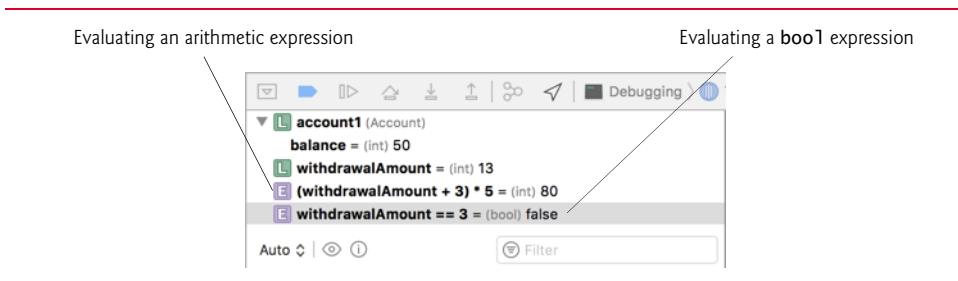
1. **Inserting breakpoints.** Clear the breakpoints at line 16 by right clicking it and selecting **Delete Breakpoint**. Then, set another breakpoint at line 23 of `figJ_03.cpp`.
2. **Starting to debug.** Click the **Run** (▶) button on the workspace window's tool bar to build the program and begin the debugging process. Type 13 at the **Enter withdrawal amount for account1:** prompt and press *Return* so that your program reads the value you just entered. The program executes until the breakpoint at line 20. At this point, line 17 has input the `withdrawalAmount` that you entered (13), lines 18–19 have output that the program will attempt to withdraw money and line 20 is the next statement that will execute.
3. **Examining data.** The left side of the **Debug** area displays the **Auto** window by default. This window automatically shows variables that are currently in scope—the  icon indicates that a variable is a local variable. This window allows you to explore the current values of variables. Click the arrow 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. Figure J.8 shows the values for `main`'s local variables `account1` and `withdrawalAmount` (13).




**Fig. J.8** | Examining the variables that are currently in scope.

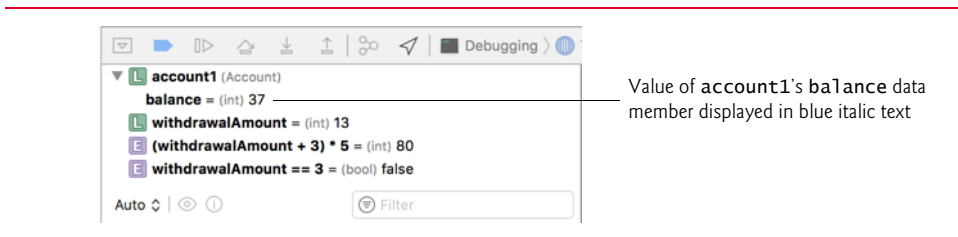
4. **Evaluating arithmetic and Boolean expressions.** You can watch the values of specific variables, arithmetic expressions and Boolean expressions using the **Auto** window. To do so, right click in the **Auto** window and select **Add Expression....** In the small window that appears, type the variable or expression to watch. For example, enter the expression `(withdrawalAmount + 3) * 5`, then press *Return*. The expression's type and value (80 in this case) are displayed to the right of the expression (Fig. J.9). Repeat this to add the expression `withdrawalAmount == 3`, then press *Return*. Ex-

pressions 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. J.9), because `withdrawalAmount` currently contains 13, not 3. The  icon indicates that a line in the **Auto** window represents an expression that you are watching. When configuring a variable or expression to watch, if you check the **Show in All Stack Frames** checkbox, the expression will be displayed in the **Auto** window *throughout* the debugging process; otherwise, the expression will be displayed only in the *scope* where you created the expression. You can delete a watched expression by right clicking it and selecting **Delete Expression**.




**Fig. J.9** | Examining the values of expressions.

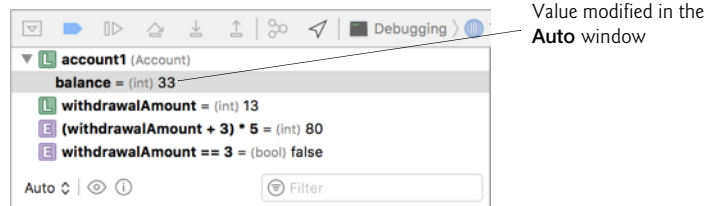
5. **Resuming execution.** Click the **Continue program execution** () button to resume execution. Line 20 withdraws `withdrawalAmount`, and the debugger reenters break mode at line 23. The updated `balance` is now displayed (Fig. J.10).



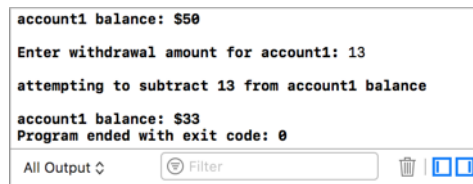
**Fig. J.10** | Displaying the value of local variables.

6. **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 **Auto** 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 **Auto** window, expand the `account1` node, select `balance` then click its value (37) to make it editable. Type 33, then press *Return*. The debugger changes the value of `balance` (Fig. J.11).
7. **Viewing the program result.** Click the **Continue program execution** () button to resume execution. The program displays the result. Notice that the result is \$33 (Fig. J.12). This shows that *Step 7* changed the value of `balance` from the calculated value (37) to 33.





**Fig. J.11** | Modifying the value of a variable.



**Fig. J.12** | Output displayed after modifying the account1 variable.

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.

## J.4 Controlling Execution Using the Step Into, Step Over and Step Out 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. **Removing a breakpoint.** Remove the breakpoint at line 23 from Section J.3.
2. **Starting to debug.** Click the **Run** (▶) button on the workspace window's tool bar to build the program and begin the debugging process. Type 13 at the **Enter withdrawal amount for account1:** prompt and press *Return* so that your program reads the value you just entered. The program executes until the breakpoint at line 20.
3. **Using the Step Into command.** The **Step Into** command executes the next statement in the program (line 20), 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. Click the **Step Into** (⏴) button in the **Debug** area's toolbar to enter the `withdraw` function. The green arrow indicating the next statement to execute is positioned at line 28 of `Account.cpp`.
4. **Using the Step Over command.** Click the **Step Over** (⏵) button to execute the current statement (line 28) and transfer control to line 29. The **Step Over** com-

**mand** 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 8*.

5. *Using the Step Out command.* Click the **Step Out** (↶) button to execute the remaining statements in the function and return control to calling function (line 20 in Fig. J.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. *Completing the program's execution.* Click the **Continue program execution** (▶▶) button to complete the program's execution.
7. *Starting the debugger.* Start the debugger again, as you did in *Step 2*, and enter 13 in response to the prompt. The debugger enters break mode at line 20.
8. *Using the Step Over command.* Click the **Step Over** (↷) button. 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 green arrow advances to the next executable line (after the function call) in the current function. In this case, the debugger executes line 20, located in `main` (Fig. J.3). Line 20 calls the `withdraw` function. The debugger then pauses execution at line 23, the next executable line in the current function, `main`.
9. *Stopping the debugger.* Click the **Stop** button on the Xcode toolbar.

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.

## J.5 Wrap-Up

In this appendix, you learned how to insert, disable and remove breakpoints in the Xcode 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 **Auto** window 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 program execution** that can be used to determine whether a function is executing correctly.