基本要义

不积跬步，无以至千里。不积小流，无以成江海。

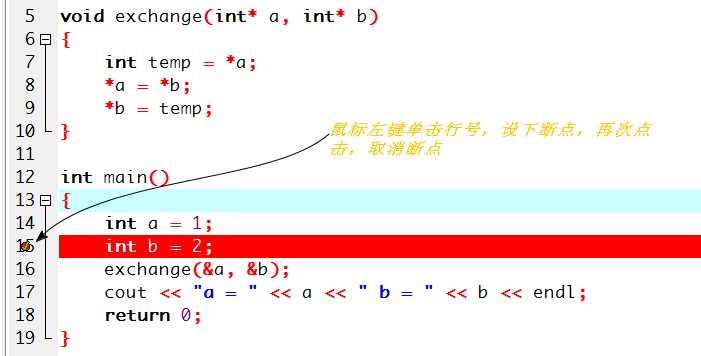
## 调试

调试是编码最基本不过的东西了，在这里，我以一个简单的程序为例，稍微来讲解一下dev-c++中调试的基本技巧。

我的代码如下，非常简单的一个交换值的程序：

|  |
| --- |
| #include <iostream>  using namespace std;  void exchange(int\* a, int\* b)  {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  int main()  {  int a = 1;  int b = 2;  exchange(&a, &b);  cout << "a = " << a << " b = " << b << endl;  return 0;  } |

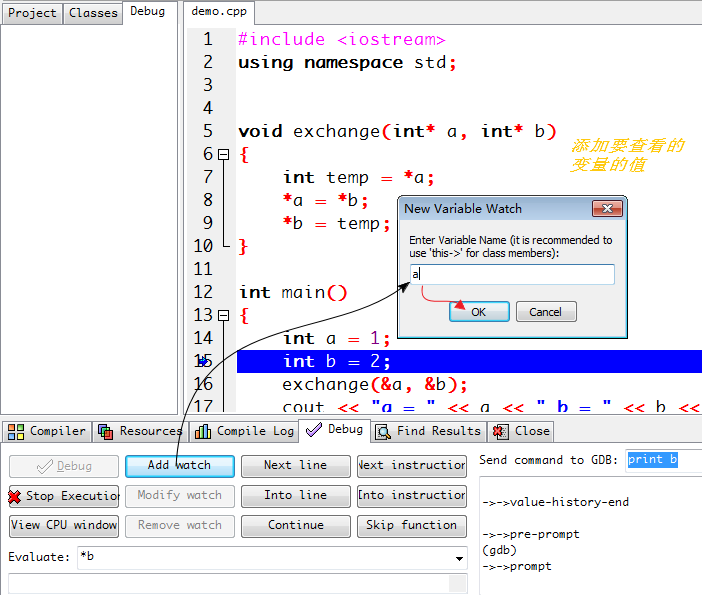
### 断点



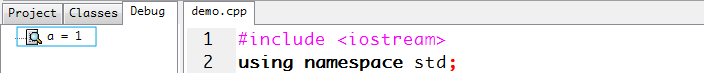
所谓断点，指的就是在你debug的时候，程序运行到你的断点处，会暂时停下来，不会继续运行。在你的代码中，如果你怀疑你的代码哪里可能出错了，你就可以在那里设置断点。

### watch

debug的话自然是少不了查看变量的值的工具，下面是一个例子，当我真正开始debug的时候，程序在断点处，也就是蓝色箭头指示的15行处，暂时停止运行，此时，如果我想查看变量a的值的话，我就要对变量添加watch了，如下图所示。



添加完成后，就可以在左边的框中查看变量的值了。

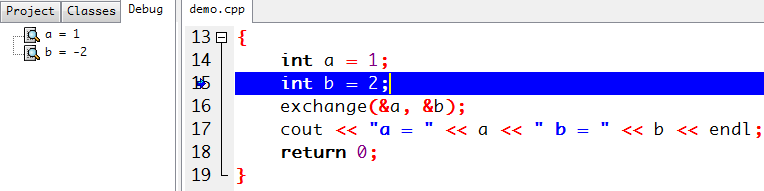


值得一提的是，变量的值会随之程序的运行发生变化。

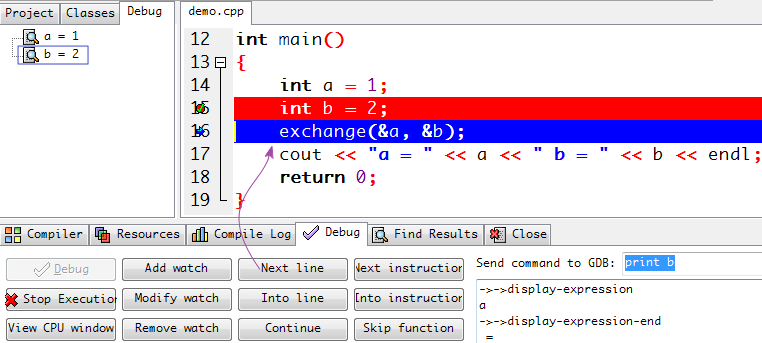
### 单步执行与单步进入

如果我想仔细查看端点处代码究竟干了一些什么事情，那么debug的另一个基本的命令单步执行我们就必须要知晓了。

这是单步执行之前的状态：

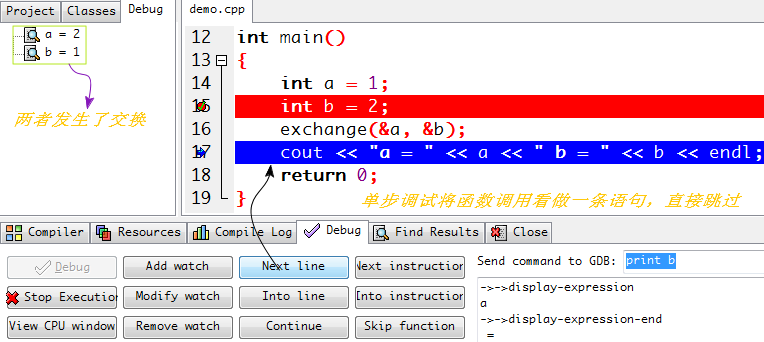


单步执行之后的状态，b的值发生了变化，指示将要运行的代码的蓝色箭头往下移动了一步（这就是所谓的单步）。

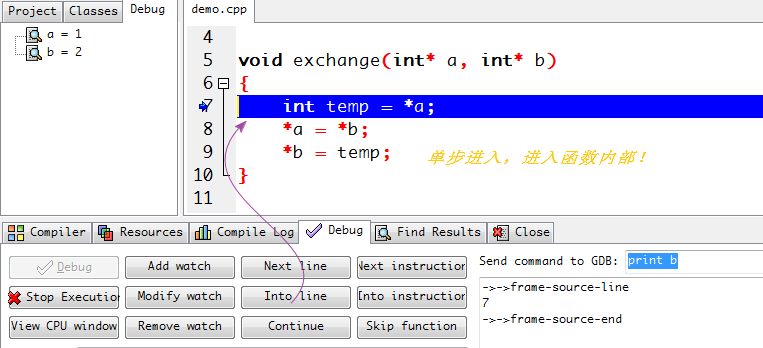


与单步执行非常近似的一个概念叫做单步进入，在没有函数调用的时候，单步执行和单步进入没有什么区别，但是有函数调用的时候区别就来了。

上面的exchange是一个函数调用。我们就以此为例，看看两者的区别：



单步调试将函数调用看做一条语句，直接跳过，而单步进入会进入函数的内部：



相信你已经掌握了两者的用法了。

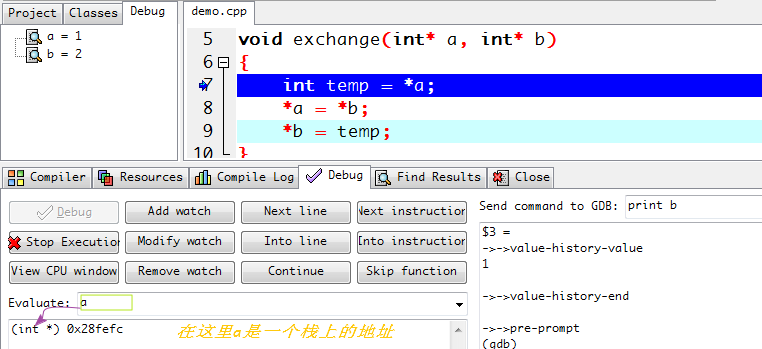
### 继续执行

另外一个比较重要的调试命令是上面图中的Continue,这个命令很简单，它代表继续执行，一直到碰到下一个断点，当然，如果没有断点的话，会直接执行完整个程序。

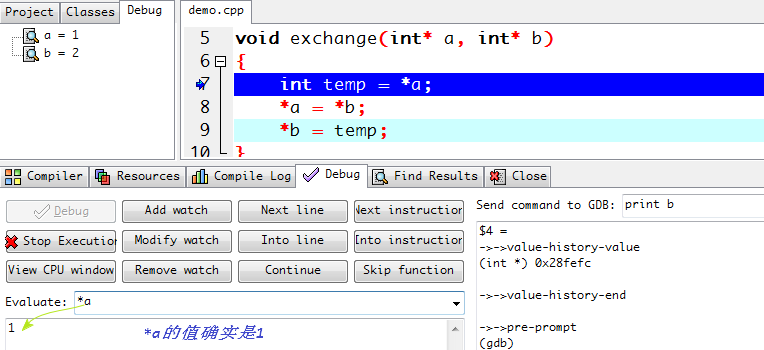
掌握了上面的几个调试技巧，个人觉得差不多了。

### Evaluate

这个不算命令啦，顶多算是一个小插曲而已。



看到上面的Evaluate没有，这里可以显示变量的值。在比如说\*a：



其实在Send command to GDB那里可以输入调试的命令，可是并不推荐，如果感兴趣，我写过一篇linux下gdb调试的总结的文章，当做附录送给你好了。

好了，总结一下吧，程序输出结果不对是经常有的事情，所以程序员大部分的时间里都在调试，调试的时候，最好拿出一页纸，将你预想的结果和程序实际运行的结果对比一下，这样你就能很快找到bug所在的地方。Good luck！

## 常用函数

我并不会讲很多的函数，其余的函数大家可以去查api，这里只会讲一些平时用得特别多的一些函数的用法。

### 常用的字符函数

#### strcpy用于复制字符串

|  |
| --- |
| include <string.h>  char \*strcpy(char \*dest, const char \*src);  char \*strncpy(char \*dest, const char \*src, size\_t n); |

用源地址处(src)取出字符串复制到目的地址处(dest),strcpy函数要求你自己保证dest处有足够的空间。而strncpy多出的一个参数n用于指示dest处最大的可用空间的数目（单位是字节）。如果src的字符数多余n的话，不会全部复制，只会复制n个字符到dest处。

#### strlen用于检测字符的长度

|  |
| --- |
| #include <string.h>  size\_t strlen(const char \*s); |

用与获取字符串s的长度，但是不包括结尾的’\0’字符，返回长度值。

#### strcmp用于比较两个字符串的函数

|  |
| --- |
| #include <string.h>  int strcmp(const char \*s1, const char \*s2);  int strncmp(const char \*s1, const char \*s2, size\_t n); |

很简单，对s1和s2两个串逐个字符进行比较，如果s1中的某个字符的ascii码的值大于s2中对应的字符的ascii码的值，返回的值大于0，小于的话，返回值小于0，对于strcmp函数而言，如果一直比较到s1的结尾字符’\0’,还没有返回的话，返回0。对于strncmp，比较了指定的n个字符还未返回的话，就返回0。

#### strstr获取子串

|  |
| --- |
| #include <string.h>  char \*strstr(const char \*haystack, const char \*needle); |

strstr函数用于寻找子串needle在haystack串中第一次出现的位置。

如果在haystack中找到了needle子串，那么返回haystack中子串开始位置的指针，否则的话，返回NULL。

### 常用的输入输出函数

#### 输出函数

|  |
| --- |
| #include <stdio.h>  int printf(const char \*format, ...);  int sprintf(char \*str, const char \*format, ...);  int snprintf(char \*str, size\_t size, const char \*format, ...); |

printf函数向标准输出输出数据。sprintf将数据输出到str指针指向的字符串。snprintf多了一个n,说明这是一个安全版本的sprintf函数,可以避免向str指向的字符数组写入过多的数据，前提是你要将size填充为str指向字符串的长度!

我这里给出一个例子。

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {  char str[512];  float f = 1.23455678;  sprintf(str, "%f", f);  cout << str << endl;  return 0;  } |

输出如下：



其实这个函数只是将字符数组当做了输出对象而已，而printf函数是将标准输出当做了输出对象。

#### 输出函数

|  |
| --- |
| #include <stdio.h>  int scanf(const char \*format, ...);  int sscanf(const char \*str, const char \*format, ...); |

这里主要讲一下sscanf函数吧。这个函数，一个例子足矣。

|  |
| --- |
| #include <iostream>  using namespace std;  char buf[1024] = "hi\_girls 1.234 12";  int main()  {  char str[512];  float f = 0;  int i = 0;  sscanf(buf, "%s %f %d", str, &f, &i);  cout << str << ' ' << f << ' ' << i << endl;  return 0;  } |

输出的结果如下：

C:\Users\Administrator\Desktop\图像 2.png

请仔细体会一下这个函数的神奇之处，这个函数直接将字符数组str当做了输入，而scanf函数是将标准输入当做了输入，这个函数有些时候很好用。希望你能够用到吧。

#### 获取一行输入数据的函数

|  |
| --- |
| #include <stdio.h>  char \*gets(char \*s); |

gets从stdin中读入一行内容到s指定的buffer中，当遇到换行符或EOF时读取结束。读取成功时，返回s地址；失败时返回null。需要注意的是，gets会将行末尾的'\n'字符或EOF替换成'\0'，这样，gets读取的内容中不包括'\n'字符。当然，这个函数在实际中是被废弃掉的，因为很容易缓冲区溢出。不过ccf里用一下问题也不是很大。

|  |
| --- |
| #include <stdio.h>  char \*fgets(char \*s, int size, FILE \*stream); |

fgets从stream中读取最多size-1大小的内容到s指定的buffer中，当遇到换行符或EOF时读取结束。读取成功时，返回s地址；失败时返回null。需要注意的是，fgets会在所读取的内容后面添加'\0'，这样，fgets读取的内容中会包括行末尾的'\n'字符。如果要获取读取字符串的长度，可以调用strlen函数获得。

## 排序

关于排序，不得不提一下sort函数。下面是sort函数的原型。

|  |
| --- |
| #include <algorithm>  template<class \_RanIt, class \_Pr>  void sort(\_RanIt \_First, \_RanIt \_Last, \_Pr \_Pred) |

了解了这个sort函数，ccf里面关于数组排序的题都应该没有什么难度了。

当然，这个函数使用到了泛型，所谓的泛型，其实很简单，对于sort函数来说，原来只能往里面放int，char等具体的类型，并且一种类型我们要写一个对应的sort来排序，但是泛型的话，你只需要写一个sort即可，不管什么类型都可以往里面放，这个函数可不管你放的是什么东西，你只需要告诉我怎么排序即可。这也是后面为什么有一个\_Pred的原因。

\_First存放你要排序的数组的首地址，然后\_Last存放你要排序的数组的尾部的地址，\_Pred用于指定应该如何排序。

我举一个栗子。

我们经常会有对结构体进行排序的需求，假设有这么一个结构体：

|  |
| --- |
| struct STU {  int id; // 学生id号  int grades; // 学生成绩  }; |

给定一个描述学生信息的STU类型的长度为k的数组stuinfo，如果我要求你这么来排序，首先按照成绩从高低排序，如果成绩相同，那么id号比较小的排在前面，你如何来排序呢？

我不太推荐自己去写排序函数，因为容易出错，并且非常耗时间。怎么办呢？

用sort函数来排序，sort(stuinfo, stuinfo + k, …),现在的问题来了，就是前面的\_Pred这个东西应该如何来写。

我这里要说一点，sort实现用的是优化后的快排，你自己写的排序函数速度很难达到sort函数的级别。类似于我们平常写的排序，我们通过 a > b 这样的比较来确定a和b的大小，如果a > b返回真的话，我们可以说a比b要大，否则就要小。sort函数里面也需要通过\_Pred(a, b)的形式来确定a和b的大小\_Pred(a, b)返回true，那么sort函数会认为，a比b要“大”，a应该排在前面。所以，你要写排序函数的话，可以参照下面的形式。

|  |
| --- |
| bool cmp(const STU& l, const STU& r)  {  if (l.grade != r.grade) // 成绩不等的话  return l.grade > r.grade; // 如果左边的成绩比右边的好，左边的排在前面  else  return l.id < r.id; // 成绩相等，如果左边的id小于右边的id，那么排在前面  // 总之，如果这个函数返回true的话，说明l比r“大”，应该排在前面  // 这个大的含义是多样的，如果我我们将小看做大的话  // return l.grade < r.grade; 将会实现成绩从低到高排序  // 这样来理解，如果l的成绩小的话，那么应该排在前面  } |

当然，比较函数还有非常多花哨的写法，这里我就不再叙述。

写成sort(stuinfo, stuinfo + k, cmp),即可以实现我们的目的。如果写成sort(stuinfo + 1, stuinfo + k, cmp),当然这里k应该大于等于1，这样的话，可以实现对k – 1个元素的排序。依此类推。

## 一些细枝末节的东西

### 输入的数据有若干组

这里举一个例子，有的题目有时候会有这样的描述：输入的数据有若干组。但是就是不告诉你输入数据的组数。这样的题目，我们读入的输入最终都会返回一个EOF，即文件结束符。我们怎样处理这种状况？其实很简单。

我这里给出一个很小的例子。

|  |
| --- |
| 假设我输入的数据有若干组，我现在要求你原样输出我输入的数据。  输入  100 200 300  输出  100  200  300 |

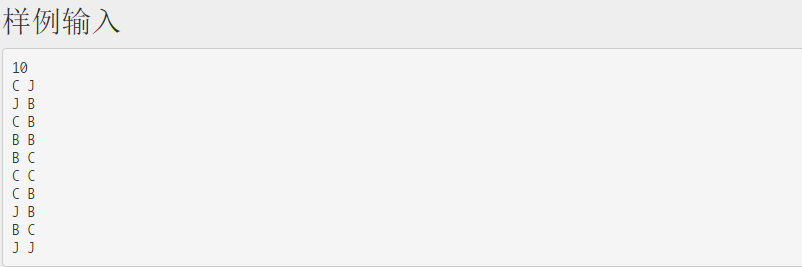
代码片段：

|  |
| --- |
| int num;  while (cin >> num)  {  cout << num << endl;  } |

我这里来说明一下，第一次调用cin >> num，这个时候缓冲区里是有数据的，这个式子会返回cin本身的一个引用(所以不为false，继续循环)，将num置为100，直至num为300，然后到了结尾的地方，也就是EOF的位置，这个时候调用cin >> num的话，会返回null，跳出while循环。也就达到了我们的目的。

### 大批量的输入

如果题目中碰到像下面的大批量的输入数据，我们大可不必一个一个地输入，直接复制下下面的数据。



然后，点击程序左上方的图标，选择编辑->粘贴即可。



### 输入EOF

那么如何输入文件结尾符呢？也就是我们常说的EOF，按ctrl + z即可。

### 字符和数字的转换

在C中，char类型其实也是一个数字，只不过它的字长比较短而已。我们可以这样将char类型的数字转换为int类型的数字。

|  |
| --- |
| char a = '1';  int b = a - '0'; |

这个时候b的值就变成了1。之所以可以这么干，是因为ASCII表中数字其实是按照数字的大小连续地排列在一起的。

将int类型的数字转换为char类型的数字也很简单，下面是一个例子：

|  |
| --- |
| int a = 1;  char b = 1 + '0'; |

这个时候b的值就已经是’1’了。

# 附录

## gdb常用命令

在调试程序的时候，gdb是一柄利器，恰当的使用gdb可以解决掉程序的许多bug。

gdb并不检查语法错误，那是gcc或者g++的事情，gdb干的是调试的事情。

说明：

1. gdb 程序名 [corefile]之类的是代表命令的用法，[]中间的内容是可选项，即你可以加，也可以不加。
2. 如果需要重复执行一条命令，不需要每次都键入命令，gdb记住了最后一个被执行的命令，只要简单的按enter键就可以重复执行最后的命令。

### gdb命令

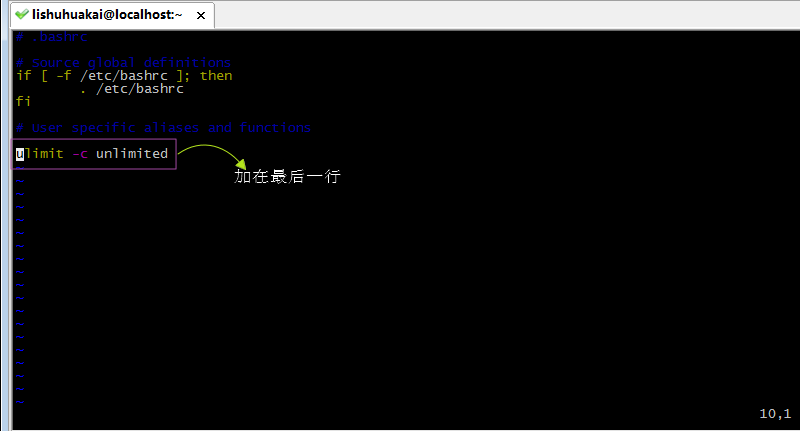
该命令主要用来启动调试。

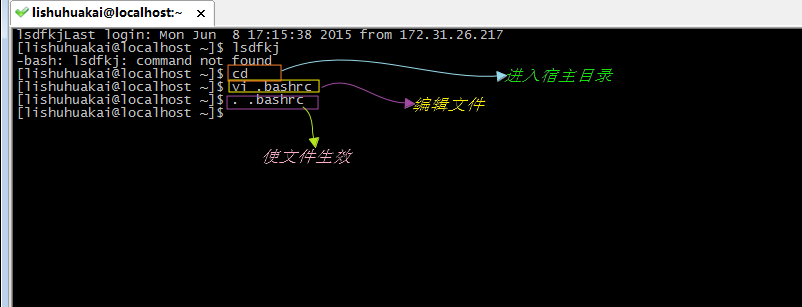
gdb 程序名 [corefile]

corefile是可选的，但能增强gdb的调试能力。Linux默认是不生成corefile的，所以需要在.bashrc文件中添加

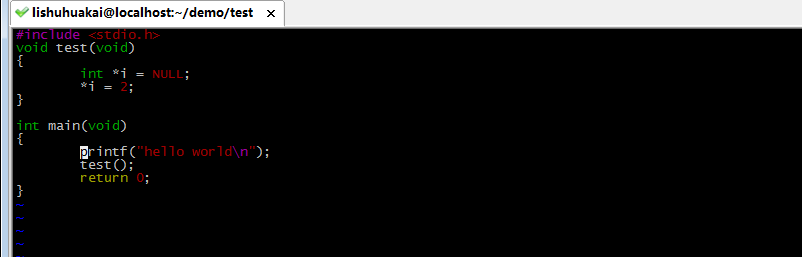
ulimit -c unlimited

修改完.bashrc文件后记得. .bashrc让修改生效。





下面是一个没有语法错误，但是存在逻辑错误的代码：



一运行立马就会提示错误：

Segmentation fault (core dumped)

我们列出当前目录下的文件，发现多了一个core.\*之类的文件，这就是系统给我们生成的core文件。

我们现在可以启动gdb进行调试了。

gdb 1 core.1997

其中1是代码生成的程序，core.1997是出错后系统给我们生成的core文件。

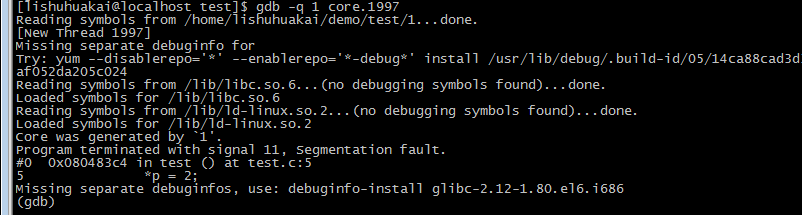
如果你不喜欢一大堆的软件信息，可以通过-q参数关闭软件信息

gdb -q 1 core.1997

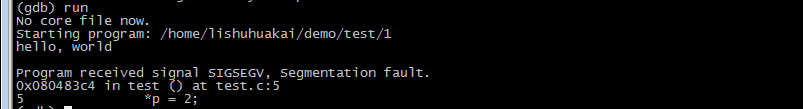
#0 0x080483c4 in test () at test.c:5

5 \*p = 2;

可以看到gdb通过core告诉你，程序哪条语句出现问题



### run命令



该命令使得程序跑起来，需要注意：gdb命令并没有运行程序，只是进入了gdb状态。

### continue命令

与run相对的是continue命令，记住，run是开始执行，continue是继续执行，两者是不同的，程序在断点处听下之后，你如果输入run，程序会重新启动，而输入continue，程序会从断点处向下继续执行。

### where命令

where命令，可以显示导致段错误的执行函数处。

C:\Users\李懿虎\Desktop\Image 7.png

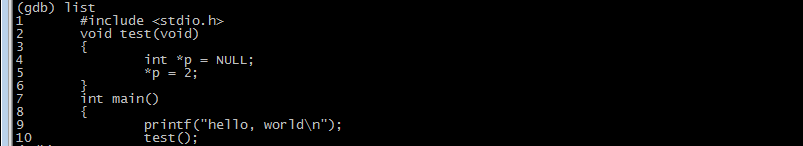
#0 0x080483c4 in test () at test.c:5

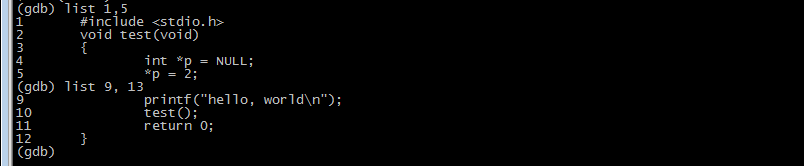
#1 0x080483e6 in main () at test.c:10

### list命令

知道函数出错行的上下文对调试程序是很有帮助的。

list [m,n]，m,n是要显示包含错误首次出现位置的起始行和结尾行。不带参数的list将显示附近的10行代码。





### break命令

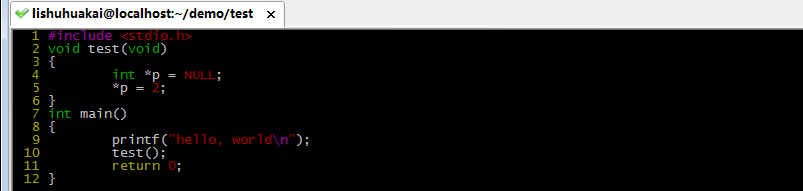
break命令主要用来设置断点。具体用法如下：

break linenum在文件的linenum行设置断点；

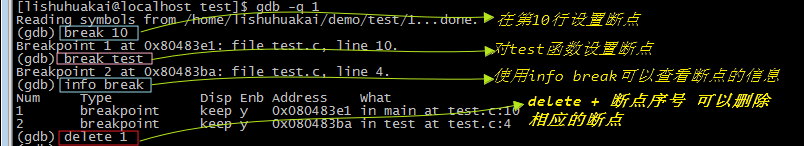
break funcname对funcname函数设置断点，每次该函数被调用都会触发断点；

break filename:linenum在filename文件的linenum行设置断点；

break filename:funcname在filename文件对funcname函数设置断点。



对于上面的一段代码，我们对test函数设置断点，在第10行设置断点：



info break可以查看已有的断点的信息。

delete + 断点序号 可以删除断点。

### 单步调试命令

step命令：step顾名思义，就是一步一步执行。当遇到一个函数的时候，step将进入函数，每次执行一条语句，相当于step into。

next命令：当遇到一个函数的时候，next将执行整个函数，相当于step over。

### print命令

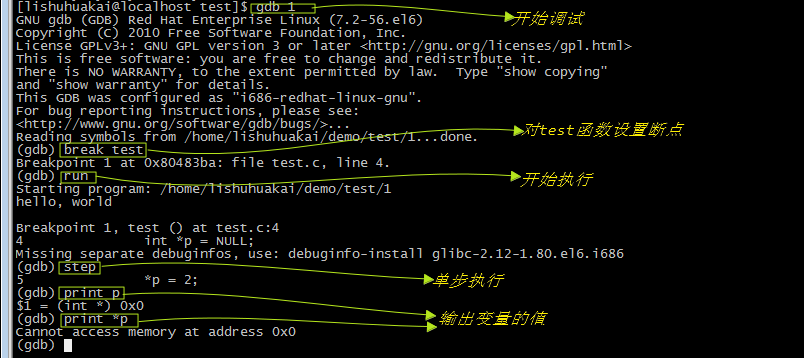
gdb最有用的功能之一就是它可以显示被调试程序中任何表达式、变量的值。

print 变量，表达式。

print ‘file’::变量，表达式,‘’是必须的，以便让gdb知道指的是一个文件名。

print funcname::变量，表达式

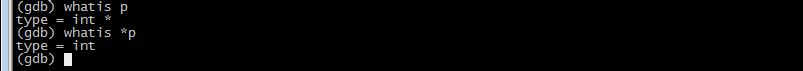
我们先对test函数设置断点，然后单步执行，然后输出i的值：



我们可以看到，print命令确实强大，方便地输出了变量的值。

### whatis命令

whatis 命令可以告诉你变量的类型， ptype 告诉你结构的定义。

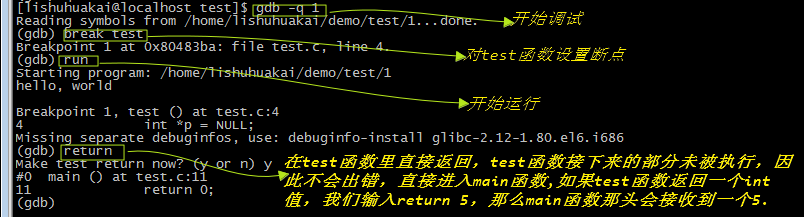


### return命令

return [value]

停止执行当前函数，将value返回给调用者，相当于step return。

执行该命令，会让当前的函数立马退出，并且返回。

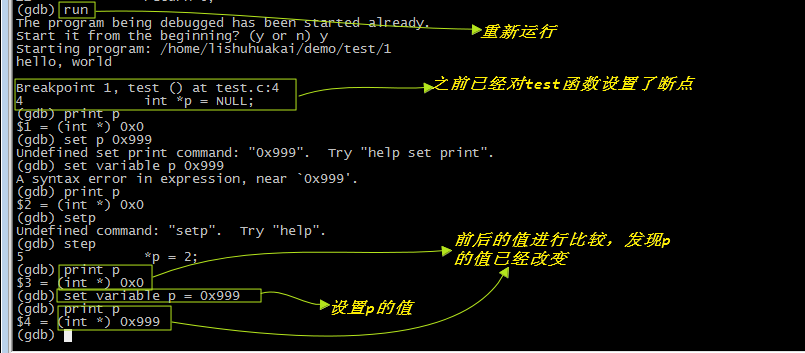


### set命令

该命令可以改变一个变量的值。

set variable varname = value

varname是变量名称，value是变量的新值。



当然gdb还有非常多复杂的命令，不过它们用到的机率非常低，熟练地掌握了上面的命令，一般应付大部分的调试都不存在问题。