使用 GDB

一般来说 GDB 主要调试的是 C/C++的程序。要调试 C/C++的程序，首先在编译时，我们必须要把调试信息加到可执行文件中。使用编译器（cc/gcc/g++）的 -g 参数可以做到这一点。如：

> cc -g hello.c -o hello

> g++ -g hello.cpp -o hello

如果没有-g，你将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。当你用-g 把调试信息加入之后，并成功编译目标代码以后，让我们来看看如何用 gdb 来调试他。

进入 GDB

启动 GDB 的方法有以下几种：

1、gdb program

program 也就是你的执行文件，一般在当然目录下。

2、gdb program core

用 gdb 同时调试一个运行程序和 core 文件，core 是程序非法执行后 core dump 后产生的文件。

3、gdb 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。

GDB 启动时，可以加上一些 GDB 的启动开关，详细的开关可以用 gdb -help 查看。我在下面只例举一些比较常用的参数：

参数	说明
-symbols [file](-s)	读取文件中的符号表
-exec [file] (-e)	调试一个可执行文件
-se [file]	上二者的缩写
-core [file](-c)	读入一个 core dump 文件
-pid number (-p)	启动 attach 模式，除错一个执行中的行程。number 是目标行程的 pid

-directory [directory] (-d)　　将 directory 加入原始码的搜寻路行

-readnow (-r)　　一次读取完所有的符号表，这会让启动 gdb 的时间变长，但在执行往后的除错动作会较快速。

下列还有部分选择性的参数，我列出几个目前用的到的：

-quiet -silent -q　　安静模式，启动时 gdb 将不会显示版权页。

-windows -w　　与下一选项相反，这会启动 GUI

-nowindows -nw　　如果 gdb 有编入 GUI 的话，这个选项会关掉它。

-cd [directory]　　切换工作目录为 directory 而不是现在的目录

-tty [device](-t)　　指定 device 为程式的标准输出入

--args　　这个参数要当作命令列的最后一个参数，其后跟随的参数都会被视为「欲传给将调试的程序的参数」

当然我们也不一定要在启动时指定调试来源。

file 指令可以指定要调试的程序，功能就与-se 一样，其实这样指定是比较方便的。 在执行程序前，我们可以先指定命令列参数，如同--args 参数的效果。

程序运行参数

set args　　指定运行时的命令列参数

show args　　查看现在的命令列参数是什么。由于 gdb 调试的对象是已经编译好的可执行文件，所以这里我们不必像 VS.NET 一样等半天。有一点必须注意，gdb 传给程序的命令列参数是程序开始执行前的那一份命令列参数，如果程序已经开始执行，就算用中断点中断执行然后改变命令列参数，也只会在下一次执行时变更才会生效。

运行环境

path　　设定程序的运行路径。

show paths　　查看程序的运行路径。

set environment varname [=value]　　设置环境变量，如 set env USR =

show environment [varname]　　查看环境变量。

工作目录

cd　　相当于 shell 的 cd 命令。

Pwd 显示当前的所在目录。

程序的输入输出

info terminal 显示你程序用到的终端的模式。

使用重定向控制程序的输出。如:run > outfile

tty 命令可以指定写输入输出的终端设备。如:tty /dev/ttyb

调试已运行的程序 两种方法：

1、在 UNIX 下用 ps 查看正在运行的程序的 PID（进程 ID），然后用 gdb PID 格式挂接正在运行的程序。

2、先用 gdb 关联上源代码，并进行 gdb，在 gdb 中用 attach 命令来挂接进程的 PID。并用 detach 来取消挂接的进程。

暂停 / 恢复程序运行

我们在调试的过程中，会推测某些代码是否出了问题，如果要测试该段代码，可以让程序执行到那区段前暂停，然后我们来测试看看是出了什么问题。

一、设置断点（BreakPoint）

参数	说明
break [function]	在某函数的进入点设中断点。C++中可以使用 class::function 或 function(type,type)格式来指定函数名。
break +offset | -offset	当程序停止时，在停止位置的前/后第 offset 行设中断点
break linenum	指定行号设中断点
break filename:linenum	在某 source file 的第几行或指定函数设定中断点
break	在下一个要执行的指令设中断点
break [args] if [cond]	当[cond]这个运算式为真，设定中断点。 args 可能是上列的任一种情形。
tbreak args	只会生效一次的中断点
rbreak regex	使用正则运算来找寻可能的函数，并在其进入点设中断点。 EX:(gdb) rbreak . 这样每个函数开头都有中断点了。The syntax of the regular expression is the standard one

used with tools like `grep'. Note that this is different from the syntax used by shells, so for instance foo* matches all functions that include an fo followed by zero or more os. There is an implicit .* leading and trailing the regular expression you supply, so to match only functions that begin with foo, use ^foo. When debugging C++ programs, rbreak is useful for setting breakpoints on overloaded functions that are not members of any special classes.

break *address　　在程序运行的内存地址设置断点

break filename:function

info break [n]　　查看断点，n 表示断点号

值得一提的是，C++允许 overloading，所以直接指定函数名称有时候 gdb 无法辨认要把中断点设在哪。 举例：

```
int takeout(int i,int j){...}
int takeout(float i,float j){...}
```

这时候我们可以设定 break takeout(int,int)，这样就会在上面的函数的进入点设定中断点。 当然假如我们只输入 break takeout，gdb 会显示一个互动示选单显示所有可能的函式的原型，让我们挑选要将中断点设在哪个函式。break 的简写是 b，通常只输入简写方便多了。

二、设置观察点（WatchPoint）

观察点一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停住程序。我们有下面的几种方法来设置观察点：

watch expr

为表达式（变量）expr 设置一个观察点。一量表达式值有变化时，马上停住程序。

rwatch expr

当表达式（变量）expr 被读时，停住程序。

awatch expr

当表达式（变量）的值被读或被写时，停住程序。

info watchpoints

列出当前所设置了的所有观察点。

三、设置捕捉点(CatchPoint)

你可设置捕捉点来补捉程序运行时的一些事件。如：载入共享库(动态链接库)或是C++的异常。设置捕捉点的格式为：

catch event

当event发生时，停住程序。event可以是下面的内容：

1、throw 一个C++抛出的异常。(throw为关键字)

2、catch 一个C++捕捉到的异常。(catch为关键字)

3、exec 调用系统调用exec时。(exec为关键字，目前此功能只在HP-UX下有用)

4、fork 调用系统调用fork时。(fork为关键字，目前此功能只在HP-UX下有用)

5、vfork 调用系统调用vfork时。(vfork为关键字，目前此功能只在HP-UX下有用)

6、load 或load 载入共享库(动态链接库)时。(load为关键字，目前此功能只在HP-UX下有用)

7、unload 或unload 卸载共享库(动态链接库)时。(unload为关键字，目前此功能只在HP-UX下有用)

tcatch event

只设置一次捕捉点，当程序停住以后，应点被自动删除。

使用info break命令列出当前捕捉点。

四、维护停止点

上面说了如何设置程序的停止点，GDB中的停止点也就是上述的三类。在GDB中，如果你觉得已定义好的停止点没有用了，你可以使用delete、clear、disable、enable这几个命令来进行维护。

clear

清除停止点，后面跟(文件名)行号或函数名,未指定则表示清除所有断点

比删除更好的一种方法是disable停止点，disable了的停止点，GDB不会删除，当你还需要时，enable即可，就好像回收站一样。

disable [breakpoints] [range...]

disable 所指定的停止点，breakpoints 为停止点号。如果什么都不指定，表示 disable 所有的停止点。简写命令是 dis.

enable [breakpoints] [range...]

enable 所指定的停止点，breakpoints 为停止点号。

enable [breakpoints] once range...

enable 所指定的停止点一次，当程序停止后，该停止点马上被 GDB 自动 disable。

enable [breakpoints] delete range...

enable 所指定的停止点一次，当程序停止后，该停止点马上被 GDB 自动删除。

五、停止条件维护

前面在说到设置断点时，我们提到过可以设置一个条件，当条件成立时，程序自动停止，这是一个非常强大的功能，这里，我想专门说说这个条件的相关维护命令。一般来说，为断点设置一个条件，我们使用 if 关键词，后面跟其断点条件。并且，条件设置好后，我们可以用 condition 命令来修改断点的条件。（只有 break 和 watch 命令支持 if，catch 目前暂不支持 if）

condition bnum expression

修改断点号为 bnum 的停止条件为 expression。

condition bnum

清除断点号为 bnum 的停止条件。

还有一个比较特殊的维护命令 ignore，你可以指定程序运行时，忽略停止条件几次。

ignore bnum count

表示忽略断点号为 bnum 的停止条件 count 次。

六、为停止点设定运行命令

我们可以使用 GDB 提供的 command 命令来设置停止点的运行命令。也就是说，当运行的程序在被停止住时，我们可以让其自动运行一些别的命令，这很有利行自动化调试。对基于 GDB 的自动化调试是一个强大的支持。

commands [bnum]

... command-list ...

end

为断点号 bnum 指写一个命令列表。当程序被该断点停住时，gdb 会依次运行命令列表中的命令。

例如：

```
break foo if x>0
commands
printf "x is %d\n",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。

七、断点菜单

在 C++中，可能会重复出现同一个名字的函数若干次（函数重载），在这种情况下，break 不能告诉 GDB 要停在哪个函数的入口。当然，你可以使用 break 也就是把函数的参数类型告诉 GDB，以指定一个函数。否则的话，GDB 会给你列出一个断点菜单供你选择你所需要的断点。你只要输入你菜单列表中的编号就可以了。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:String.cc; line number:735
> 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.

Multiple breakpoints were set.

Use the "delete" command to delete unwanted

breakpoints.

(gdb)

可见，GDB 列出了所有 after 的重载函数，你可以选一下列表编号就行了。0 表示放弃设置断点，1 表示所有函数都设置断点。

八、恢复程序运行和单步调试

continue(c)

继续执行直到下一个中断点或结束

step(s)

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 step in。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。

next(n)

同样单步跟踪，如果有函数调用，他不会进入该函数。很像 VC 等工具中的 step over。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。

set step-mode on

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。

set step-mod off

关闭 step-mode 模式

finish

运行程序，直到当前函数完成返回。并打印函数返回时的堆栈地址和返回值及参数值等信息。

until(u)

执行一行程序，若此时程序是在 for/while/do loop 循环的最后一行，则一直执行到循环结束后的第一行程序后停止。

nexti

单步执行一条指令，会跟踪进入子程序内部，但不打印出子程序内部的语句

stepi

单步执行一条指令，会跟踪进入子程序内部，会提示下面要运行语句的代码

值得一提的是，如果 step 执行到一个函数，但该函数的调试符号信息没有编进二进制文件里的话，那 step 也不会跳进这个函数里，而是单纯的将它看作一行代码(如同 next)，比如说是标准函式库提供的函数，大概都会这样。

九、信号（Signals）

信号是一种软中断，是一种处理异步事件的方法。一般来说，操作系统都支持许多信号。尤其是 UNIX，比较重要应用程序一般都会处理信号。UNIX 定义了许多信号，比如 SIGINT 表示中断字符信号，也就是 Ctrl+C 的信号，SIGBUS 表示硬件故障的信号；SIGCHLD 表示子进程状态改变信号；SIGKILL 表示终止程序运行的信号，等等。信号量编程是 UNIX 下非常重要的一种技术。

GDB 有能力在你调试程序的时候处理任何一种信号，你可以告诉 GDB 需要处理哪一种信号。你可以要求 GDB 收到你所指定的信号时，马上停住正在运行的程序，以供你进行调试。你可以用 GDB 的 handle 命令来完成这一功能。

handle signal keywords...

在 GDB 中定义一个信号处理。信号可以以 SIG 开头或不以 SIG 开头，可以用定义一个要处理信号的范围（如：SIGIO-SIGKILL，表示处理从 SIGIO 信号到 SIGKILL 的信号，其中包括 SIGIO，SIGIOT，SIGKILL 三个信号），也可以使用关键字 all 来标明要处理所有的信号。一旦被调试的程序接收到信号，运行程序马上会被 GDB 停住，以供调试。其可以是以下几种关键字的一个或多个。

nostop

当被调试的程序收到信号时，GDB 不会停住程序的运行，但会打出消息告诉你收到这种信号。

stop

当被调试的程序收到信号时，GDB 会停住你的程序。

print

当被调试的程序收到信号时，GDB 会显示出一条信息。

noprint

当被调试的程序收到信号时，GDB 不会告诉你收到信号的信息。

pass

noignore

当被调试的程序收到信号时，GDB 不处理信号。这表示，GDB 会把这个信号交给被调试程序会处理。

nopass

ignore

当被调试的程序收到信号时，GDB 不会让被调试程序来处理这个信号。

info signals

info handle

查看有哪些信号在被 GDB 检测中。

十、线程（Thread Stops）

如果你程序是多线程的话，你可以定义你的断点是否在所有的线程上，或是在某个特定的线程。GDB 很容易帮你完成这一工作。

break linespec thread threadno

break linespec thread threadno if ...

linespec 指定了断点设置在的源程序的行号。threadno 指定了线程的 ID，注意，这个 ID 是 GDB 分配的，你可以通过“info threads”命令来查看正在运行程序中的线程信息。如果你不指定 thread 则表示你的断点设在所有线程上面。你还可以为某线程指定断点条件。如：

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

当你的程序被 GDB 停住时，所有的运行线程都会被停住。这方便你你查看运行程序的总体情况。而在你恢复程序运行时，所有的线程也会被恢复运行。那怕是主进程在被单步调试时。

查看栈信息

当程序被停住了，你需要做的第一件事就是查看程序是在哪里停住的。当你的程序调用了一个函数，函数的地址，函数参数，函数内的局部变量都会被压入“栈”（Stack）中。你可以用 GDB 命令来查看当前的栈中的信息。

下面是一些查看函数调用栈信息的 GDB 命令：

backtrace (bt)

打印当前的函数调用栈的所有信息。如：

(gdb) bt

#0 func (n=250) at tst.c:6

#1 0x08048524 in main (argc=1, argv=0xbffff674) at tst.c:30

#2 0x400409ed in __libc_start_main () from /lib/libc.so.6

从上可以看出函数的调用栈信息：__libc_start_main --> main() --> func()

backtrace (bt) n

n 是一个正整数，表示只打印栈顶上 n 层的栈信息。

Backtrace (bt) <-n>

-n 表一个负整数，表示只打印栈底下 n 层的栈信息。

如果你要查看某一层的信息，你需要在切换当前的栈，一般来说，程序停止时，最顶层的栈就是当前栈，如果你要查看栈下面层的详细信息，首先要做的是切换当前栈。

frame (f) n

n 是一个从 0 开始的整数，是栈中的层编号。比如：frame 0，表示栈顶，frame 1，表示栈的第二层。

frame (f) addr

Select the frame at address addr.

up n

表示向栈的上面移动 n 层，可以不打 n，表示向上移动一层。

down n

表示向栈的下面移动 n 层，可以不打 n，表示向下移动一层。

上面的命令，都会打印出移动到的栈层的信息。如果你不想让其打出信息。你可以使用这三个命令：

select-frame 对应于 frame 命令。

up-silently n 对应于 up 命令。

down-silently n 对应于 down 命令。

查看当前栈层的信息，你可以用以下 GDB 命令：

frame (f)

会打印出这些信息：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。

info frame(f)

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存地址。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、局部变量的地址等等。如：

(gdb) info f

Stack level 0, frame at 0xbffff5d4:

eip = 0x804845d in func (tst.c:6); saved eip 0x8048524

called by frame at 0xbffff60c

source language c.

Arglist at 0xbffff5d4, args: n=250

Locals at 0xbffff5d4, Previous frame's sp is 0x0

Saved registers:

ebp at 0xbffff5d4, eip at 0xbffff5d8

info frame(f) addr

Print a verbose description of the frame at address addr, without selecting that frame.

info args

打印出当前函数的参数名及其值。Print the arguments of the selected frame, each on a separate line.

info locals

打印出当前函数中所有局部变量及其值。

info catch

打印出当前的函数中的异常处理信息。

查看源程序

一、显示源代码(list)

list linenum

显示程序第 linenum 行的周围的源程序。

list function

显示函数名为 function 的函数的源程序。

list

显示当前行后面的源程序。

list -

显示当前行前面的源程序。

一般是打印当前行的上 5 行和下 5 行，如果显示函数是是上 2 行下 8 行，默认是 10 行，当然，你也可以定制显示的范围，使用下面命令可以设置一次显示源程序的行数。

set listsize count

设置一次显示源代码的行数。

show listsize

查看当前 listsize 的设置。

list 命令还有下面的用法：

list first,last

显示从 first 行到 last 行之间的源代码。

list ,last

显示从当前行到 last 行之间的源代码。

list first,

显示从 first 行到当前行之间的源代码

一般来说在 list 后面可以跟以下这们的参数：行号，<+offset> 当前行号的正偏移量，<-offset> 当前行号的负偏移量，哪个文件的哪一行，函数名，哪个文件中的哪个函数，

<*address> 程序运行时的语句在内存中的地址。

二、搜索源代码

forward-search (search) regexp

向前面搜索。

reverse-search regexp

全部搜索。

三、指定源文件的路径

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。

directory (dir) 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“;”。

directory

清除所有的自定义的源文件搜索路径信息。

show directories

显示定义了的源文件搜索路径。

四、源代码的内存（Source and machine code）

你可以使用 info line 命令来查看源代码在内存中的地址。info line 后面可以跟“行号”，“函数名”，“文件名:行号”，“文件名:函数名”，这个命令会打印出所指定的源码在运行时的内存地址，如：

```
(gdb) info line tst.c:func
Line 5 of ″tst.c″ starts at address 0x8048456  and ends at 0x804845d .
```

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。

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 : push %ebp
0x8048451 : mov %esp,%ebp
0x8048453 : sub $0x18,%esp
0x8048456 : movl $0x0,0xfffffffc(%ebp)
0x804845d : movl $0x1,0xfffffff8(%ebp)
0x8048464 : mov 0xfffffff8(%ebp),%eax
0x8048467 : cmp 0x8(%ebp),%eax
0x804846a : jle 0x8048470
0x804846c : jmp 0x8048480
0x804846e : mov %esi,%esi
0x8048470 : mov 0xfffffff8(%ebp),%eax
0x8048473 : add %eax,0xfffffffc(%ebp)
```

```
0x8048476 : incl 0xffffffff8(%ebp)
0x8048479 : jmp 0x8048464
0x804847b : nop
0x804847c : lea 0x0(%esi,1),%esi
0x8048480 : mov 0xfffffffc(%ebp),%edx
0x8048483 : mov %edx,%eax
0x8048485 : jmp 0x8048487
0x8048487 : mov %ebp,%esp
0x8048489 : pop %ebp
0x804848a : ret
End of assembler dump.
```

查看运行时的数据

在你调试程序时，当程序被停住时，你可以使用 print 命令（简写命令为 p），或是同义命令 inspect 来查看当前程序的运行数据。print 命令的格式是：

```
print
print /
```

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），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。

一、表达式

print 和许多 GDB 的命令一样，可以接受一个表达式，GDB 会根据当前的程序运行的数据来计算这个表达式，既然是表达式，那么就可以是当前程序运行中的 const 常量、变量、函数等内容。可惜的是 GDB 不能使用你在程序中所定义的宏。

表达式的语法应该是当前所调试的语言的语法，由于 C/C++是一种大众型的语言，所以，本文中的例子都是关于 C/C++的。（而关于用 GDB 调试其它语言的章节，我将在后面介绍）

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。

{}

表示一个指向内存地址的类型为 type 的一个对象。

二、程序变量

在 GDB 中，你可以随时查看以下三种变量的值：

1、全局变量（所有文件可见的）

2、静态全局变量（当前文件可见的）

3、局部变量（当前 Scope 可见的）

如果你的局部变量和全局变量发生冲突（也就是重名），一般情况下是局部变量会隐藏全局变量，也就是说，如果一个全局变量和一个函数中的局部变量同名时，如果当前停止点在函数中，用 print 显示出的变量的值会是函数中的局部变量的值。如果此时你想查看全局变量的值时，你可以使用“::”操作符：

file::variable

function::variable

可以通过这种形式指定你所想查看的变量，是哪个文件中的或是哪个函数中的。例如，查看文件 f2.c 中的全局变量 x 的值：

gdb) p 'f2.c'::x

当然，“::”操作符会和 C++中的发生冲突，GDB 能自动识别“::” 是否 C++的操作符，所以你不必担心在调试 C++程序时会出现异常。

另外，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。这个是很正常的，因为优化程序会删改你的程序，整理你程序的语句顺序，剔除一些无意义的变量等，所以在 GDB 调试这种程序时，运行时的指令和你所编写指令就有不一样，也就会出现你所想象不到的结果。对付这种情况时，需要在编译程序时关闭编译优化。一般来说，几乎所有的编译器都支持编译优化的开关，例如，GNU 的 C/C++编译器 GCC，你可以使用“-gstabs”选项来解决这个问题。关于编译器的参数，还请查看编译器的使用说明文档。

三、数组

有时候，你需要查看一段连续的内存空间的值。比如数组的一段，或是动态分配的数据的大

小。你可以使用GDB的“@”操作符，“@”的左边是第一个内存的地址的值，“@”的右边则你你想查看内存的长度。例如，你的程序中有这样的语句：

```
int *array = (int *) malloc (len * sizeof (int));
```

于是，在GDB调试过程中，你可以以如下命令显示出这个动态数组的取值：

```
p *array@len
```

@的左边是数组的首地址的值，也就是变量array所指向的内容，右边则是数据的长度，其保存在变量len中，其输出结果，大约是下面这个样子的：

```
(gdb) p *array@len
$1 = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40}
```

如果是静态数组的话，可以直接用print数组名，就可以显示数组中所有数据的内容了。

四、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。

```
(gdb) p i
$21 = 101

(gdb) p/a i
```

$22 = 0x65

(gdb) p/c i
$23 = 101 'e'

(gdb) p/f i
$24 = 1.41531145e-43

(gdb) p/x i
$25 = 0x65

(gdb) p/t i
$26 = 1100101

五、查看内存

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：

x/nfu addr

n、f、u 是可选的参数。

n(the repeat count) 是一个正整数，默认为 1，表示从 addr 开始显示内存的长度(按照 units u 计数)，也就是说从当前地址向后显示几个地址的内容。

f(the display format) 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，如果地址是指令地址，那么格式可以是 i。默认是'x'(十六进制)，但默认值将随你使用 x 或 print。

u(the unit size) 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 w(4 个 bytes)。u 参数可以用下面的字符来代替，b(Bytes)表示单字节，h(Halfwords)表示双字节，w(Words)表示四字节，g(Giant word)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。每次指定一个 unit size，在你下一次使用 x 命令时这个 size 将变成默认的 unit。

addr(starting display address)表示一个内存地址。

The expression need not have a pointer value (though it may); it is always

interpreted as an integer address of a byte of memory. The default for addr is usually just after the last address examined--but several other commands also set the default address: info breakpoints (to the address of the last breakpoint listed), info line (to the starting address of a line), and print (if you use it to display a value from memory).

n/f/u 三个参数可以一起使用。例如：

命令 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。

六、自动显示

你可以设置一些自动显示的变量，当程序停住时，或是在你单步跟踪时，这些变量会自动显示。相关的 GDB 命令是 display。

```
display  expr
display/ fmt  expr
display/fmt addr
```

expr 是一个表达式，fmt 表示显示的格式，addr 表示内存地址，当你用 display 设定好了一个或多个表达式后，只要你的程序被停下来，GDB 会自动显示你所设置的这些表达式的值。

格式 i 和 s 同样被 display 支持，一个非常有用的命令是：

```
display/i $pc
```

$pc 是 GDB 的环境变量，表示着指令的地址，/i 则表示输出格式为机器指令码，也就是汇编。于是当程序停下后，就会出现源代码和机器指令码相对应的情形，这是一个很有意思的功能。

下面是一些和 display 相关的 GDB 命令：

```
undisplay dnums ...
delete display dnums ...
```

删除自动显示，dnums 意为所设置好了的自动显式的编号。如果要同时删除几个，编号可以用空格分隔，如果要删除一个范围内的编号，可以用减号表示（如：2-5）

disable display dnums ...

enable display dnums ...

disable 和 enalbe 不删除自动显示的设置，而只是让其失效和恢复。

display

Display the current values of the expressions on the list, just as is done when your program stops.

info display

查看 display 设置的自动显示的信息。GDB 会打出一张表格，向你报告当然调试中设置了多少个自动显示设置，其中包括，设置的编号，表达式，是否 enable。

七、设置显示选项

GDB 中关于显示的选项比较多，这里我只例举大多数常用的选项。

set print address

set print address on

打开地址输出，当程序显示函数信息时，GDB 会显出函数的参数地址。系统默认为打开的，如：

```
(gdb) f
#0 set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
at input.c:530
530 if (lquote != def_lquote)
```

set print address off

关闭函数的参数地址显示，如：

```
(gdb) set print addr off
(gdb) f
```

```
#0 set_quotes (lq="<<", rq=">>") at input.c:530
530 if (lquote != def_lquote)
```

```
show print address
```

查看当前地址显示选项是否打开。

```
set print array
set print array on
```

打开数组显示，打开后当数组显示时，每个元素占一行，如果不打开的话，每个元素则以逗号分隔。这个选项默认是关闭的。与之相关的两个命令如下，我就不再多说了。

```
set print array off
show print array
```

```
set print elements
```

这个选项主要是设置数组的，如果你的数组太大了，那么就可以指定一个来指定数据显示的最大长度，当到达这个长度时，GDB 就不再往下显示了。如果设置为 0，则表示不限制。

```
show print elements
```

查看 print elements 的选项信息。

```
set print null-stop
```

如果打开了这个选项，那么当显示字符串时，遇到结束符则停止显示。这个选项默认为 off。

```
set print pretty on
```

如果打开 printf pretty 这个选项，那么当 GDB 显示结构体时会比较漂亮。如：

```
$1 = {
next = 0x0,
flags = {
sweet = 1,
```

```
sour = 1
},
meat = 0x54 "Pork"
}
```

```
set print pretty off
```

关闭 printf pretty 这个选项，GDB 显示结构体时会如下显示：

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, meat = 0x54 "Pork"}
```

```
show print pretty
```

查看 GDB 是如何显示结构体的。

```
set print sevenbit-strings
```

设置字符显示，是否按“\nnn”的格式显示，如果打开，则字符串或字符数据按\nnn 显示，如“\065”。

```
show print sevenbit-strings
```

查看字符显示开关是否打开。

```
set print union
```

设置显示结构体时，是否显式其内的联合体数据。例如有以下数据结构：

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
Bug_forms;

struct thing {
Species it;
union {
Tree_forms tree;
```

```
Bug_forms bug;
} form;
};

struct thing foo = {Tree, {Acorn}};
```

当打开这个开关时，执行 p foo 命令后，会如下显示：

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

当关闭这个开关时，执行 p foo 命令后，会如下显示：

```
$1 = {it = Tree, form = {...}}
```

```
show print union
```

查看联合体数据的显示方式

```
set print object
```

在 C++中，如果一个对象指针指向其派生类，如果打开这个选项，GDB 会自动按照虚方法调用的规则显示输出，如果关闭这个选项的话，GDB 就不管虚函数表了。这个选项默认是 off。

```
show print object
```

查看对象选项的设置。

```
set print static-members
```

这个选项表示，当显示一个 C++对象中的内容是，是否显示其中的静态数据成员。默认是 on。

```
show print static-members
```

查看静态数据成员选项设置。

```
set print vtbl
```

当此选项打开时，GDB 将用比较规整的格式来显示虚函数表时。其默认是关闭的。

show print vtbl

查看虚函数显示格式的选项。

八、历史记录

当你用 GDB 的 print 查看程序运行时的数据时，你每一个 print 都会被 GDB 记录下来。GDB 会以$1, $2, $3 .....这样的方式为你每一个 print 命令编上号。于是，你可以使用这个编号访问以前的表达式，如$1。这个功能所带来的好处是，如果你先前输入了一个比较长的表达式，如果你还想查看这个表达式的值，你可以使用历史记录来访问，省去了重复输入。

九、GDB 环境变量

你可以在 GDB 的调试环境中定义自己的变量，用来保存一些调试程序中的运行数据。要定义一个 GDB 的变量很简单只需使用 GDB 的 set 命令。GDB 的环境变量和 UNIX 一样，也是以$起头。如：

```
set $foo = *object_ptr
```

使用环境变量时，GDB 会在你第一次使用时创建这个变量，而在以后的使用中，则直接对其赋值。环境变量没有类型，你可以给环境变量定义任一的类型。包括结构体和数组。

show convenience

该命令查看当前所设置的所有的环境变量。

这是一个比较强大的功能，环境变量和程序变量的交互使用，将使得程序调试更为灵活便捷。例如：

```
set $i = 0
print bar[$i++]->contents
```

于是，当你就不必，print bar[0]->contents, print bar[1]->contents 地输入命令了。

输入这样的命令后，只用敲回车，重复执行上一条语句，环境变量会自动累加，从而完成逐个输出的功能。

十、查看寄存器

要查看寄存器的值，很简单，可以使用如下命令：

info registers

查看寄存器的情况。（除了浮点寄存器）

info all-registers

查看所有寄存器的情况。（包括浮点寄存器）

info registers regname ...

查看所指定的寄存器的情况。

GDB has four "standard" register names that are available (in expressions) on most machines--whenever they do not conflict with an architecture's canonical mnemonics for registers. The register names $pc and $sp are used for the program counter register and the stack pointer.$fp is used for a register that contain a pointer to the current stack frame,and $ps is used for a register that contains the processor status.For example,you could print the program counter in hex with

p /x $pc

or print the instruction to be executed next with

x /i $pc

or add four to the stack pointer with

set $sp += 4

寄存器中放置了程序运行时的数据，比如程序当前运行的指令地址（ip），程序的当前堆栈地址（sp）等等。你同样可以使用 print 命令来访问寄存器的情况，只需要在寄存器名字前加一个$符号就可以了。如：p $eip。

改变程序的执行

一旦使用 GDB 挂上被调试程序，当程序运行起来后，你可以根据自己的调试思路来动态地在 GDB 中更改当前被调试程序的运行线路或是其变量的值，这个强大的功能能够让你更好的调试你的程序，比如，你可以在程序的一次运行中走遍程序的所有分支。

一、修改变量值

修改被调试程序运行时的变量值，在 GDB 中很容易实现，使用 GDB 的 print 命令即可完成。如：

```
(gdb) print x=4
```

x=4 这个表达式是 C/C++的语法，意为把变量 x 的值修改为 4，如果你当前调试的语言是 Pascal，那么你可以使用 Pascal 的语法：x:=4。

在某些时候，很有可能你的变量和 GDB 中的参数冲突，如：

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

因为，set width 是 GDB 的命令，所以，出现了“Invalid syntax in expression”的设置错误，此时，你可以使用 set var 命令来告诉 GDB，width 不是你 GDB 的参数，而是程序的变量名，如：

```
(gdb) set var width=47
```

另外，还可能有些情况，GDB 并不报告这种错误，所以保险起见，在你改变程序变量取值时，最好都使用 set var 格式的 GDB 命令。

二、跳转执行

一般来说，被调试程序会按照程序代码的运行顺序依次执行。GDB 提供了乱序执行的功能，也就是说，GDB 可以修改程序的执行顺序，可以让程序执行随意跳跃。这个功能可以由 GDB 的 jump 命令来完：

jump

指定下一条语句的运行点。可以是文件的行号，可以是 file:line 格式，可以是+num 这种偏移量格式。表式着下一条运行语句从哪里开始。

jump *address

这里的

是代码行的内存地址。

注意，jump 命令不会改变当前的程序栈中的内容，所以，当你从一个函数跳到另一个函数时，当函数运行完返回时进行弹栈操作时必然会发生错误，可能结果还是非常奇怪的，甚至于产生程序 Core Dump。所以最好是同一个函数中进行跳转。

熟悉汇编的人都知道，程序运行时，有一个寄存器用于保存当前代码所在的内存地址。所以，jump 命令也就是改变了这个寄存器中的值。于是，你可以使用“set $pc”来更改跳转执行的地址。如：

set $pc = 0x485

三、产生信号量

使用 singal 命令，可以产生一个信号量给被调试的程序。如：中断信号 Ctrl+C。这非常方便于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用 GDB 产生一个信号量，这种精确地在某处产生信号非常有利程序的调试。

语法是：signal ，UNIX 的系统信号量通常从 1 到 15。所以取值也在这个范围。

single 命令和 shell 的 kill 命令不同，系统的 kill 命令发信号给被调试程序时，是由 GDB 截获的，而 single 命令所发出一信号则是直接发给被调试程序的。

四、强制函数返回

如果你的调试断点在某个函数中，并还有语句没有执行完。你可以使用 return 命令强制函数忽略还没有执行的语句并返回。

return
return expression

使用 return 命令取消当前函数的执行，并立即返回，如果指定了，那么该表达式的值会被认作函数的返回值。

五、强制调用函数

call expr

表达式中可以一是函数，以此达到强制调用函数的目的。并显示函数的返回值，如果函数返回值是 void，那么就不显示。

另一个相似的命令也可以完成这一功能——print，print 后面可以跟表达式，所以也可以用他来调用函数，print 和 call 的不同是，如果函数返回 void，call 则不显示，print 则显示函数返回值，并把该值存入历史数据中。

在不同语言中使用 GDB

GDB 支持下列语言：C, C++, Fortran, PASCAL, Java, Chill, assembly, 和 Modula-2。一般说来，GDB 会根据你所调试的程序来确定当然的调试语言，比如：发现文件名后缀为".c"的，GDB 会认为是 C 程序。文件名后缀为".C, .cc, .cp, .cpp, .cxx, .c++"的，GDB 会认为是 C++程序。而后缀是".f, .F"的，GDB 会认为是 Fortran 程序，还有，后缀为如果是".s, .S"的会认为是汇编语言。

也就是说，GDB 会根据你所调试的程序的语言，来设置自己的语言环境，并让 GDB 的命令跟着语言环境的改变而改变。比如一些 GDB 命令需要用到表达式或变量时，这些表达式或变量的语法，完全是根据当前的语言环境而改变的。例如 C/C++中对指针的语法是*p，而在 Modula-2 中则是 p^。并且，如果你当前的程序是由几种不同语言一同编译成的，那到在调

试过程中，GDB 也能根据不同的语言自动地切换语言环境。这种跟着语言环境而改变的功能，真是体贴开发人员的一种设计。

下面是几个相关于 GDB 语言环境的命令：

show language

查看当前的语言环境。如果 GDB 不能识为你所调试的编程语言，那么，C 语言被认为是默认的环境。

info frame

查看当前函数的程序语言。

info source

查看当前文件的程序语言。

如果 GDB 没有检测出当前的程序语言，那么你也可以手动设置当前的程序语言。使用 set language 命令即可做到。

当 set language 命令后什么也不跟的话，你可以查看 GDB 所支持的语言种类：

```
(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

于是你可以在 set language 后跟上被列出来的程序语言名，来设置当前的语言环境。

GDB 的本站：http://www.gnu.org/software/gdb/

GDB 的官方手册下载区：http://www.gnu.org/software/gdb/documentation/

国外的一个 gdb 除错示范：http://www-2.cs.cmu.edu/~gilpin/tutorial/

Study-area 的除错教学：http://www.study-area.org/cyril/opentools/opentools/debug.html