# GDB ????

- gdb □□□□

# gdb ????

## ??

```
SH$ gdb program             # program是你的可执行文件，一般在当前目录
$ gdb program core        # gdb同时调试运行程序和core文件，core是程序非法执行产生的文件
$ gdb program pid         # 如果你的程序是一个服务程序，那么你可以指定这个服务程序运行时的进程ID。gdb会自动attach上去，并调试他。program应该在PATH环境变量中搜索得到。
```

## ??

```
SH(gdb) r/run               # 开始运行程序
(gdb) c/continue          # 继续运行
(gdb) n/next              # 下一行，不进入函数调用
(gdb) s/step              # 下一行，进入函数调用
(gdb) ni/si               # 吓一跳指令，ni和si区别同上
(gdb) fini/finish         # 继续运行至函数退出/当前栈帧
(gdb) u/util              # 继续运行至某一行，在循环中，u可以实现运行至循环刚刚退出，但这取决于循环的实现

(gdb) set args            # 设置程序启动参数，如: set args 10 20 30
(gdb) show args           # 查看程序启动参数
(gdb) path <dir>          # 设置程序的运行路径
(gdb) show paths          # 查看程序的运行路径
(gdb) set env <name=val># 设置环境变量，如: set env USER=chen
(gdb) show env [name]     # 查看环境变量
(gdb) cd <dir>            # 相当于shell的cd命令
(gdb) pwd                 # 显示当前所在目录

(gdb) shell <commond>     # 执行shell命令
```

## ??/????(breakpoint)

```
SH(gdb) b/break linenum/func       # 在第linenum行或function处停住
(gdb) b/break +/-offset          # 在当前行号后/前offset行停住
(gdb) b/break filename:linenum   # 在源文件filename的linenum行停住
(gdb) b/break filename:func      # 在源文件的function入口停住
(gdb) b/break *address           # 在内存地址address处停住
(gdb) b/break                    # 没有参数，表示下一跳指令处停住
(gdb) b/break if <condition>     # 条件成立是停住，如在循环中: break if i=100

(gdb) info break [n]             # 查看断点， n表示断点号
```

# ??/?????(watchpoint)

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) watch <expr>              # □□□□□□□□
(gdb) rwatch <expr>             # □expr□□□□□□□
(gdb) awatch <expr>             # □expr□□□□□□□□□□

(gdb) info watchpoints          # □□□□□□□
```

# ??/?????(catchpoint)

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□C++□□□□
(gdb) tcatch <event>            # □□□□□□□□
(gdb) catch <event>             # □event□□□□□□□□□□□□
# throw             □□c++□□□□□□throw□□□□□
# catch             □□C++□□□□□□□catch□□□□□
# exec              □□□□□□exec□□□□HP-UX□□□□
# fork              □□□□□□fork□□□□HP-UX□□□□
# vfork             □□□□□□vfork□□□□HP-UX□□□□
# load [file]       □□□□□□□□□□□□□□□□HP-UX□□□□
# unload [libname]  □□□□□□□□□□□□□□□□HP-UX□□□□
#
```

# ?????

```
SH# □□□□□□□□□□□□□□□□□□gdb□□□□□□□□□□□□□□□□□□□□□□□□delete□clear□disable□enable□□□□
(gdb) clear                     # □□□□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) clear <fuction>           # □□□□□□□□□□□□□□
(gdb) clear <file:line>         # □□□□□□□□□□□□□□□
(gdb) d/delete [n]/[m-n]        # □□□□□□□□□□□□□□□□□□□□m-n

# □□□□□□□□□□□disable□□□□disable□□□□□□GDB□□□□□□□□□□□□enable□□□□□□□□□□□□
(gdb) disable [n]/[m-n]         # disable□□□□□n□□□□□disable□□□□□□□□m-n

(gdb) enable [n]/[m-n]          # enable□□n□□□□□□m-n
(gdb) enable once [n]/[m-n]     # enable□□n□□□□□□□□□□disable□□□□□□m-n
(gdb) enable delete [n]/[m-n]   # enable□□□□□□□□□□□□□□□□□m-n
```

# ???????

```
SH# □□□□□□breakpoint□□□□□□□□□□□□□□□□□□□□□
(gdb) condition <bunm> <expr>   # □□□□□bnum□□□□□
(gdb) condition <bnum>          # □□□□□bnum□□□□□

# ignore □□□□□□□□□□□□□□□□□□
(gdb) ignore <bnum> <count>     # □□□□□hnum□□□□□count□
```

# ?????????

SH# 当程序停住时，我们可以通过command设置其自动执行的命令，这很利于自动化调试。

```
(gdb) commands [bnum]
> ... commands list ...
> end                    # 这里为断点号bnum设置一个命令列表
```

如:

```
(gdb) break foo if x>0
(gdb) commands
> printf "x is %dn",x
> continue
> end
```

# 断点设置在函数foo中，断点条件是x>0，如果程序被断住后，也就是，一旦x的值在foo函数中大于0，GDB会自动打印出x的值，并继续运行程序。

# 如果你要清除断点上的命令序列，那么只要简单的执行一下commands命令，并直接在打个end就行了。

## ????

SH# 如果你使用c++，有可能下断点时遇到相同名字的函数，gdb会为你列出该函数菜单供你选择。

如:

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
```

## ???????????

SH# 当程序被停住了，你可以用c/continue恢复运行，或下一个断点到来。也可以使用step或next命令单步跟踪程序。

```
(gdb) c/continue [ignore-count]     # 恢复程序运行，ignore-count忽略后面断点数

# 单步跟踪，如果有函数调用，他会进入该函数。进入函数的前提是，此函数被编译有debug信息。很像VC等工具中的stepin。后面可以加count
也可以不加，不加表示一条条地执行，加表示执行后面的count条指令，然后再停住。
(gdb) step <count>

# 打开step-mode模式，于是，在进行单步跟踪时，程序不会因为没有debug信息而不停住。这个参数有很利于查看机器码。
(gdb) set step-mode on

# 当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。
(gdb) u/until

# 单步跟踪一条机器指令！一条程序代码有可能由数条机器指令完成，stepi和nexti可以单步执行机器指令。与之一样有相同功能的命令是“display/i $pc”
，当运行完这个命令后，单步跟踪会在打出程序代码的同时打出机器指令（也就是汇编代码）
(gdb) si/stepi
(gdb) ni/stepi
```

## ??

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□UNIX□□□□□□□□□□□□□□□□□□UNIX□□□□□□□□□□SIGINT
□□□□□□□□□□□□Ctrl+C□□□□SIGBUS□□□□□□□□□□SIGCHLD□□□□□□□□□□□□SIGKILL□□□□□□□□□□□□□□□□□□□□□UNIX
□□□□□□□□□□□
# GDB□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□
handle□□□□□□□□□□
(gdb) handle <signal> <keywords...>
# □GDB□□□□□□□□□□□□<signal>□□□SIG□□□□□SIG□□□□□□□□□□□□□□□□□□□□□SIGIO-SIGKILL□□□□□□SIGIO□□□SIGKILL
□□□□□□□□SIGIO□SIGIOT□SIGKILL□□□□□□□□□□□□□□all□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□□
<keywords>□□□□□□□□□□□□□□□□□
  nostop            # □□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□
  stop          # □□□□□□□□□□□□□GDB□□□□□□□□
  print         # □□□□□□□□□□□□□GDB□□□□□□□□□
  noprint           # □□□□□□□□□□□□□GDB□□□□□□□□□□□□□
  pass/noignore # □□□□□□□□□□□□□GDB□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□
  nopass/ignore # □□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□

# □□□□□□□□□GDB□□□□
(gdb) info signals
(gdb) info handle
```

## ?????

```
SH# □□singal□□□□□□□□□□□□□□□□□□□□□□□□□□Ctrl+C□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□GDB
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) signal <singal>
# UNIX□□□□□□□□□1□15□□□<singal>□□□□□□□□□
# single□□□shell□kill□□□□□□□□□□kill□□□□□□□□□□□□□GDB□□□□□□□single□□□□□□□□□□□□□□□□□□□
```

## ??

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) info threads                          # □□□□
(gdb) break <line> thread <threadno>        # □□□□□line□□□□threadno□□
(gdb) break <line> thread <threadno> if...  # □□□□□line□□□□threadno□□□□□□□

□□
(gdb) break frik.c:13 thread 28 if bartab > lim
```

## ?????

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□“□”□Stack□□□□□□□GDB
□□□□□□□□□□□□□□
(gdb) bt/backtrace          # □□□□□□□□□□□□□□□□□
(gdb) bt/backtrace <n>      # □n□□□□□□□□n□□□□□□□□□□n□

# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

```
(gdb) f/frame <n>              # n从0开始，是栈中的编号
(gdb) up <n>                   # 向栈的上面移动n层。如无n，向上移动一层
(gdb) down <n>                 # 向栈的下面移动n层。如无n，向下移动一层

# 栈的层编号，当前的函数名，函数参数值，函数所在文件及行号，函数执行到的语句。
(gdb) f/frame

#
这个命令会打印出更为详细的当前栈层的信息，只不过，大多数都是运行时的内内地址。比如：函数地址，调用函数的地址，被调用函数的地址，目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变[
(gdb) info f/frame

(gdb) info args                # 打印当前函数的参数名及值
(gdb) info locals              # 打印当前函数中所有局部变量及值
(gdb) info catch               # 打印当前函数中的异常处理信息
```

# ????

```
SH# 在程序编译时一定要加上-g的参数，把源程序信息编译到执行文件中。不然就看不到源程序了。
(gdb) l                              # 显示当前行前后的源码
(gdb) l -                            # 显示当前行前的源码
(gdb) l +                            # 显示当前行后的源码
(gdb) l/list <linuenum/func>         # 查看第linenum行或者function所在行附近的10行
(gdb) l/list                         # 查看上一次list命令列出的代码后面的10行
(gdb) l/list m,n                     # 查看从第m行到第n行的源码。m不填，则从当前行到n
(gdb) l/list -/+offset               # 查看想对当前行偏移offset源码
(gdb) l/list <file:line>             # 查看文件file的line行的源码
(gdb) l/list <func>                  # 查看函数名func源码
(gdb) l/list <file:func>             # 查看文件file的函数func源码
(gdb) l/list <*address>              # 查看运行时内存地址address的源码

(gdb) set listsize                   # 设置一次显示源码的行数
(gdb) show listsize                  # 查看listsize的值
```

# ?????

```
SH(gdb) forward-search <regexp>    # 向前搜索，regexp是一个正则表达式
(gdb) search <regexp>              # 向前搜索
(gdb) reverse-search <regexp>      # 全局搜索
```

# ???????

```
SH# 某些时候，用-g编译过后的执行程序中只是包括了源文件的名字，没有路径名。GDB提供了可以让你指定源文件的路径的命令，以便GDB进行搜索。
(gdb) dir/directory <dirname ... >  # 加一个源文件路径到当前路径的前面。如果你要指定多个路径，UNIX下你可以使用“:”，Windows下你可以使用
“;”。
(gdb) dir/directory                   # 清除所有的自定义的源文件搜索路径信息。
(gdb) show directories                # 显示定义了的源文件搜索路径。
```

# ?????

```
SH(gdb) info line                       # 查看源代码在内存中的地址，还可以：
(gdb) info line <num>
(gdb) info line <file:num>
(gdb) info line <func>
(gdb) info line <file:func>

# 还有一个命令（disassemble）你可以查看源程序的当前执行时的机器码，这个命令会把目前内存中的指令dump出来。如下面的示例表示查看函数func的汇编代码。
(gdb) disassemble func
```

# ???????

```
SH# <expr>是表达式，是你所调试的程序的语言的表达式（GDB可以调试多种编程语言），<f>是输出的格式，比如，如果要把表达式按16进制的格式输出，那么就是/x。
(gdb) p/print <expr>              # expr可以是const常量、变量、函数等内容
(gdb) p/print /<f> <expr>

# 在表达式中，有几种GDB所支持的操作符，它们可以用在任何一种语言中。
@                       是一个和数组有关的操作符，在后面会有更详细的说明。
::                      指定一个在文件或是一个函数中的变量。
{<type>} <addr>         表示一个指向内存地址<addr>的类型为type的一个对象。
# 需要注意的是，如果你的程序编译时开启了优化选项，那么在用GDB调试被优化过的程序时，可能会发生某些变量不能访问，或是取值错误码的情况。
SH# 输出格式
# 一般来说，GDB会根据变量的类型输出变量的值。但你也可以自定义GDB的输出的格式。例如，你想输出一个整数的十六进制，或是二进制来查看这个整型变量的中的位的情况。要做到这样，你可以使用GDB的数据显示格式：
  x 按十六进制格式显示变量。
  d 按十进制格式显示变量。
  u 按十六进制格式显示无符号整型。
  o 按八进制格式显示变量。
  t 按二进制格式显示变量。
  a 按十六进制格式显示变量。
  c 按字符格式显示变量。
  f 按浮点数格式显示变量。

如：
(gdb) p i
$21 = 101
(gdb) p/a i
$22 = 0x65
```

# ????

```
SH# 你可以使用examine命令（简写是x）来查看内存地址中的值。x命令的语法如下所示：
(gdb) x/<n/f/u> <addr>            # n, f, u可选

n           是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容。
f           表示显示的格式，参见上面。如果地址所指的是字符串，那么格式可以是s，如果地十是指令地址，那么格式可以是i。
u           表示从当前地址往后请求的字节数，如果不指定的话，GDB默认是4个bytes。u参数可以用下面的字符来代替，b表示单字节，h表示双字节，w表示四字节，g
表示八字节。当我们指定了字节长度后，GDB会从指内存定的内存地址开始，读写指定字节，并把其当作一个值取出来。
<addr>      表示一个内存地址。

# n/f/u三个参数可以一起使用。例如：
(gdb) x/3uh 0x54320               #表示，从内存地址0x54320读取内容，h表示以双字节为一个单位，3表示三个单位，u表示按十六进制显示。
```

## ????

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□□□display□
(gdb) display <expr>
(gdb) display/<fmt> <expr>
(gdb) display/<fmt> <addr>
# expr□□□□□□□fmt□□□□□□□□addr□□□□□□□□□□display□□□□□□□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□

# □□i□s□□□display□□□□□□□□□□□□□□
(gdb) display/i $pc
# $pc□GDB□□□□□□□□□□□□□□□/i□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□display□□□GDB□□□
(gdb) undisplay <dnums...>
(gdb) delete display <dnums...>
# □□□□□□□□□dnums□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
# □□□□□□□□□□□□□□□□□2-5□
(gdb) disable display <dnums...>
(gdb) enable display <dnums...>
# disable□enalbe□□□□□□□□□□□□□□□□□□□□□□
(gdb) info display
# □□display□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□enable□
```

## ??????

```
SH# GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□
# 1□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□□
(gdb) set print address
(gdb) set print address on
(gdb) set print address off     # □□□□□□□□□□□
(gdb) show print address        # □□□□□□□□□□□□□□□

# 2□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) set print array
(gdb) set print array on
(gdb) set print array off
(gdb) show print array
(gdb) show print elements       # □□print elements□□□□□□□□
(gdb) set print elements <number-of-elements>
# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□<number-of-elements>□□□□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□0
□□□□□□□□

# 3□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□off
(gdb) set print null-stop <on/off>

# 4□□□□□printf pretty□□□□□□□□GDB□□□□□□□□□□□□
(gdb) set print pretty on
(gdb) show print pretty         # □□GDB□□□□□□□□□□\

# 5□□□□□□□□□□□“nnn”□□□□□□□□□□□□□□□□□□□□□nnn□□□□“65”□
(gdb) set print sevenbit-strings <on/off>
(gdb) show print sevenbit-strings   # □□□□□□□□□□□□□
```

```
# 6□□□□□□□□□□□□□□□□□□□□□□□
(gdb) set print union <on/off>
(gdb) show print union              # □□□□□□□□□□□□
□□
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}   # □
$1 = {it = Tree, form = {...}}                          # □

# 7□□C++□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□off□
(gdb) set print object <on/off>
(gdb) show print object             # □□□□□□□□□□

# 8□□□□□□□□□□□□□C++□□□□□□□□□□□□□□□□□□□□□□□□□on□
(gdb) set print static-members <on/off>
(gdb) show print static-members     # □□□□□□□□□□□□□

# 9□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) set print vtbl <on/off>
(gdb) show print vtbl               # □□□□□□□□□□□□□
```

## ????

```
□□□GDB□print□□□□□□□□□□□□□□□□print□□□GDB□□□□□GDB□□$1, $2, $3 .....□□□□□□□□□□print
□□□□□□□□□□□□□□□□□□□□□□□□□□□□$1
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

## ????

```
SH# □□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□ □□GDB□set□□□GDB□□□□□□UNIX□□□□□□$
□□□□□
(gdb) set $foo = *object_ptr
# □□□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) show convenience
# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) set $i = 0
(gdb) print bar[$i++]->contents
# □□□□□□□□□print bar[0]->contents, print bar[1]->contents
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

## ?????

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ip□□□□□□□□□□□□□□sp□□□□□□□□□print□□□□□□□□□□□□□□□□□□□□□□□□□$
□□□□□□□□□p $eip□
(gdb) info registers        # □□□□□□□(□□□□□□)
(gdb) info all-registers    # □□□□□□□□□
(gdb) info registers regname# □□□□□□□□□□□□info rbp
```

## ???????

```
SH# □□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□GDB
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

# □□□□□□□
# □□□□□□□□□□□□□□□□GDB□□□□□□□□□GDB□print□□□□□□□□□
(gdb) print x=4
# x=4□□□□□C/C++□□□□□□□□□□x□□□□4□□□□□□□□□□□□□Pascal□□□□□□□□Pascal□□□□x:=4□
# □□□□□□□□□□□□□□□GDB□□□□□□□□□
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
# □□□set width□GDB□□□□□□□□□□“Invalid syntax in expression”□□□□□□□□□□□□□□set
# var□□□□□GDB□width□□□GDB□□□□□□□□□□□□□□□
(gdb) set var width=47
# □□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□set var□□□GDB□□□
```

## ????

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□GDB□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□GDB□jump
□□□□□
(gdb) jump <linespec>
# □□□□□□□□□□□□<linespce>□□□□□□□□□□□□file:line□□□□□□+num□□□□□□□□□□□□□□□□□□□□□□□□
(gdb) jump <address>
# □□□<address>□□□□□□□□□□□□□jump
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Core Dump
□□□□□□□□□□□□□□□□□
# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□jump□□□□□□□□□□□□□□□□□□□□□□□□□“set $pc”□□□□□□□□□□□□□
(gdb) set $pc = 0x485
```

## ??????

```
SH# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□return□□□□□□□□□□□□□□□□□□□□
(gdb) return
(gdb) return <expression>
# □□return□□□□□□□□□□□□□□□□□□□□□□□<expression>□□□□□□□□□□□□□□□□□□□□
```

# ??????

```
SH(gdb) call <expr>
# □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□void□□□□□□□□
# □□□□□□□□□□□□□□□□□——print□print□□□□□□□□□□□□□□□□□□□□□□print□call□□□□□□□□□□□void□call□□□□□print
□□□□□□□□□□□□□□□□□□□□□
```

## GDB????

```
SH(gdb) show language
# [illegible]GDB[illegible]C[illegible]
(gdb) info frame
# [illegible]
(gdb) info source
# [illegible]GDB[illegible]set language[illegible]set language
[illegible]GDB[illegible]
(gdb) set language
The currently understood settings are:
local or auto Automatic setting based on source file
c Use the C language
c++ Use the C++ language
asm Use the Asm language
chill Use the Chill language
fortran Use the Fortran language
java Use the Java language
modula-2 Use the Modula-2 language
pascal Use the Pascal language
scheme Use the Scheme language
# [illegible]set language[illegible]
```

## ??

```
(gdb) q/quit                # [illegible]GDB[illegible]
```