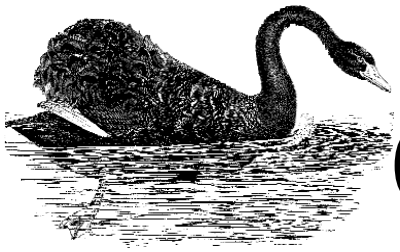
第六章



C 及 C++ 程式除錯器

本章提要

- gdb 的編譯
- 啟動 gdb
- gdb 的基本命令
- 變數的範圍及本文
- 在呼叫堆疊中上下移動
- C++ 程式
- Emacs 介面
- 命令補正及縮寫
- 附加到現有行程
- 快速參考

本章介紹 C 及 C++ 程式除錯程序一個強有力的工具— gdb。就某些方面來說，gdb 類似 BSD UNIX 上之原始程式碼除錯器 dbx。但就像其它典型的 FSF 工具一樣，gdb 功能更多，包括 C++ 原始程式的除錯，及執行中行程的除錯。（System V 的除錯器 sdb 已經年代久遠，gdb 及 dbx 的功能遠超過它。）

gdb 可讓你執行程式，在程式中設定中斷點，在執行過程中檢視並修改變數，呼叫函式，並追蹤程式的執行過程。它也有類似 bash（GNU shell）及 emacs 中命令列編輯及命令歷史等功能。我們先以最簡單的術語描述 gdb 的功能及命令。剛開始會盡可能透過簡單的範例，然後再深入困難的部份。

和其它除錯器一樣，有些執行中斷點及追蹤的選項設定，會使你程式的執行極為緩慢，慢到幾乎無法忍受而失去實用價值。但這些選項往往是最有用處的。不要沮喪！有些旁門左道雖會限制這些選項設定的彈性，卻能給予可以接受（甚至於驚人）的執行效果。我們會指出那些除錯功能有這類慢如牛步的傾向，並告訴你如何避開這些陷阱。

要得到所有 gdb 命令的完整文件，可閱讀 GDB 手冊或向 FSF 訂購。

gdb 的編譯

要使用 gdb 進行程式除錯之前，先以 -g 選項編譯你的程式。這會使編譯器產生額外的符號表。例如下列命令：

```
% gcc -g file1.c file2.c file3.o
```

編譯 C 原始程式 file1.c 及 file2.c，會產生一擴充之符號表以供 gdb 使用。這些檔案與 file3.o 連結，這是一個編譯過的物件檔。則 gdb 可用於對 file1.c 及 file2.c 之原始碼除錯，但不能對 file3.c 的原始碼除錯，除非它也是以 -g 選項編譯的。

編譯器的 -g 及 -O 選項並無不相容的問題；你可同時兼顧編譯最佳化與除錯能力。不像其它許多除錯器，gdb 產生的結果甚至更精密，但經過最佳化後的程式，除錯會比較困難；在本質上，最佳化的過程會對程式本身作出一些改變，原始程式中指令間的關係在可執行程式中已經改變。你也許會發現，某些變數（甚至數行程式指令）會因最佳化而消失，有些指定指令的動作也沒在你預期的地方執行。所以在進行最佳化之前，最好盡可能徹底的先進行除錯。完全依照 ANSI C 標準的程式也最容易除錯；不恪遵標準的程式可能比較困難。在某些系統上對這類程式進行最佳化，甚至會暴露一些原本並不明顯的錯誤。

啟動 gdb

對編譯過的程式進行除錯，使用下列命令：

```
% gdb program [ core-dump ]
```

porgram 是你要除錯的執行檔名稱，core-dump 則是上次執行你的程式所留下的 core-dump 檔名稱。用 gdb 檢視 core-dump，你可以發現程式當在甚麼地方，以及為何會當掉。例如下面的命令執行 gdb，參數為 qsort2 檔案及 core.2957 core-dump：

```
% gdb qsort2 core.2957
GDB is free software and you are welcome to 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.
GDB 4.15.2-96q3 (sparc-sun-solaris2),
Copyright 1996 Free Software Foundation, Inc.
Core was generated by ‘qsort2’.
Program terminated with signal 7, Emulator trap.
#0 0x2734 in qsort2 (l=93643, u=93864, start=1) at qsort2.c:118
118   do i++; while(i <= u && x[i] < t);
(gdb) quit
```

冗長的開場白告訴你 gdb 的版本（註）。然後告訴你 core 檔是如何產生的（因為 qsort2 程式在執行時收到 7 號訊號，模擬器中斷陷阱），程式當掉時正在做甚麼（執行到 118 行）。提示符號（gdb）告訴你可以輸入命令了。本例中，我們直接離開。

註　　依照預設，gdb 啟動時會顯示冗長的“free software”訊息。現在你可以在啟動時省略這些訊息。只要使用命令 gdb -q 即可。

執行檔及 core 檔的參數都不是必要的。你可以稍後再用 core 命令叫出 core 檔。任何一種狀況下，只要你輸入的檔案名稱不存在，或檔案名稱的格式不對，gdb 會顯示如下的警告訊息：

```
% gdb nosuchfile
"/home/los/mikel/cuser/nosuchfile": No such file or directory
(gdb)
```

你可在 gdb 啟動時附加一些參數，下列是最常用的一些：

-d dir

告訴 gdb 去 dir 目錄找原始程式檔。稍後會告訴你用處何在。

-x file

在接受任何命令之前，gdb 會讀入並執行 file 中的命令，讓你可設定 gdb 執行一些你每次都會做的指令，如在某一指令設中斷點（註）。 你可以使用多個 -x 參數，不同的檔案會依序執行。

-nx

不要執行初始環境設定檔（通常是 .gdbinit）中的命令。

-q

不要在啟動時顯示開場白及版權訊息。

-help

顯示所有命令選項之後結束。

註　　你也可以在你的 home 目錄中的 .gdbinit 檔案裡放入啟動命令的方式，當作gdb的初始設定。

-batch

以批次模式執行；先執行任何初始環境設定檔（除非你要求它不做），然後執行 -x 選項指定的檔案，結束執行並傳回狀態 0。

gdb 的基本命令

只要用到少數幾個 gdb 命令，你就可以完成大部分的工作。基本工作包括檢視原始程式、設定中斷點、執行程式、及檢查變數。

假如你忘記某些命令（或要查查不常用的特殊功能），可用內建之輔助說明工具。你可查詢特定命令（如 help print）或指定下列主題之一，gdb 會顯示相關命令：

help aliases

help breakpoints

help data

help files

help internals

help obscure

help running

help stack

help status

help support

help user

列示檔案

要觀看原始程式，使用命令 list：

```
% gdb qsort2
(gdb) list
13   void qsort2();
14   void swap();
15   void gen_amd_sort();
16   void init_organ();
17   void init_random();
18   void print_array();
19
20   main()
21   {
22           int power=1;
```

若你只打 list 而未加任何參數，gdb 會從剛執行到的指令開始顯示原始程式。因我們尚未執行程式，gdb 顯示 main 函式的進入點。左邊的數字為原始程式之行號；你之後會在許多除錯命令中用到這些行號。gdb 將所有程式列加上編號，包括註解及空白行。

要看你正除錯檔案中的特定幾行，使用下列這種 list 命令：

```
(gdb) list line1,line2
```

要看某函式的前面 10 行，使用下列的 list 命令：

```
(gdb) list routine-name
```

若你重複執行 list 命令而不加任何參數，gdb 會顯示接下來幾行，每次執行皆會如此，直到檔案結尾。原則上，gdb 每次 list 命令會顯示 10 行。本章之後會說明如何改變這種設定。

gdb 無法分辨條件編譯（#ifdef 指令）及表頭檔（#include 指令）所造成的程式流程改變。你會在程式列示中看到這些指令，你必須知道它們如何作用。特別是程式編譯的結果，那些指令被含入，及那些會跳過去。

若 gdb 無法找到指定的原始程式檔案，會顯示下列訊息：

```
Function name not defined
```

訊息中，gdb 說它無法找到包含 name 函式的檔案。

執行程式

run 命令用來執行你要除錯的程式。程式的參數，包括標準輸出入符號 < 及 >，及 shell 萬用字元（*、?、[、及]），也要接著輸入。C shell 歷史（!）及管道（|）命令在此無法接受。你執行的程式會有一完整的 shell 環境（如環境變數 SHELL 所定義）；你也可以用 set 及 unset 環境變數的命令，定義或刪除其它的環境變數。

如下例，經由 gdb 去執行 exp 程式。下列命令執行 exp，參數為 -b，將 invalues 當成標準輸入，並將標準輸出重導至 outtable 檔：

```
% gdb exp
(gdb) run -b < invalues > outtable
```

若無設定任何中斷點或其它除錯功能，exp 將執行到程式結束，不管結果正確與否。執行完畢後，gdb 顯示下列訊息：

```
Program exited with code n.
(gdb)
```

n 是程式傳回的結束值。此時提示符號（gdb）出現，等待輸入下一命令。

若你的 run 命令沒有參數，gdb 重新利用上次 run 命令的參數。這用處蠻大的，因為你在除錯程式時常會重複執行程式。你可用 set args 命令改變傳程式的參數；也可用 show args 命令看看目前的參數列。如下例：

```
(gdb) set args -b -x
(gdb) show args
Arguments to give program being debugged when it is started is "-b -x".
(gdb)
```

要中止正在 gdb 控制下執行的程式，按 Ctrl-C 即可，將控制權交回 gdb，並出現提示符號等待新命令；它並不會回到 shell。要離開 gdb，就輸入 quit：

```
(gdb) run
CTRL-C
Program received sinal SIGINT, Interrupt.
Qsort2 (l=95136, u=97479, start=1) at qsort2.c:119
119  do j--; while (x[j] > t);
(gdb) quit
The program is running. Quit anyway (and kill it)? (y or n) y
```

若你正除錯的程式是"可執行的"（即它正停在中斷點，因 Ctrl-Z 而暫停，或停在其它任何狀態，而你可繼續執行），gdb 會要求你確認是否要離開。

如果你正在除錯的程式不正常的結束，控制權會交回 gdb，這時可用 gdb 命令找出程式為何結束。下例顯示一個程式因存取程式堆疊範圍以外的資料，而導致記憶區段蓋寫（segmentation violation）而中止。backtrace 命令會回溯堆疊內容，以顯示程式在當掉時的行為：

```
% gdb badref
(gdb) run
Starting program: /home/los/mikel/cuser/badref

0x22c8 in march_to_infinity () at badref.c:16
16    h |= *p;

(gdb) backtrace
```

```
#0 0x22c8 in march_to _infinity () at badref.c:16
#1 0x2324 in setup () at badref.c:25
#2 0x2340 in main () at badref.c:30
(gdb)
```

backtrace 列出所有正在執行的行程及其參數，從最後被呼叫的行程開始。它顯示了程式當在 march_to_infinity 函式，此函式是由 setup 函式呼叫，而它本身又為 main 函式所呼叫。剩下的工作，是要找出 march_to_infinity 到底出了甚麼漏子，這正是我們下一節的主題。

列示資料

print 命令可讓你檢查變數的內容。我們看前面的程式到底發生了甚麼事。首先把程式列印出來：

```
(gdb) list
8
9     p=&j;
10    /* march off the end of the world*/
11    for (i = 0; i < VERYBIG; i++)
12    {
13            h |= *p;
14            p++;
15    }
16    printf("h: %d\n",h);
```

看起來很清楚了。p 是某種指標；我們可用 whatis 命令測試它，顯示它的宣告：

```
(gdb) whatis p
type = int *
(gdb) print p
$1 = (int *) 0xf8000000
```

```
(gdb) print *p
$2 = Cannot access memory at address 0xf8000000.
(gdb) print h
$3 = -1
(gdb)
```

當我們檢視 p 時，可以看到它指到天邊去了。當然，沒有任何特定的方式可以確定 p 的這種值是否合法。但我們可以看是否能讀到 p 指到的資料，就如我們的程式嚐試去做的。而當我們下達命令 print *p，我們看到它指到無法存取的資料。

print 是 gdb 強大的功能之一。你可用來觀看任何除錯中運算式的結果，只要運算式合乎程式語法。除了程式中的變數以外，運算式可以包括：

* 呼叫程式中的任何函式；這些函式也許有副作用（side effect）。（它們可能會修改一些全域變數，你會在後續的執行中發現）。

  ```
  (gdb) print find_entry(1.0)
  $1 = 3
  ```

* 資料結構及其它複雜的物件

  ```
  (gdb) print *table_start
  $8 = {e_reference = '\000' <repeat 79 times>, location = 0x0, next = 0x0}
  ```

* “歷史值”元素（本節後面會提到）

* “假陣列”（本節後面會提到）

在我們列印指標的範例中，兩次 print 命令的結果分別是 $1、$2、$3，它們是“歷史值”識別字；這表示你可在未來的運算式中用 $1 代表 0xf8000000，$3 代表常數 -1。（$2 代表字串常數“Cannot access…”，對我們用處不大）

“歷史值”的用處何在？舉例來說，你看不出為何位址 0xf8000000 是錯誤的；因此必須針對這個數字再做些計算。若你想將此整數指標下移一位，看看是否能成為有用的指標，可以重新輸入此值，那樣你必須打許多 0，還要檢查有沒有打錯。或是你可利用“歷史值”，保證用到的是 gdb 曾經印出來的位址。下面就是它的作法：

```
(gdb) print $1-1
$4 = (int *) 0xf7fffffc
(gdb) print *$4
$5 = 0
```

首先我們將 $1 減一，只是試試看結果如何。結果可能出乎你意料，但應該不會：因為 $1 本來是個整數指標，減 1 表示減去一個整數大小之位址數，即四個位元組。gdb 通常設法以 C 語言的方式執行。

當我們印出新位址的內容，結果是 0，而非存取錯誤。這種作法讓我們可以進行測試而不需重打任何資料，這樣比較簡單，而且避免錯誤發生。

人工陣列（Artificial Array）提供一個列印一塊記憶體（部分陣列，或動態配置的一塊記憶體）的方式。早期的除錯器無法將任何指標轉成陣列；例如，dbx可印出 p 指標及後續位址的內容，但必須一次一個，無法整批印出。所以要有動態陣列。

我們可以示範一下，看看變數 h 所在記憶體後面的十個整數。動態陣列的語法為：

```
基礎位址 @ 長度
```

要印出 h 後之十個元素，命令為 h@10：

```
(gdb) print h@10
$13 = {-1, 349, 0, 0, 0, 0, 536903697, 32831, 1, -131080192}
(gdb)
```

注意，這個人工陣列整個又變成一歷史值了；如果你要參考它的第七個元素，可用下列方式：

```
(gdb) print $13[6]
$14 = 536903697
```

（請記住，在 C 語言中陣列是從 0 開始）或要再看接下來十個元素，我們會這樣做：

```
(gdb) print $13[10]@10
%15 = {-131876248, 0, 65545, -131080268, 57344, 64, 2, 0, -134219000, 9020}
(gdb)
```

若要依垂直排列的方式印出，可以用下列命令：

```
(gdb) set print array
```

以上是一些在非常狀況下有用的 set 命令；你可從 gdb 文件中找到更多其它命令。

中斷點（Breakpoints）

中斷點可讓你在程式執行中暫時停止。當程式停在中斷點時，你可檢查或修改變數，呼叫函式或執行其它任何 gdb 命令。它可以讓你檢查程式當時執行的狀態。之後你可以繼續往下執行。

break 命令（可縮寫成 b）在程式中設定中斷點。它有下列四種方式：

break line-number

在程式執行到指定行前停止。

break function-name

在程式執行到進入指定函式前停止。

break routine-name

在特定程序的入口設中斷點。程式執行時，gdb 暫停在指定函式的第一行可執行指令前。

break line-or-function if condition

若條件符合，在程式執行到進入指定行號或函式前停止。

以最後一種形式為例，下面的 break 命令在 init_random 函式的入口處設一中斷點。然後 run 命令會執行到函式開頭為止。停在 init_random 中第一行可執行指令上，在本例中為原始程式第 155 行之 for 迴圈：

```
% gdb qsort2
(gdb) break init_random
Breakpoint 1 at 0x28bc: file qsort2.c, line 155.
(gdb) run
Starting program: /home/los/mikel/cuser/qsort2
Tests with RANDOM inputs and FIXED pivot

Breakpoint 1, init_random (number=10) at qsort2.c:155
155  for (i= 0; i < number; i++) {
(gdb)
```

中斷點設定之後，gdb 會給予一獨一無二之識別號碼（本例中為 1），印出一些重要的相關資訊。每當執行到中斷點，gdb 印出中斷點的識別號碼、相關描述及目前行號。若你已設定許多中斷點，識別號碼可讓你知道程式停在那個中斷點，然後告訴你程式停在哪一行。

要讓程式停在某一行，可用 break line-number 命令。例如下面的 break 命令，將中斷點設定在程式的第 155 行：

```
(gdb) break 155
Note: breakpoint 1 also set at pc 0x28bc
Breakpoint 2 at 0x28bc: file qsort2.c, line 155.
(gdb)
```

因為這個中斷點和前一個設定在同一個地方，我們接到警告，但 gdb 還是設定中斷點。兩個中斷點設定在同一地方不會有甚麼後遺症。事實上還有些好處，例如你可能想依不同執行狀況設定兩個中斷點（馬上就會談到）。

若程式包含許多原始程式檔，下面是如何將中斷點設定在目前原始程式以外的其它檔案：

```
(gdb) break filename: line-number
(gdb) break filename: function-name
```

要設定情況中斷點，命令為 break if。如下所示：

```
(gdb) break line-or-function if expr
```

expr 表合法的 gdb 運算式。gdb 在 expr 的值為非 0（即 true），且到達某一行或函式入口才會停止程式。這便於設定要停在迴圈當中之特定循環。如下面的命令要停在這個詭異迴圈的第二輪：

```
% gdb qsort2
(gdb) list 43,47
43   printf("Tests with RANDOM inputs and FIXED pivot\n");
44   /*random input, factor of 10 bigger each time*/
45   for(testsize = 10; testsize <= MAXSIZE; testsize *=10) {
46            gen_and_sort(testsize,RANDOM,FIXED);
47   }
(gdb break 46 if testsize == 100;
```

```
Brakpoint 1 at 0x2394: file qsort2.c, line 46.
(gdb) run
Starting program: /home/los/mikel/cuser/qsort2
Tests with RANDOM inputs and FIXED pivot
test of 10 element: user + sys time, ticks: 0

Breakpoint1, main() at qsort2.c:46
46    gen_and_sort(testsize,RANDOM,FIXED);
(gdb)
```

gdb 還支援另一種稱作“監看點”(watchpoint) 之中斷點，它有點類似我們剛提過的“break-if”中斷點，只不過它不是設定在特定一行或函式入口。程式只要在運算式為 true 時就會停住：如下面命令設定在變數 testsize 大於 100000 時停止。

```
(gdb) watch testsize > 100000
```

監看點的理念很好，但用起來不是很方便。它們在下列情況特別有用：程式當了，而你知道是因為某重要變數被破壞，但你確定相關程式沒有錯，所以一定是其它地方搞鬼，這時你正是需要監看點。問題在於若無特別的硬體支援（只有少數工作站才有），設定監看點可讓你的程式慢上 100 倍左右。所以若你非做不可的話，依下列步驟：

1. 在你的程式中盡量靠近當掉的地方設定一般中斷點。
2. 設定監看點。
3. 以 continue 命令讓程式繼續執行。
4. 讓你的程式執行一整夜。

另外值得一提的是 tbreak 命令，可以設定“暫時的”中斷點。它們跟一般短中斷點一樣，只是它們在使程式停止後就失效了。但它們並沒被刪除；你可以 enable 命令使暫態中斷點再度生效。

從中斷點繼續執行

停在中斷點後，你可以 continue 命令（可縮寫成 c）繼續執行：

```
% gdb qsort2
(gdb) break init_random
Breakpoint 1 at 0x28bc: file qsort2.c, line 155.
(gdb) run
Starting program: /home/los/mikel/cuser/qsort2
Tests with RANDOM inputs and FIXED pivot

Breakpoint 1, init_random (number=10) at qsort2.c:155
155  for (i = 0; i < number; i++) {
(gdb) continue
Continuing.
test of 10 elements: user + sys time, ticks: 0

Breakpoint 1, init_random (number=100) at qsort2.c:155
155  for (i = 0; i < number; i++) {
(gdb)
```

程式繼續執行到結束，這時若不是遇到中斷點，便是錯誤發生。

若你在沒有程式執行時下達 continue 命令，gdb 顯示下列訊息：

```
The program is not being run.
```

若你還沒下達 run 命令，或程式已執行完畢，不論正常與否，你就會看到這個訊息。

你可以按 return 鍵以跑到下一個中斷點，而不必重複打 continue 命令。（原則上按 return 鍵可重複執行前一個命令，雖然有很多例外）

你也可以在 continue 後面加上數字參數，代表“跳過下面 n 個中斷點”。例如 c 5 跳過下面 5 個中斷點而停在第六個。

管理中斷點

要刪除中斷點，需要兩個命令：info breakpoints 及 delete。info breakpoints 命令列出你在程式中設定的所有中斷點及監看點。如下：

```
(gdb) info breakpoints
Num   Type             Disp Enb         Address  What
1     breakpoint       keep y           0x000028bc       in init_random at qsort2.c:155
3     breakpoint       keep y           0x0000291c       in init_organ at qsort2.c:168
4     breakpoint       keep y           0x00002544       in gen_and_sort at qsort2.c:79
(gdb)
```

這顯示目前有三個中斷點，識別號碼分別為 1，3，4。上面列表也顯示每一中斷點是如何定義的：它的“屬性”(是否在停住後就會失效)、是否在有效狀態、記憶體位址及它的定義（函式名、檔案名、行號)。

一旦你知道中斷點的識別號碼，就可用 delete 命令加以刪除。例如 delete 1 刪除 qsort2.c 第 155 行之中斷點。同樣的命令也可用來刪除監看點。若 delete 後沒有任何參數，則所有的中斷點都會被刪除。

clear 命令也很類似，或許更好用。以行號作為參數，clear 可以刪除那行上的所有中斷點。若以函式名為參數，clear 刪除函式入口上的所有中斷點。若沒有參數，表示刪除目前這行上的所有中斷點。

gdb 也可以使中斷點生效或失效。如果你已經花了許多精神去輸入一個相當長的中斷點命令，再加上許多條件設定須輸入，也許現在你想要忽略（disable）它，但也許你也會想要保留，以便稍後再度使用（使有效)。這裡是一些相關的命令：

disable number

使第 number 個中斷點失效。它仍會在列表 (list) 中，但直到使它能作用之前，它都在休止狀態。number 參數可以是一或多個中斷點號碼。若你省略 number 參數，則所有的中斷點都會失效。

enable number

使中斷點能作用，因而再度使能中斷執行。number 參數可以是一或多個中斷點號碼。若你省略 number 參數，則所有中斷點都適用。

enable once number

使中斷點能暫時作用一次。在下次程式執行到此時會暫停，然後這中斷點會再度失效。number 參數可以是一或多個中斷點號碼。

在某些系統上，gdb 可支援執行緒。執行緒是行程（process）的一部分，它們就像不同的行程一樣獨立執行，但卻以同一行程的方式共用全域資料及其它資源。若你的系統有執行緒（thread）函式庫，gdb 可以在單一執行緒中設定中斷點。

檢視及設定變數的值

當程式停在中斷點時，你可以看看程式到底是如何執行的。基本的變數處理命令包括：

whatis

判斷變數或陣列的型別。

set variable

設定變數的值。

print

除了印出變數的值外，print 也可設定值。

要知道任何變數的型別，可用下列命令：

```
(gdb) whatis variable-name
```

variable-name 是使用中變數或陣列名稱。例如：

```
(gdb) where
#0  init_random (number=0) at qsort2.c:156
#1  0x2584 in gen_and_sort (numels=10, genstyle=0, start=1) at qsort2.c:86
#2  0x23a8 in main () at qsort2.c:46
(gdb) whatis x
type = int (1000000)
```

這裡我們看到 x 是一個包含 1000000 個元素的整數陣列。你也可以用 whatis 去查詢資料結構，雖然 ptype 命令的功能更勝一籌。舉例來說，whatis 雖能給你資料結構的名稱，ptype 卻能告訴你資料結構的定義。例如：

```
(gdb) whatis s
type = struct tms *
(gdb) ptype s
type = struct tms {
     long tms_utime;
     long tms_stime;
     long tms_cutime;
     long tms_cstime;
} *
(gdb)
```

我們前面看過，print 會印出任何變數或運算式的值。如下面的例子：

```
(gdb) print vec(a,1.0)
$4 = 2.2360679774997898
(gdb) print x[2]
$5 = 1027100827
```

第一例計算呼叫 vec 函式的結果。因為函式正在執行中，所以你除了看到傳回值外，也因為執行了 vec 而產生內部的副作用。另一命令，print x[2]，印出此陣列的值。

print 命令也可用來做指定運算。例如，我們不喜歡 x[2] 的值，可將它指定為：

```
(gdb) print x[2] = 4 + x[1]
$6 = 143302918
```

因此印出 4+x[1] 之運算結果，並指定到 x[2]。看到運算式的結果，可以幫助你發現一些打字錯誤；但你若不想看運算式的值，可改用 set variable 命令。下列的 set 命令跟上面的 print 命令效果相同：

```
(gdb) set variable x[2] = 4 + x[1]
(gdb) print x[2]
$8 = 143302918
```

若程式已作過最佳化，也許不能印出變數的值或將其改指定新值。例如已指定到暫存器或已從程式中刪除的變數，gdb 就不可能看得到。whatis 則永遠可以看到變數的型別。你也可用更高級的 info address name 命令去找出資料存在那裡；這樣可以告訴你是否變數已指定到暫存器去了。

單步執行

gdb 有兩種方式可進行單步執行。當 next 命令碰到函式呼叫時，會執行完整個函式，而 step 命令則會進入函式並一步執行一指令。為了解釋兩種命令的差別，看看下面這個簡單程式的除錯過程，分別觀察它們的表現：

```
% gdb qsort2
(gdb) break main
Breakpoint 6 at 0x235c: file qsort2.c, line 40
(gdb) run
Breakpoint 6 main () at qsort2.c:40
40   int power=1;
```

```
(gdb) step
43   print("Tests with RANDOM inputs and FIXED pivot\n");
(gdb) step
45   for (testsize = 10; testsize <= MAXSIZE; testsize *= 10) {
(gdb) step
46            gen_and_sort(testsize,RANDOM,FIXED);
(gdb) step
gen_and_sort (numels=10, genstyle=0, start=1) at qsort2.c:79
79   s = &start_time;
(gdb)
```

我們在 main 函式設一中斷點並作單步執行。幾步之後，我們到了呼叫 gen_and_sort 處。這時 step 命令會進入 gen_and_sort 函式，突然間，我們正在執行第 79 行而非第 46 行。它“走進”函式，而非整個執行完畢。

相形之下，下列命令執行完整個 gen_and_sort 而不停止：

```
(gdb) run
The program being debugged has been started already.
Start it from the begging? (y or n) y
Starting program: /home/los/mikel/cuser/qsort2

Breakpoint 7, main () at qsort2.c:40
40   int power=1;
(gdb) next
43   print("Tests with RANDOM inputs and FIXED pivot\n");
(gdb) next
Tests with RANDOM inputs and FIXED pivot
45   for (testsize = 10; testsize <= MAXSIZE; testsize *= 10) {
(gdb) next
```

```
46            gen_and_sort(testsize,RANDOM,FIXED);
(gdb) next
test of 10 elements: user + sys time, ticks: 0
45   for (testsize = 10; testsize <= MAXSIZE; testsize *= 10) {
(gdb)
```

next 命令後，gdb 準備執行程式中的下一行了。它執行完第 46 行，包括呼叫 geb_and_sort。

step 及 next 甚至在沒有原始程式檔案的情況下仍可以作用。但 step 命令不會走進內建或是函式庫函式。

呼叫函式

gdb 可以讓你單獨執行函式呼叫。有許多命令可處理單一函式：

call name

呼叫並執行函式。

finish

結束目前函式執行並印出傳回值（假如有的話）。

return value

取消目前的函式執行，並傳回 value。

call 命令執行指定函式，幾乎同於 print，但若函式沒有傳回值的話，它就不會印出來。命令形式如下：

```
(gdb) call function(arguments)
```

舉例說，下列命令：

```
(gdb) call gen_and_sort(1234,1,0)
test of 1234 elements: user + sys time, ticks: 3
(gdb)
```

執行 gen_and_sort，參數為 1234、1 和 0。中斷點、監看點及其它 gdb 功能都仍舊能使用。如下例：

```
(gdb) break 79
Breakpoint 11 at 0x2544: file qsort2.c, line 79
(gdb) call gen_and_sort(1234,1,0)

Breakpoint 10, gen_and_sort (numels=1234, genstyle=1, start=0)
     at qsort2.c:79
79            s = &start_time
The program being debugged stpped while in a function called from GDB.
The expression which contained the function call has been discarded.
Unable to restore previously selected frame.
```

函式在中斷點停止；你可以用 continue 繼續執行，或設定其它中斷點等動作。

finish 命令會繼續目前的函式執行，直到函式傳回控制時便停止執行，若有傳回值的話，並印出來。如下例：

```
(gdb) run
Starting program: /home/los/mikel/cuser/qsort2
tests with RANDOM inputs and FIXED pivot

Breakpoint 11, gen_and_sort (numels=1-, genstyle=0, start=1) at qsort2.c:79
79    s = &start_time;
(gdb) finish
Run till exit from #0 gen_and_sort (numels=10, genstyle=0, start=1)
     at qsort2.c:79
```

```
test of 10 elements: user + sys time, ticks: 0
main () at qsort2.c:45
45   for (testsize = 10; testsize <= MAXSIZE; testsize *= 10) {
(gdb)
```

我們在 gen_and_sort 函式中碰到中斷點，然後經由 finish 命令，我們繼續執行到結束。此時程式停在 main() 中第 45 行。gen_and_sort 沒有傳回值，所以甚麼都沒印。若你在函式中碰到中斷點而停止，return 命令停止函式執行，而程式停在函式傳回處。聽來像 finish，不過 return 不會執行完函式剩下的部分。這樣做當然很危險，因為跳過一些程式碼可能會造成一些副作用。無論如何，return 的參數（若有的話）會傳回給呼叫者。像這樣用：

```
(gdb) return return-value
```

自動執行命令

gdb 可以讓你在中斷點及監看點後附加一串命令。這樣做的用途很多。一般用法中，它們可以用來作“自動”輸出；例如你可以在碰到某個中斷點時查看 i 及 j 的值。使用命令串，而非手動輸入 print 命令去印出這些值，可以節省不少打字的功夫。在特殊應用上，可以用來修正錯誤資料後繼續除錯，讓你收集更多的資料而不必重新編譯程式。

下列是命令串的文法：

```
(gdb) commands number
…list-of-commands
…list-of-commands
end
```

命令串附在第 number 中斷點（或監看點）上。省略 number 的話，命令串則附在上一個中斷點上。

例如我們懷疑關於時間的計算有錯，可在程式第 94 行設定中斷點：

```
(gdb) break 94
Breakpoint 12 at 0x25e4: file qsort2.c, line 94.
(gdb) run
Starting program: /home/los/mikel/cuser/qsort2
Tests with RANDOM inputs and FIXED pivot

Breakpoint 12, gen_and_sort (numels=10, genstyle=0, start=1) at qsort2.c:94
94   printf(“test of %d elements: user + sys time, ticks: %d\n”,
(gdb) list
89   times(s);
90   qsort2(0,numels-1,start); /* do the sort */
91   times(e);
92   begin = (s->tms_utime + s->tms_stime);
93   end =    ( e->tms_utime + e->tms_stime);
94   printf(“test of %d elements: user + sys time, ticks: %d\n”,
95            numels,end-begin);
96   }
97
```

我們想在碰到中斷點時檢查 *s 及 *e 的值，所以補充下列命令：

```
(gdb) commands 12
Type commands for when breakpoint 12 is hit, one per line.
End with a line saying just “end”.
Echo value of s (start time) \n
print *s
echo value of e (end time) \n
print *e
end
```

注意，我們加上 echo 命令以便使中斷點輸出更為清楚。繼續執行之後，會看到下面的訊息：

```
(gdb) continue
Continuing.
test of 1000 elements: user + sys time, ticks: 2
Breakpoint 12, gen_and_sort (numels=10000, genstyle=0, start=1)
at qsort2.c:94
94    printf("test of %d elements: user + sys time, ticks: %d\n",
value of s (start time)
$28 = {tms_utime = 13, tms_stime = 21, tms_cutime = 0, tms_cstime = 0}
value of e (end time)
$29 = {tms_utime = 37, tms_stime = 21, tms_cutime = 0, tms_cstime = 0}
```

當我們碰到中斷點，時間資料就自動顯示出來了。命令串中可以放入任何命令，這裡有些特別值得一提：

silent

告訴 gdb 碰到中斷點時不要“宣佈”；silent 只能放在命令串的最前面。最好不要用 silent，除非你在命令串中有加上 print 之類的命令。

continue

continue 可以放在命令串的結尾以便自動繼續執行。你可以得到中斷點產生的任何輸出，但事實上程式並未停止。

值得一提的是“自動顯示”的功能，在程式暫停時可以用來計算並印出運算式。例如下面的 display 命令跟前面的例子有類似的效果：

```
(gdb) display *s
1: *s = { tms_utime = 11, tms_stime = 11, tms_cutime = 0, tms_cstime = 0}
(gdb) display *e
2: *e = { tms_utime = 11, tms_stime = 11, tms_cutime = 0, tms_cstime = 0}
```

每一的顯示均包含一個“顯示號碼”，以便識別。顯示和命令串最大的不同，在於顯示在每當程式停止，而非任何特定中斷點時，就會印出值。（有一個例外：區域變數只會在定義範圍所在的程式暫停時才會出現。）

要刪除一個顯示，使用 undisplay number 命令，number 是你在設定顯示時被指定的識別數字。要觀察那些顯示正在作用中，可以用 info display 命令。

變數之範圍及本文

本節說明 gdb 中提到變數，函式及運算式時的定義及觀念。通常你可以直接使用變數及函式的名稱，像目前所有的例子一樣。但有時 gdb 表現並不如你預期：它會抱怨某個變數“不在目前本文中”，或抓到的是不同函式中的變數。下面我們解釋為何有這些衝突，以及如何化解。

活動中及休止中之變數

變數若非在活動中便是在休止中。在除錯時，你只能抓到目前本文中所有的變數。若你要抓沒有定義在目前本文中的變數，gdb 會回應下列訊息：

```
No symbol “i” in context.
```

下列是一些判斷變數是否「在本文中」的規則：

* 全域變數永遠是在活動中，不論程式是否在執行。
* 若程式不在執行中，所有非全域變數均非在活動中。程式啟動之後，結束前均可視為“執行中”（停在中斷點時亦是）。程式在結束後便不算在執行中。
* 區域變數在包含的程序，或程序所呼叫的任何函式正在執行時均算在活動中。例如程序 g1 中的變數 i 在 g1 執行時是在活動中的。它在 g1 或 g1 所呼叫的程序執行時都在活動中。

檢查程式異常結束後留下的 core dump 檔案，算是一個特例。當你這樣做時，gdb 允許你查看程式中斷時所有活動中的變數。gdb 不准你改變 core 檔案中變數的值；這樣做毫無意義，因為在異常結束後你無法繼續執行。

變數名稱及範圍

多個靜態變數可以共用同樣的名稱，因為靜態變數的範圍僅限於所在之原始程式檔案。為避免混淆，gdb 讓你明確指定所要的變數。

完整的變數名稱格式如下：

```
file-or-function::name
```

name 是變數名稱，file-or-function 是變數定義所在的檔案或函式的名稱。例如要印出 trans.c 中 trans 函式裡的變數 foo 值，你可以用下列命令之一：

```
(gdb) print trans::foo
(gdb) print ‘trans.c’::foo
```

注意第二例中，檔案名稱加註單引號。

在呼叫堆疊中上下移動

許多提示性命令會因所在程式位置而有不同表現；它們的參數及輸出依目前所在程式單元而定。通常目前單元指的是正在暫停的函式。但有時你會想顯示其它函式中的變數。

up 及 down 命令讓你在目前的呼叫堆疊中上下移動一層。up n 及 down n 命令讓你在目前的呼叫堆疊中上下移動 n 層。往下表示從 main 函式越往下呼叫；往上表示離 main 越近。藉由 up 及 down，你可以檢查堆疊中任何函式，包括遞迴呼叫的區域變數。當然，你必須先往上才能往下；因為你正在目前執行的函式中，那裡是堆疊的最下面。

例如，在 qsort2 中，main 呼叫 gen_and_sort，其又呼叫 qsort2，又再呼叫 swap。若你停在 swap 中的中斷點，where 命令可以給你如下的報告：

```
(gdb) where
#0   swap (i=3, j=7) at qsort2.c:134
#1   0x278c in qsort2 (l=0, u=9, start=1) at qsort2.c:121
#2   0x25a8 in gen_and_sort (numels=10, genstyle=0, start=1) at qsort2.c:90
#3   0x23a8 in main () at qsort2.c:46
(gdb)
```

up 命令將 gdb 的注意力導向 qsort2 之堆疊框，所以你可以檢視 qsort2 中之區域變數；否則之前它們是看不到的。再一個 up 就到了 gen_and_sort 的堆疊框了；down 命令則可讓你回到 swap。如果你忘了在甚麼地方，frame 命令會總結目前堆疊框的內容：

```
(gdb) frame
#1   0x278c in qsort2 (l=0, u=9, start=1) at qsort2.c:121
121  swap(i,j);
```

本例顯示我們正在檢視 qsort2 的堆疊框，目前正要呼叫 swap 函式。

機器語言功能

gdb 提供一些特殊命令來處理機器語言。首先，info line 命令告訴你原始程式中某一行的物件碼開始及結束位址。如下例：

```
(gdb) info line 121
Line 121 of “qsort2.c” starts at pc 0x277c and ends at 0x278c.
```

然後可以用 disassemble 命令來觀察這行的機器碼：

```
(gdb) disassemble 0x260c 0x261c
Dump of assembler code from 0x260c to 0x261c:
0x260c <qsort2>: save  %sp, -120, %sp
0x261c <qsort2+4>:     st %i0, [ %fp + 0x44 ]
0x261c <qsort2+8>:      st %i1, [ %fp + 0x48 ]
0x261c <qsort2+12>:     st %i2, [ %fp + 0x4c ]
End of assembler dump.
```

disassemble 的參數指定一段位址去進行反組合。若你想反組合整個函式，可以將函式名稱作為參數；例如 disassemble swap 產生 swap 函式的機器碼。

stepi 及 nexti 命令就像 step 及 next 命令，只不過是機器語言指令層次，而非原始指令。stepi 命令執行下一個機器語言指令；nexti 命令也是，但在呼叫函式時會執行完整個函式。

記憶體檢查命令 x (examine) 可印出記憶體的內容。有兩種用法：

```
(gdb) x/nfu addr
(gdb) x addr
```

前一種有指定格式；後一種則依預設（就是前一次 x 或 print 命令所用格式；若第一次執行的話，用十六進位）。addr 是位址。

nfu 是格式資訊，依序是下列三項：

* n 表示要印出多少資料項
* f 表示何種輸出格式
* u 代表資料單位的大小（位元組，字組等）

表 6-1 顯示可用的格式（f 值）

Format code	Output format
x	十六進位
d	有正負之十進位
u	無正負之十進位
o	八進位
t	二進位
a	位址；印出十六進位數字，表示離最近符號的位移
c	字元
f	浮點數字

表 6-2 顯示可以用的大小單位

Size code	Data size
b	位元組
h	半字組（兩位元組）
w	字組（四位元組）
g	大字組（兩字組，八位元組）

舉例來說，讓我們看看程式中第 79 行的 s 變數。print 顯示它是 struct tms 的指標：

```
79    s = &start_time;
(gdb) print s
$1 = (struct tms *) 0xf7fffae8
```

要來就是 print *s 命令，這會顯示該資料結構中所有的欄位：

```
(gdb) print *s
$2 = {tms_utime = 9, tms_cutime = 0, tms_cstime = 0}
```

為了示範起見，我們用 x 來檢查資料。struct tms（在 time.h 中定義）包含四整數欄位；所以我們須要印出四個十進位數字。可以用命令 x/4dw，由 s 位址開始：

```
(gdb) x/4wd s
0xf7fffae8 <_end+-138321592>:  9        14       0        0
```

從 s 開始的四個字組分別為 9,14,0,0，和 print 顯示的一樣。

一群特殊的 gdb 變數可以讓你檢查及修改電腦的一般暫存器。現代處理器中幾乎都包含四個通用的暫存器，gdb 給予了標準名稱：

表 6-3 標準暫存器名稱

Name	Register
$pc	程式計數器
$fp	堆疊框指標（目前堆疊框）
$sp	堆疊指標
$ps	處理器狀態

除此之外，不同的計算機架構各自有其暫存器命名方式。要知道你的電腦如何命名暫存器，最好的辦法就是 info registers 命令，它會顯示所有暫存器的內容。注意，有些標準名稱其實是較長名稱的縮寫；例如在 SPARC 系統上，狀態暫存器稱作 $ps 或 $psr。info registers 命令會採用較長的名稱。

信號（signal）

gdb 通常能抓到所有系統訊號。抓到信號之後，gdb 決定如何處理你正執行的程序。例如，Ctrl-C 送出中斷信號給 gdb，通常應終止 gdb 執行；但其實你不是要中斷 gdb，你真正要中斷的是 gdb 正在除錯的程式。因此，gdb 抓到信號並停止上面執行的程式，這樣你可以進行一些除錯工作。

handle 命令控制信號的處理。它需要兩個參數：信號名稱及收到信號後應採取的動作。它的值可能是下列之一：

nonstop

收到信號後，直接傳給程式而不停止程式。

stop

收到信號後停止程式，讓你能除錯。顯示信號已到達的訊息（除非不讓訊息顯示）

print

收到信號後顯示訊息。

nonprint

收到信號後不顯示訊息（而且不停掉程式）。

pass

將信號傳給程式，讓你的程式去處理，當掉，或採取其它行動。

nopass

停掉程式，但不將信號傳給程式。

例如你想攔住 SIGPIPE 信號，而不讓你正在除錯的程式收到它；但當信號發生時，你希望程式停止並作些修改。要這麼做，可用下列命令：

```
(gdb) handle SIGPIPE stop print
```

注意，UNIX 信號名稱永遠是大寫的！你可以用號碼來代替信號名稱。

假如你的程式有作任何信號處理，你需要測試這些信號處理程式；要這麼做，你需要一個簡便的方式送給程式一個信號。這就是 signal 命令的作用。它的參數為數字或是 SIGINT 之類的名稱。例如你的程式有 SIGINT（鍵盤輸入中斷，或 Ctrl-C；信號 2）的信號處理程序來作些大掃除動作。要測試這些程式，你可以設定中斷點並給予下列命令：

```
(gdb) signal 2
Continuing with signal SIGINT (2).
```

程式仍然繼續執行（如同 continue 命令一樣），但信號已立即送給你的處理程序去執行了。

信號種類隨系統而不同，甚至並非所有的 UNIX 都相同。要知道你的作業系統提供那些信號，及 gdb 如何處理它們，可用 info signals 命令。

便利變數（Convenience Variables）

gdb 可以定義用在運算式及指定指令中的便利變數，便利變數的格式如 $name，name 可以是數字以外的任何字元。先前介紹的暫存器變數也是便利變數，它們的名稱（pc,sr,sp,fp）及其它硬體相關的名稱不應該在除錯中重複定義。相對的，參考歷史資料時是用數字（$1,$2 等）而非便利變數。

set 命令定義及指定值給 gdb 變數，用法如下：

```
(gdb) set variable = epression
```

它將運算式 expression 的運算結果指定到 variable 變數，若無 = 運算元，則 variable 只是被定義，而不會指定運算結果。

便利變數就像任何其它變數一樣可以用在運算式中，所以像 print $foo++ 之類的命令是合法的：結果是印出 $foo 的值然後加 1。事實上，print 可用來產生並設定便利變數的值（例如 print $bar=123）；只有在你不需要印出結果的時候，set 才有用。

處理原始程式檔案

要找到原始程式中特定的一行，可以用 gdb 的尋找命令。search text 命令會印出目前檔案中下一個包含 text 的指令行；同樣的，reverse-search text 印出前一個包含 text 的指令行，text 可以是任何 UNIX 的正常運算式。例如，search return 命令會尋找檔案中下一個 return 指令。

大型開發專案的原始程式常會分佈在不同的目錄中。例如某個名為 digest 的大型程式位於 /work/bin，此外並用到 /work/phase1、/work/phase2 及 /work/gastro 三個程式目錄。你希望 gdb 在顯示程式的時候會到三個目錄中尋找原始程式。

依照預設狀況，gdb 只會在目前工作目錄及程式編譯目錄（記錄在符號表中）尋找原始程式，但如果你啟動 gdb 的時候加上一個或以上的 -d 參數，被指定的目錄會加到搜尋目錄列中；所以在剛剛提到的例子裡，可以這樣啟動 gdb：

```
% gdb -d /work/phase1 -d /work/phase2 -d /work/gastro digest
```

gdb 中的 directory 命令用來改變搜尋目錄列，文法如下：

```
(gdb) directory list-of-directory
```

gdb 將加到搜尋路徑的前面，例如下列命令：

```
(gdb) directory /home/src
Source dierectories searched: /home/src:$cdir:$cwd
```

命令下達後，gdb 會先搜尋 /home/src，再搜尋目前目錄。

假如不加任何參數，directory 命令會在提示你確認後重設搜尋目錄列至預設值：

```
(gdb) directory
Reinitialize source path to empty? (y or n) y
Source directories searched: $cdir:$cwd
(gdb)
```

個人化設定

就像大部分 FSF 的產品一樣，gdb 可輕易的進行個人化設定。基本上有三種方式增加 gdb 的功能：

* 使用 define 命令去增加一個“使用者定義的命令”
* 使用 define 命令來寫一個“攔截 (hook)”（正常 gdb 命令的擴充）
* 寫命令檔（或 script）以便用 source 命令執行

每當 gdb 開始執行，它會在目前目錄找 .gdbinit 檔案（註）。.gdbinit 內含一串 gdb 命令以供 gdb 在啟動時執行，也是一個定義你自己的命令或攔截的理想位置。注意，與其它大部分的公用程式不同的是，gdb 有兩個起始檔案。大部分情況下，針對一個專案，你只會在一個目錄底下作除錯。所以你大可以將起始命令分成“專案特定”啟動（處理某特定程式用到的命令）及“一般用途”設定（如鍵盤偏好等）。

使用者定義的命令及攔截（hooks）

使用者命令產生方式如下：

```
(gdb) define command-name
...command...
...command...
end
```

註　　在某些特殊環境下，起始檔案有不同名稱。

command-name 是你要定義的新檔案名稱，以後只要你一執行這命令，gdb 就會自動執行其中所有命令。使用者定義的命令不能接受參數。舉例來說，當你停在中斷點時，常常繼續執行到下一個中斷點，然後單步執行四個機器語言指令。看下列如何將過程自動化：

```
(gdb)define runstep
Redefine command “runstep”? (y or n) y
type commands for definition of “runstep”.
End with a line saying just “end”.
continue
stepi
stepi
stepi
stepi
end
(gdb) runstep
test of 100 elements: user + sys time, ticks: 1

Breakpoint 1, gen_and_sort (numels=1000, genstyle=0, start=1)
     at qsort2.c:79
79   s = &start_time;
0x2548        79       s = &start_time
80   e = &end_time
0x2550        80       e = &end_time
83   if (genstyle == ORGANPIPE) {
(gdb)
```

使用者定義的命令可以非常複雜，也可以只是簡單的“三行指令”用來定義常用命令的別名。當然，gdb 的命令已經很好用，你其實不需要去定義別名。

原則上，攔截（hook）類似使用者定義的命令，所謂攔截，是一串在原本正常的命令之前先執行的 gdb 命令。例如，你希望在單步執行前先印出 i 值，可以寫如下的命令，名為 hook-step：

```
(gdb) define hook-step
Type commands for definition of “hook-step”.
End with a line saying just “end”.

print i
end
(gdb)
```

注意，這和一般的使用者定義命令一模一樣；事實上你可以當作正常的使用者定義命令來執行。唯一特別的是規定你在命令前加上 hook- 字樣。要看這樣做有何效果，再試試單步執行：

```
(gdb) step
$2 = 0
155  for (i = 0; i < number; i++) {
(gdb) step
$3 = 0
156           x[i] = random();
(gdb) step
$4 = 1
155  for (i = 0; i < number; i++) {
```

可以看到 gdb 在每一 step 動作前會先印 i 值。

文稿（Scripts）

文稿是在你下達 source filename 命令後執行的一連串 gdb 命令。有兩件事情需要注意：

* 在其中的命令發生錯誤時，script 會停止執行
* 當 gdb 命令作為文稿的一部分而執行時，通常需要再確認的命令就不會再提示你做確認了

例如你想要在除錯某程式時設定一些中斷點。你可以在檔案 setbkpts.gdb 中放入一系列 break 命令；然後用下列命令自動執行：

```
(gdb) source setbkpts.gdb
```

（本例中，將中斷點命令放在工作目錄中的 .gdbinit 起始檔案還比較方便）

UNIX 介面

Shell 命令啟動 UNIX shell。輸入 Ctrl-D 可以離開 shell，回到 gdb。

shell shell-command 命令執行指定的 shell 命令，而後立刻回到 gdb。例如：

```
(gdb) shell date
Mon Apr6  16:51:20 EST 1987
(gdb)
```

這些命令可以讓你在不離開除錯器的情況下，很快的執行 shell 命令。

依預設，gdb 啟動 Bourne shell。可以設定 SHELL 環境變數來改變這預設，使用不同的 shell。例如下列命令：

```
% setenv SHELL /bin/bash
```

要求改用 GNU 的 Bourne Again shell （bash）。

C++ 程式

假如你寫的是 C++，編譯時就用 g++，你會發現 gdb 是個很好的作業環境。它完全瞭解 C++ 的文法，在觀念上也配合類別延伸 C 結構的方式。讓我們透過一個小程式，看看 gdb 如何處理類別（class）及建構函式（constructor）。

雖然下列程式做甚麼無關緊要，但為了方便，我們還是解釋一下：它是個很小公用程式，用來處理文件的索引項。索引存在類別中的 e_text 成員裡，程式用來參考這資料的獨特短字串存在 e_reference 成員裡。本程式只產生一個索引項：

```
(gdb) list 1,30
1    #include <fstream.h>
2    #include <strings.h>
3    #include <stdio.h>
4
5    const unsigned int REF_SIZE = 80;
6
7    class entry {
8             char *e_text;
9             char e_reference[REF_SIZE];
10   public:
11            entry(const char *text,
12                     const unsigned int length,
13                     const char *ref) {
14                     e_text = new char(length+1);
15                     strncpy(e_text, text, length+1);
16                     strncpy(e_reference, ref, REF_SIZE);
17            }
18   };
19
20   main(int argc, char *argv[])
21   {
```

```
22            char *text_1 = “Finding errors in C++ program”;
23            char *ref_1 = “errc++”;
24            entry entry_1(text_1, strlen(text_1), ref_1);
25    }
```

為了觀察程式動作，我們在第 24 行的 entry 指令處設定中斷點。這個宣告會啟動一個函式，當然就是 entry 建構函式。

```
(gdb) b 24
Breakpoint 1 at 0x23e4: file ref.C, line 24
(gdb) run
Starting program: /home/los/mikel/crossref/ref

Breakpoint 1, main (argc=1, argv=0xeffffd8c) at ref.C:24
24 entry entry_1(text_1, strlen(text_1), ref_1);
```

現在我們進入函式。使用 step 命令，就像進入 C 函式：

```
(gdb) step

entry::entry (this=0xeffffcb8, text=0x2390, ”Finding errors in C++ programs”,
     length=30, ref=0x23b0 “errc++”) at ref.C:14
14            e_text = new char(length+1);
```

gdb 跑到 entry 建構元的第一行，顯示呼叫函式時所傳的參數。當我們回到主程式時，可以印出 entry_1 變數，就像任何其它資料結構一樣：

```
(gdb) print entry_1
$1 = {e_text = 0x6128 “Finding errors in C++ programs”,
     e_reference = “errc++”, ’\000’ <repeats 73 times>}
```

所以除錯 C++ 就像除錯 C 一樣直接。

Emacs 介面

Emacs 提供特別的模式，使得使用 gdb 特別容易。啟動方式是 ESC x gdb 命令。Emacs 會提示你輸入檔案名稱：

```
Run gdb (like this): gdb
```

輸入完畢後，Emacs啟動一特別視窗來執行 gdb，你將看到：

```
Current directory is /home/los/mikel/cuser/
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type “show copying” to se the conditions.
There is absolutely no warranty for GDB; type “show warranty” for details.
GDB 4.16 (sparc-sun-solaris2.4), Copyright 1996 Free Software Foundation, Inc.
(gdb)
```

你現在可以下達任何 gdb 命令。當停在中斷點時，gdb 自動產生一視窗來顯示原始程式碼並標示中斷點所在，如下所示：

```
struct tms end_time, *e;
     int begin, end;

=>   s = &start_time;
     e = &end_time;

     /* initialize x according to right style */
```

=> 標誌顯示要執行的下一行。它的位置隨每次 gdb 中斷而改變，也就是每個單步執行及每次 continue 後等等。你也許永遠不需再用到 list 命令！

原始碼緩衝區即是正常 Emacs 緩衝區；你可以編輯、儲存、搜尋等等。=> 標誌只會顯示在螢幕上，不用擔心它會被加到你的程式裡。

Emacs gdb 介面有許多其它的便利。例如，你可用鍵序 C-c C-s 作為 step 的縮寫。要利用更多關於命令結合及其它功能，可用命令 C-h m。雖然我們覺得輸入“傳統的”gdb 命令比較容易，但能夠隨時看到最新原始碼顯示，卻是 Emacs 除錯模式最強的地方。

命令補正（Completion）及縮寫

本章介紹許了多命令。gdb 有兩個稍微重複的功能，可以節省打字所花費的功夫。

最常用的命令有一字縮寫。例如 print 命令可縮寫成 p。表 6-4 列舉了前面介紹過的命令。

表 6-4 gdb 命令縮寫

縮寫	指令	功能
b	break	設定中斷點
c	continue	從中斷點繼續執行
d	delete	刪除中斷點（或其它物件）
f	frame	顯示堆疊框
h	help	顯示命令輔助說明
i	info	顯示相關資訊
l	list	表列原始碼
n	next	單步執行（跳過函式）
p	print	顯示變數或運算式
q	quit	離開 gdb
r	run	執行程式
s	step	單步執行（走進函式）

命令補正是較通用的特色。所有命令皆可以縮寫到最短而又不混淆的程度。例如，ptype 可以縮寫成 pt。你不能縮寫成 p，因為它不是獨一無二的，p 已經是 print 的縮寫了。

若你想知道某縮寫是甚麼命令，輸入定位鍵即可。gdb 然後會填滿命令（同時發出嗶嗶聲，告訴你做完了），但不會執行它。假如 gdb 沒有填滿，有兩種可能：

* 命令已經完整，輸入換行鍵來執行它。
* 命令仍然混淆，再輸入定位鍵來看所有可能的結尾；然後再加上一或多個字元以使命令獨特。

命令補正是非常強大的功能。你幾乎可以用在任何地方。例如不用輸入 info breakpoints，只要 ib 命令即可。i 是 info 的縮寫，而 breakpoints 是唯一 b 開頭的 info 子命令。

你也可以用在變數名稱上、函式名稱、及幾乎所有需要完整輸入的地方。變數等名稱的輸入就像命令補正一樣；輸入不會混淆的前置字元，然後用定位鍵完成剩下部分（如果程式中變數太多的話，可能會拖些時間）。

命令編輯

另一個有用的功能是能夠編輯命令以修正打字的錯誤。gdb 提供一部分 Emacs 命令讓你在命令行來回移動。底下是其中最重要的：

表 6-5 gdb 命令編輯

鍵盤按鍵	指令
C-c	移回一字元
C-f	移前一字元
C-a	移到本行頭
C-e	移到本行尾
ESC f	移前一字
ESC b	移回一字
DEL	刪除游標左邊字元
C-d	刪除游標所在字元
C-_	復原上一指令
C-l	清除螢幕

例如下列命令：

```
(gdb) stop in gen_and_sort
```

這些你應該很熟悉了，這是個 dbx 命令。我們其實要輸入 break gen_and_sort。要修改它，我們先打 C-a，然後輸入 ESC d 兩次，然後輸入 break，再按輸入鍵來執行：

```
(gdb) break gen_and_sort
Breakpoint 1 at 0x2544: file qsort2.c, line 79.
(gdb)
```

若你在 emacs 中執行 gdb，就不需要命令編輯了。

命令歷史（Command History）

gdb 也可以叫出之前的命令。它跟 C shell 或 bash shell 的歷史功能非常類似。要起用命令歷史，用 set history expansion on 命令（註）。以後要叫出前一個命令，輸入 !!。

同樣的，!c 可以叫出上一個 c 開頭的命令。例如下面關於亂數產生器的例子：

```
(gdb) set history expansion on
(gdb) print random() + random()
$1 = -1537112880
(gdb) !!
print random() + random()
$2 = 14567834598
(gdb) !p
print random(0 + random()
$3 = 23984783743
```

註　命令歷史預設是關閉的，因為它會干擾 C 邏輯運算元（! 及 !=）的解譯。當使用命令歷史時，在邏輯運算式中的！後加上空白可以減少麻煩。

若你打開命令歷史，也可以用 Emacs 式的命令在前面的命令中移動。C-p 叫出前一個命令以供你編輯或執行。C-n 叫出“下一個”命令（假如有的話）。很明顯的，C-n 只在你已回到歷史命令時才有意義。

你會發現在 emacs 中執行 gdb 時，命令歷史很方便。我們發現使用 emacs 命令比較容易移回去，“剪下”所要命令，當你需要時再“貼上”。就如 C shell，gdb 提供一些相當複雜的方式來編輯命令，但我們覺得這些命令太複雜而不實用；有興趣的話，可參考 FSF 的技術文件。

附加到現有行程

gdb 一個高級用法，是對伺服程式這種執行中的行程進行除錯。當啟動 gdb 時，給它執行檔名及行程識別碼。可以將它附加到執行中的程式。例如：

```
% gcc -o qsort2 -q qsort2.c
% qsort2 &
% ps | grep qsort2
2912 p4 R     0:24 qsort2
2914 p4 S     0:00 gre qsort2
% gdb qsort2 2912
/home/los/mikel/cuser/2912: No such file or directory.
Attaching program `/home/los/mikel/cuser/qsort2', pid 2912
0x2734 in qsort2 (l=673395, u=673356, start=1) at qsort2:119
119           do j--; while (x[j] > t);
(gdb)
```

首先，我們看到警告說明 2912 檔案並不存在。不用管它，gdb 總是先檢查確定沒有 core 檔存在。若沒有時，它附加到執行中程序 2912（就是 qsort2）上，停在中斷點（第 119 行），讓你開始除錯。等到你離開後，gdb“脫離”讓函式庫繼續執行。

附加到執行中的行程需要大量的作業系統支援。大部分 UNIX 系統應該沒有問題，但在許多跨平台的環境就無法適用了。

你也可以在啟動 gdb 後再用 attach 命令來附加到行程上。你可以用 detach 命令來脫離行程。

若你在遠端系統，如嵌入（embedded）系統，執行，可以用 gdb 去除錯。如我們前面提到，並非所有的 gdb 功能都能適用，因為遠端系統也許很陽春。但你可以用下列命令去選擇裝置：

```
(gdb) target remote /dev/device-name
```

以及載入程式到遠端系統：

```
(gdb) load file
```

快速參考

表 6-6 顯示 gdb 中最重要的命令。雖然表中列的是完整名稱，你通常可以輸入一或兩個字即可執行。例如 b 代表 breakpoint。

info、set 及 show 命令有許多參數在本章並未提到；較少用的命令可參考 gdb 文件。

表 6-6：一般 gdb 命令

指令	動作
backtrace	顯示程式中目前位置及堆疊記錄（同義字：where）
breakpoint	設定中斷點
cd	變更目前目錄
clear	刪除上一個中斷點
commands	表列遇到中斷點時要執行的命令
continue	從中斷點繼續執行
delete	刪除中斷點或監看點；也和其它命令並用

表 6-6（續）

指令	動作
display	在程式中斷時顯示變數或運算式
down	下移堆疊框到另一個函式
frame	選擇下一個 continue 命令的堆疊框
info	顯示程式相關資訊。如 info breakpoints 顯示所有中斷點及監看點
jump	跳到原始碼中其它地方執行
kill	結束 gdb 控制下執行的行程
list	原始程式碼列表
next	執行下一行，不會走進函式
print	印出變數或運算式的值
pwd	顯示目前目錄
ptype	顯示資料型別的內容，如結構或 C++ 類別
quit	離開 gdb
reverse-search	在原始程式檔中往前尋找運算式
run	執行程式
search	在原始程式檔中尋找運算式
set variable	設定變數值
signal	發出信號給執行中程序
step	執行下一行，會走進函式
undisplay	取消 display 命令；不顯示運算式
until	完成目前迴圈
up	上移堆疊框到另一個函式
watch	設定監看點（資料中斷點）
whatis	顯示變數或函式的型別