線上章節

I

And so shall I catch the fly.
—William Shakespeare
We are built to make mistakes, coded for error.
—Lewis Thomas
What we anticipate seldom occurs; what we least expect generally happens.
—Benjamin Disraeli

學習目標

在本章中，你將學到：

- 使用 `run` 指令，在偵錯器內執行程式。
- 使用 `break` 指令，設定中斷點。
- 使用 `continue` 指令，繼續執行程式。
- 使用 `print` 指令，求運算式的值。
- 使用 `set` 指令，在程式執行時修改變數的值。
- 使用 `step`、`finish`、`next` 指令，控制程式執行。
- 使用 `watch` 指令，在程式執行時觀察一個資料成員如何被修改。
- 使用 `delete` 指令，取消中斷點或監看點。

本章綱要

摘要｜術語｜自我測驗｜自我測驗解答

I.1　簡介

你在第 2 章學到兩種錯誤：編譯錯誤和邏輯錯誤，也學到如何從你的程式碼排除編譯錯誤。邏輯錯誤並不會讓程式無法編譯，可是在執行時會讓程式產生錯誤的結果。GNU 有稱爲**偵錯器 (debugger)** 的軟體，讓你可以觀察程式的執行，藉以找到邏輯錯誤並排除。

偵錯器將是你最重要的程式開發工具之一。很多整合開發環境提供自己的偵錯器，功能和 GNU 內附的類似，或是提供 GNU 偵錯器的圖形介面。這篇附錄示範 GNU 偵錯器的主要功能，附錄 H 會討論 Visual Studio 偵錯器的特性和功能。我們的 C++資源中心 (`www.deitel.com/cplusplus/`) 提供一些教學網站的連結，幫助學生學生和老師熟悉其他開發工具提供的偵錯器。

I.2　中斷點和 `run`、`stop`、`continue`、`print` 命令

我們從研究**中斷點 (breakpoint)** 開始學習偵錯器，這是可以設定在任一行執行碼的標記。當程式執行時遇到中斷點會暫停，讓你可以檢查變數的值，有助於判斷是否有邏輯錯誤。例如，你可以檢查儲存計算結果的變數的值，判斷計算是否執行正確。請注意，在不能執行一行程式碼 (例如註解) 設定中斷點的話，實際上的中斷點是設定在這個函式下一行可執行的程式碼。

爲了示範偵錯器的功能，我們使用 `Account` 類別 (圖 I.1-I.2) 以及圖 I.3 的程式建立並操作 `Account` 類別的物件。從 `main` (圖 I.3 第 12-30 行) 開始執行，第 10 行建立一個 `Account` 物件，初始餘額是$50.00。`Account` 的建構子 (圖 I.2，第 8-20 行) 接收一個引數，指定 `Account` 的初始 `balance`。圖 I.3 的第 13 行使用 `Account` 的成員函式 `getBalance` 印出初始的帳戶餘額。第 15 行宣告一個區域變數

`withdrawalAmount`，儲存從使用者讀取的提款金額。第 17 行提示使用者輸入提款金額，第 18 行把金額輸入 `withdrawalAmount`。第 21 行使用 `Account` 的成員函式 `debit`，從 `balance` 扣除提款金額。最後，第 24 顯示新的 `balance`。

```
1  // Fig. I.1: Account.h
2  // Definition of Account class.
3  class Account
4  {
5  public:
6     Account( int ); // constructor initializes balance
7     void credit( int ); // add an amount to the account balance
8     void debit( int ); // subtract an amount from the account balance
9     int getBalance(); // return the account balance
10 private:
11    int balance; // data member that stores the balance
12 }; // end class Account
```

圖 I.1　　Account 類別的標頭檔

```
1  // Fig. I.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  {
10    balance = 0; // assume that the balance begins at 0
11
12    // if initialBalance is greater than 0, set this value as the
13    // balance of the Account; otherwise, balance remains 0
14    if ( initialBalance > 0 )
15       balance = initialBalance;
16
17    // if initialBalance is negative, print error message
18    if ( initialBalance < 0 )
19       cout << "Error: Initial balance cannot be negative.\n" << endl;
20 } // end Account constructor
21
22 // credit (add) an amount to the account balance
23 void Account::credit( int amount )
24 {
25    balance = balance + amount; // add amount to balance
26 } // end function credit
27
28 // debit (subtract) an amount from the account balance
29 void Account::debit( int amount )
30 {
```

圖 I.2　　Account 類別的定義

```
31      if ( amount <= balance ) // debit amount does not exceed balance
32         balance = balance - amount;
33      else // debit amount exceeds balance
34         cout << "Debit amount exceeded account balance.\n" << endl;
35   } // end function debit
36
37   // return the account balance
38   int Account::getBalance()
39   {
40      return balance; // gives the value of balance to the calling function
41   } // end function getBalance
```

圖 I.2　Account 類別的定義 (續)

```
1   // Fig. I.3: figI_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   {
10     Account account1( 50 ); // create Account object
11
12     // display initial balance of each object
13     cout << "account1 balance: $" << account1.getBalance() << endl;
14
15     int withdrawalAmount; // stores withdrawal amount read from user
16
17     cout << "\nEnter withdrawal amount for account1: "; // prompt
18     cin >> withdrawalAmount; // obtain user input
19     cout << "\nattempting to subtract " << withdrawalAmount
20        << " from account1 balance\n\n";
21     account1.debit( withdrawalAmount ); // try to subtract from account1
22
23     // display balances
24     cout << "account1 balance: $" << account1.getBalance() << endl;
25  } // end main
```

圖 I.3　要偵錯的測試類別

依照下列步驟，你可以使用中斷點和各種偵錯器指令，檢視圖 H.3 第 15 行宣告的變數 `withdrawalAmount` 的值。

1. **編譯偵錯用的應用程式。**想要使用偵錯器時，你必須用 **-g** 選項編譯你的程式，這個選項產生的資訊可供偵錯器在偵錯時使用。請輸入：

```
g++ -g -o figI_03 figI_03.cpp Account.cpp
```

2. **啓動偵錯器**鍵入 gdb figI_03 (圖 I.4)。**gdb 指令**會啓動 GNU 偵錯器，並且顯示提示符號(gdb)，你可以在此輸入指令。
3. **在偵錯器內執行應用程式。**輸入 **run**，可以透過偵錯器執行程式 (圖 I.5)。如果在偵錯器執行你的程式之前，沒有設定任何中斷點的話，程式會執行到結束。
4. **使用 GNU 偵錯器插入中斷點。**輸入 break 13，在 FigI_03.cpp 的第 13 行設定一個中斷點。**break 指令 (break command)** 在指定的行號設定一個中斷點。你可以視需要加任意多個中斷點。每個中斷點是依照建立的順序指定，第一個建立的中斷點稱爲 Breakpoint 1。輸入 break 21，在第 21 行設定另一個中斷點 (圖 I.6)。新的中斷點稱爲 Breakpoint 2。當程式執行時，會在任何有中斷點的那行暫停執行，程式稱爲處在**中斷模式 (break mode)**。甚至在開始偵錯之後也可以設定中斷點。[請注意：如果你沒有附行號的程式碼，可以使用 **list 指令**輸出附有行號的程式碼。可以在 gdb 提示符號之後輸入 **help** list，得到更多關於 list 指令的資訊。]

```
$ gdb FigI_03
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db
library "/ lib/tls/i686/cmov/libthread_db.so.1".

(gdb)
```

圖 I.4　　啓動偵錯器執行程式

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

圖 I.5　　執行沒有設定中斷點的程式

```
(gdb) break 13
Breakpoint 1 at 0x80486f6: file FigI_03.cpp, line 13.
(gdb) break 21
Breakpoint 2 at 0x8048799: file FigI_03.cpp, line 21.
(gdb)
```

圖 I.6　在程式設定二個中斷點

5. **執行程式，開始偵錯。**輸入 `run` 執行你的程式，開始偵錯 (圖 I.7)。當執行到第 13 行的中斷點，程式會進入中斷模式。此時偵錯器會通知你，已經遇到一個中斷點，並且顯示第 13 行的原始碼，這是下一行要執行的程式。

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03

Breakpoint 1, main () at FigI_03.cpp:13
13          cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

圖 I.7　執行程式直到第一個中斷點

6. **使用 `Continue` 指令繼續執行。**輸入 `continue`。**`continue`** 指令會讓程式繼續執行，直到下一個中斷點爲止 (第 21 行)。在提示符號輸入 13。當執行到第二個中斷點時，偵錯器會通知你 (圖 I.8)。請注意，`figI_03` 的正常輸出會夾雜偵錯器的訊息。

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 2, main () at FigI_03.cpp:21
21          account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

圖 I.8　繼續執行直到第二個中斷點

7. **檢視變數的值。**輸入 `print withdrawalAmount`，會印出目前存在 `withdrawalAmount` 變數的值 (圖 I.9)。**`print` 指令**可以讓你偷看電腦內你的變

數的值。可以幫助你找到程式碼的邏輯錯誤並排除。請注意顯示的值是 13，這個值在圖 I.3 第 18 行讀取並設定給 `withdrawalAmount`。使用 `print` 指令輸出 `account1` 物件的內容。透過偵錯器的 `print` 指令輸出一個物件時，物件的資料成員在印出時會加上大括號。此例只有一個資料成員 `balance`，其值爲 50。

8. **使用方便變數。**使用 `print` 指令時，結果會存在一個方便變數，例如$1。方便變數是偵錯器所產生的暫時變數，名稱是錢字號加數字。方便變數可以用來執行算術運算，或是求布林運算式的值。輸入 `print $1`，偵錯器會顯示$1 的值 (圖 I.10)，這個變數包含 `withdrawalAmount` 的值。請注意印出$1 的值會建立一個新的方便變數$3。

```
(gdb) print withdrawalAmount
$2 = 13
(gdb) print account1
$3 = {balance = 50}
(gdb)
```

圖 I.9　印出變數的值

```
(gdb) print $1
$3 = 13
(gdb)
```

圖 I.10　印出一個方便變數

9. **繼續執行程式。**輸入 `continue`，繼續程式執行，偵錯器沒有再遇到其他的中斷點，所以會繼續執行直到結束 (圖 I.11)。

```
(gdb) continue
Continuing.
account1 balance: $37

Program exited normally.
(gdb)
```

圖 I.11　結束程式執行

10. **取消中斷點。**你可以輸入 **`info break`**，會列出程式中所有的中斷點。想要取消一個中斷點，要輸入 **`delete`**，加上一個空白和要取消的中斷點編號。輸入 `delete 1` 取消第一個中斷點，然後也照樣取消第二個中斷點。現在輸入 `info break`，會列出程式中剩餘的中斷點。偵錯器應該會指出沒有設定中斷點 (圖 I.12)。

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080486f6 in main at FigI_03.cpp:13
        breakpoint already hit 1 time
2   breakpoint     keep y   0x08048799 in main at FigI_03.cpp:21
        breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

圖 I.12　觀察和取消中斷點

11. **執行沒有中斷點的程式。**輸入 `run` 執行程式。在提示符號輸入數值 13。因爲你成功地取消兩個中斷點，所以會顯示程式的結果，而且偵錯器不會進入中斷模式(圖 I.13)。

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

圖 I.13　執行沒有設定中斷點的程式

12. **使用 `quit` 指令。**使用 **`quit`** 指令結束偵錯流程 (圖 I.14)。這個指令會讓偵錯器終止。

```
(gdb) quit
$
```

圖 I.14　使用 `quit` 指令離開偵錯器

你在這一節學到如何使用 `gdb` 指令啓動偵錯器，以及如何使用 `run` 指令開始偵錯程式。你在 `main` 函式的特定行號設定中斷點。`break` 指令也可以在另一個檔案或是特定的函式設定中斷點。輸入 `break` 以及檔名、冒號、行號，會在另一個檔案的那行設定中斷點。輸入 `break` 以及函式名稱，會讓偵錯器在這個函式呼叫時進入中斷模式。

你也在這一節看到使用 `help list` 指令，可以列出更多關於 `list` 指令的資訊。如果你有更何關於偵錯器或是其指令的問題，可以輸入 `help`，或是 `help` 之後加上指令名稱，可以得到更多資訊。

最後，你學到如何使用 `print` 指令檢查變數，以及使用 `delete` 指令取消中斷點。你學到如何在遇到中斷點之後，使用 `continue` 指令繼續執行。還有如何使用 `quit` 指令結束偵錯器。

I.3 `print` 和 `set` 指令

你在前一節學到如何使用偵錯器的 `print` 指令，在程式執行時檢視變數的值。在這一節，你會學到如何使用 `print` 指令，檢視更複雜的運算式。你也會學到 **`set` 指令**，這個命令讓程式設計師指派新的值給變數。我們假設你位在附錄範例的目錄，而且已經使用 `-g` 選項編譯。

1. **開始偵錯。**輸入 `gdb figI_03`，啓動 GNU 偵錯器。
2. **插入中斷點。**輸入 `break 21`，在原始碼的第 21 行設定一個中斷點 (圖 I.15)。

```
(gdb) break 21
Breakpoint 1 at 0x8048799: file FigI_03.cpp, line 21.
(gdb)
```

圖 I.15　在程式設定一個中斷點

3. **執行程式到達中斷點。**輸入 `run`，開始偵錯流程 (圖 I.16)。讓 `main` 開始執行，直到第 21 行的中斷點，程式暫停執行，並且切換成中斷模式。第 25 行的敘述式是下一個要執行的敘述式。

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 1, main () at FigI_03.cpp:21
21          account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

圖 I.16　執行程式直到第 25 行的中斷點

17. **求算術和布林運算式的值。**請回憶第 I.2 節，一旦程式進入中斷模式，你可以使用偵錯器的 `print` 指令，瀏覽程式中變數的值。你也可以使用 print 指令，求算術和布林運算式的值。鍵入 `print withdrawalAmount -2`。這個運算式傳回數值 11 (圖 I.17)。然而，這個指令實際上沒有改變 `withdrawalAmount` 的值。請輸入 `print withdrawalAmount == 11`。包含 `==` 符號的運算式，會回傳布林值。傳回的值是 `false` (圖 I.17)，因爲 `withdrawalAmount` 目前的值仍是數值 13。

```
(gdb) print withdrawalAmount - 2
$1 = 11
(gdb) print withdrawalAmount == 11
$2 = false
(gdb)
```

圖 I.17　使用偵錯器印出運算式

5. **修改數值。**當程式在偵錯器中執行時，你可以修改變數的值。這是很有用的，可以實驗不同的數值，找到程式的邏輯錯誤。你可以使用偵錯器的 `set` 指令，修改變數的值。鍵入 `set withdrawalAmount = 42`，改變 `withdrawalAmount` 的值，接著鍵入 `print withdrawalAmount` 顯示新值 (圖 I.18) 。

```
(gdb) set withdrawalAmount = 42
(gdb) print withdrawalAmount
$3 = 42
(gdb)
```

圖 I.18　在中斷模式設定變數的值

6. **觀察程式結果。**輸入 `continue`，繼續程式執行，接下來會執行圖 I.3 的第 21 行，把 `withdrawalAmount` 傳遞給 `Account` 的成員函式 `debit`。然後 `main` 函式會顯示新的餘額。請注意結果是 $8 (圖 I.19)。這顯示前一個步驟把 `withdrawalAmount` 的值，從你輸入的值 (13) 改成 42。

```
(gdb) continue
Continuing.
account1 balance: $8

Program exited normally.
(gdb)
```

圖 I.19　在程式執行時使用修改過的變數

7. 使用 quit 指令。使用 quit 指令結束偵錯流程 (圖 I.20)。這個指令會讓偵錯器終止。

```
(gdb) quit
$
```

圖 I.20　使用 quit 指令離開偵錯器

在這一節，你學到如何使用偵錯器的 print 指令，求算術和布林運算式的值。也學到如何使用 set 指令，在程式執行時修改變數的值。

I.4 使用 step、finish、next 指令，控制程式執行

有時候你會需要逐行執行程式，才可以找到錯誤並修正。以這種方式經歷程式的一部分，可以幫助你檢驗某個函式的程式碼執行正確。在這學到的指令讓你可以逐行執行一個函式、立刻執行一個函式的所有敘述式、或是只執行一個函式剩餘的敘述式 (如果你已經執行這個函式的一些敘述式)。

1. **啓動偵錯器**輸入 gdb figI_03，啓動偵錯器。
2. **設定中斷點。**輸入 break 21，在第 21 行設定中斷點。
3. **執行程式。**輸入 run，開始執行程式，然後在提示訊息之後輸入 13。在程式印出兩個訊息之後，偵錯器指示已經到達中斷點，並且印出第 21 行的程式碼。然後偵錯器和程式都會暫停，等待下一個命令輸入。
4. **使用 step 指令。step 指令**執行程式的下一個敘述式。如果下一個要執行的敘述式是函式呼叫，控制權會轉移到被呼叫的函式。step 指令讓你可以進入一個函式，研究這個函式的每一行敘述式。例如，你可以使用 print 和 set 指令，觀察和修改函式內的變數。輸入 step，進入 Account 類別 (圖 I.2) 的成員函式 debit。偵錯器會指示這個步驟已經完成，並且顯示下一個要執行的敘述式 (圖 I.21)，以這個例子來說是 Account 類別 (圖 I.2) 的第 31 行。

```
(gdb) step
Account::debit (this=0xbff81700, amount=13) at Account.cpp:31
31          if ( amount <= balance ) // debit amount does not exceed balance
(gdb)
```

圖 I.21　使用 step 指令進入函式

5. **使用 `finish` 指令。**在你進入 `debit` 成員函式之後，請輸入 `finish`。這個指令會執行這個函式剩餘的敘述式，然後把控制權傳回呼叫這個函式的地方。`finish` 指令會執行成員函式 `debit` 剩餘的敘述式，然後在 `main` 的第 24 行暫停 (圖 I.22)。在一些比較長的函式，你可能想要看一些關鍵的程式碼，然後繼續呼叫者的程式碼偵錯。有些情況下，你不想逐行執行整個函式，這時 `finish` 指令就很有用。

```
(gdb) finish
Run till exit from #0  Account::debit (this=0xbff81700, amount=13) at
   Account.cpp:31
0x080487a9 in main () at FigI_03.cpp:21
21         account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

圖 I.22　使用 `finish` 指令完成函式執行，然後返回呼叫函式

6. **使用 `continue` 指令繼續執行。**輸入 `continue` 指令繼續執行，直到程式結束。
7. **再次執行程式。**中斷點會保留到整個偵錯流程結束。因此，你在步驟 2 設定的中斷點，在下次執行程式的時候還會在。輸入 `run`，開始執行程式，然後在提示訊息之後輸入 13。就像步驟 3 一樣，程式會執行到第 21 行的中斷點爲止，然後偵錯器會暫停，等待下一個命令 (圖 I.23)。

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 1, main () at FigI_03.cpp:21
21         account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

圖 I.23　重新執行程式

8. **使用 `next` 命令。輸入 `next`**，這個命令就像 `step` 指令一樣，除非下一個要執行的敘述式是函式呼叫。在這種狀況下，被呼叫的函式會整個執行完畢，程式會前進到函式呼叫後的下一行 (圖 I.24)。在步驟 4，`step` 指令會進入被呼叫的函式。在這個範例，`next` 命令讓 `Account` 的成員函式 `debit` 執行，然後偵錯器暫停在第 24 行。

```
(gdb) next
24          cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

圖 I.24　　使用 next 指令執行整個函式。

9. **使用 quit 指令。**使用 quit 指令結束偵錯流程 (圖 I.25)。當程式執行時，這個指令會讓程式立刻結束，而不會執行 main 的其餘敘述式。

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

圖 I.25　　使用 quit 指令離開偵錯器

在這一節，你學到如何使用偵錯器的 step 和 finish 命令，在程式執行時進行呼叫函式偵錯。你看到如何使用 next 指令，經過函式呼叫。你也學到 quit 指令可以終止偵錯流程。

I.5　watch 指令

watch 指令可以要求偵錯器監視一個資料成員。當這個資料成員將要改變時，偵錯器會通知你。在這一節，你會學到如何使用 watch 指令，在程式執行時觀察 Account 物件的資料成員 balance 遭到修改。

1. **啓動偵錯器**輸入 gdb figI_03，啓動偵錯器。
2. **在程式設定一個中斷點，執行程式。**輸入 break 10，在第 10 行設定中斷點。輸入 run，開始執行程式。偵錯器和程式會暫停在第 10 行 (圖 I.26)。

```
(gdb) break 10
Breakpoint 1 at 0x80486e5: file FigI_03.cpp, line 10.
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03

Breakpoint 1, main () at FigI_03.cpp:10
10          Account account1( 50 ); // create Account object
(gdb)
```

圖 I.26　　執行程式直到第一個中斷點

3. **監視類別的資料成員。**輸入 `watch account1.balance`，在 `account1` 的資料成員 `balance` 設定監看點 (圖 I.27)。這個監看點標記成 `watchpoint 2`，因爲監看點和中斷點用同樣的編號序列。在執行偵錯器時，生存範圍內的任何變數或資料成員，你都可以設定監看點。當監看變數的值改變時，偵錯器會進入中斷模式，並且通知你這個值已經改變。

```
(gdb) watch account1.balance
Hardware watchpoint 2: account1.balance
(gdb)
```

圖 I.27　在資料成員設定監看點

4. **執行建構子。**使用 next 指令執行建構子，將 `account1` 物件的 `balance` 資料成員初始化。偵錯器現在會通知你，資料成員 `balance` 即將改變，顯示出舊值和新值，並在第 18 行進入中斷模式(圖 I.28)。

```
(gdb) next
Hardware watchpoint 2: account1.balance

Old value = 0
New value = 50
Account (this=0xbfcd6b90, initialBalance=50) at Account.cpp:18
18         if ( initialBalance < 0 )
(gdb)
```

圖 I.28　進入建構式

5. **結束建構子。**輸入 `finish`，結束建構子的執行，回到 `main`。
6. **從帳戶提款。**輸入 `continue`，繼續執行，然後在提示符號輸入提款金額，程式會正常執行。圖 I.3 的第 21 行會呼叫 `Account` 的成員函式 `debit`，從 `Account` 物件的 `balance` 扣除指定的 `amount`。debit 函式在圖 I.2 的第 32 行會改變餘額的值，偵錯器會通知你這項改變，然後進入中斷模式 (I.29)。

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Hardware watchpoint 2: account1.balance

Old value = 50
New value = 37
0x0804893b in Account::debit (this=0xbfcd6b90, amount=13) at Account.cpp:32
32            balance = balance - amount;
(gdb)
```

圖 I.29　當變數改變時會進入中斷模式

7. **繼續執行。**輸入 `continue`，因爲程式再也沒有改變 `balance`，所以會結束執行 `main` 函式。當 `main` 函式結束時，因爲 `account1` 物件離開生存範圍，所以偵錯器會取消 `account1` 資料成員 `balance` 的監看點。取消這個監看點會讓偵錯器進入中斷模式，再輸入 `continue`，完成程式執行 (圖 I.30)。

```
(gdb) continue
Continuing.
account1 balance: $37

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
0xb7da0595 in exit () from /lib/tls/i686/cmov/libc.so.6
(gdb) continue
Continuing.

Program exited normally.
(gdb)
```

圖 I.30　　繼續執行到程式結束

8. **重新啓動偵錯器，在變數重設監視點。**輸入 `run`，重新啓動偵錯器。再次輸入 `watch account1.balance`，在 `account1` 的資料成員 balance 設定監看點，這個監看點標記成 `watchpoint 3`。輸入 `continue`，繼續程式執行 (圖 I.31)。

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03

Breakpoint 1, main () at FigI_03.cpp:10
10          Account account1( 50 ); // create Account object
(gdb) watch account1.balance
Hardware watchpoint 3: account1.balance
(gdb) continue
Continuing.
Hardware watchpoint 3: account1.balance

Old value = 0
New value = 50
Account (this=0xbfd8eb90, initialBalance=50) at Account.cpp:18
18          if ( initialBalance < 0 )
(gdb)
```

圖 I.31　　重新在資料成員設定監看點

9. **移除資料成員的監看點**假設你只要在部分程式執行時，監視一個資料成員。你可以輸入 `delete 3`，取消 `balance` 變數的監看點 (圖 I.32)。輸入 `continue`，程式會執行完成，不再進入中斷模式。

```
(gdb) delete 3
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

圖 I.32　取消監看點

在這一節，你學會如何使用 `watch` 指令，在程式執行過程中，讓偵錯器通知你一個資料成員的值改變。你也學到如何使用 `delete` 指令，在程式結束前取消資料成員的監看點。

I.6　總結

在本附錄中，你學會如何在偵錯器中插入、停用、取消中斷點。中斷點可以暫停程式執行，讓你使用偵錯器的 print 指令檢視變數的值，找出邏輯錯誤並修正它們。你學會使用 `print` 指令檢視運算式的值，以及 `set` 指令改變變數的值。你也學會如何使用偵錯器指令 (`step`、`finish` 和 `next` 指令) 判斷函式是否正確執行。你學會如何利用 `watch` 指令，在資料成員的使用域追蹤它們的值。最後你學到了如何使用 `info break` 指令，列出所有的中斷點和監看點，以及 `delete` 指令，移除個別的中斷點和監看點。

摘要

I.1　簡介

- GNU 有稱爲偵錯器的軟體，讓你可以觀察程式的執行，藉以找到邏輯錯誤並排除。

I.2　中斷點和 `run`、`stop`、`continue`、`print` 命令

- GNU 偵錯器只可以處理用 `-g` 選項編譯的執行檔，這個選項產生的資訊可供偵錯器在偵錯時使用。
- `gdb` 指令會啓動 GNU 偵錯器，讓你可以使用他的功能。`run` 指令會透過偵錯器執行程式。

- 中斷點是一個標示，可以設在任一行可執行的程式碼。當程式執行時遇到中斷點，就會暫停執行。
- `break` 指令在指定的行號設定一個中斷點。
- 當程式執行時，會在任何有中斷點的那行暫停執行，程式稱爲處在中斷模式。
- `continue` 指令會讓程式繼續執行，直到下一個中斷點爲止。
- `print` 指令可以讓你偷看電腦內你的變數的值。
- 使用 `print` 指令時，結果會存在一個方便變數，例如$1。方便變數是暫存變數，可以在偵錯過程中使用，拿來執行算術運算，或是求布林運算式的值。
- 你可以輸入 `info break`，會列出程式中所有的中斷點。
- 想要取消一個中斷點，要輸入 `delete`，加上一個空白和要取消的中斷點編號。

I.3　`print` 和 `set` 指令

- 使用 `quit` 指令結束偵錯流程。
- `set` 指令讓程式設計師可以把一個變數設定新的值。

I.4　使用 `step`、`finish`、`next` 指令，控制程式執行

- `step` 指令執行程式的下一個敘述式。如果下一個要執行的敘述式是函式呼叫，控制權會轉移到被呼叫的函式。`step` 指令讓你可以進入一個函式，研究這個函式的每一行敘述式。
- `finish` 指令會執行這個函式剩餘的敘述式，然後把控制權傳回呼叫這個函式的地方。
- `next` 命令就像 `step` 指令一樣，除非下一個要執行的敘述式是函式呼叫。在這種狀況下，被呼叫的函式會整個執行完畢，程式會前進到函式呼叫後的下一行。

I.5　`watch` 指令

- 在執行偵錯器時，生存範圍內的任何變數或資料成員，你都可以用 `watch` 指令設定監看點。當監看變數的值改變時，偵錯器會進入中斷模式，並且通知你這個值已經改變。

術語

`break` 指令 (break command)
中斷模式 (break mode)
中斷點 (breakpoint)
`continue` 指令 (continue command)
偵錯器 (debugger)
`delete` 指令 (delete command)
`finish` 指令 (finish command)
`-g` 編譯器選項 (-g compiler option)
`gdb` 指令 (gdb command)
`help` 指令 (`help` command)

`info break` 指令 (info break command)
`list` 指令 (`list` command)
`next` 指令 (next command)
`print` 指令 (print command)
`quit` 指令 (quit command)
`run` 指令 (run command)
`set` 指令 (set command)
`step` 指令 (step command)
`watch` 指令 (watch command)

自我測驗習題

I.1 填充題：

a) 中斷點不能設定在_____。
b) 你可以用偵錯器的_____視窗，檢查運算式的值。
c) 你可以用偵錯器的_____指令，修改變數的值。
d) 在偵錯過程中，_____指令會執行目前函式剩餘的敘述式，並且傳回控制權到呼叫這個函式的地方。
e) 如果下一個要執行的敘述式沒有函式呼叫的話，偵錯器的_____指令就像 `step` 指令一樣。
f) 偵錯器的 `watch` 指令，讓你可以監看一個_____的所有變化。

I.2 說明下列何者爲對，何者爲錯。如是錯的，請解釋爲什麼。

a) 當程式到達中斷點而暫停執行時，下一個要執行的敘述式就是中斷點後的這一個。
b) 可以用偵錯器的 `remove` 指令取消監看點。
c) 當編譯偵錯使用的程式時，必須用 `-g` 編譯器選項。

自我測驗解答

I.1 a) 不能執行的程式碼。 b) `print`。 c) set。 d) `finish`。 e) `next`。 f) 資料成員。

I.2 a) 錯。當程式到達中斷點而暫停執行時，下一個要執行的敘述式是中斷點所在的這一個。
b) 錯。可以用偵錯器的 `delete` 指令取消監看點。c) 對。