手把手教你玩转GDB

Zesheng Wu (武泽胜)
<wuzesheng@vip.qq.com>

2010.11

主要内容

- 1. 温故知新---程序的秘密
- 2. 牛刀小试---GDB初探
- 3. 大显身手---玩转GDB
- 4. 学而时习之---总结回顾

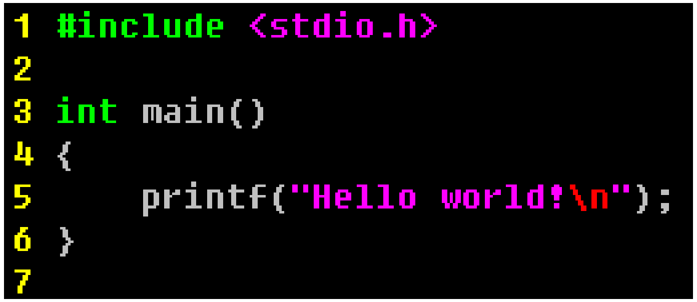
1. 温故知新---程序的秘密

- （1）Declaration
- （2）GCC做了什么
- （3）进程地址空间

（1）Declaration

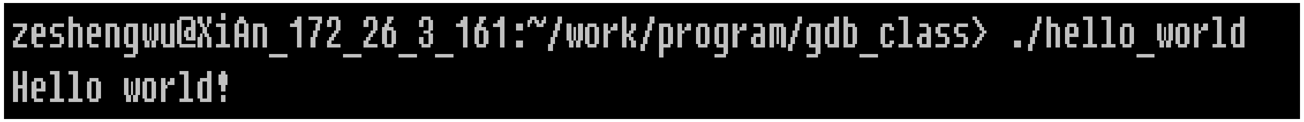
- 本课程所讲内容都是基于x86 32位平台, 在64位平台上某些内容可能会略有差别，请大家注意区别！

（2）GCC做了什么



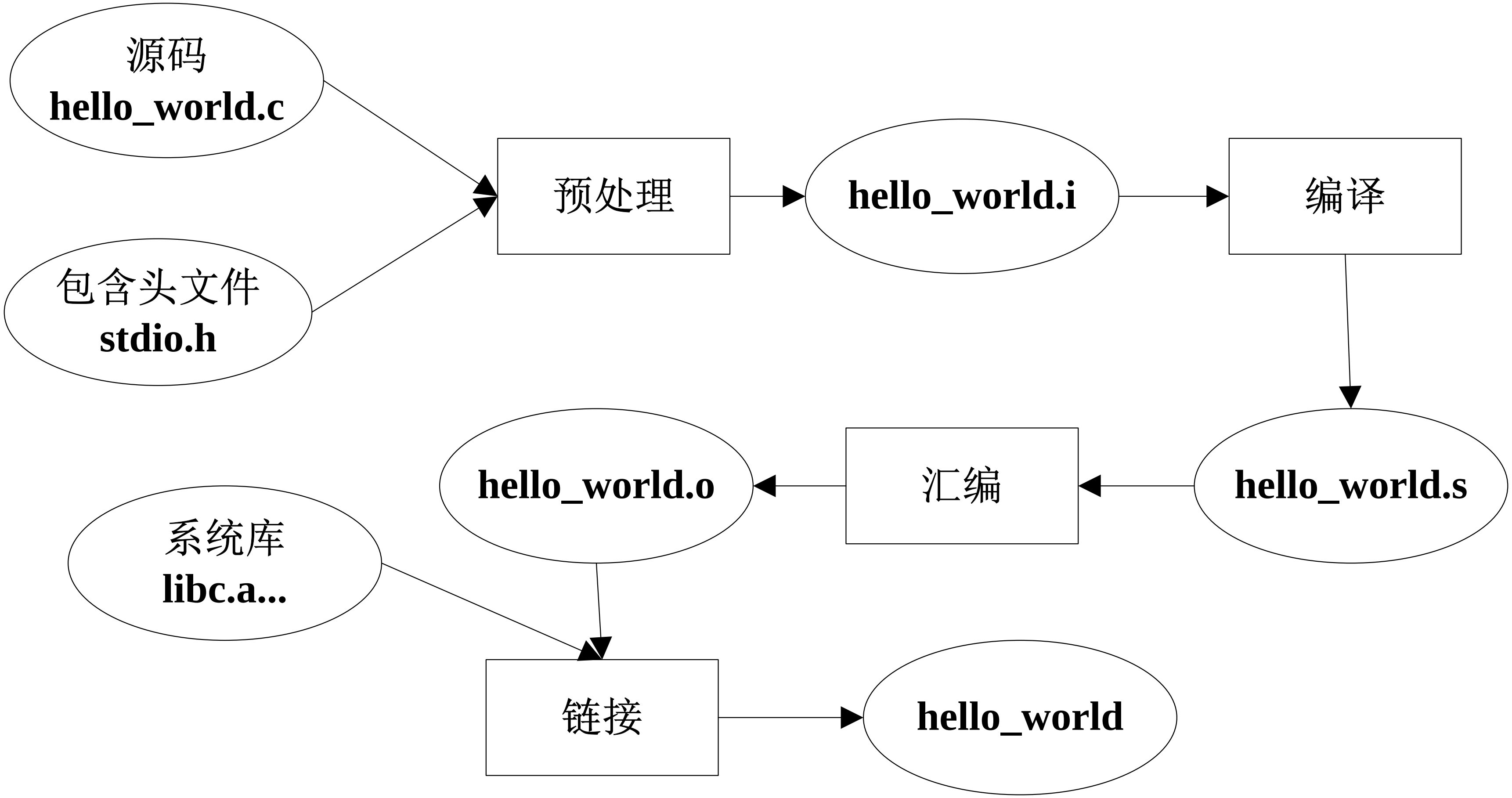
```c
#include <stdio.h>

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!
```

（2）GCC做了什么


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

（2）GCC做了什么

- A. 预处理

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

任务：展开宏，替换头文件，删除注释

- B. 编译

gcc –S

任 代码

总结—GCC实际上只是对多个工具的包装，它会根据不同的参数，去调用cpp、ccl(cclplus)、as或者ld去完成程序编译过程中的一系列工作

- C. 汇编

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

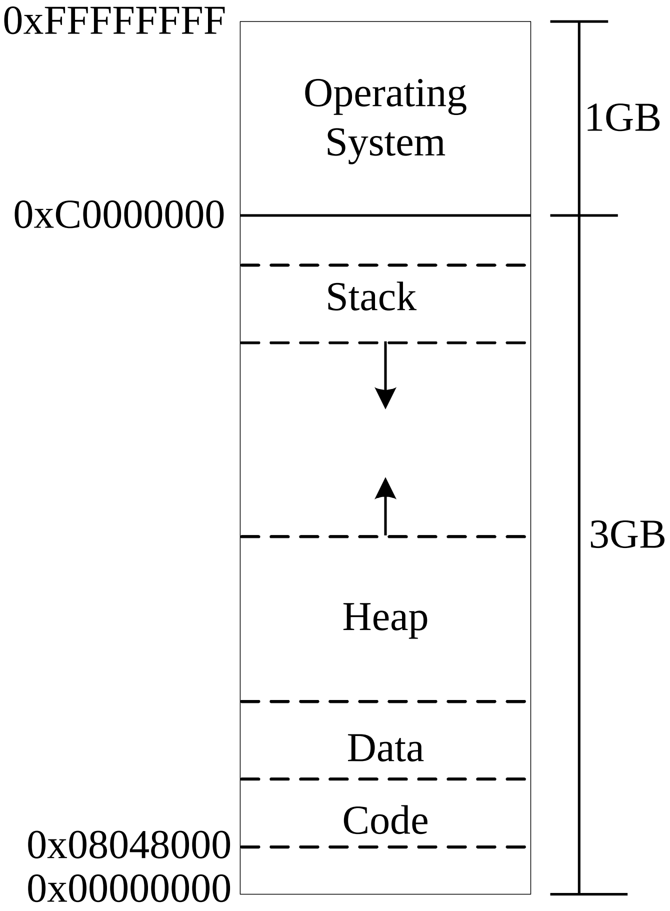
任务：将汇编代码转换成为机器可以执行指令

- D. 链接

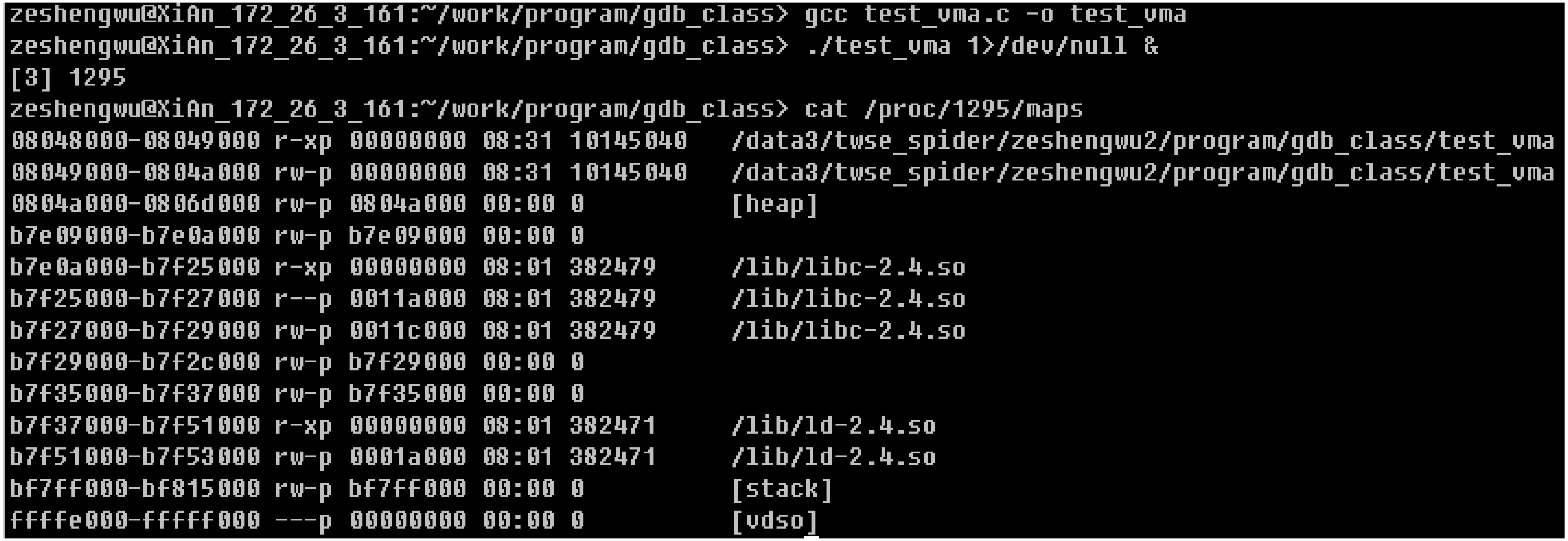
gcc hello_world.o –o hello_world(调用ld完成)

任务：地址和空间分配，符号决议定位，将目标文件拼装成可执行文件

（3）进程地址空间


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

（3）进程地址空间



```
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.牛刀小试---GDB初探

- （1）启动GDB开始调试
- （2）常用调试命令介绍
- （3）退出GDB结束调试
- （4）寻求帮助

（1）启动GDB开始调试

- A.准备工作

编译调试版本的可执行程序(gcc加上-g参数即可,注意不要调试加-O相关的选项)

- B.冷启动

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.热启动

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

- D. 命令行参数

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

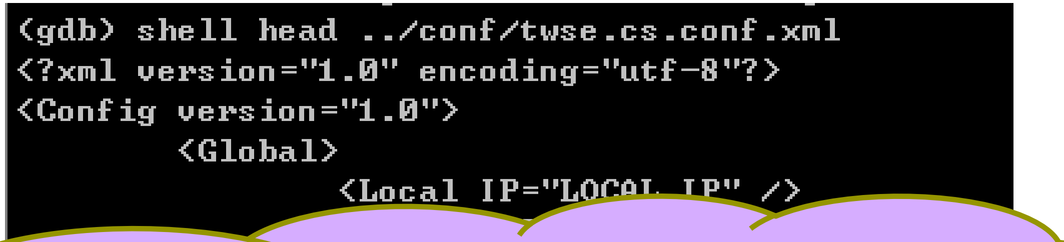
(gdb) **set args** *arglist*

(gdb) **run** *arglist*

（2）常用调试命令介绍

- A. 在GDB中执行shell命令

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



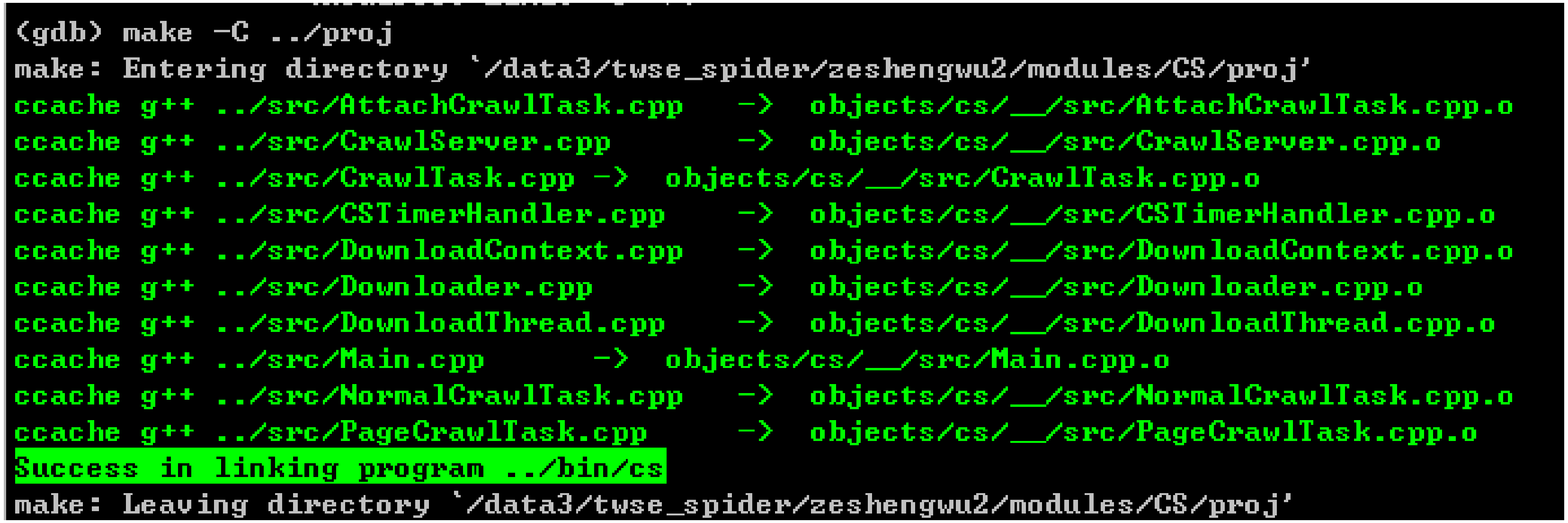
```
(gdb) shell head ../conf/twse.cs.conf.xml
<?xml version="1.0" encoding="utf-8"?>
<Config version="1.0">
        <Global>
                <Local IP="LOCAL_IP" />
```

shell小技巧—可以在GDB中直接执行shell命令，这样就会暂时退出GDB, 回到shell终端， 在shell执行完*command*后，然后在shell中执行exit命令，便可回到GDB

（2）常用调试命令介绍

- B. 在GDB中调用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）常用调试命令介绍

- C. 断点(Breakpoints)

a. 设置断点：

(gdb) **break** *function*: 在函数*funtion*入口处设置断点

(gdb) **break** *linenum*: 在当前源文件的第*linenum*行处设置断点

(gdb) **break** *filename*:*linenum*: 在名为*filename*的源文件的第*linenum*行处设置断点

(gdb) **break** *filename*:*function*: 在名为*filename*的源文件中的*function*函数入口处设置断点

(gdb) **break** *args* **if** *cond*: *args* 为上面讲到的任意一种参数，在指定位置设置一个断点，当且仅但*cond*为**true**时，该断点 生效

(gdb) **tbreak** *args*: 设置一个只停止一次的断点, *args*与**break**命令的一样。这样的断点当第一次停下来后，就会立即被删除

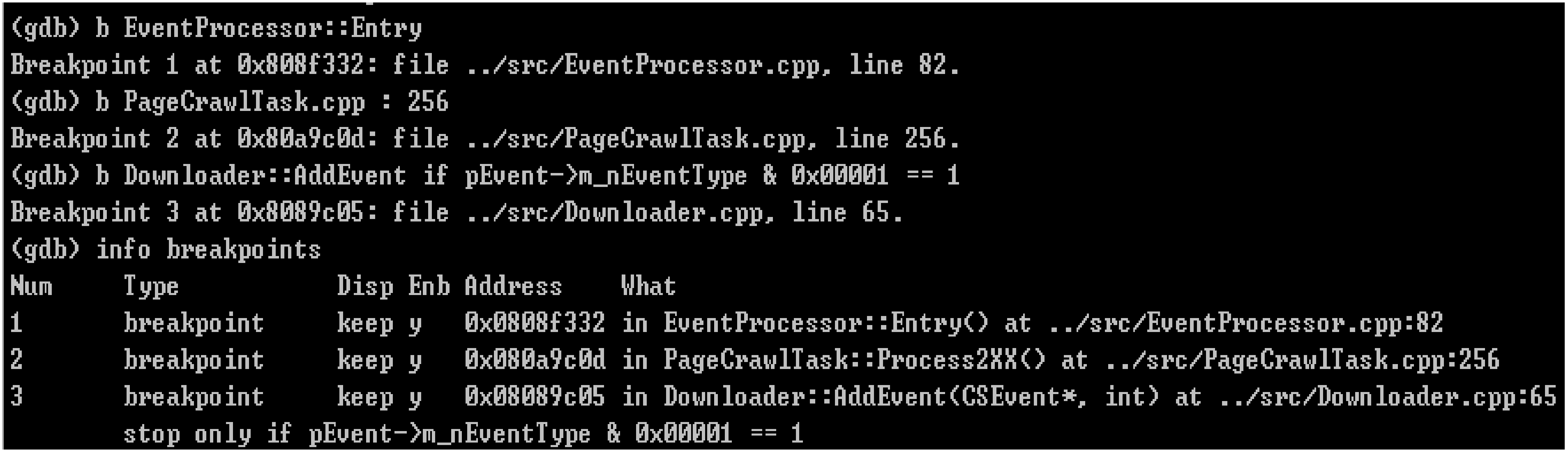
(gdb) **rbreak** *regex*: 在所有符合正则表达式*regex*的**函数**处设置breakpoint

（2）常用调试命令介绍

- C. 断点(Breakpoints)

b. 查看断点属性：

(gdb) **info breakpoints** [*n*]:查看第*n*个断点的相关信息，如果没有指定*n*，则显示所有断点的相关信息



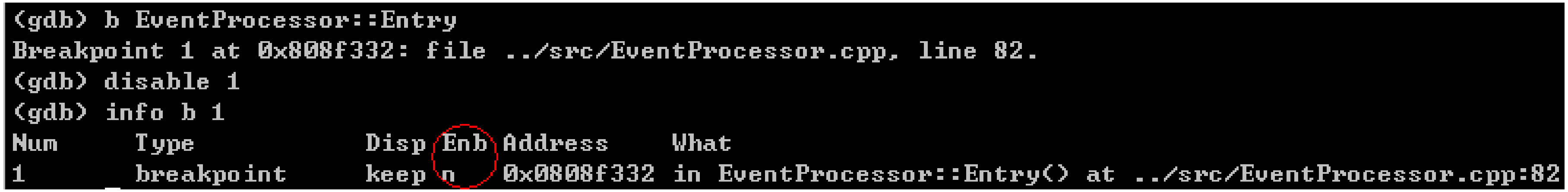
```
(gdb) b EventProcessor::Entry
Breakpoint 1 at 0x808f332: file ../src/EventProcessor.cpp, line 82.
(gdb) b PageCrawlTask.cpp : 256
Breakpoint 2 at 0x80a9c0d: file ../src/PageCrawlTask.cpp, line 256.
(gdb) b Downloader::AddEvent if pEvent->m_nEventType & 0x00001 == 1
Breakpoint 3 at 0x8089c05: file ../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）常用调试命令介绍

- C. 断点(Breakpoints)

c. 断点禁用/启用：

<span style="color:red">(gdb)</span> **disable [breakpoints]** [*range*…]: 禁用由*range*指定的范围内的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*…]: 启用由*range*指定的范围内的breakpoints

(gdb) **enable [breakpoints] once** [*range*…]: 只启用一次由*range*指定的范围内的breakpoints，等程序停下来后，自动设为禁用

(gdb) **enable [breakpoints] delete** [*range*…]: 启用*range*指定的范围内的breakpoints，等程序停下来后，这些breakpoints自动被删除

（2）常用调试命令介绍

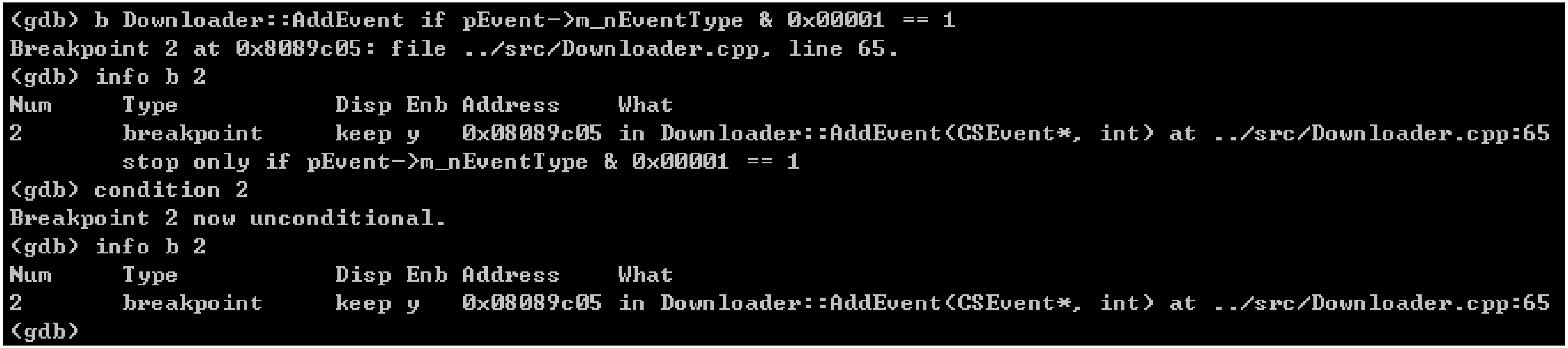
- C. 断点(Breakpoints)

d. 条件断点：

(gdb) **break** *args* **if** *cond*: 设置条件断点

(gdb) **condition** *bnum* [*cond-expr*]: 当指定*cond-expr*时，给第*bnum*个断点设置条件；当未指定*cond-expr*时，取消第*bnum*个断点的条件

(gdb) **ignore** *bnum count*: 忽略第*bnum*个断点*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）常用调试命令介绍

- C. 断点(Breakpoints)

e. 在断点处自动执行命令

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

在第*bnum*个断点处停下来后，执行由*command-list*指定的命令串，如果没有指定*bnum*，则对最后一个断点生效

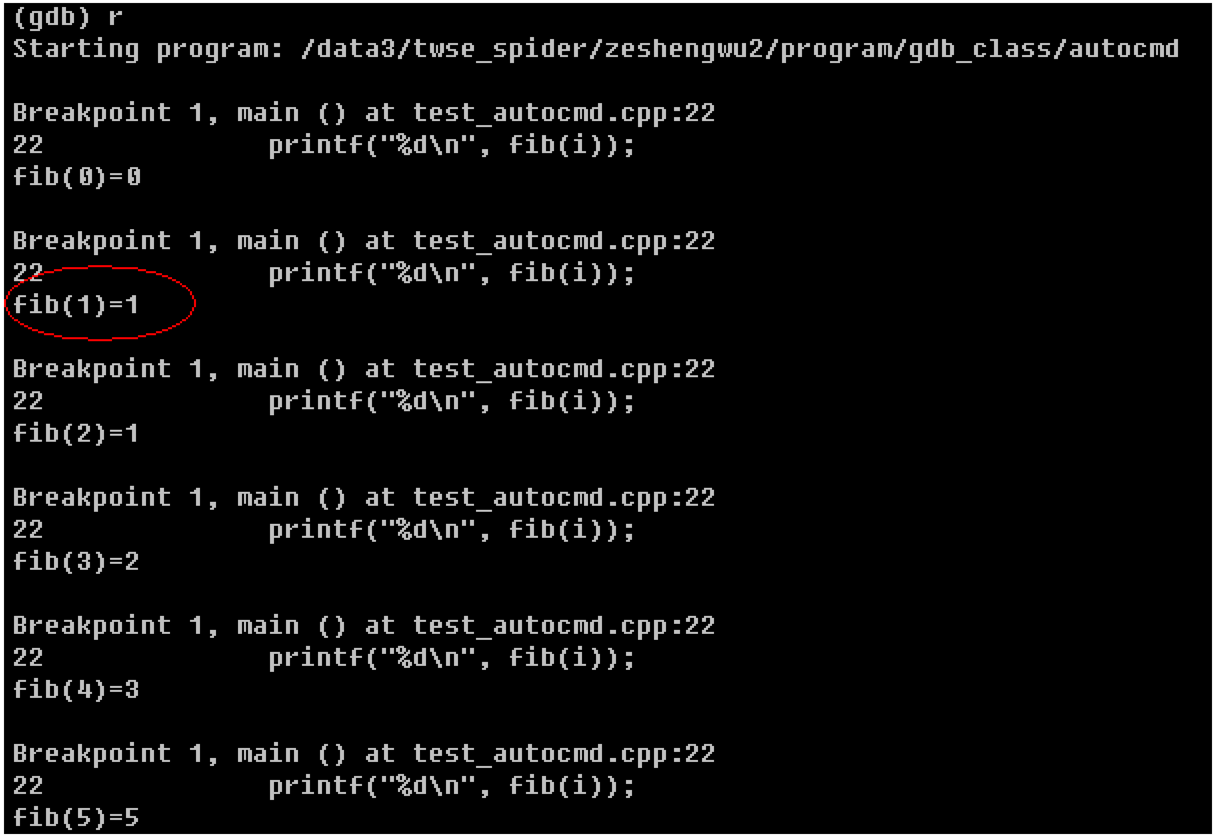
```
(gdb) commands [bnum]
         end
```

取消第*bnum*个断点处的命令列表

（2）常用调试命令介绍

- C. 断点(Breakpoints)

e. 在断点处自动执行命令



```
(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）常用调试命令介绍

- C. 断点(Breakpoints)

f. 清理断点：

(gdb) **clear** *function* & **clear** *filename:function*: 清除函数*function*入口处的断点

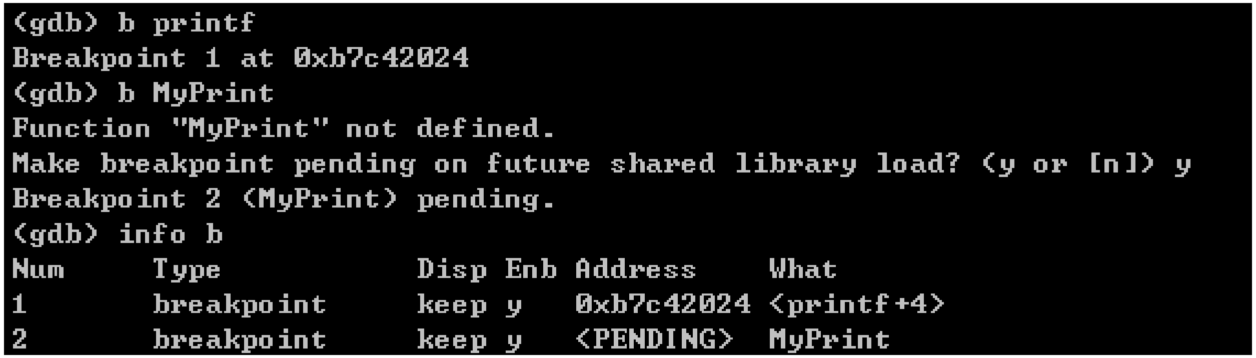
(gdb) **clear** *linenum* & **clear** *filename:linenum*: 清除第*linenum*行处的断点

(gdb) **delete** [**breakpoints**] [*range*…]: 删除由*range*指定的范围内的breakpoints，*range*范围是指断点的序列号的范围

（2）常用调试命令介绍

- C. 断点(Breakpoints)

g. 未决的断点—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缺省设置，询问用户是否要设置pending breakpoint

(gdb) **set breakpoint pending on**: GDB当前不能识别的breakpoint自动成为pending breakpoint

(gdb) **set breakpoint pending off**: GDB当前不能识别某个breakpoint时，直接报错

(gdb) **show breakpoint pending**: 查看GDB关于pending breakpoint的设置的行为(auto, on, off)

（2）常用调试命令介绍

- C. 断点(Breakpoints)

h. Watchpoints和Catchpoints：

1）Watchpoint的作用是让程序在某个表达式的值发生变化的时候停止运行，达到‘监视’该表达式的目的

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

2）Catchpoints的作用是让程序在发生某种事件的时候停止运行，比如C++中发生异常事件，加载动态库事件，系统调用事件

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

3）Watchpoints和Catchpoints都与Breakpoints很相像，都有enable/disabe/delete等操作，使用方法也与breakpoints的类似

（2）常用调试命令介绍

- D. 单步调试

a. 设置断点（参见前面《C.断点》一节）

b. next & nexti

(gdb) **next** [*count*]：如果没有指定*count*, 单步执行下一行程序；如果指定了*count*，单步执行接下来的*count*行程序

(gdb) **nexti** [*count*]：如果没有指定count, 单步执行下一条指令；如果指定了count, 单步执行接下来的count条指令

c. step & stepi

(gdb) **step** [*count*]：如果没有指定*count*, 则继续执行程序，直到到达与当前源文件行不同的行时停止执行；如果指定了*count*, 则重复行上面的过程*count*次

（2）常用调试命令介绍

- D. 单步调试

c. step & s

(gdb) ste 然

后停止；如

nexti和stepi的区别--nexti在执行某机器指令时，如果该指令是函数调用，那么程序执行直到该函数调用结束时才停止

d. continue

(gdb) **continue** [*ignore-count*]：唤醒程序，继续运行，至到遇到下一个断点，或者程序结束。如果指定ignore-count，那么程序在接下来的运行中，忽略ignore-count次断点。

e. finish & return

(gdb) **finish**：继续执行程序，直到当前被调用的函数结束，如果该函数有返回值，把返回值也打印到控制台

(gdb) **return** [*expr*]：中止当前函数的调用，如果指定了*expr*，把*expr*的值当做当前函数的返回值；如果没有，直接结束当前函数调用

（2）常用调试命令介绍

- E. 变量与内存查看

a. print：查看变量

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

f：x --- 16进制整数　d --- 10进制整数　u ---10进制无符号整数

o --- 8进制整数　t --- 2进制整数　a --- 地址　c --- 字符　f --- 浮点数

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** : 打印数组*array*的前*len*个元素

3) (gdb) **p** *file*::*variable*：打印文件*file*中的变量*variable*

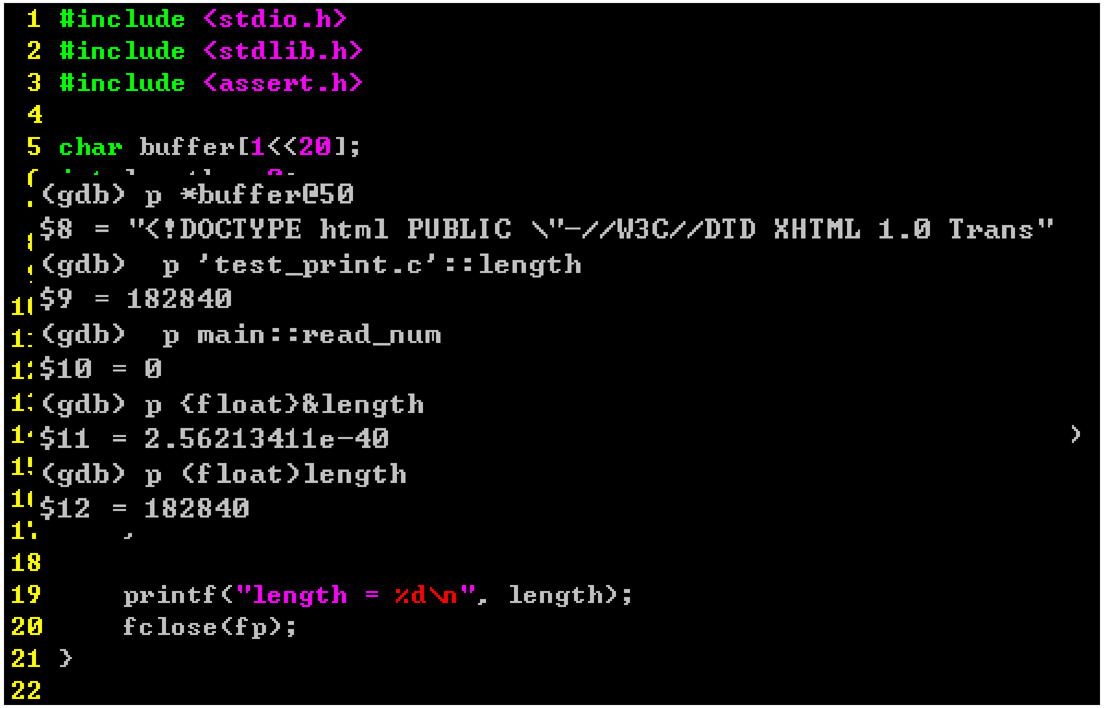
4) (gdb) **p** *function*::*variable*: 打印函数*function*中的变量*variable*

5) (gdb) **p** {*type*}*address*:把*address*指定的内存解释为*type*类型（类似于强制转型，更加强）

（2）常用调试命令介绍

- E. 变量与内存查看

a. print：查看变量



```
#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）常用调试命令介绍

- E. 变量与内存查看

b. x：查看内存

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

n: 重复次数，缺省是1

f: 打印的格式，除了print支持的格式外，还支持如下格式：

s--- C风格字符串，i---机器指令

缺省格式是x

u: 打印的单位大小，支持如下单位：

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

（2）常用调试命令介绍

- E. 变量与内存查看

c. display: 自动打印

(gdb) **display** */f expr*|*addr*: 以格式*f*，自动打印表达式*expr*或地址*addr*

(gdb) **undisplay** *dnums*: 删除掉指定的自动打印点, dnums可以为一个或者多个自动打印点的序号

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

(gdb) **disable display** *dnums*: 禁用由dnums指定的自动打印点

(gdb) **enable display** *dnums*: 启用由dnums指定的自动打印点

(gdb) **info display**: 查看当前所有自动打印点相关的信息

（2）常用调试命令介绍

- E. 变量与内存查看

d. 打印相关属性

基本用法：

(gdb) **set print** *field* **[on]**：打开*field*指定的属性

(gdb) **set print** *field* **off**：关闭*field*指定的属性

(gdb) **show print** *field* **：** 查看*filed*指定的属性的相关设置

相关属性：

1) (gdb) **set print array**：以一种比较好看的方式打印数组，缺省是关闭的

2) (gdb) **set print elements** *num-of-elements*：设置GDB打印数据时显示元素的个数，缺省为200，设为0表示不限制(unlimited)

3) (gdb) **set print null-stop**：设置GDB打印字符数组的时候，遇到NULL时停止，缺省是关闭的

（2）常用调试命令介绍

- E. 变量与内存查看

d. 打印相关属性

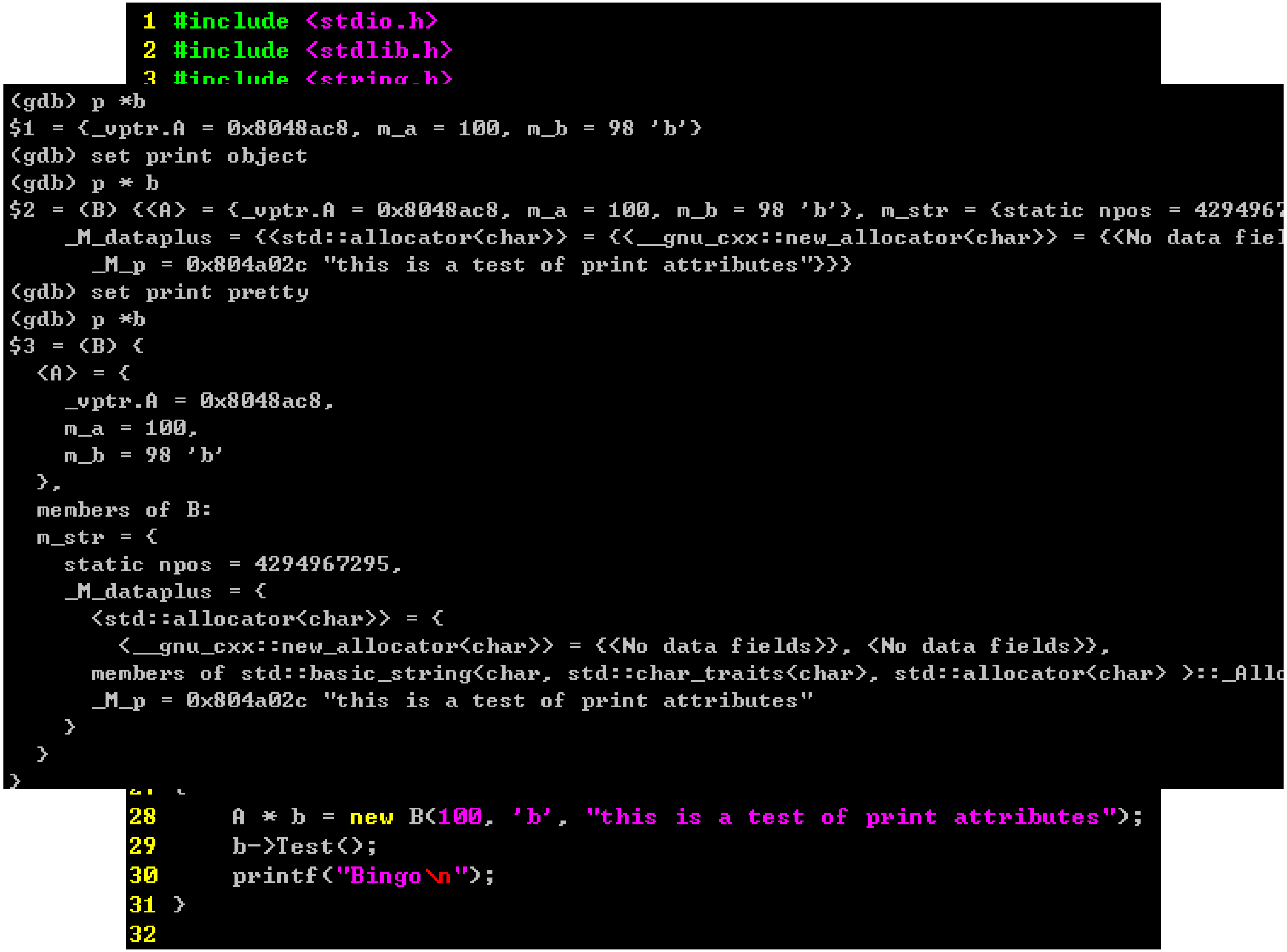
4) (gdb) **set print pretty**：设置GDB打印结构的时候，每行一个成员，并且有相应的缩进，缺省是关闭的

5) (gdb) **set print object**：设置GDB打印多态类型的时候，打印实际的类型，缺省为关闭

6) (gdb) **set print static-members**：设置GDB打印结构的时候，是否打印static成员，缺省是打开的

7) (gdb) **set print vtbl**：以漂亮的方式打印C++的虚函数表，缺省是关闭的

（2）常用调试命令介绍



```
#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结束调试

- 停止

kill小技巧--不退出GDB而对更新当前正在调试的应用程序：在GDB中用**kill**杀掉子进程，然后直接更换应用程序可执行文件，再重新执行**run**，GDB便可加载新的可执行程序启动调试

(g 的子进程

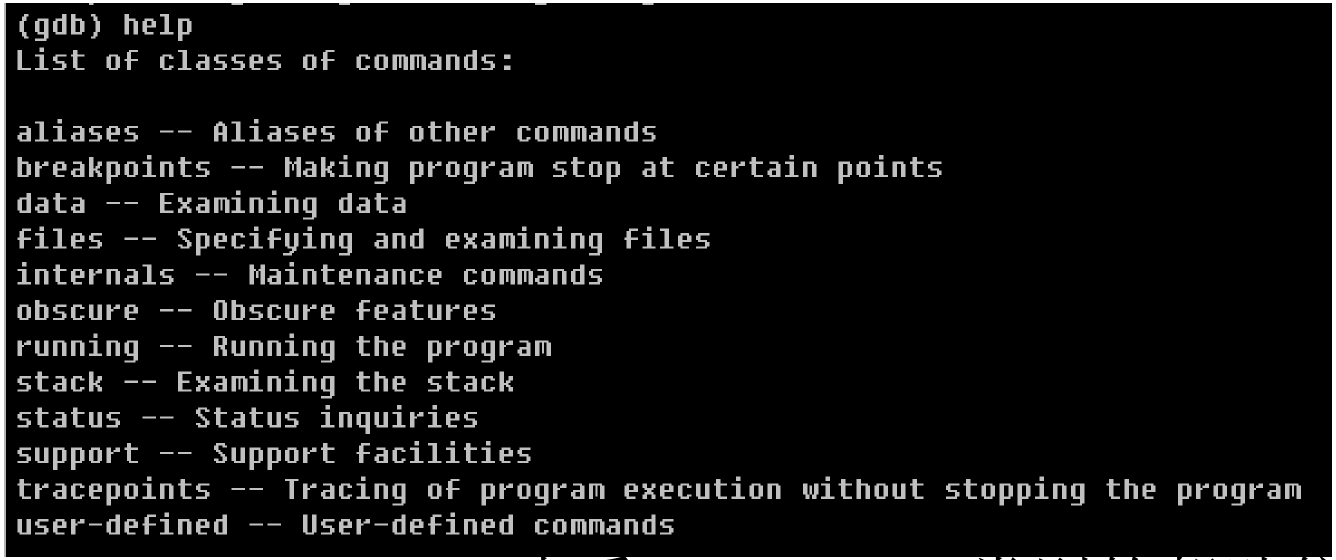
(gdb) **detach**: 调试 调试的进程，与**attach**配对试用

- 退出GDB

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

(gdb) quit

（4）寻求帮助



```
(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*类别的帮助信息

(gdb) **help all**: 查看所有类别的帮助信息

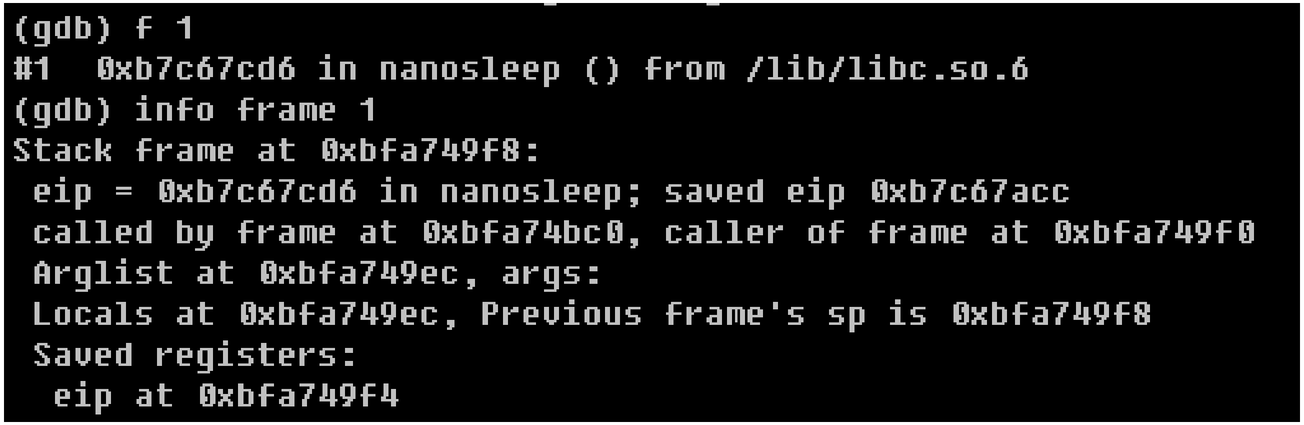
(gdb) **help** *command*: 查看*command*命令的帮助信息

(gdb) **apropos** *word*: 查看*word*关键字相关的命令

(gdb) **complete** *prefix*: 查看以*prefix*为前缀的所有命令

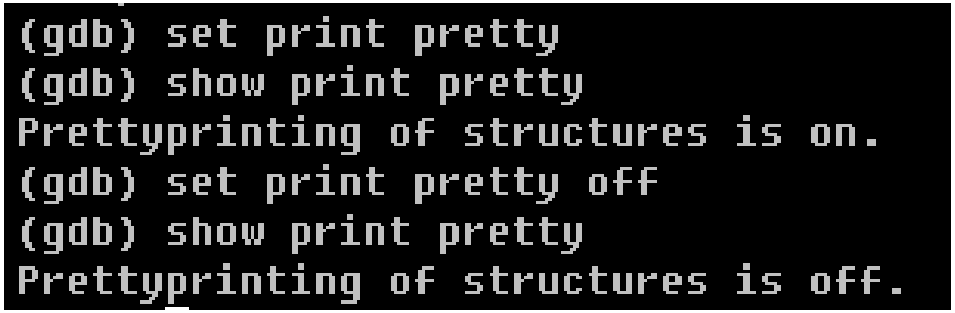
（4）寻求帮助

- **info**：查看与被调试的应用程序相关的信息



```
(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本身设置相关信息



```
(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.大显身手---玩转GDB

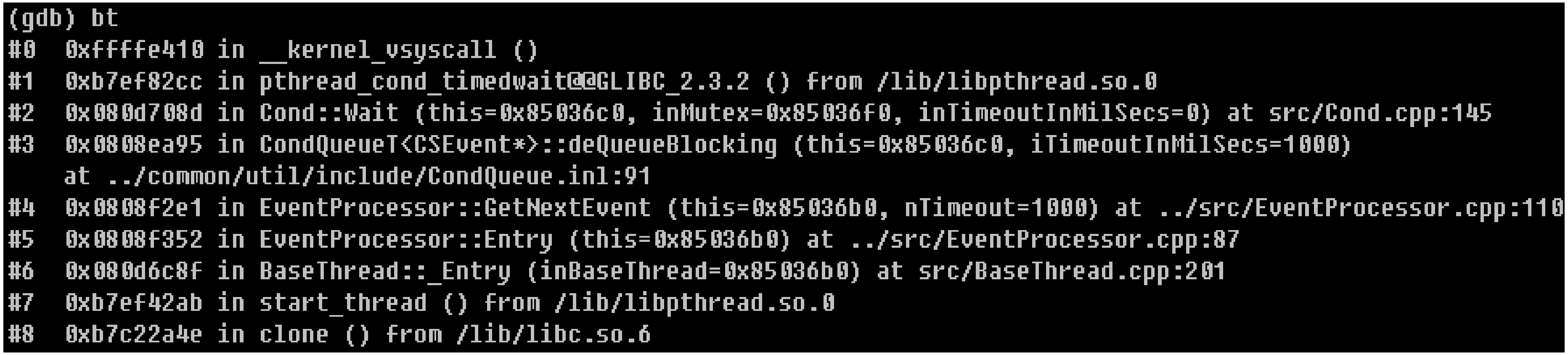
- （1）函数调用栈探密
- （2）调试中信号的响应
- （3）修改程序运行、源码
- （4）多线程调试
- （5）自定义命令

（1）函数调用栈探密

A. Stack frame(栈桢) & Call stack(调用栈)

Stack frame是指保存函数调用上下文信息的一段区域

Call stack是用来存放各个Stack frame的一块内存区域



```
(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）函数调用栈探密

B. 查看Call stack相关信息

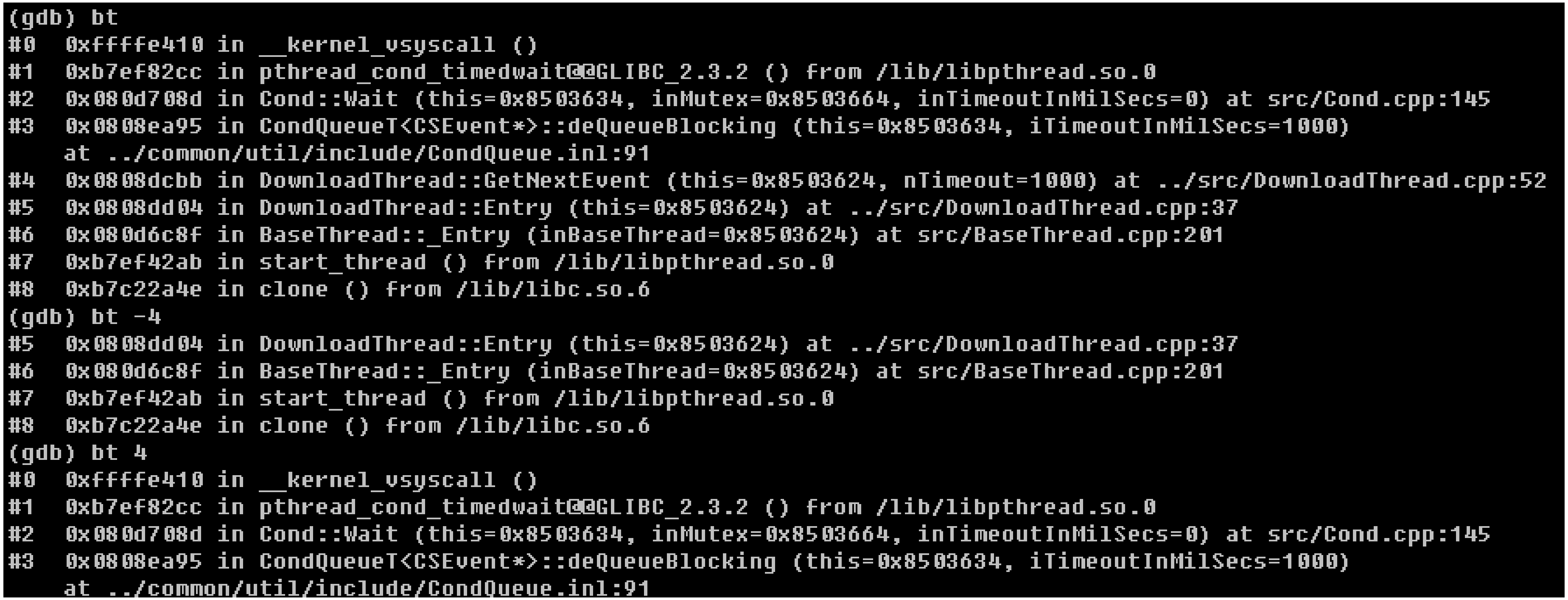
(gdb) **backtrace**:显示程序的调用栈信息，可以用**bt**缩写

(gdb) **backtrace** *n*:显示程序的调用栈信息，只显示栈顶*n*桢

(gdb) **backtrace –***n*:显示程序的调用栈信息，只显示栈底部*n*桢

(gdb) **set backtrace limit** *n*: 设置**bt**显示的最大桢层数，缺省没有限制

(gdb) **where**, **info stack**: **bt**的别名



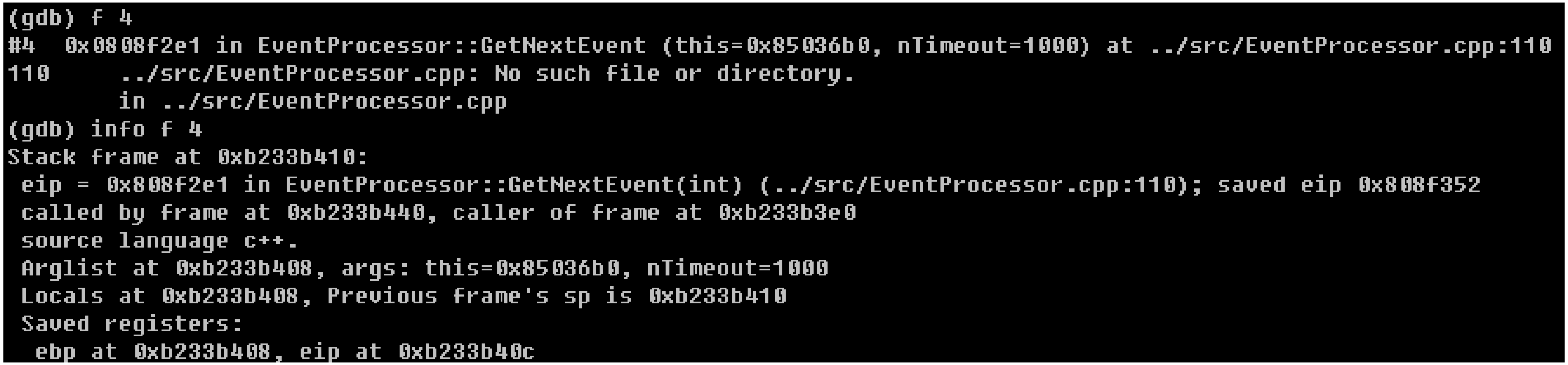
```
(gdb) bt
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7ef82cc in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x080d708d in Cond::Wait (this=0x8503634, inMutex=0x8503664, inTimeoutInMilSecs=0) at src/Cond.cpp:145
#3  0x0808ea95 in CondQueueT<CSEvent*>::deQueueBlocking (this=0x8503634, 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）函数调用栈探密

- C. 查看Stack frame信息

(gdb) **frame** *n*: 查看第n桢的简要信息

(gdb) **info frame** *n*:查看第n桢的详细信息



```
(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
```

简要信息：桢号，$pc, 函数名，函数参数名和参数值，源文件名和行号

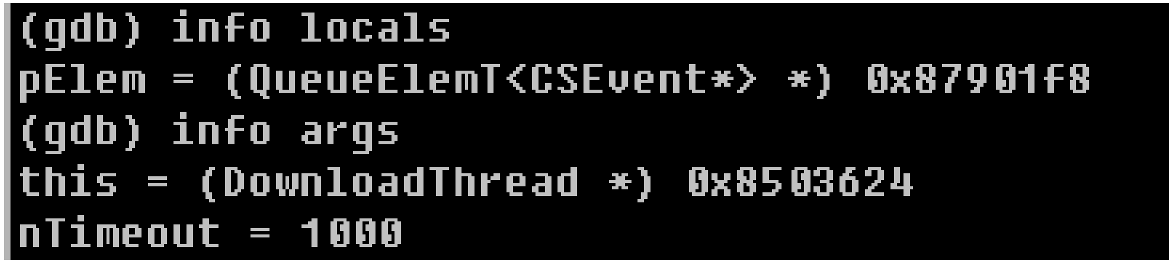
详细信息：当前桢地址，上一桢$eip(pc), 函数名，源文件名和行号，本桢的$eip，上一桢地址，下一桢地址，源码语言，参数列表地址，各参数的值，局部变量地址，上一桢的$sp，保存的一些寄存器

（1）函数调用栈探密

- C. 查看Stack frame信息

(gdb) **info locals**:查看当前桢中函数的参数相关信息

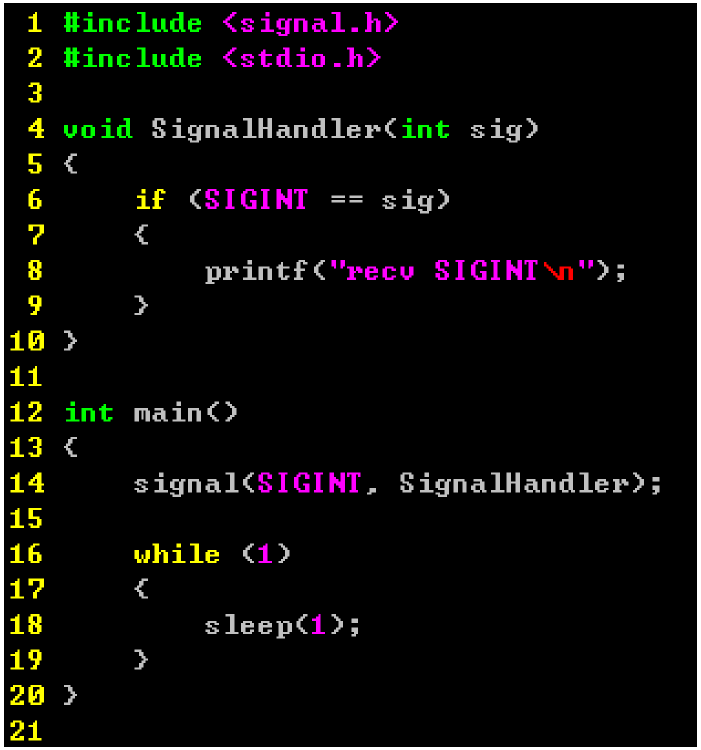
(gdb) **info args**: 查看当前桢中的局部变量相关信息



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

（2）调试中信号的响应

- GDB可以检测到应用程序运行时收到的信号，可以通过命令提前设置当收到指定信息时的处理情况。



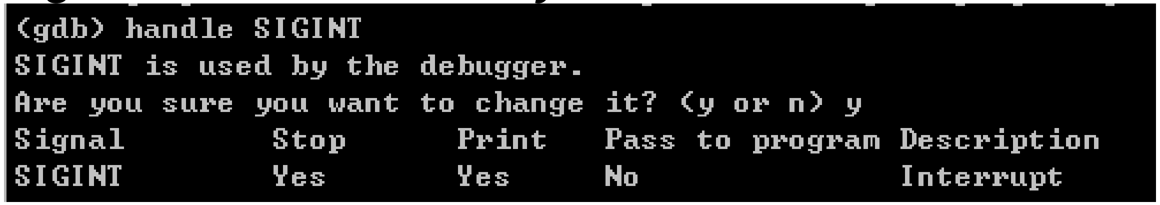
```
#include <signal.h>
#include <stdio.h>

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调试这个程序的时候，让这个程序收到SIGINT信号？

（2）调试中信号的响应

- A. **handle** *signal*

(gdb) **handle** *signal* [*keywords*]: 如果没指定*keywords*, 该命令查看GDB对*signal*的当前的处理情况；如果指定了*keywords*，则是设置GDB对*signal*的处理属性， *keywords*就是要设置的属性



```
(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: 可以为整数或符号形式的信号名，e.g. SIGINT和2是同一信号

keywords:

print & **noprint**: **print**收到指定的信号,打印出一条信息; **noprint**与**print**相反

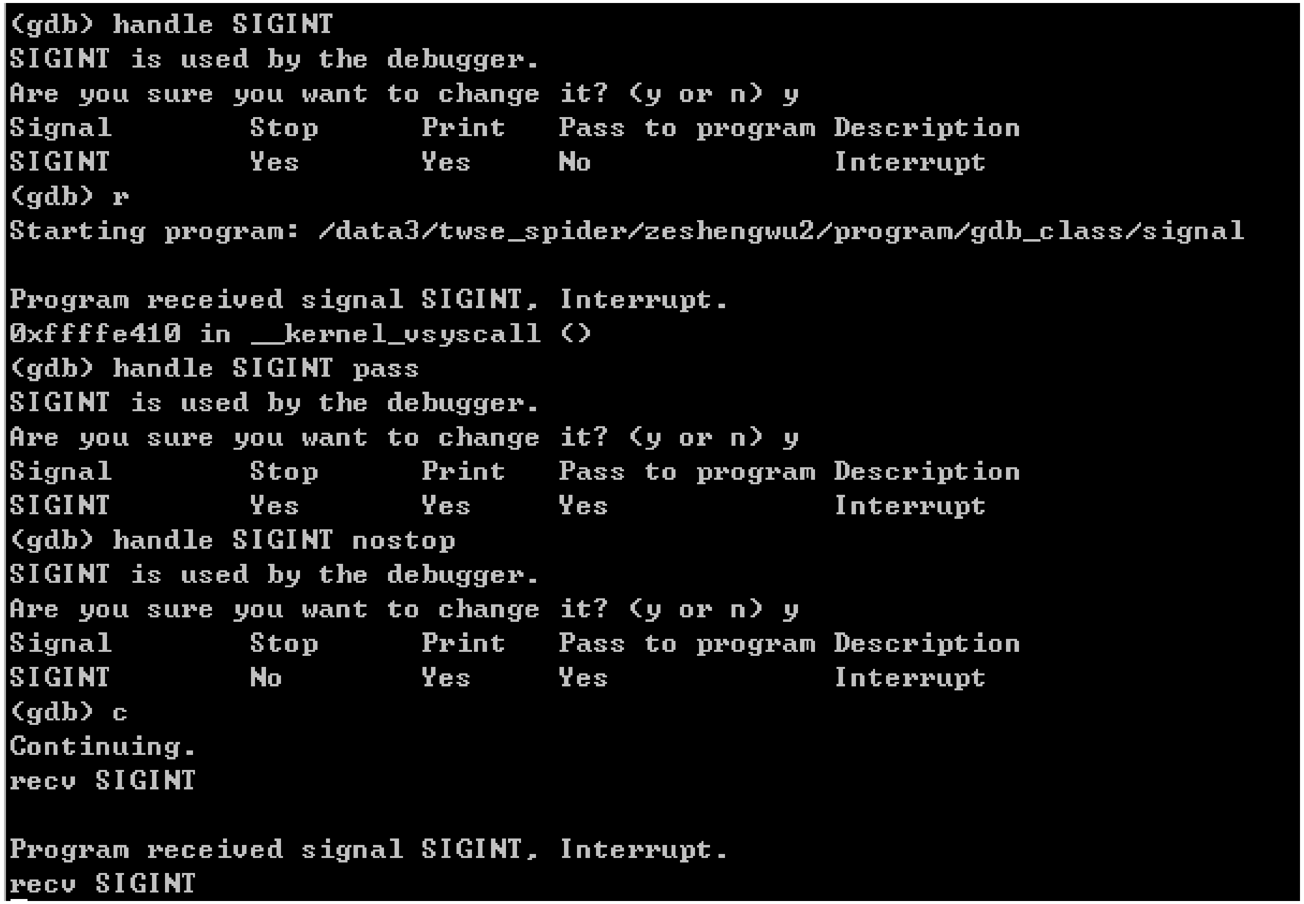
stop & **nostop**: **nostop**表示收到指定的信号，不停止程序的执行，只打印出一条收到信号的消息,因此,**nostop**也暗含**print**, **stop**与**nostop**相反

pass & **nopass**: **pass**表示收到指定的信号，把该信号通知给应用程序; **nopass**与**pass**相反

ignore & **noignore:ingore**与**noignore**分别是**nopass**和**pass**的别名

（2）调试中信号的响应

- A. **handle** *signal*

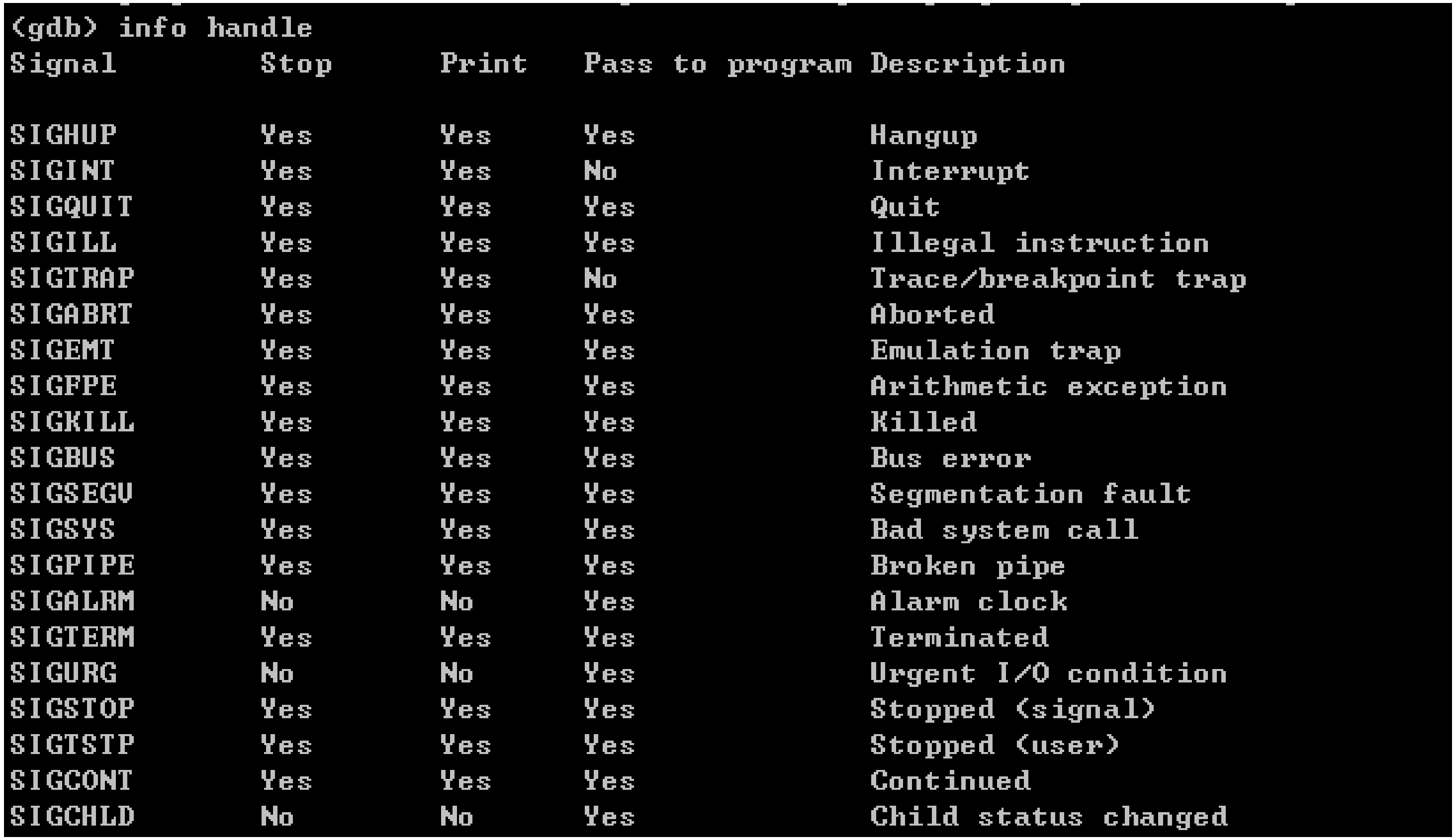


```
(gdb) handle SIGINT
SIGINT is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal        Stop      Print   Pass to program Description
SIGINT        Yes       Yes     No              Interrupt
(gdb) r
Starting program: /data3/twse_spider/zeshengwu2/program/gdb_class/signal

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）调试中信号的响应

- B. 查看GDB对各种信号的缺省处理

(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）修改程序运行、源码

- A. 修改程序的运行

(gdb) **print** *v=value*: 修改变量*v*的值并打印修改后的值

(gdb) **set [var]** *v=value*: 修改变量*v*的值，如果*v*与GDB的某个属性名一样的话，需要在前面加**var**关键字

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

(gdb) **whatis** *v*: 查看变量*v*的类型

(gdb) **signal** *sig*: 把信号*sig*发给被调试的程序

(gdb) **return** [*expression*]: 中止当前函数的执行，返回*expression*值

(gdb) **finish**: 结束当前函数的执行，打印出返回值

(gdb) **call** *function* : 调用程序中的函数*function*

（3）修改程序运行、源码

- B. 修改源码

1）设置环境变量: export EDITOR=/usr/bin/vim

2） (gdb) **edit**: 编辑当前文件

3） (gdb) **edit** *number*: 编辑当前文件的第*number*行

4） (gdb) **edi**

5） (gd *umber* 行

6） (gd 件的*function*函数

回忆— 结合我们前面介绍的shell, make, kill和本节的edit命令，我们完全可以直接在GDB中完成很多的工作！

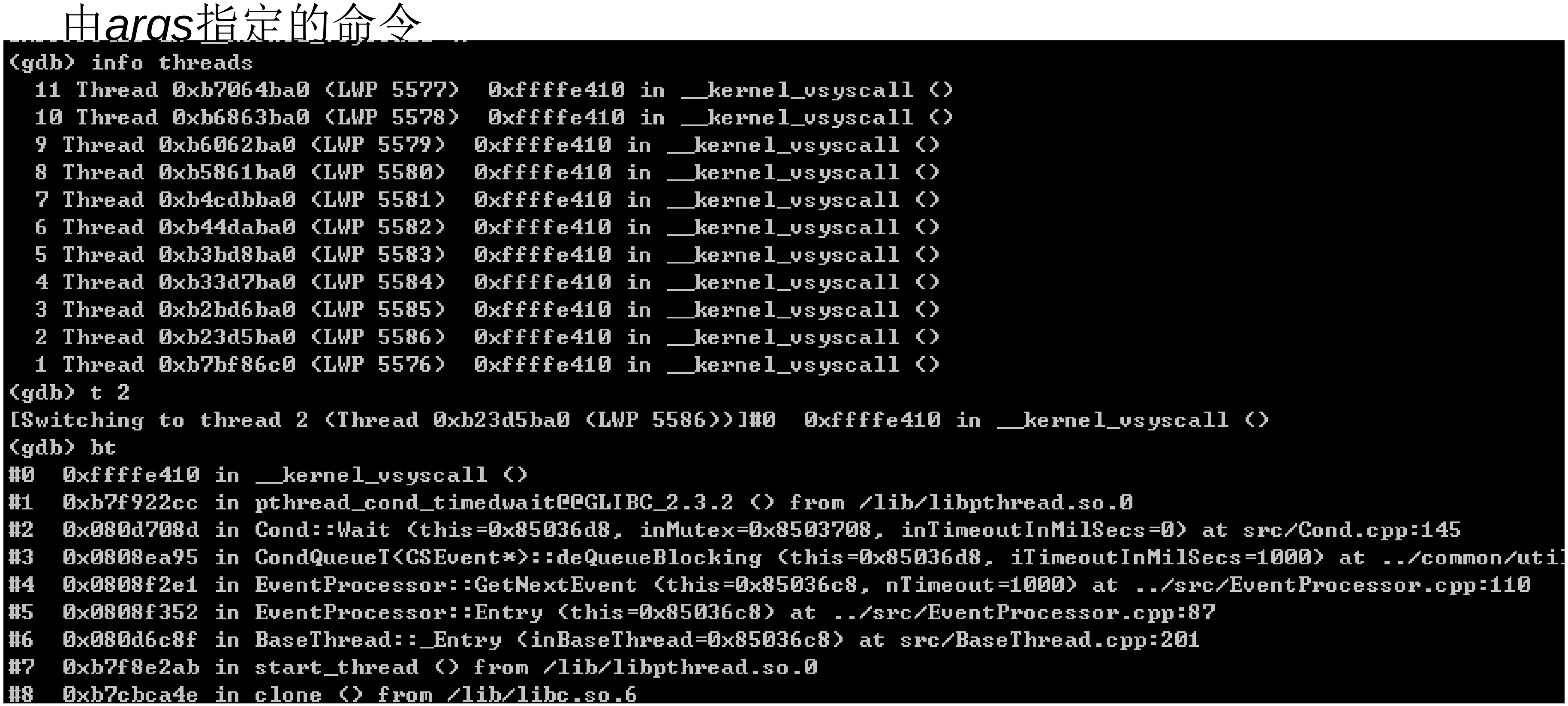
（4）多线程调试

- A. 基本命令

(gdb) **info threads**：查看GDB当前调试的程序的各个线程的相关信息

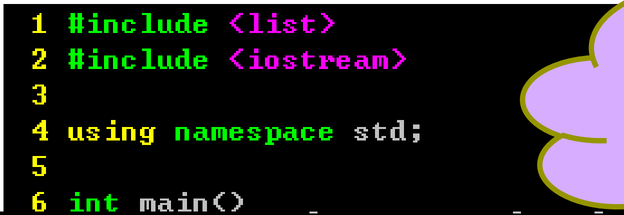
(gdb) **thread** *threadno*：切换当前线程到由**threadno**指定的线程

(gdb) **thread apply** [*threadno*] [**all**] *args*：对指定（或所有）的线程执行由*args*指定的命令



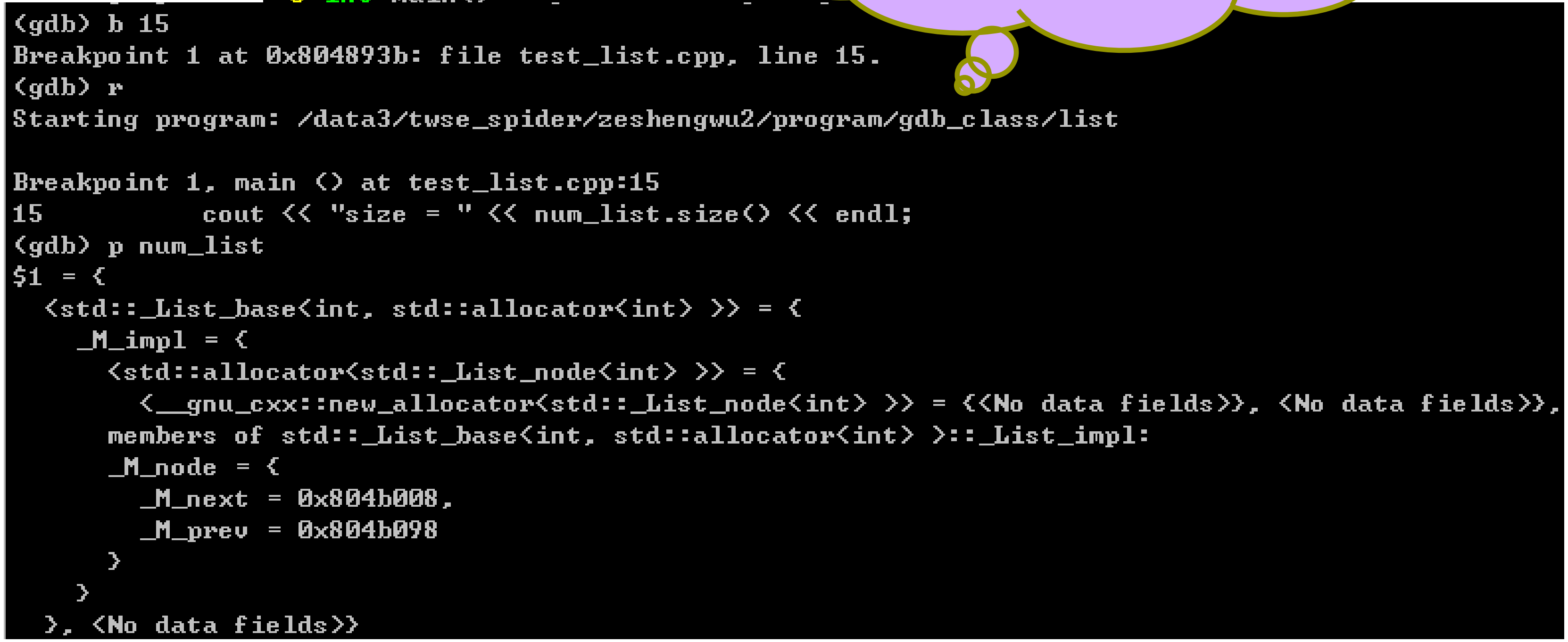
```
(gdb) info threads
  11 Thread 0xb7064ba0 (LWP 5577)  0xffffe410 in __kernel_vsyscall ()
  10 Thread 0xb6863ba0 (LWP 5578)  0xffffe410 in __kernel_vsyscall ()
  9 Thread 0xb6062ba0 (LWP 5579)  0xffffe410 in __kernel_vsyscall ()
  8 Thread 0xb5861ba0 (LWP 5580)  0xffffe410 in __kernel_vsyscall ()
  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/util
#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）自定义命令



```
1 #include <list>
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
```

Question—如何在GDB查看这个list里面的每个元素?



```
(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）自定义命令

- A. 自定义命令基本语法

1）定义一个命令

```
define commandname
 …
 end
```

2）条件语句：

```
if cond-expr
…
else
…
end
```

3）循环语句：

```
while cond-expr
…
end
```

4）定义一个命令的文档信息，在 **help** *commandname*的时候可以显示：

```
document commandname
 …
 end
```

5) *$arg0…$arg9*：表示命令行参数，最多10个

（5）自定义命令

- B. 查看用户自定义命令

(gdb) **help user-defined**：查看所有的用户自定义命令

(gdb) **show user** *commandname*：查看自定义命令*commandname*的定义

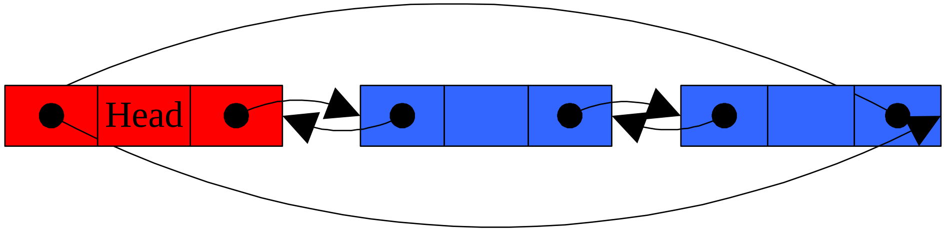
(gdb) **help** *commandname*：查看自定义命令*commandname*的帮助信息

(gdb) **show max-user-call-depth**：查看用户自定义命令的最大递归调用深度，缺省是1024

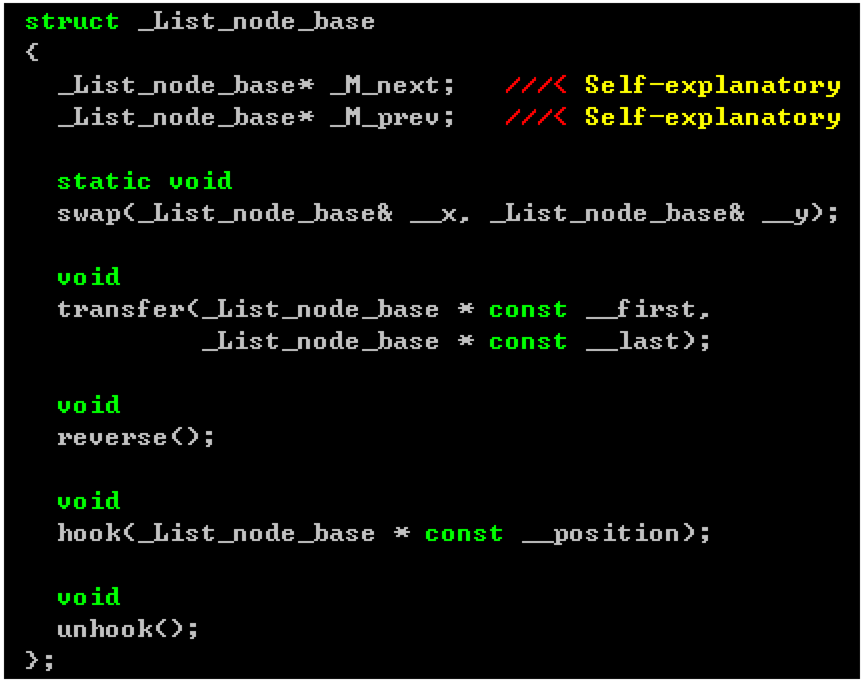
(gdb) **set max-user-call-depth**：设置用户自定义命令的最大递归调用深度

（5）自定义命令

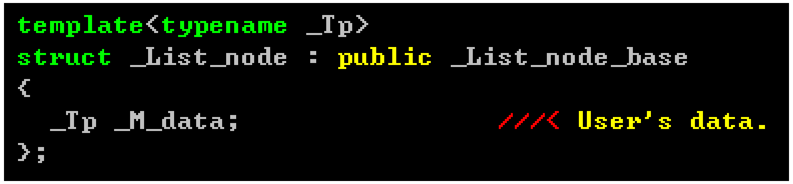
- C. plist实现



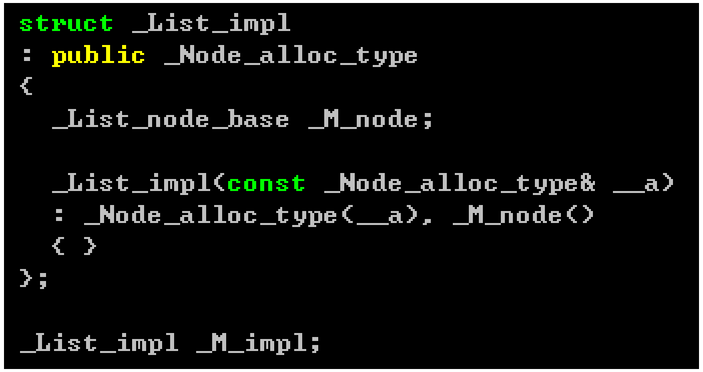
/usr/include/c++/4.1.2/bits/stl_list.h



```
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();
};
```



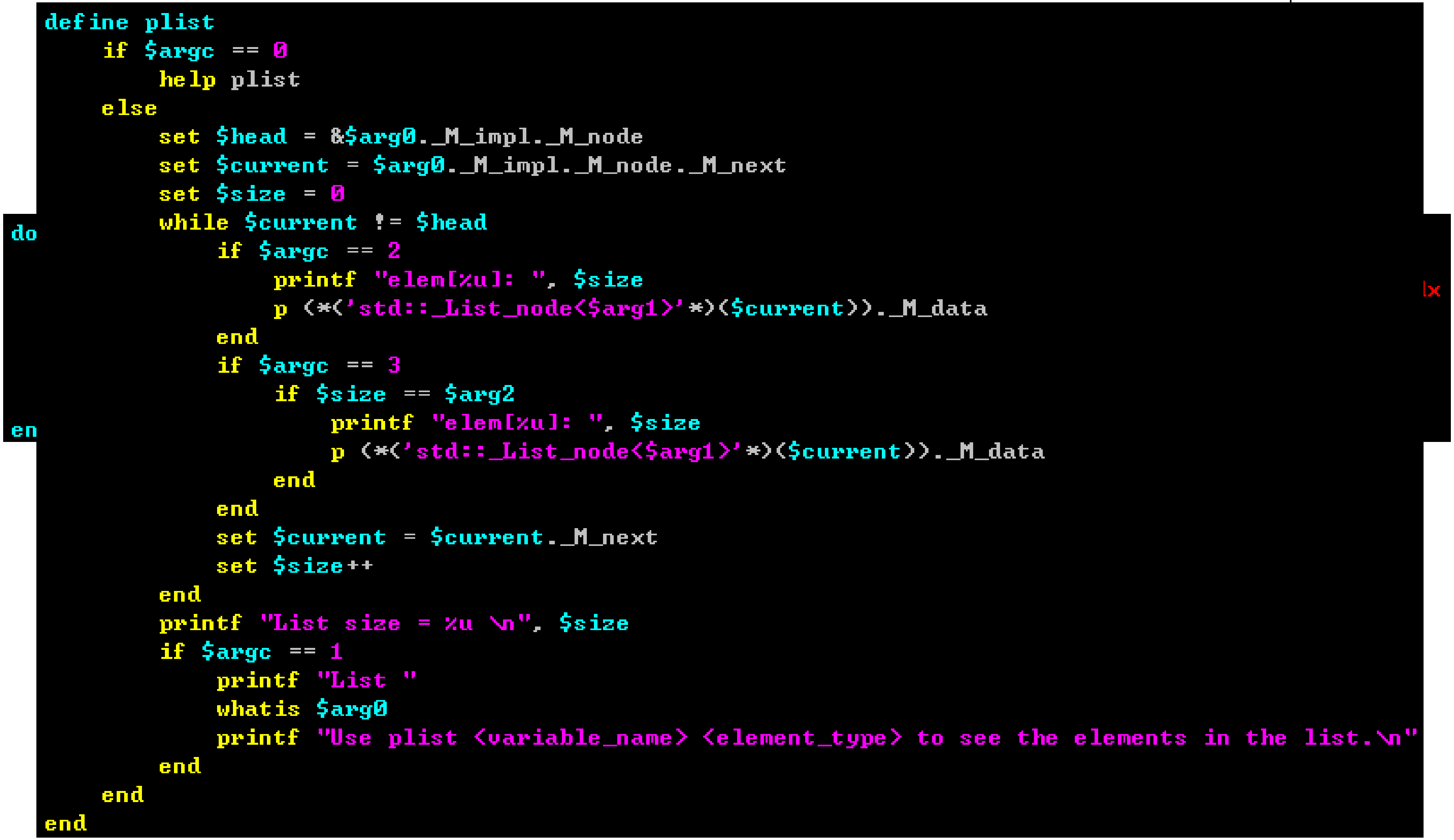
```
template<typename _Tp>
struct _List_node : public _List_node_base
{
  _Tp _M_data;                ///< User's data.
};
```



```
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）自定义命令

- C. plist实现



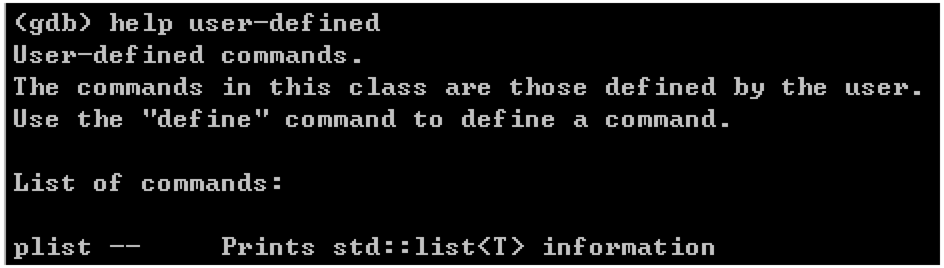
```
define plist
    if $argc == 0
        help plist
    else
        set $head = &$arg0._M_impl._M_node
        set $current = $arg0._M_impl._M_node._M_next
        set $size = 0
        while $current != $head
            if $argc == 2
                printf "elem[%u]: ", $size
                p (*('std::_List_node<$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）自定义命令

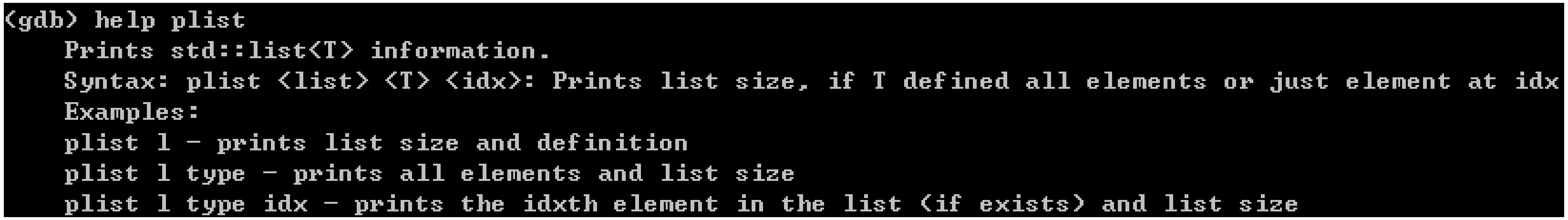
- C. plist实现

1)将plist的实现放到~/.gdbinit文件中

2）



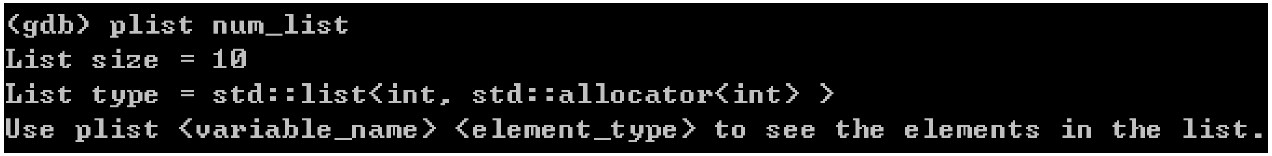
```
(gdb) help user-defined
User-defined commands.
The commands in this class are those defined by the user.
Use the "define" command to define a command.

List of commands:

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



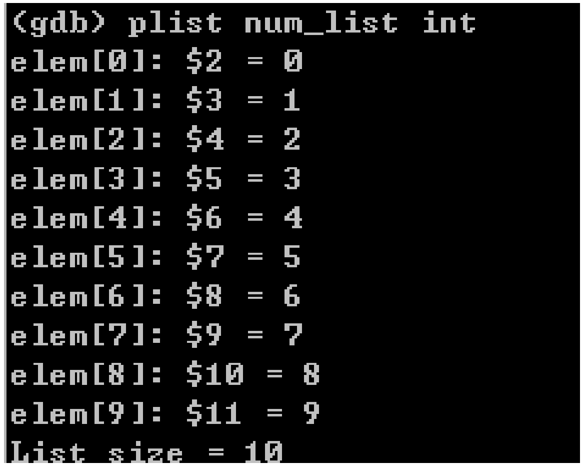
```
(gdb) help plist
    Prints std::list<T> information.
    Syntax: plist <list> <T> <idx>: Prints list size, if T defined all elements or just element at idx
    Examples:
    plist l - prints list size and definition
    plist l type - prints all elements and list size
    plist l type idx - prints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

（5）自定义命令

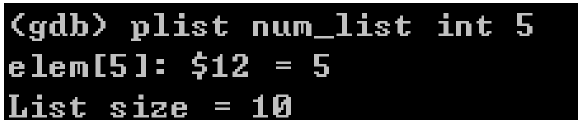
- C. plist实现



```
(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.学而时习之---总结回顾

- （1）常见的coredump原因

a. Signal 6(SIGABRT):

New失败：内存泄露造成内存不够

Delete失败：多次delete同一块内存

应用程序抛出的异常

b. Signal 11(SIGSEGV): 多为内存越界，访问已经被delete掉的内存

c. Signal 13(SIGPIPE): 写已经被删除的文件，写对方已经关闭的socket

- （2）参考资料

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

《The Art of Assembly Language》

《Understanding the Linux Kernel》

《程序员的自我修养---链接、装载与库》

Thank you!