手把手教你玩轉GDB

小武哥
<http://www.wuzesheng.com>
2010.11

主要內容

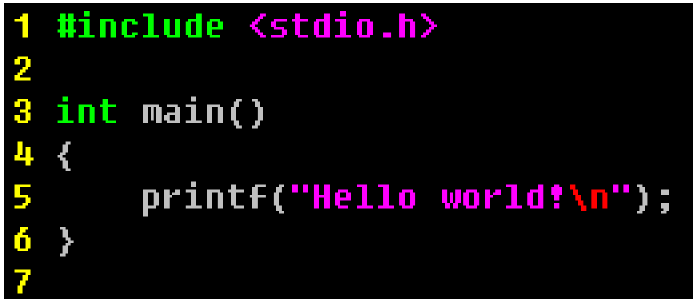
- 1. 溫故知新---程式的秘密
- 2. 牛刀小試---GDB初探
- 3. 大顯身手---玩轉GDB
- 4. 學而時習之---總結回顧

- 本課程所講內容都是基於80x86 32位平臺, 在64位元平臺上某些內容可能會略有差別，請大家注意區別!

1. 溫故知新---程式的秘密

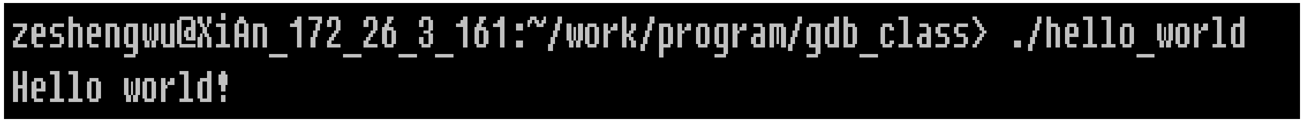
- （1）GCC做了什麼
- （2）進程位址空間

（1）GCC做了什麼



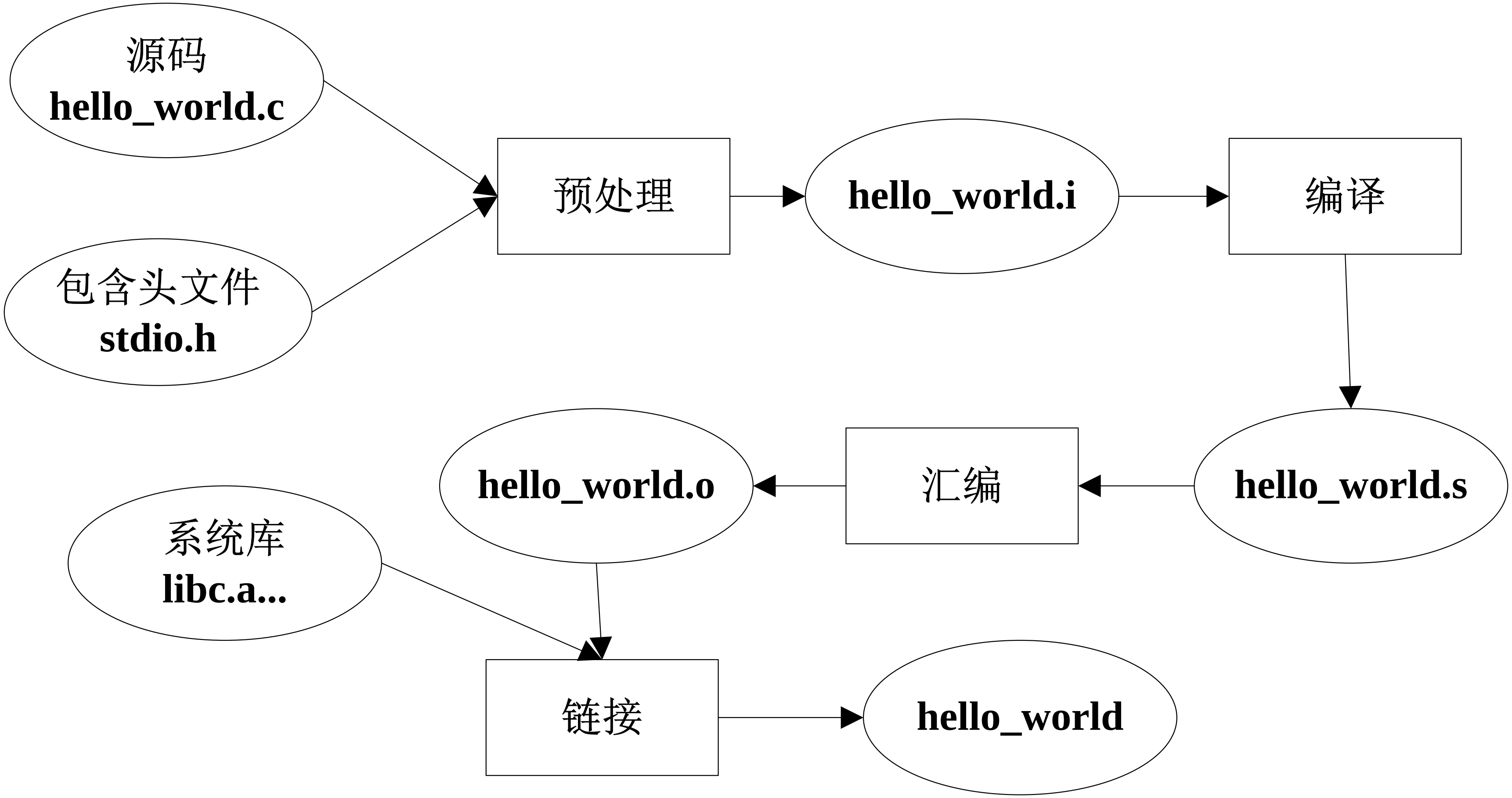
```
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
}

```

gcc hello_world.c –o hello_world



```
zeshengwu@XiAn_172_26_3_161:~/work/program/gdb_class> ./hello_world
Hello world!
```

（1）GCC做了什麼


源码
hello_world.c
包含头文件
stdio.h
预处理
hello_world.i
编译
hello_world.s
汇编
hello_world.o
系统库
libc.a...
链接
hello_world

（1）GCC做了什麼

- A. 預處理

gcc –E hello_world.c –o hello_world.i (調用cpp完成)

任務：展開宏，替換標頭檔，刪除注釋

- B. 編譯

gcc –S

任 代碼

總結—GCC實際上只是對多個工具的包裝，
它會根據不同的參數，去調用cpp、
ccl(cclplus)、as或者ld去完成程式編譯過程
中的一系列工作

- C. 彙編

gcc –c hello_world.s –o hello_world.o(調用as完成)

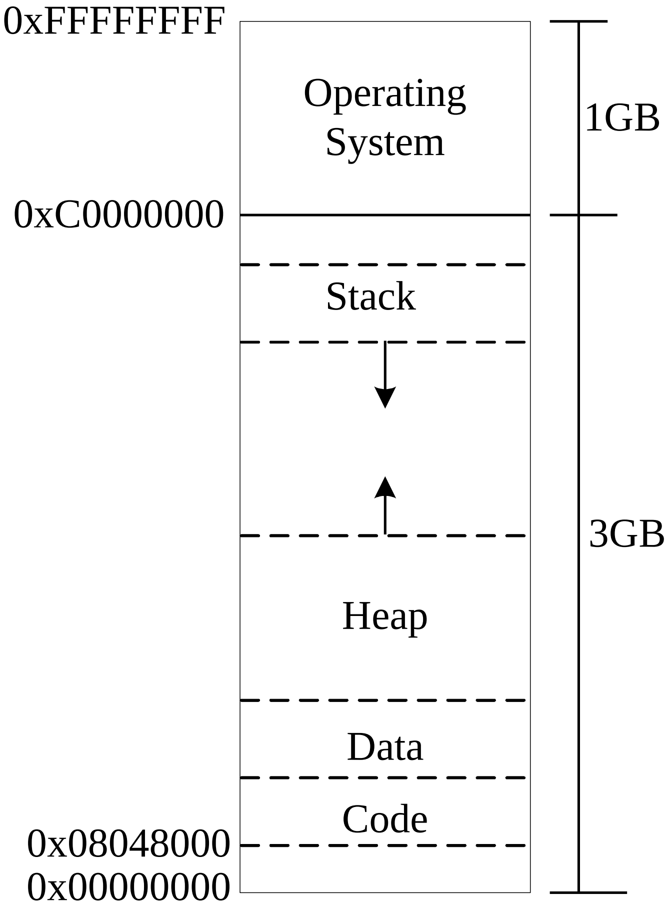
任務：將彙編代碼轉換成為機器可以執行指令

- D. 連結

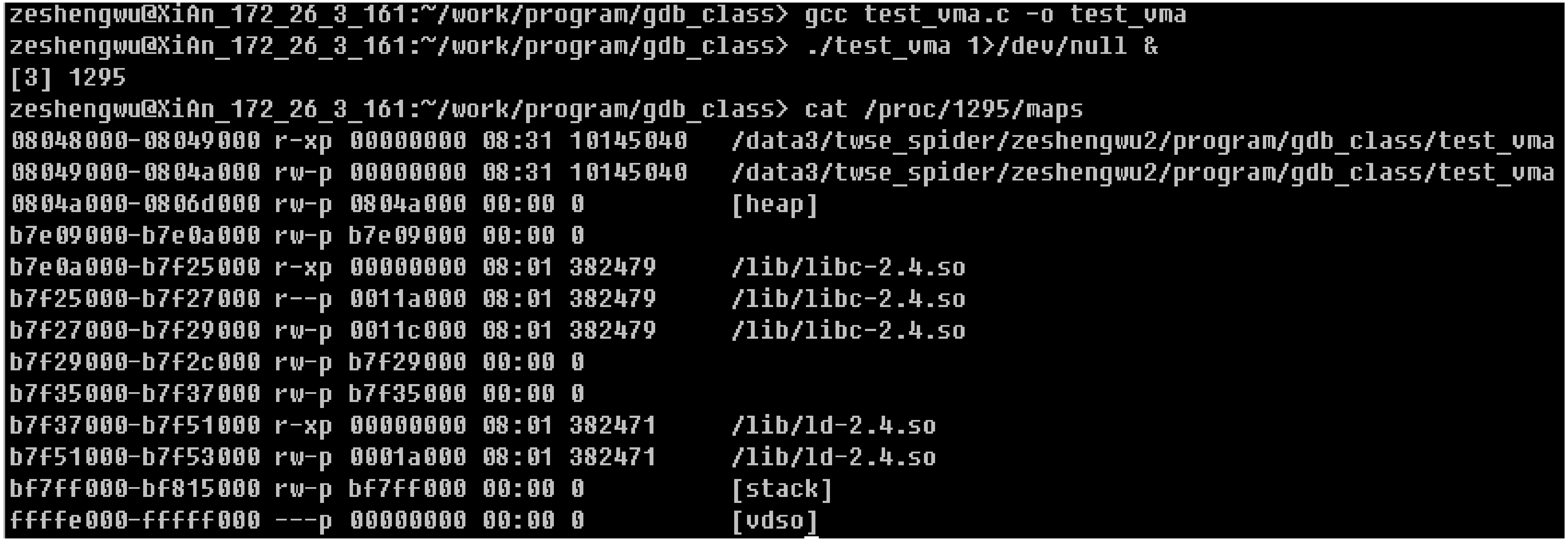
gcc hello_world.o –o hello_world(調用ld完成)

任務：位址和空間分配，符號決議定位，將目的檔案拼裝成可執行檔

（2）進程位址空間


0xFFFFFFFF
Operating System
1GB
0xC0000000
Stack
3GB
Heap
Data
Code
0x08048000
0x00000000

（2）進程位址空間



```
zeshengwu@XiAn_172_26_3_161:~/work/program/gdb_class> gcc test_vma.c -o test_vma
zeshengwu@XiAn_172_26_3_161:~/work/program/gdb_class> ./test_vma 1>/dev/null &
[3] 1295
zeshengwu@XiAn_172_26_3_161:~/work/program/gdb_class> cat /proc/1295/maps
08048000-08049000 r-xp 00000000 08:31 10145040   /data3/twse_spider/zeshengwu2/program/gdb_class/test_vma
08049000-0804a000 rw-p 00000000 08:31 10145040   /data3/twse_spider/zeshengwu2/program/gdb_class/test_vma
0804a000-0806d000 rw-p 0804a000 00:00 0          [heap]
b7e09000-b7e0a000 rw-p b7e09000 00:00 0
b7e0a000-b7f25000 r-xp 00000000 08:01 382479     /lib/libc-2.4.so
b7f25000-b7f27000 r--p 0011a000 08:01 382479     /lib/libc-2.4.so
b7f27000-b7f29000 rw-p 0011c000 08:01 382479     /lib/libc-2.4.so
b7f29000-b7f2c000 rw-p b7f29000 00:00 0
b7f35000-b7f37000 rw-p b7f35000 00:00 0
b7f37000-b7f51000 r-xp 00000000 08:01 382471     /lib/ld-2.4.so
b7f51000-b7f53000 rw-p 0001a000 08:01 382471     /lib/ld-2.4.so
bf7ff000-bf815000 rw-p bf7ff000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

2.牛刀小試---GDB初探

- （1）啟動GDB開始調試
- （2）常用調試命令介紹
- （3）退出GDB結束調試
- （4）尋求幫助

（1）啟動GDB開始調試

- A.準備工作

編譯調試版本的可執行程式(gcc加上-g參數即可,注意不要調試加-O相關的選項)

- B.冷開機

gdb *program*　　　e.g., gdb ./cs

gdb –p *pid*　　　e.g., gdb –p \`pidof cs\`

gdb *program core*　　　e.g., gdb ./cs core.xxx

- C.暖開機

(gdb) **attach** *pid*　　　e.g., (gdb) attach 2313

- D. 命令列參數

gdb *program* **--args** *arglist*

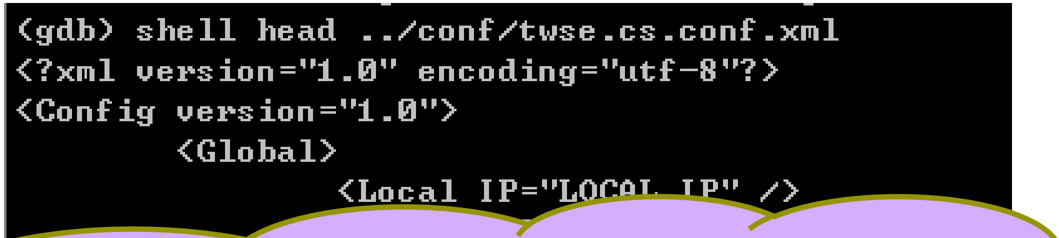
(gdb) **set args** *arglist*

(gdb) **run** *arglist*

（2）常用調試命令介紹

- A. 在GDB中執行shell命令

(gdb) **shell** *command args*



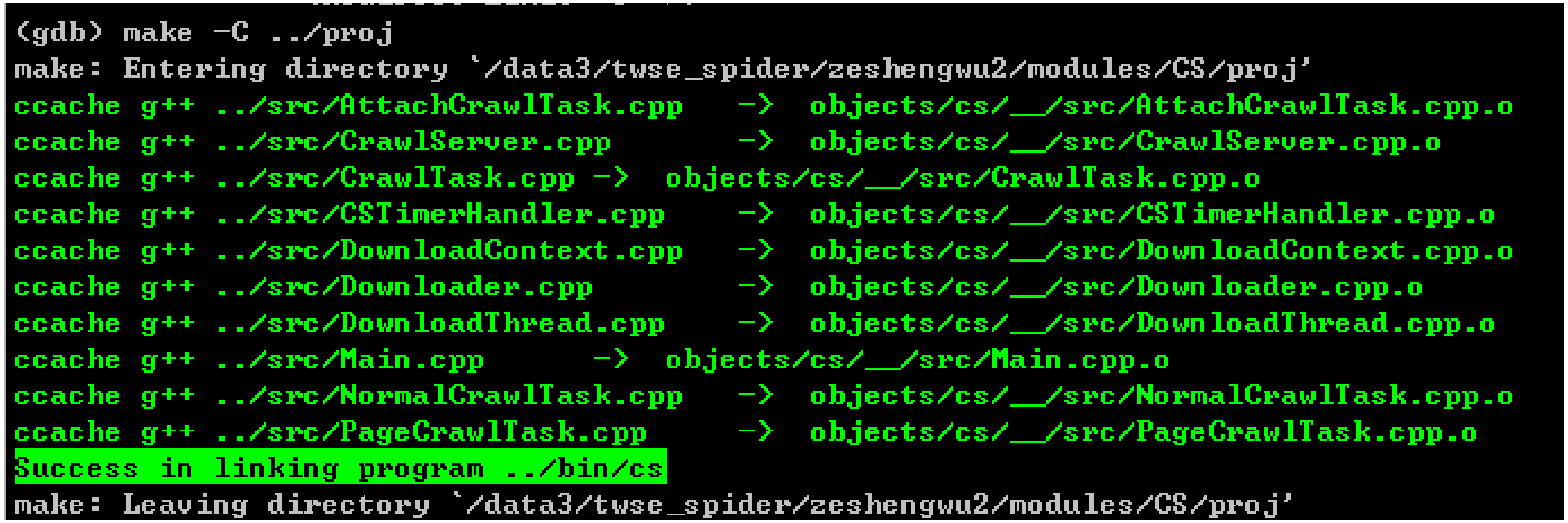
```
(gdb) shell head ../conf/twse.cs.conf.xml
<?xml version="1.0" encoding="utf-8"?>
<Config version="1.0">
        <Global>
                <Local IP="LOCAL_IP" />
```

shell小技巧—可以在GDB中直接執行shell命令，這樣就會暫時退出GDB, 回到shell終端， 在shell執行完*command*後，然後在shell中執行exit命令，便可回到GDB

（2）常用調試命令介紹

- B. 在GDB中調用make

(gdb) **make** *make-args*(=**shell make** *make-args*)



```
(gdb) make -C ../proj
make: Entering directory `/data3/twse_spider/zeshengwu2/modules/CS/proj'
ccache g++ ../src/AttachCrawlTask.cpp   ->  objects/cs/__/src/AttachCrawlTask.cpp.o
ccache g++ ../src/CrawlServer.cpp       ->  objects/cs/__/src/CrawlServer.cpp.o
ccache g++ ../src/CrawlTask.cpp ->  objects/cs/__/src/CrawlTask.cpp.o
ccache g++ ../src/CSTimerHandler.cpp    ->  objects/cs/__/src/CSTimerHandler.cpp.o
ccache g++ ../src/DownloadContext.cpp   ->  objects/cs/__/src/DownloadContext.cpp.o
ccache g++ ../src/Downloader.cpp        ->  objects/cs/__/src/Downloader.cpp.o
ccache g++ ../src/DownloadThread.cpp    ->  objects/cs/__/src/DownloadThread.cpp.o
ccache g++ ../src/Main.cpp      ->  objects/cs/__/src/Main.cpp.o
ccache g++ ../src/NormalCrawlTask.cpp   ->  objects/cs/__/src/NormalCrawlTask.cpp.o
ccache g++ ../src/PageCrawlTask.cpp     ->  objects/cs/__/src/PageCrawlTask.cpp.o
Success in linking program ../bin/cs
make: Leaving directory `/data3/twse_spider/zeshengwu2/modules/CS/proj'
```

（2）常用調試命令介紹

- C. 中斷點(Breakpoints)

a. 設置中斷點：

(gdb) **break** *function*: 在函數*funtion*入口處設置中斷點

(gdb) **break** *linenum*: 在當前原始檔案的第*linenum*行處設置斷點

(gdb) **break** *filename***:***linenum*: 在名為*filename*的原始檔案的第*linenum*行處設置斷點

(gdb) **break** *filename***:***function*: 在名為*filename*的原始檔案中的*function*函數入口處設置斷點

(gdb) **break** *args* **if** *cond*: *args* 為上面講到的任意一種參數，在指定位置設置一個斷點 , 當且僅但*cond*為**true**時，該斷點 生效

(gdb) **tbreak** *args*: 設置一個隻停止一次的斷點, *args*與**break**命令的一樣。這樣的斷點當第一次停下來後，就會立即被刪除

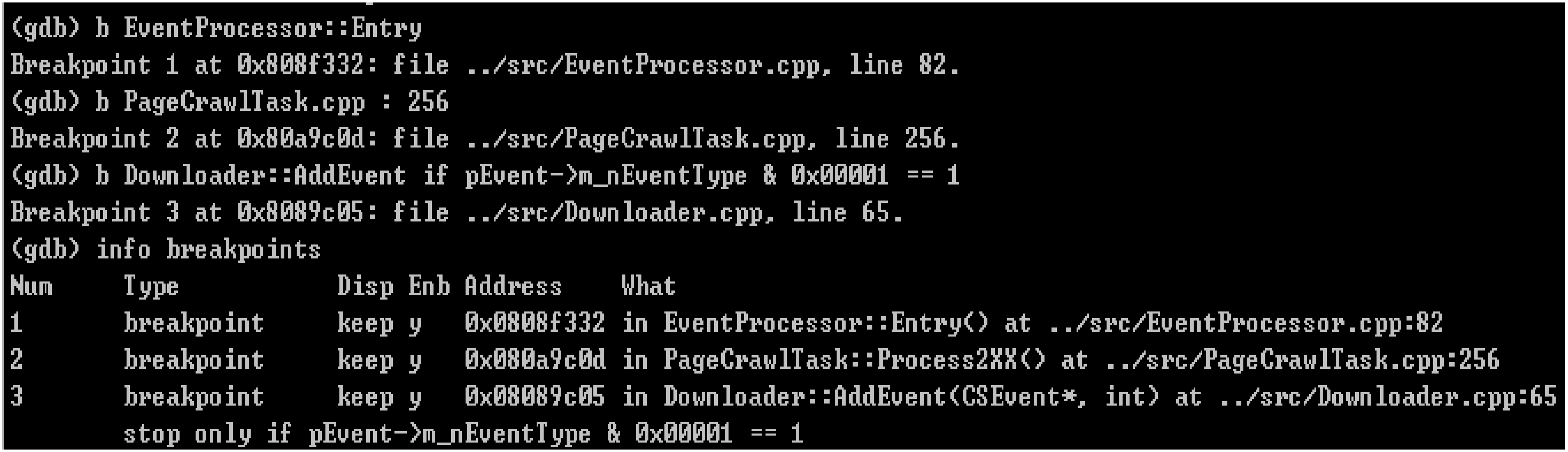
(gdb) **rbreak** *regex*: 在所有符合規則運算式*regex*的**函數**處設置breakpoint

（2）常用調試命令介紹

- C. 中斷點(Breakpoints)

b. 查看中斷點屬性：

(gdb) **info breakpoints** [*n*]:查看第*n*個斷點的相關資訊，如果沒有指定*n*，則顯示所有斷點的相關資訊



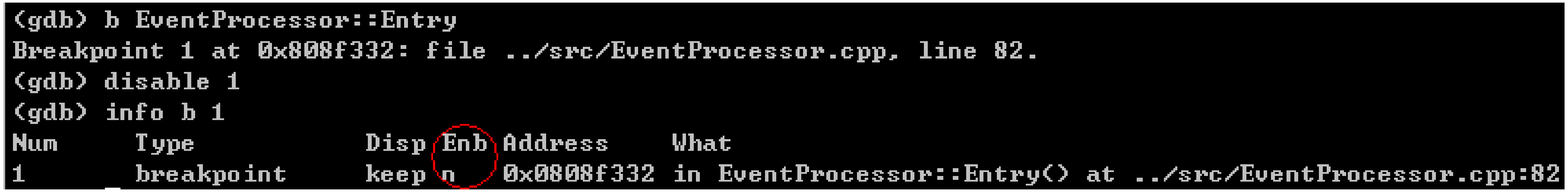
```
(gdb) b EventProcessor::Entry
Breakpoint 1 at 0x808f332: file ../src/EventProcessor.cpp, line 82.
(gdb) b PageCrawlTask.cpp : 256
Breakpoint 2 at 0x80a9c0d: file ../src/PageCrawlTask.cpp, line 256.
(gdb) b Downloader::AddEvent if pEvent->m_nEventType & 0x00001 == 1
Breakpoint 3 at 0x8089c05: file ../src/Downloader.cpp, line 65.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0808f332 in EventProcessor::Entry() at ../src/EventProcessor.cpp:82
2       breakpoint     keep y   0x080a9c0d in PageCrawlTask::Process2XX() at ../src/PageCrawlTask.cpp:256
3       breakpoint     keep y   0x08089c05 in Downloader::AddEvent(CSEvent*, int) at ../src/Downloader.cpp:65
        stop only if pEvent->m_nEventType & 0x00001 == 1
```

（2）常用調試命令介紹

- C. 中斷點(Breakpoints)

c. 中斷點禁用/啟用：

(gdb) **disable [breakpoints]** [*range*…]: 禁用由*range*指定的範圍內的breakpoints



```
(gdb) b EventProcessor::Entry
Breakpoint 1 at 0x808f332: file ../src/EventProcessor.cpp, line 82.
(gdb) disable 1
(gdb) info b 1
Num     Type           Disp Enb Address    What
1       breakpoint     keep n   0x0808f332 in EventProcessor::Entry() at ../src/EventProcessor.cpp:82
```

(gdb) **enable [breakpoints]** [*range*…]: 啟用由*range*指定的範圍內的breakpoints

(gdb) **enable [breakpoints] once** [*range*…]: 只啟用一次由*range*指定的範圍內的breakpoints，等程式停下來後，自動設為禁用

(gdb) **enable [breakpoints] delete** [*range*…]: 啟用*range*指定的範圍內的breakpoints，等程式停下來後，這些breakpoints自動被刪除

（2）常用調試命令介紹

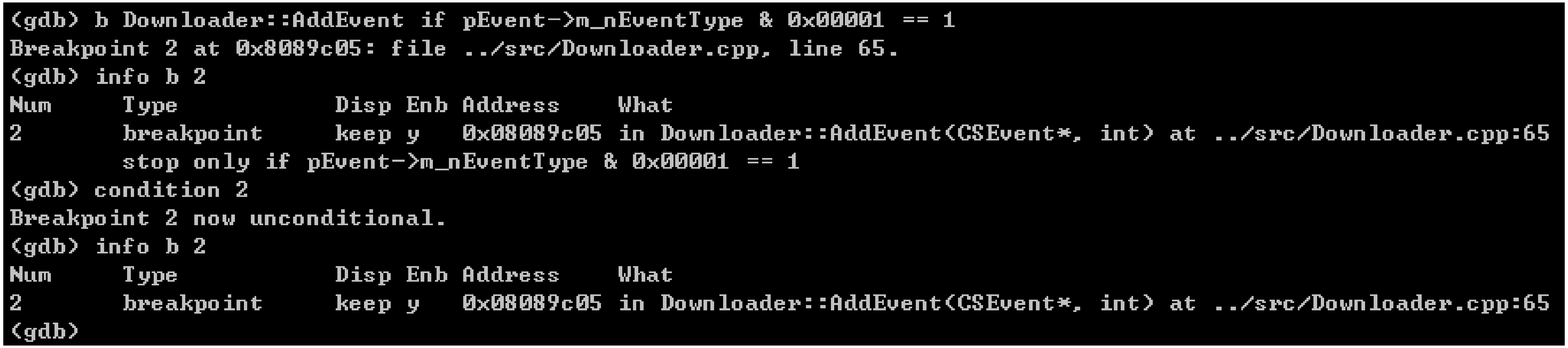
- C. 中斷點(Breakpoints)

d. 條件中斷點:

(gdb) **break** *args* **if** *cond*: 設置條件中斷點

(gdb) **condition** *bnum* [*cond-expr*]: 當指定*cond-expr*時，給第*bnum*個中斷點設置條件；當未指定*cond-expr*時，取消第*bnum*個中斷點的條件

(gdb) **ignore** *bnum count*: 忽略第*bnum*個中斷點*count*次



```
(gdb) b Downloader::AddEvent if pEvent->m_nEventType & 0x00001 == 1
Breakpoint 2 at 0x8089c05: file ../src/Downloader.cpp, line 65.
(gdb) info b 2
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x08089c05 in Downloader::AddEvent(CSEvent*, int) at ../src/Downloader.cpp:65
        stop only if pEvent->m_nEventType & 0x00001 == 1
(gdb) condition 2
Breakpoint 2 now unconditional.
(gdb) info b 2
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x08089c05 in Downloader::AddEvent(CSEvent*, int) at ../src/Downloader.cpp:65
(gdb)
```

（2）常用調試命令介紹

- C. 中斷點(Breakpoints)

e. 在中斷點處自動執行命令

```
(gdb) commands [bnum]
        … command-list …
        end
```

在第*bnum*個斷點處停下來後，執行由*command-list*指定的命令串，如果沒有指定*bnum*，則對最後一個斷點生效

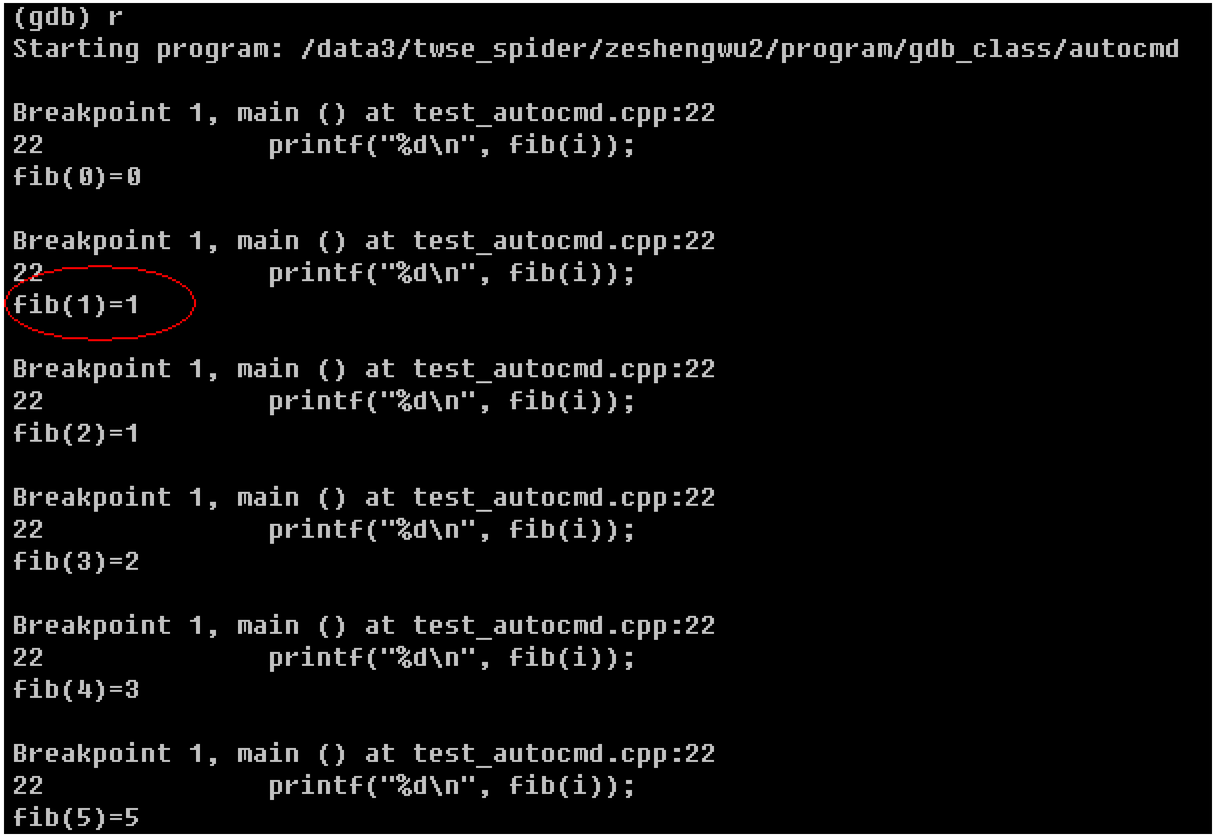
```
(gdb) commands [bnum]
         end
```

取消第*bnum*個中斷點處的命令列表

（2）常用調試命令介紹

- C. 中斷點(Breakpoints)

e. 在中斷點處自動執行命令



```
(gdb) r
Starting program: /data3/twse_spider/zeshengwu2/program/gdb_class/autocmd

Breakpoint 1, main () at test_autocmd.cpp:22
22                printf("%d\n", fib(i));
fib(0)=0

Breakpoint 1, main () at test_autocmd.cpp:22
22                printf("%d\n", fib(i));
fib(1)=1

Breakpoint 1, main () at test_autocmd.cpp:22
22                printf("%d\n", fib(i));
fib(2)=1

Breakpoint 1, main () at test_autocmd.cpp:22
22                printf("%d\n", fib(i));
fib(3)=2

Breakpoint 1, main () at test_autocmd.cpp:22
22                printf("%d\n", fib(i));
fib(4)=3

Breakpoint 1, main () at test_autocmd.cpp:22
22                printf("%d\n", fib(i));
fib(5)=5
```

（2）常用調試命令介紹

- C. 中斷點(Breakpoints)

f. 清理中斷點：

(gdb) **clear** *function* & **clear** *filename:function*: 清除函數*function*入口處的斷點

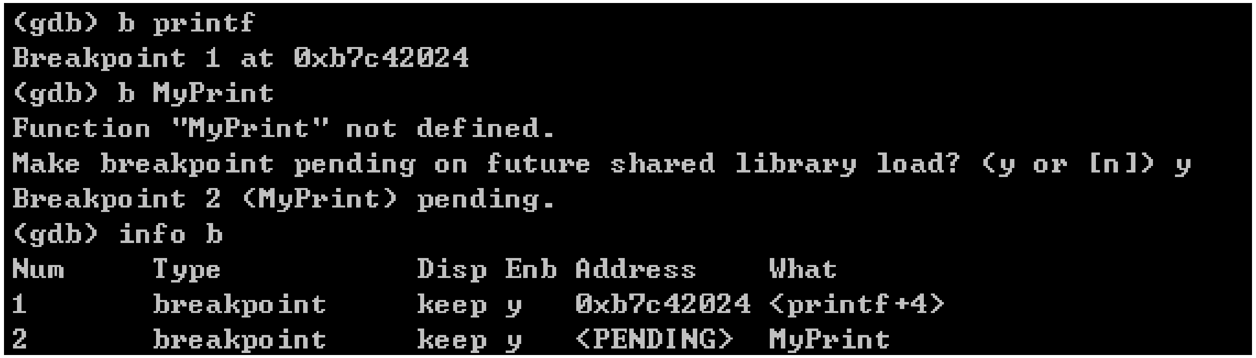
(gdb) **clear** *linenum* & **clear** *filename:linenum*: 清除第*linenum*行處的斷點

(gdb) **delete [breakpoints]** [*range*…]: 刪除由*range*指定的範圍內的breakpoints，*range*範圍是指斷點的序號的範圍

（2）常用調試命令介紹

- C. 中斷點(Breakpoints)

g. 未決的中斷點—pending breakpoints：



```
(gdb) b printf
Breakpoint 1 at 0xb7c42024
(gdb) b MyPrint
Function "MyPrint" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (MyPrint) pending.
(gdb) info b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0xb7c42024 <printf+4>
2       breakpoint     keep y   <PENDING>  MyPrint
```

(gdb) **set breakpoint pending auto**: GDB缺省設置，詢問用戶是否要設置pending breakpoint

(gdb) **set breakpoint pending on**: GDB當前不能識別的breakpoint自動成為pending breakpoint

(gdb) **set breakpoint pending off**: GDB當前不能識別某個breakpoint時，直接報錯

(gdb) **show breakpoint pending**: 查看GDB關於pending breakpoint的設置的行為(auto, on, off)

（2）常用調試命令介紹

- C. 中斷點(Breakpoints)

h. Watchpoints和Catchpoints：

1）Watchpoint的作用是讓程式在某個運算式的值發生變化的時候停止運行，達到‘監視’該運算式的目的

(gdb) **watch** *expr*　　e.g. watch CrawlServer::m_nTaskNum

2）Catchpoints的作用是讓程式在發生某種事件的時候停止運行，比如C++中發生異常事件，載入動態庫事件，系統調用事件

(gdb) **catch** *event*　　e.g. catch throw

3）Watchpoints和Catchpoints都與Breakpoints很相像，都有enable/disabe/delete等操作，使用方法也與breakpoints的類似

（2）常用調試命令介紹

- D. 單步調試

a. 設置中斷點（參見前面《C.中斷點》一節）

b. next & nexti

(gdb) **next** [*count*]：如果沒有指定*count*, 單步執行下一行程式；如果指定了*count*，單步執行接下來的*count*行程式

(gdb) **nexti** [*count*]：如果沒有指定count, 單步執行下一條指令；如果指定了count, 單步執行接下來的count條指令

c. step & stepi

(gdb) **step** [*count*]：如果沒有指定*count*, 則繼續執行程式，直到到達**與當前原始檔案行不同的行**時停止執行；如果指定了*count*, 則重複行上面的過程*count*次

（2）常用調試命令介紹

- D. 單步調試

c. step & s

nexti和stepi的區別--nexti在執行某機器指令時，如果該指令是函式呼叫，那麼程式執行直到該函式呼叫結束時才停止

(gdb) **ste** 然

後停止，

d. continue

(gdb) **continue** [*ignore-count*]：喚醒程式，繼續運行，至到遇到下一個中斷點，或者程式結束。如果指定ignore-count，那麼程式在接下來的運行中，忽略ignore-count次中斷點。

e. finish & return

(gdb) **finish**：繼續執行程式，直到當前被調用的函數結束，如果該函數有返回值，把返回值也列印到控制台

(gdb) **return** [*expr*]：中止當前函數的調用，如果指定了*expr*，把*expr*的值當做當前函數的返回值；如果沒有，直接結束當前函式呼叫

（2）常用調試命令介紹

- E. 變數與記憶體查看

a. print：查看變數

(gdb) **print** [/*f*] *expr*：以*f*指定的格式列印*expr*的值

f：x --- 16進制整數 d --- 10進制整數 u ---10進制不帶正負號的整數

o --- 8進制整數 t --- 2進制整數 a --- 地址 c --- 字元 f --- 浮點數

expr:

1) Any kind of **constant**, **variable** or **operator** defined by the programming language you are using is valid in an expression in GDB.

2) (gdb) **p** ******array@len* : 列印陣列*array*的前*len*個元素

3) (gdb) **p** *file*::*variable*：列印檔案*file*中的變數*variable*

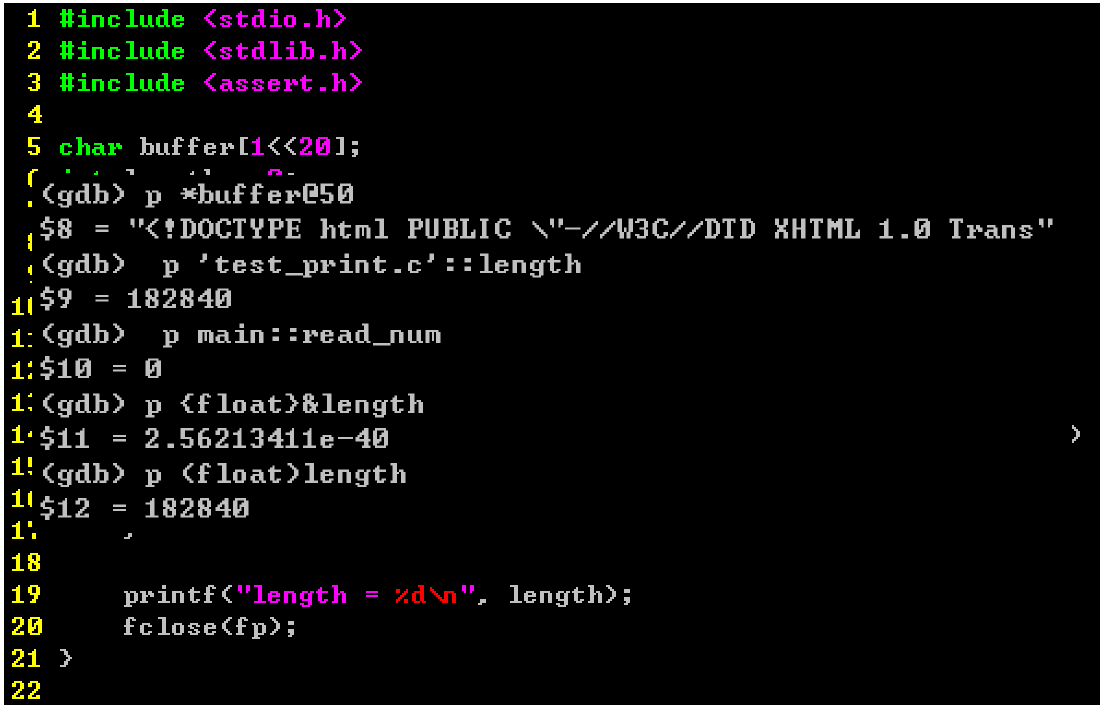
4) (gdb) **p** *function*::*variable*: 列印函數*function*中的變數*variable*

5) (gdb) **p** {*type*}*address*:把*address*指定的記憶體解釋為*type*類型（類似於強制轉型，更加強）

（2）常用調試命令介紹

- E. 變數與記憶體查看

a. print：查看變數



```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

char buffer[1<<20];
(gdb) p *buffer@50
$8 = "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Trans"
(gdb)  p 'test_print.c'::length
$9 = 182840
(gdb)  p main::read_num
$10 = 0
(gdb) p {float}&length
$11 = 2.56213411e-40
(gdb) p (float)length
$12 = 182840

    printf("length = %d\n", length);
    fclose(fp);
}
```

（2）常用調試命令介紹

- E. 變數與記憶體查看

b. x：查看記憶體

(gdb) **x** */nfu addr*

n: 重複次數，缺省是1

f: 列印的格式，除了print支援的格式外，還支援如下格式：

s--- C風格字串，i---機器指令

缺省格式是x

u: 列印的單位大小，支援如下單位：

b---byte, h---halfwords(2bytes), w---words(4bytes), g---giantwords(8bytes)

（2）常用調試命令介紹

- E. 變數與記憶體查看

c. display: 自動列印

(gdb) **display** */f expr*|*addr*: 以格式*f*，自動列印運算式*expr*或地址*addr*

(gdb) **undisplay** *dnums*: 刪除掉指定的自動列印點, dnums可以為一個或者多個自動列印點的序號

(gdb) **delete display** *dnums*: 與 **undisplay** *dnums*同

(gdb) **disable display** *dnums*: 禁用由dnums指定的自動列印點

(gdb) **enable display** *dnums*: 啟用由dnums指定的自動列印點

(gdb) **info display**: 查看當前所有自動列印點相關的資訊

（2）常用調試命令介紹

- E. 變數與記憶體查看

d. 列印相關屬性

基本用法：

(gdb) **set print** *field* **[on]**：打開*field*指定的屬性

(gdb) **set print** *field* **off**：關閉*field*指定的屬性

(gdb) **show print** *field* **：** 查看*filed*指定的屬性的相關設置

相關屬性：

1) (gdb) **set print array**：以一種比較好看的方式列印陣列，缺省是關閉的

2) (gdb) **set print elements** *num-of-elements*：設置GDB列印資料時顯示元素的個數，缺省為200，設為0表示不限制(unlimited)

3) (gdb) **set print null-stop**：設置GDB列印字元陣列的時候，遇到NULL時停止，缺省是關閉的

（2）常用調試命令介紹

- E. 變數與記憶體查看

d. 列印相關屬性

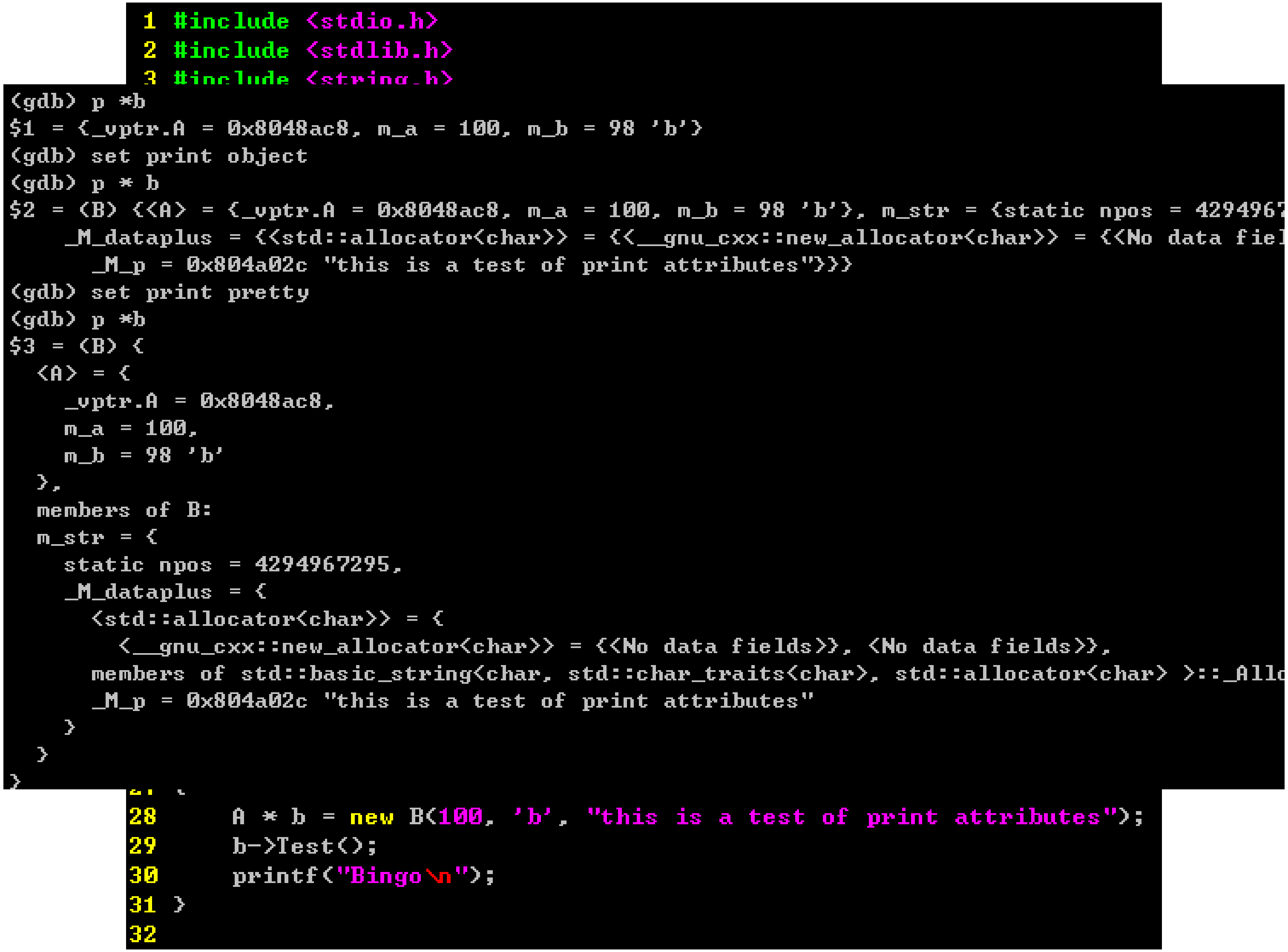
4) (gdb) **set print pretty**：設置GDB列印結構的時候，每行一個成員，並且有相應的縮進，缺省是關閉的

5) (gdb) **set print object**：設置GDB列印多態類型的時候，列印實際的類型，缺省為關閉

6) (gdb) **set print static-members**：設置GDB列印結構的時候，是否列印static成員，缺省是打開的

7) (gdb) **set print vtbl**：以漂亮的方式列印C++的虛函數表，缺省是關閉的

（2）常用調試命令介紹



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
(gdb) p *b
$1 = {_vptr.A = 0x8048ac8, m_a = 100, m_b = 98 'b'}
(gdb) set print object
(gdb) p * b
$2 = (B) {<A> = {_vptr.A = 0x8048ac8, m_a = 100, m_b = 98 'b'}, m_str = {static npos = 4294967
    _M_dataplus = {<std::allocator<char>> = {<__gnu_cxx::new_allocator<char>> = {<No data fiel
      _M_p = 0x804a02c "this is a test of print attributes"}}}
(gdb) set print pretty
(gdb) p *b
$3 = (B) {
  <A> = {
    _vptr.A = 0x8048ac8,
    m_a = 100,
    m_b = 98 'b'
  },
  members of B:
  m_str = {
    static npos = 4294967295,
    _M_dataplus = {
      <std::allocator<char>> = {
        <__gnu_cxx::new_allocator<char>> = {<No data fields>}, <No data fields>},
      members of std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_Allo
      _M_p = 0x804a02c "this is a test of print attributes"
    }
  }
}
```

```
{
      A * b = new B(100, 'b', "this is a test of print attributes");
      b->Test();
      printf("Bingo\n");
}

```

（3）退出GDB結束調試

- 停止

(g 的子進

程

(gdb) **detach**: 調試的進程， 與**attach**配對試用

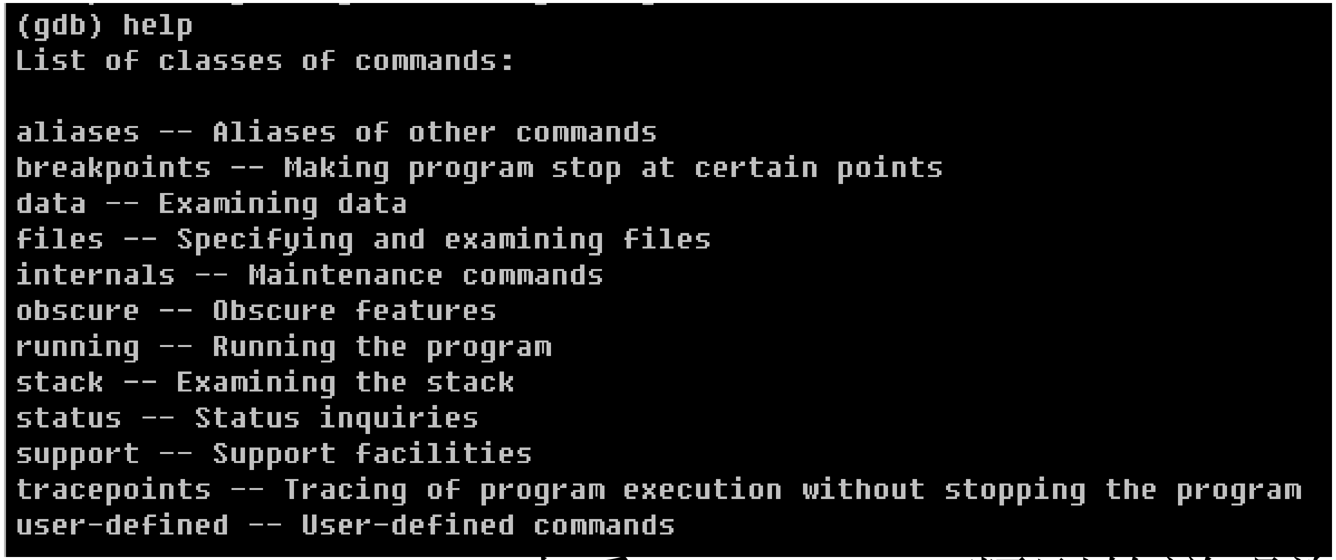
kill小技巧--不退出GDB而對更新當前正在調試的應用程式：在GDB中用**kill**殺掉子進程，然後直接更換應用程式可執行檔，再重新執行**run**，GDB便可載入新的可執行程式啟動調試

- 退出GDB

(gdb) **End-of-File**(ctrl+d)

(gdb) quit

（4）尋求幫助



```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
```

(gdb) **help** *class-name*: 查看*class-name*類別的說明資訊

(gdb) **help all**: 查看所有類別的說明資訊

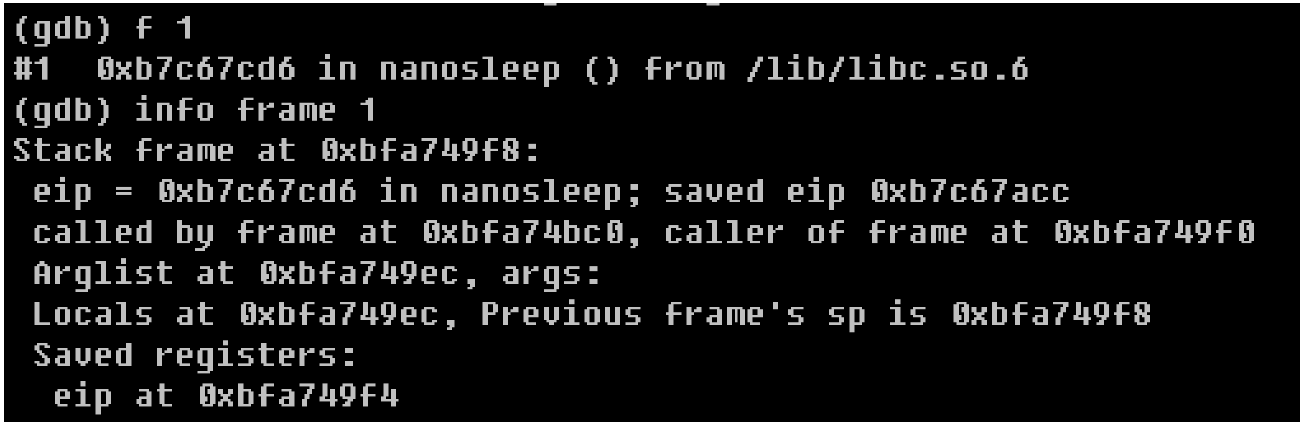
(gdb) **help** *command*: 查看*command*命令的說明資訊

(gdb) **apropos** *word*: 查看*word*關鍵字相關的命令

(gdb) **complete** *prefix*: 查看以*prefix*為首碼的所有命令

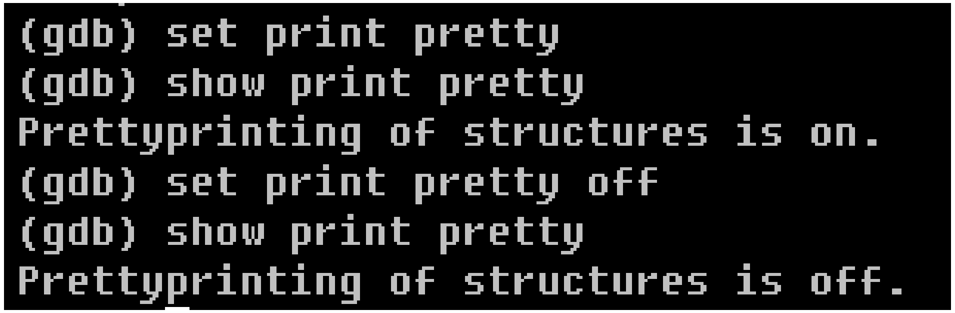
（4）尋求幫助

- **info**：查看與被調試的應用程式相關的資訊



```
(gdb) f 1
#1  0xb7c67cd6 in nanosleep () from /lib/libc.so.6
(gdb) info frame 1
Stack frame at 0xbfa749f8:
 eip = 0xb7c67cd6 in nanosleep; saved eip 0xb7c67acc
 called by frame at 0xbfa74bc0, caller of frame at 0xbfa749f0
 Arglist at 0xbfa749ec, args:
 Locals at 0xbfa749ec, Previous frame's sp is 0xbfa749f8
 Saved registers:
  eip at 0xbfa749f4
```

- **show**：查看GDB本身設置相關資訊



```
(gdb) set print pretty
(gdb) show print pretty
Prettyprinting of structures is on.
(gdb) set print pretty off
(gdb) show print pretty
Prettyprinting of structures is off.
```

3.大顯身手---玩轉GDB

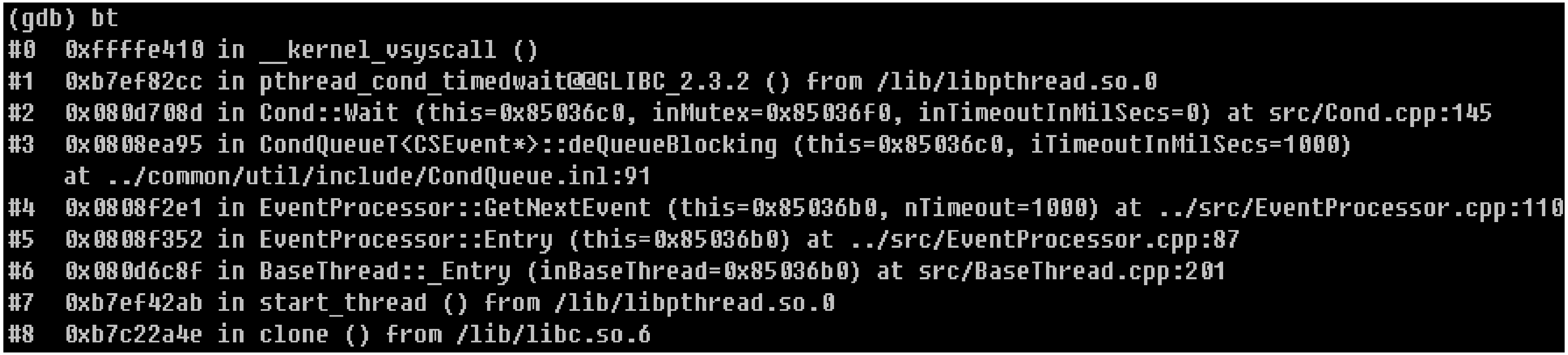
- （1）函式呼叫棧探密
- （2）調試中信號的回應
- （3）修改程式運行、源碼
- （4）多執行緒調試
- （5）自訂命令

（1）函式呼叫棧探密

A. Stack frame(棧楨) & Call stack(調用棧)

Stack frame是指保存函式呼叫上下文資訊的一段區域

Call stack是用來存放各個Stack frame的一塊記憶體區域



```
(gdb) bt
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7ef82cc in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x080d708d in Cond::Wait (this=0x85036c0, inMutex=0x85036f0, inTimeoutInMilSecs=0) at src/Cond.cpp:145
#3  0x0808ea95 in CondQueueT<CSEvent*>::deQueueBlocking (this=0x85036c0, iTimeoutInMilSecs=1000)
    at ../common/util/include/CondQueue.inl:91
#4  0x0808f2e1 in EventProcessor::GetNextEvent (this=0x85036b0, nTimeout=1000) at ../src/EventProcessor.cpp:110
#5  0x0808f352 in EventProcessor::Entry (this=0x85036b0) at ../src/EventProcessor.cpp:87
#6  0x080d6c8f in BaseThread::_Entry (inBaseThread=0x85036b0) at src/BaseThread.cpp:201
#7  0xb7ef42ab in start_thread () from /lib/libpthread.so.0
#8  0xb7c22a4e in clone () from /lib/libc.so.6
```

（1）函式呼叫棧探密

B. 查看Call stack相關資訊

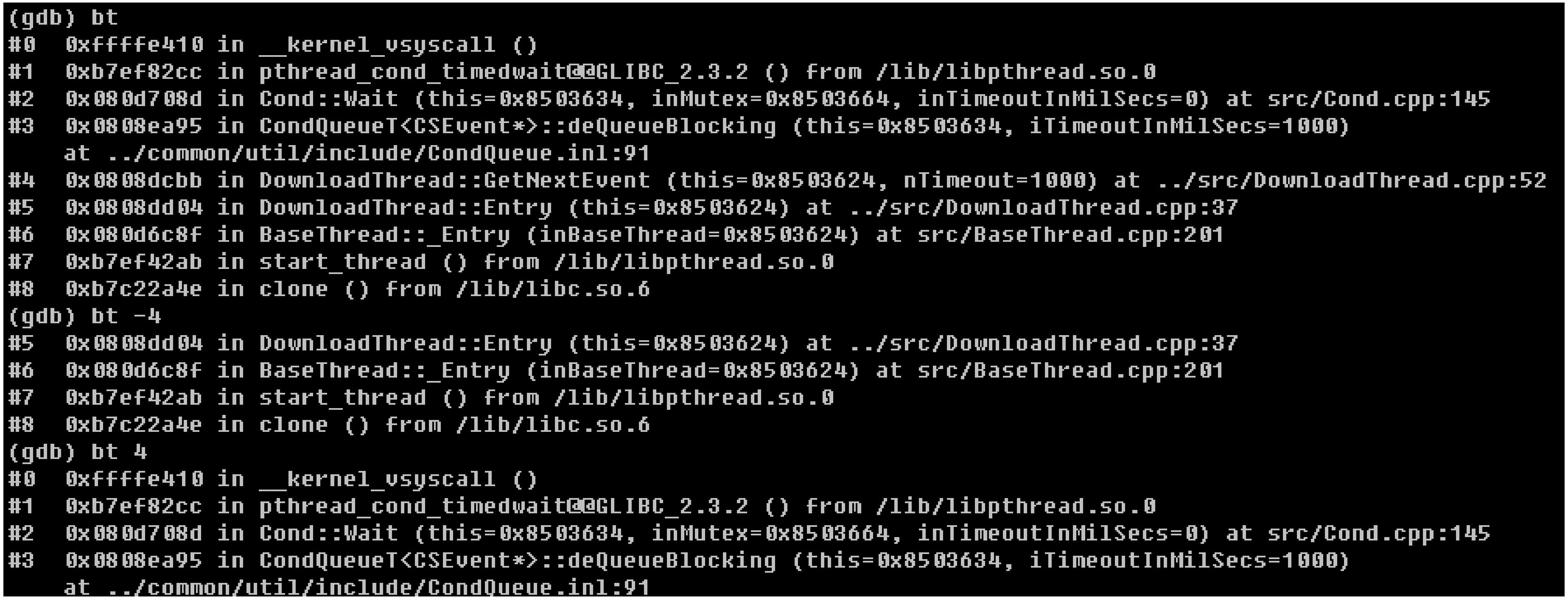
(gdb) **backtrace**:顯示程式的調用棧資訊，可以用**bt**縮寫

(gdb) **backtrace** *n*:顯示程式的調用棧資訊，只顯示棧頂*n*楨

(gdb) **backtrace –***n*:顯示程式的調用棧資訊，只顯示棧底部*n*楨

(gdb) **set backtrace limit** *n*: 設置**bt**顯示的最大楨層數，缺省沒有限制

(gdb) **where**, **info stack**: **bt**的別名



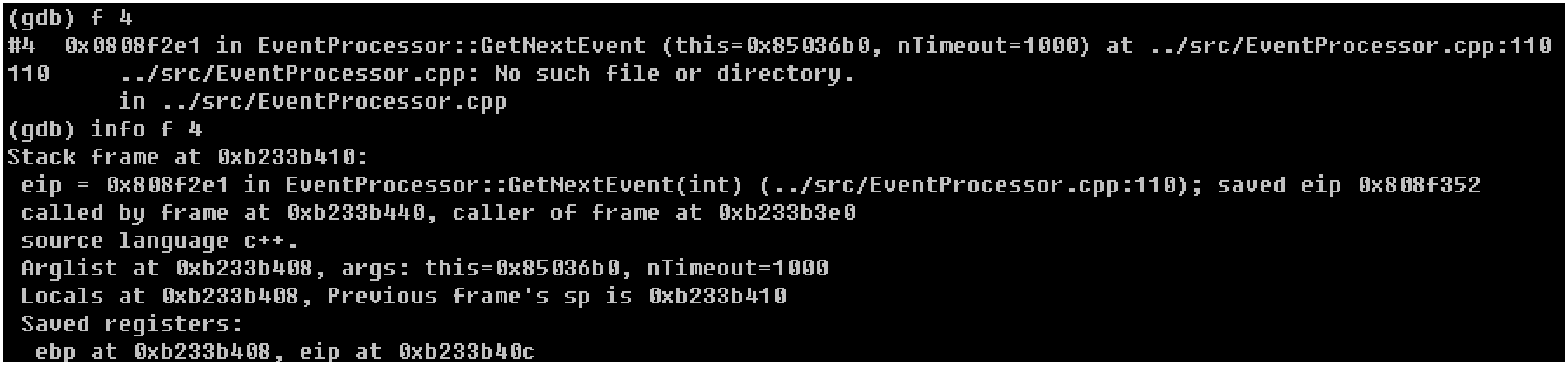
```
(gdb) bt
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7ef82cc in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x080d708d in Cond::Wait (this=0x8503634, inMutex=0x8503664, inTimeoutInMilSecs=0) at src/Cond.cpp:145
#3  0x0808ea95 in CondQueueT<CSEvent*>::deQueueBlocking (this=0x8503634, iTimeoutInMilSecs=1000)
    at ../common/util/include/CondQueue.inl:91
#4  0x0808dcbb in DownloadThread::GetNextEvent (this=0x8503624, nTimeout=1000) at ../src/DownloadThread.cpp:52
#5  0x0808dd04 in DownloadThread::Entry (this=0x8503624) at ../src/DownloadThread.cpp:37
#6  0x080d6c8f in BaseThread::_Entry (inBaseThread=0x8503624) at src/BaseThread.cpp:201
#7  0xb7ef42ab in start_thread () from /lib/libpthread.so.0
#8  0xb7c22a4e in clone () from /lib/libc.so.6
(gdb) bt -4
#5  0x0808dd04 in DownloadThread::Entry (this=0x8503624) at ../src/DownloadThread.cpp:37
#6  0x080d6c8f in BaseThread::_Entry (inBaseThread=0x8503624) at src/BaseThread.cpp:201
#7  0xb7ef42ab in start_thread () from /lib/libpthread.so.0
#8  0xb7c22a4e in clone () from /lib/libc.so.6
(gdb) bt 4
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7ef82cc in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x080d708d in Cond::Wait (this=0x8503634, inMutex=0x8503664, inTimeoutInMilSecs=0) at src/Cond.cpp:145
#3  0x0808ea95 in CondQueueT<CSEvent*>::deQueueBlocking (this=0x8503634, iTimeoutInMilSecs=1000)
    at ../common/util/include/CondQueue.inl:91
```

（1）函式呼叫棧探密

- C. 查看Stack frame信息

(gdb) **frame** *n*: 查看第n楨的簡要信息

(gdb) **info frame** *n*:查看第n楨的詳細資訊



```
(gdb) f 4
#4  0x0808f2e1 in EventProcessor::GetNextEvent (this=0x85036b0, nTimeout=1000) at ../src/EventProcessor.cpp:110
110     ../src/EventProcessor.cpp: No such file or directory.
        in ../src/EventProcessor.cpp
(gdb) info f 4
Stack frame at 0xb233b410:
 eip = 0x808f2e1 in EventProcessor::GetNextEvent(int) (../src/EventProcessor.cpp:110); saved eip 0x808f352
 called by frame at 0xb233b440, caller of frame at 0xb233b3e0
 source language c++.
 Arglist at 0xb233b408, args: this=0x85036b0, nTimeout=1000
 Locals at 0xb233b408, Previous frame's sp is 0xb233b410
 Saved registers:
  ebp at 0xb233b408, eip at 0xb233b40c
```

簡要信息：楨號，$pc, 函數名，函數參數名和參數值，原始檔案名和行號

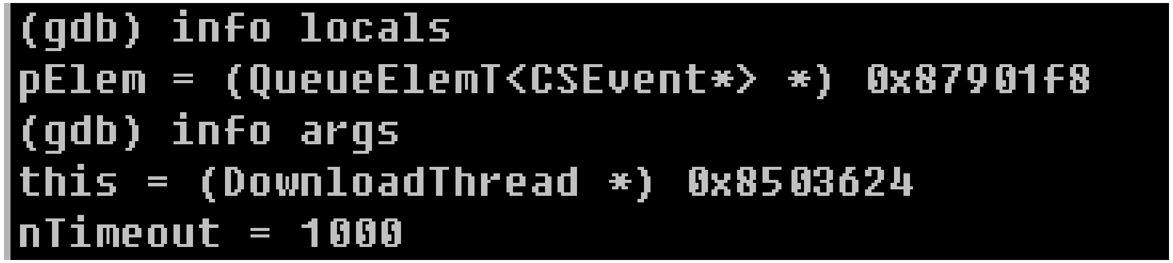
詳細資訊：當前楨地址，上一楨$eip(pc), 函數名，原始檔案名和行號，本楨的$eip，上一楨地址，下一楨位址，源碼語言，參數清單位址，各參數的值，區域變數地址，上一楨的$sp，保存的一些寄存器

（1）函式呼叫棧探密

- C. 查看Stack frame信息

(gdb) **info locals**:查看當前楨中函數的參數相關信息

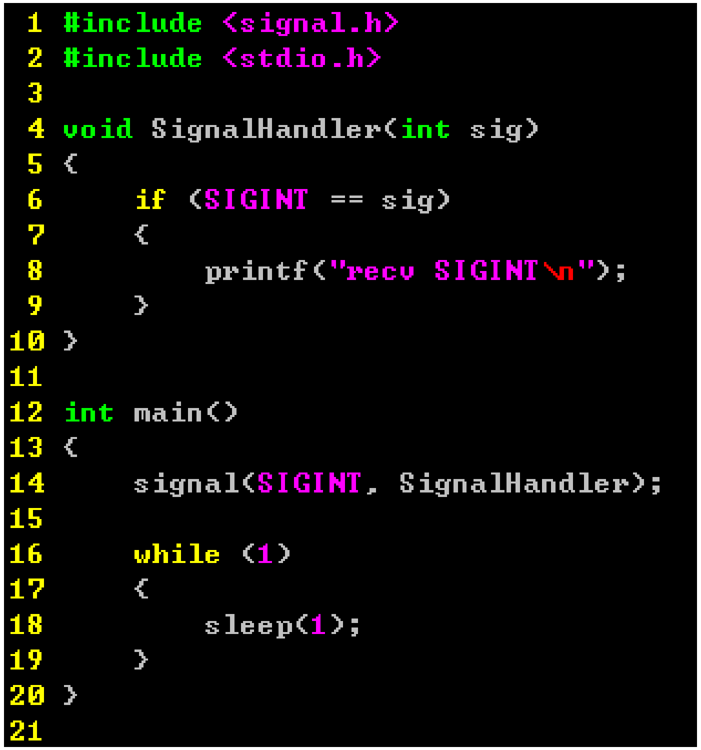
(gdb) **info args**: 查看當前楨中的區域變數相關信息



```
(gdb) info locals
pElem = (QueueElemT<CSEvent*> *) 0x87901f8
(gdb) info args
this = (DownloadThread *) 0x8503624
nTimeout = 1000
```

（2）調試中信號的回應

- GDB可以檢測到應用程式運行時收到的信號，可以通過命令提前設置當收到指定資訊時的處理情況。



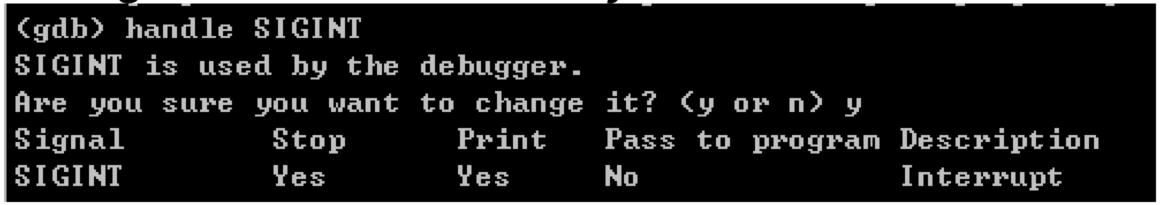
```
#include <signal.h>
#include <stdio.h>

void SignalHandler(int sig)
{
    if (SIGINT == sig)
    {
        printf("recv SIGINT\n");
    }
}

int main()
{
    signal(SIGINT, SignalHandler);

    while (1)
    {
        sleep(1);
    }
}

```

Question—如何在GDB調試這個程式的時候，讓這個程式收到SIGINT信號?

（2）調試中信號的回應

- A. **handle** *signal*

(gdb) **handle** *signal* [*keywords*]: 如果沒指定*keywords*, 該命令查看GDB對*signal*的當前的處理情況；如果指定了*keywords*，則是設置GDB對*signal*的處理屬性， *keywords*就是要設置的屬性



```
(gdb) handle SIGINT
SIGINT is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal        Stop      Print   Pass to program Description
SIGINT        Yes       Yes     No              Interrupt
```

signal: 可以為整數或符號形式的信號名，e.g. SIGINT和2是同一信號

keywords:

print & **noprint**: **print**收到指定的信號,列印出一條資訊; **noprint**與**print**相反

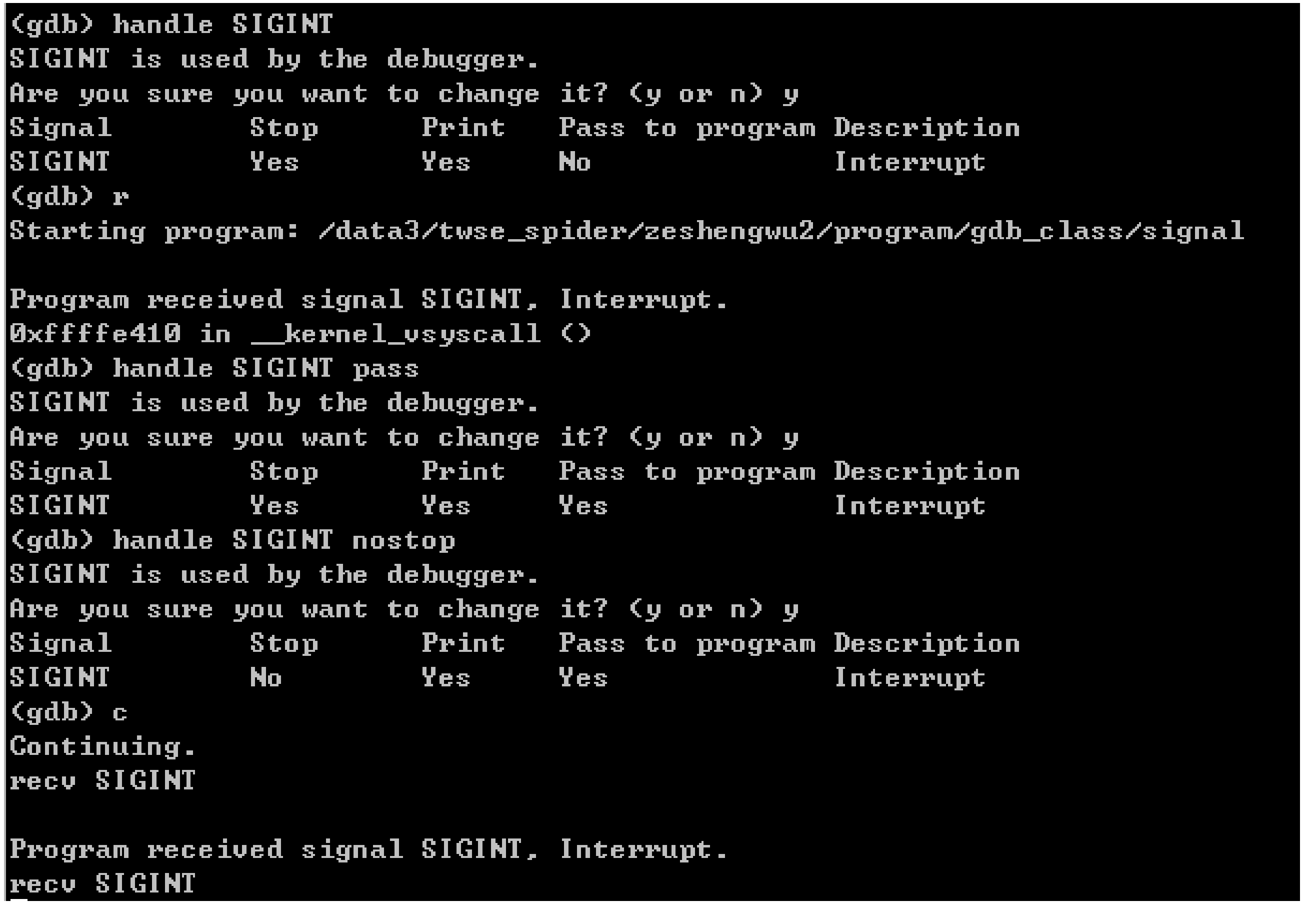
stop & **nostop**: **nostop**表示收到指定的信號，不停止程式的執行，只列印出一條收到信號的消息,因此,**nostop**也暗含**print**, **stop**與**nostop**相反

pass & **nopass**: **pass**表示收到指定的信號，把該信號通知給應用程式; **nopass**與**pass**相反

ignore & **noignore:ingore**與**noignore**分別是**nopass**和**pass**的別名

（2）調試中信號的回應

- A. **handle** *signal*

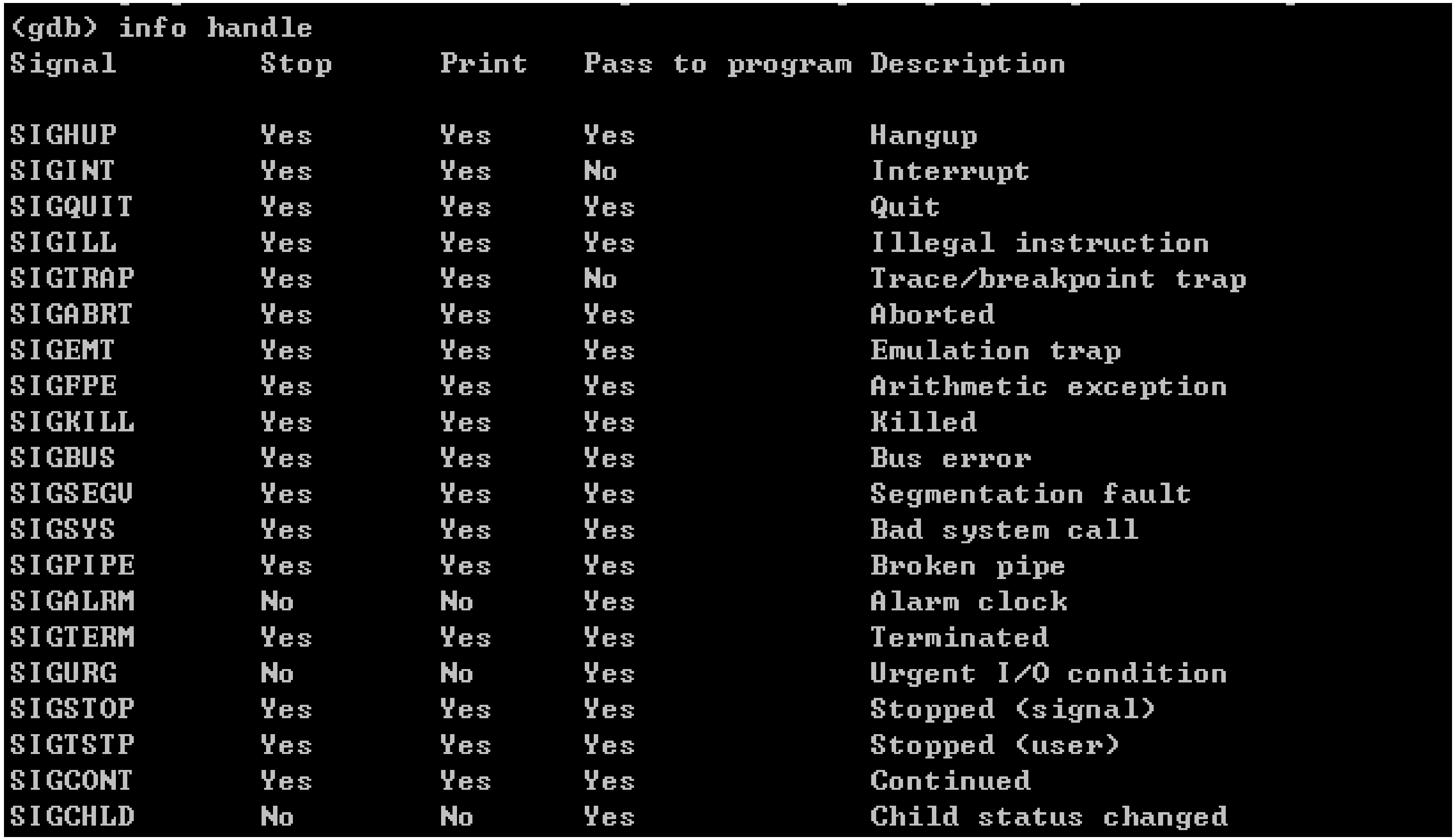


```
(gdb) handle SIGINT
SIGINT is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal        Stop      Print   Pass to program Description
SIGINT        Yes       Yes     No              Interrupt
(gdb) r
Starting program: /data3/twse_spider/zeshengwu2/program/gdb_class/signal

Program received signal SIGINT, Interrupt.
0xffffe410 in __kernel_vsyscall ()
(gdb) handle SIGINT pass
SIGINT is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal        Stop      Print   Pass to program Description
SIGINT        Yes       Yes     Yes             Interrupt
(gdb) handle SIGINT nostop
SIGINT is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal        Stop      Print   Pass to program Description
SIGINT        No        Yes     Yes             Interrupt
(gdb) c
Continuing.
recv SIGINT

Program received signal SIGINT, Interrupt.
recv SIGINT
```

（2）調試中信號的回應

- B. 查看GDB對各種信號的缺省處理

(gdb) **info handle** & (gdb) **info signals**



```
<gdb> info handle
Signal        Stop      Print   Pass to program Description

SIGHUP        Yes       Yes     Yes             Hangup
SIGINT        Yes       Yes     No              Interrupt
SIGQUIT       Yes       Yes     Yes             Quit
SIGILL        Yes       Yes     Yes             Illegal instruction
SIGTRAP       Yes       Yes     No              Trace/breakpoint trap
SIGABRT       Yes       Yes     Yes             Aborted
SIGEMT        Yes       Yes     Yes             Emulation trap
SIGFPE        Yes       Yes     Yes             Arithmetic exception
SIGKILL       Yes       Yes     Yes             Killed
SIGBUS        Yes       Yes     Yes             Bus error
SIGSEGV       Yes       Yes     Yes             Segmentation fault
SIGSYS        Yes       Yes     Yes             Bad system call
SIGPIPE       Yes       Yes     Yes             Broken pipe
SIGALRM       No        No      Yes             Alarm clock
SIGTERM       Yes       Yes     Yes             Terminated
SIGURG        No        No      Yes             Urgent I/O condition
SIGSTOP       Yes       Yes     Yes             Stopped (signal)
SIGTSTP       Yes       Yes     Yes             Stopped (user)
SIGCONT       Yes       Yes     Yes             Continued
SIGCHLD       No        No      Yes             Child status changed
```

（3）修改程式運行、源碼

- A. 修改程式的運行

(gdb) **print** *v=value*: 修改變數*v*的值並列印修改後的值

(gdb) **set [var]** *v=value*: 修改變數*v*的值，如果*v*與GDB的某個屬性名一樣的話，需要在前面加**var**關鍵字

e.g. (gdb) **set var** print=1

(gdb) **whatis** *v*: 查看變數*v*的類型

(gdb) **signal** *sig*: 把信號*sig*發給被調試的程式

(gdb) **return** [*expression*]: 中止當前函數的執行，返回*expression*值

(gdb) **finish**: 結束當前函數的執行，列印出返回值

(gdb) **call** *function*：調用程式中的函數*function*

（3）修改程式運行、源碼

- B. 修改源碼

1）設置環境變數: export EDITOR=/usr/bin/vim

2） (gdb) **edit**: 編輯當前檔

3） (gdb) **edit** *number*: 編輯當前檔的第*number*行

4） (gdb) **edi**

5） (gd *umber* 行

6） (gd 件的*function*函數

回憶— 結合我們前面介紹的shell, make, kill和本節的edit命令，我們完全可以直接在GDB中完成很多的工作！

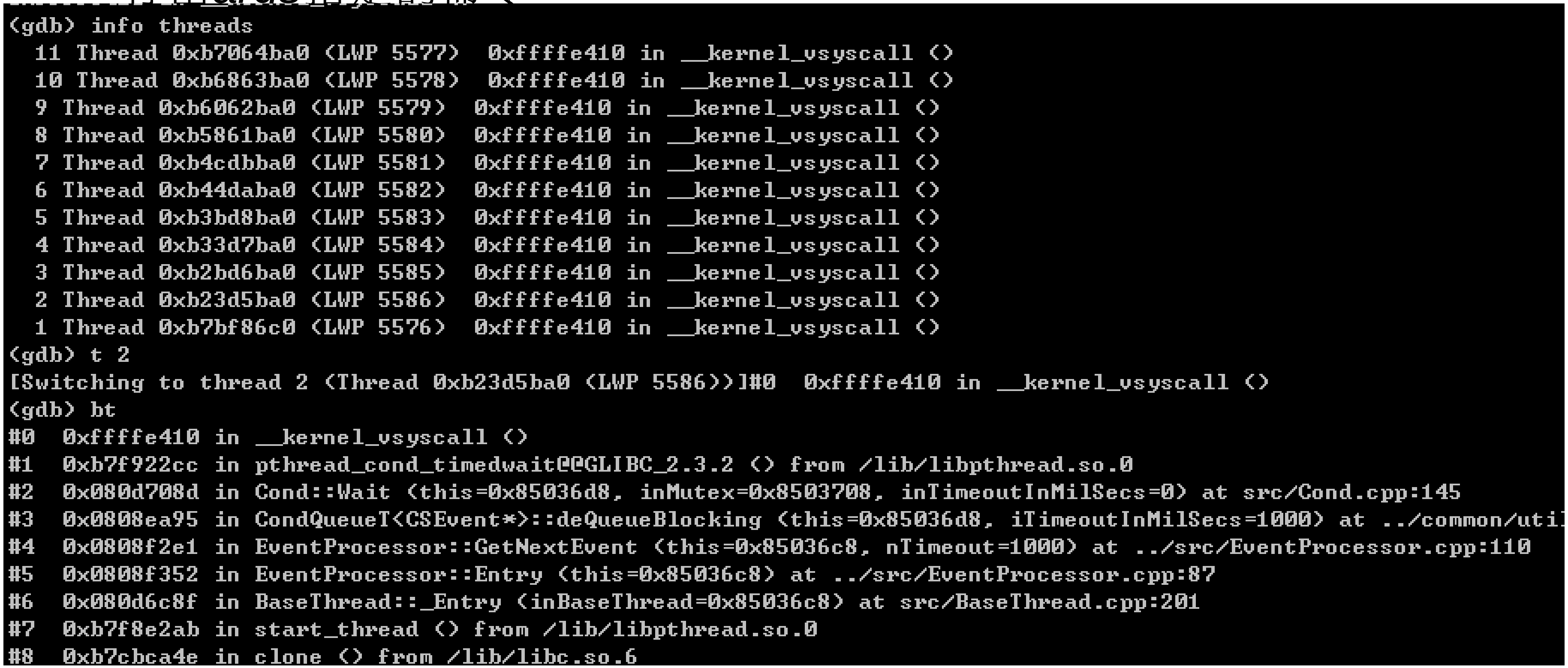
（4）多執行緒調試

- A. 基本命令

(gdb) **info threads**：查看GDB當前調試的程式的各個執行緒的相關資訊

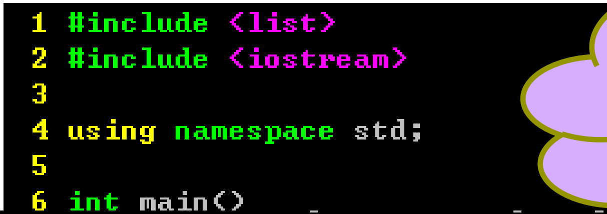
(gdb) **thread** *threadno*：切換當前執行緒到由**threadno**指定的執行緒

(gdb) **thread apply** [*threadno*] [**all**] *args*：對指定（或所有）的執行緒執行由*args*指定的命令



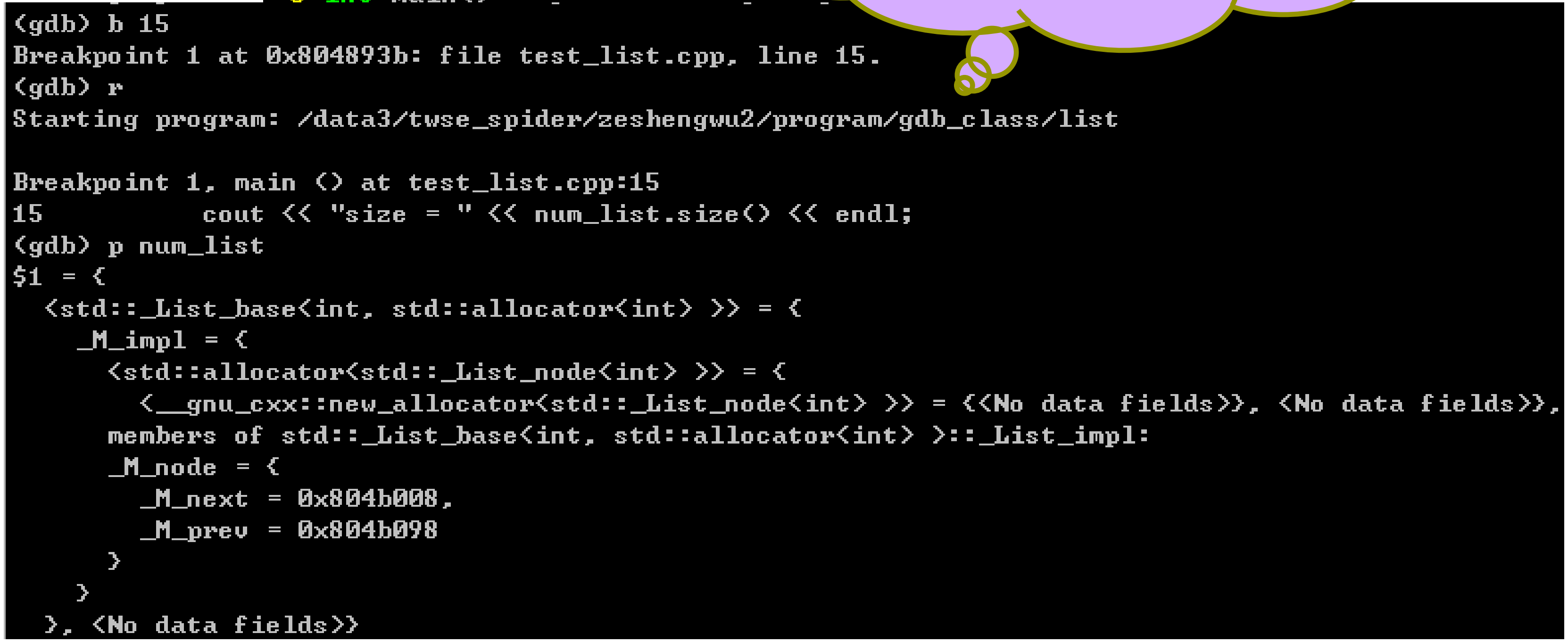
```
(gdb) info threads
  11 Thread 0xb7064ba0 (LWP 5577)  0xffffe410 in __kernel_vsyscall ()
  10 Thread 0xb6863ba0 (LWP 5578)  0xffffe410 in __kernel_vsyscall ()
  9 Thread 0xb6062ba0 (LWP 5579)  0xffffe410 in __kernel_vsyscall ()
  8 Thread 0xb5861ba0 (LWP 5580)  0xffffe410 in __kernel_vsyscall ()
  7 Thread 0xb4cdbba0 (LWP 5581)  0xffffe410 in __kernel_vsyscall ()
  6 Thread 0xb44daba0 (LWP 5582)  0xffffe410 in __kernel_vsyscall ()
  5 Thread 0xb3bd8ba0 (LWP 5583)  0xffffe410 in __kernel_vsyscall ()
  4 Thread 0xb33d7ba0 (LWP 5584)  0xffffe410 in __kernel_vsyscall ()
  3 Thread 0xb2bd6ba0 (LWP 5585)  0xffffe410 in __kernel_vsyscall ()
  2 Thread 0xb23d5ba0 (LWP 5586)  0xffffe410 in __kernel_vsyscall ()
  1 Thread 0xb7bf86c0 (LWP 5576)  0xffffe410 in __kernel_vsyscall ()
(gdb) t 2
[Switching to thread 2 (Thread 0xb23d5ba0 (LWP 5586))]#0  0xffffe410 in __kernel_vsyscall ()
(gdb) bt
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7f922cc in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x080d708d in Cond::Wait (this=0x85036d8, inMutex=0x8503708, inTimeoutInMilSecs=0) at src/Cond.cpp:145
#3  0x0808ea95 in CondQueueT<CSEvent*>::deQueueBlocking (this=0x85036d8, iTimeoutInMilSecs=1000) at ../common/uti
#4  0x0808f2e1 in EventProcessor::GetNextEvent (this=0x85036c8, nTimeout=1000) at ../src/EventProcessor.cpp:110
#5  0x0808f352 in EventProcessor::Entry (this=0x85036c8) at ../src/EventProcessor.cpp:87
#6  0x080d6c8f in BaseThread::_Entry (inBaseThread=0x85036c8) at src/BaseThread.cpp:201
#7  0xb7f8e2ab in start_thread () from /lib/libpthread.so.0
#8  0xb7cbca4e in clone () from /lib/libc.so.6
```

（5）自訂命令



```
#include <list>
#include <iostream>

using namespace std;

int main()
```

Question—如何在GDB查看這個list裡面的每個元素?



```
(gdb) b 15
Breakpoint 1 at 0x804893b: file test_list.cpp, line 15.
(gdb) r
Starting program: /data3/twse_spider/zeshengwu2/program/gdb_class/list

Breakpoint 1, main () at test_list.cpp:15
15            cout << "size = " << num_list.size() << endl;
(gdb) p num_list
$1 = {
  <std::_List_base<int, std::allocator<int> >> = {
    _M_impl = {
      <std::allocator<std::_List_node<int> >> = {
        <__gnu_cxx::new_allocator<std::_List_node<int> >> = {<No data fields>}, <No data fields>},
      members of std::_List_base<int, std::allocator<int> >::_List_impl:
      _M_node = {
        _M_next = 0x804b008,
        _M_prev = 0x804b098
      }
    }
  }, <No data fields>}
```

（5）自訂命令

- A. 自訂命令基本語法

1）定義一個命令

```
define commandname
 …
 end
```

2）條件陳述式：

```
if cond-expr
…
else
…
end
```

3）迴圈語句：

```
while cond-expr
…
end
```

4）定義一個命令的文檔資訊，在 **help** *commandname*的時候可以顯示：

```
document commandname
 …
 end
```

5) *$arg0…$arg9*：表示命令列參數，最多10個

（5）自訂命令

- B. 查看用戶自訂命令

(gdb) **help user-defined**：查看所有的用戶自訂命令

(gdb) **show user** *commandname*：查看自訂命令*commandname*的定義

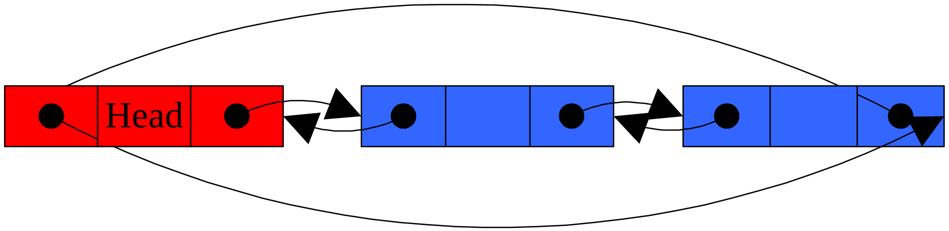
(gdb) **help** *commandname*：查看自訂命令*commandname*的說明資訊

(gdb) **show max-user-call-depth**：查看用戶自訂命令的最大遞迴呼叫深度，缺省是1024

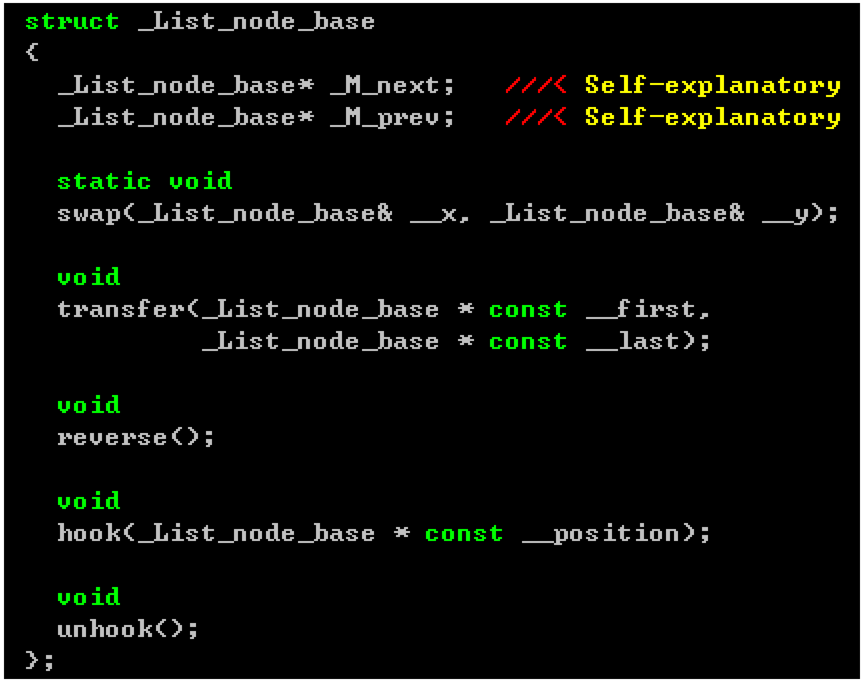
(gdb) **set max-user-call-depth**：設置用戶自訂命令的最大遞迴呼叫深度

（5）自訂命令

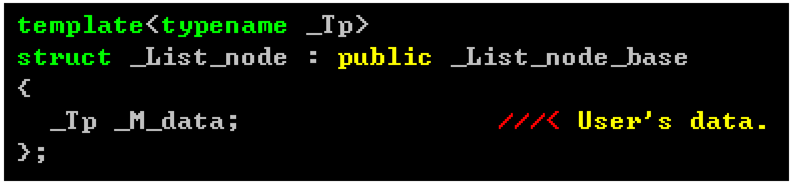
- C. plist實現



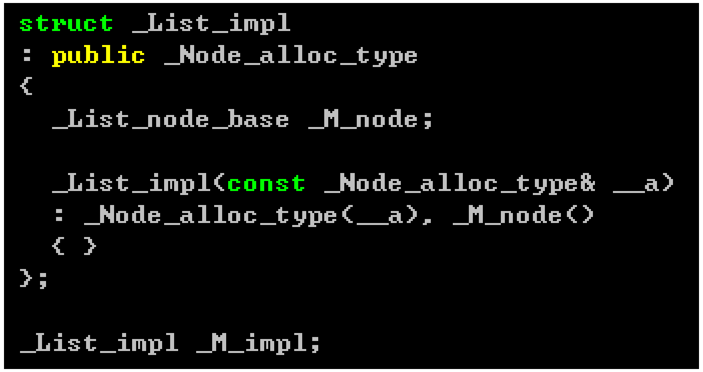
/usr/include/c++/4.1.2/bits/stl_list.h



```
struct _List_node_base
{
  _List_node_base* _M_next;   ///< Self-explanatory
  _List_node_base* _M_prev;   ///< Self-explanatory

  static void
  swap(_List_node_base& __x, _List_node_base& __y);

  void
  transfer(_List_node_base * const __first,
           _List_node_base * const __last);

  void
  reverse();

  void
  hook(_List_node_base * const __position);

  void
  unhook();
};
```



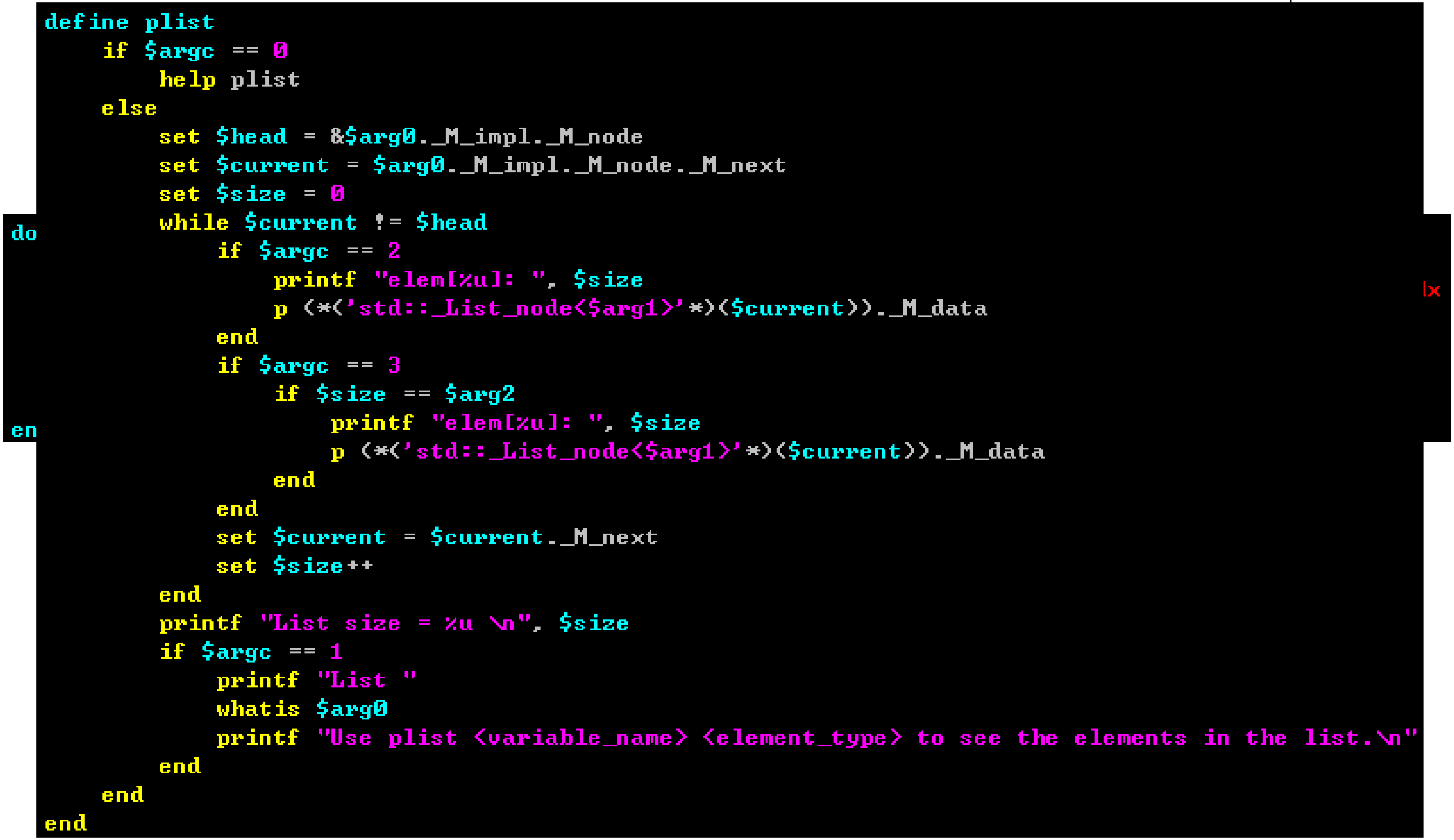
```
template<typename _Tp>
struct _List_node : public _List_node_base
{
  _Tp _M_data;                  ///< User's data.
};
```



```
struct _List_impl
: public _Node_alloc_type
{
  _List_node_base _M_node;

  _List_impl(const _Node_alloc_type& __a)
  : _Node_alloc_type(__a), _M_node()
  { }
};

_List_impl _M_impl;
```

（5）自訂命令

- C. plist實現



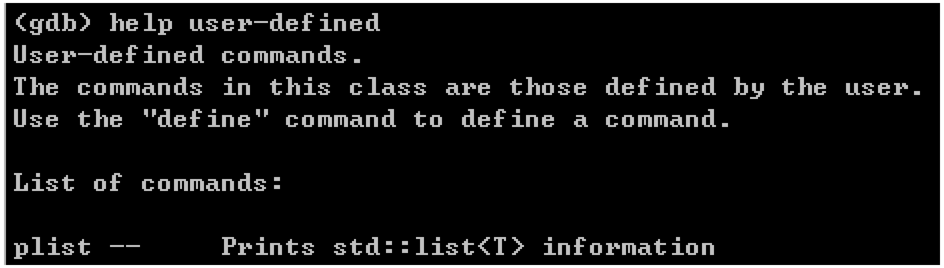
```
define plist
    if $argc == 0
        help plist
    else
        set $head = &$arg0._M_impl._M_node
        set $current = $arg0._M_impl._M_node._M_next
        set $size = 0
        while $current != $head
            if $argc == 2
                printf "elem[%u]: ", $size
                p (*('std::_List_node<$arg1>'*)($current))._M_data
            end
            if $argc == 3
                if $size == $arg2
                    printf "elem[%u]: ", $size
                    p (*('std::_List_node<$arg1>'*)($current))._M_data
                end
            end
            set $current = $current._M_next
            set $size++
        end
        printf "List size = %u \n", $size
        if $argc == 1
            printf "List "
            whatis $arg0
            printf "Use plist <variable_name> <element_type> to see the elements in the list.\n"
        end
    end
end
```

（5）自訂命令

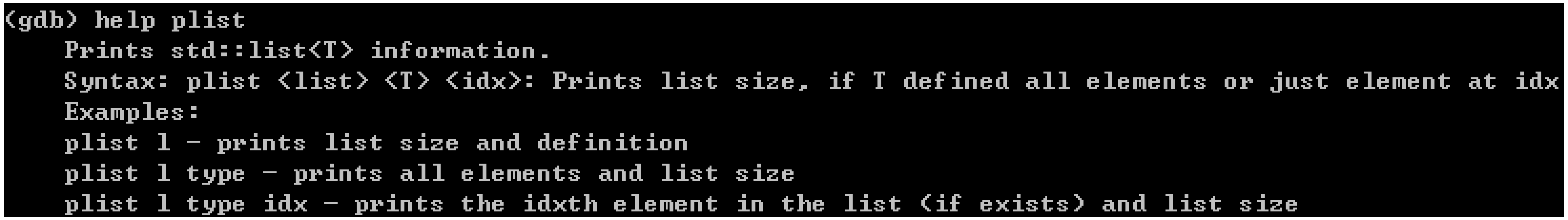
- C. plist實現

1)將plist的實現放到~/.gdbinit文件中

2）



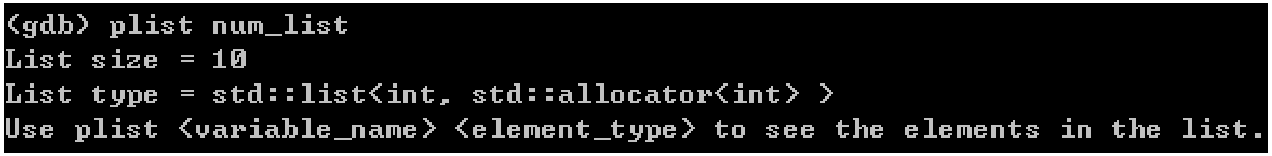
```
(gdb) help user-defined
User-defined commands.
The commands in this class are those defined by the user.
Use the "define" command to define a command.

List of commands:

plist --      Prints std::list<T> information
```



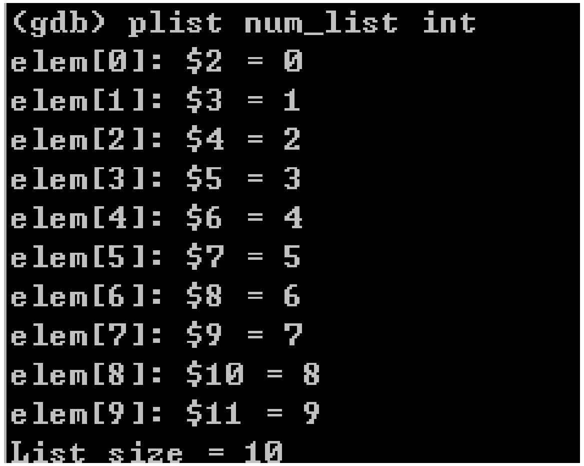
```
(gdb) help plist
    Prints std::list<T> information.
    Syntax: plist <list> <T> <idx>: Prints list size, if T defined all elements or just element at idx
    Examples:
    plist l - prints list size and definition
    plist l type - prints all elements and list size
    plist l type idx - prints the idxth element in the list (if exists) and list size
```

（5）自訂命令

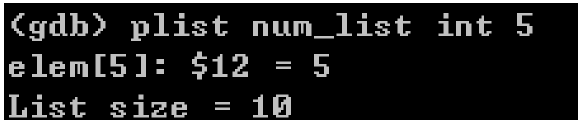
- C. plist實現



```
(gdb) plist num_list
List size = 10
List type = std::list<int, std::allocator<int> >
Use plist <variable_name> <element_type> to see the elements in the list.
```



```
(gdb) plist num_list int
elem[0]: $2 = 0
elem[1]: $3 = 1
elem[2]: $4 = 2
elem[3]: $5 = 3
elem[4]: $6 = 4
elem[5]: $7 = 5
elem[6]: $8 = 6
elem[7]: $9 = 7
elem[8]: $10 = 8
elem[9]: $11 = 9
List size = 10
```



```
(gdb) plist num_list int 5
elem[5]: $12 = 5
List size = 10
```

4.學而時習之---總結回顧

- （1）常見的coredump原因

a. Signal 6(SIGABRT):

New失敗：記憶體洩露造成記憶體不夠

Delete失敗：多次delete同一塊記憶體

應用程式拋出的異常

b. Signal 11(SIGSEGV): 多為記憶體越界，訪問已經被delete掉的記憶體

c. Signal 13(SIGPIPE): 寫已經被刪除的檔，寫對方已經關閉的socket

- （2）參考資料

http://www.gnu.org/software/gdb/documentation/

《The Art of Assembly Language》

《Understanding the Linux Kernel》

《程式師的自我修養---連結、裝載與庫》

Thank you!