

基本要义

不积跬步，无以至千里。不积小流，无以成江海。

一. 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。

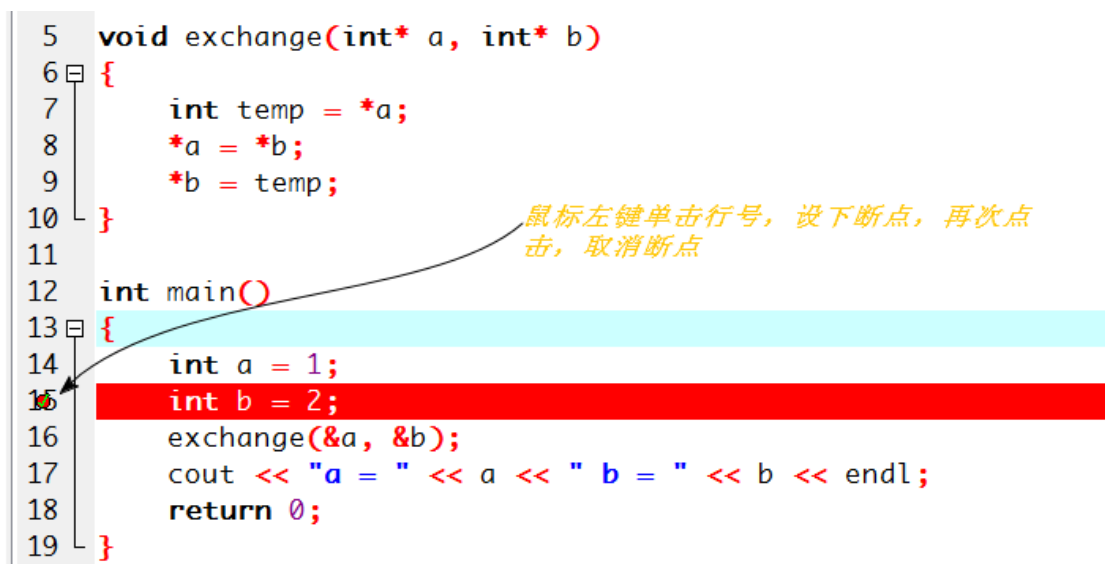
我的代码如下，非常简单的一个交换值的程序：

```
#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;
}
```

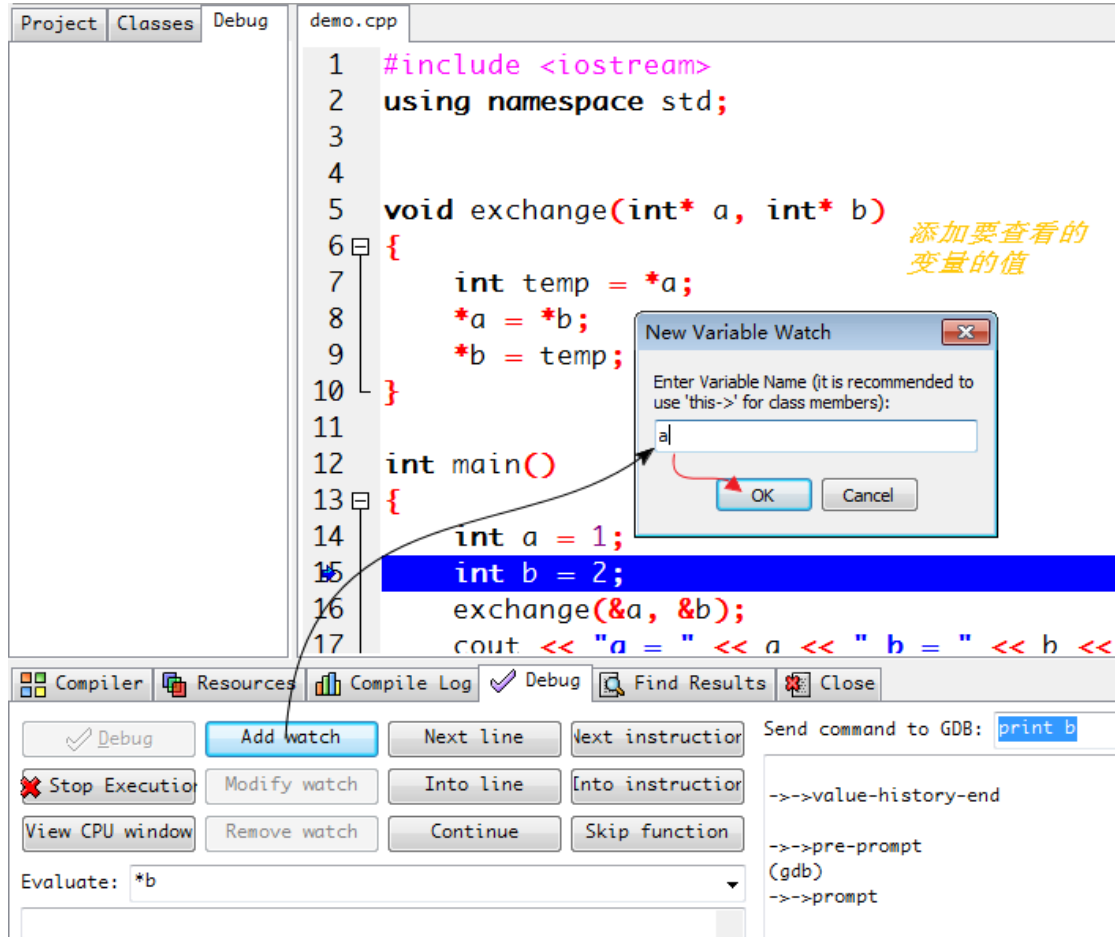
1. 断点



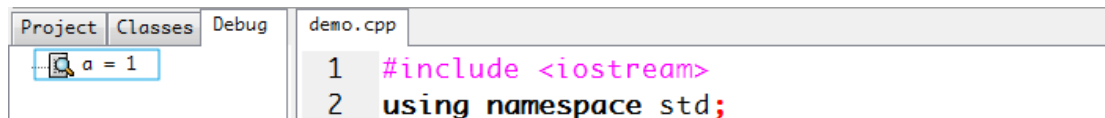
所谓断点，指的就是在你 debug 的时候，程序运行到你的断点处，会暂时停下来，不会继续运行。在你的代码中，如果你怀疑你的代码哪里可能出错了，你就可以在那里设置断点。

2. 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。

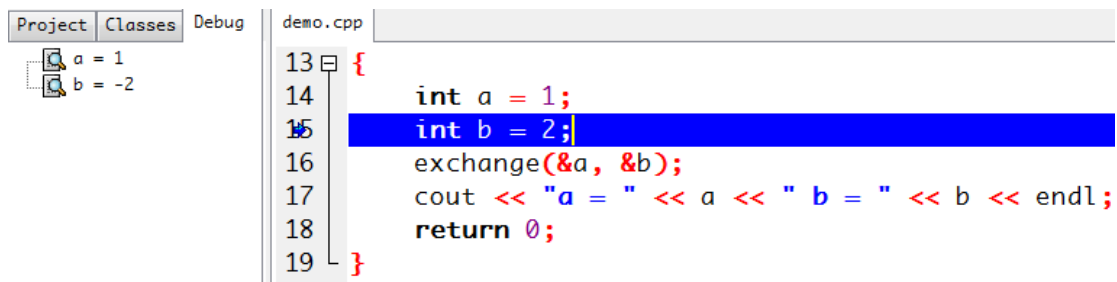


值得一提的是，变量的值会随之程序的运行发生变化。

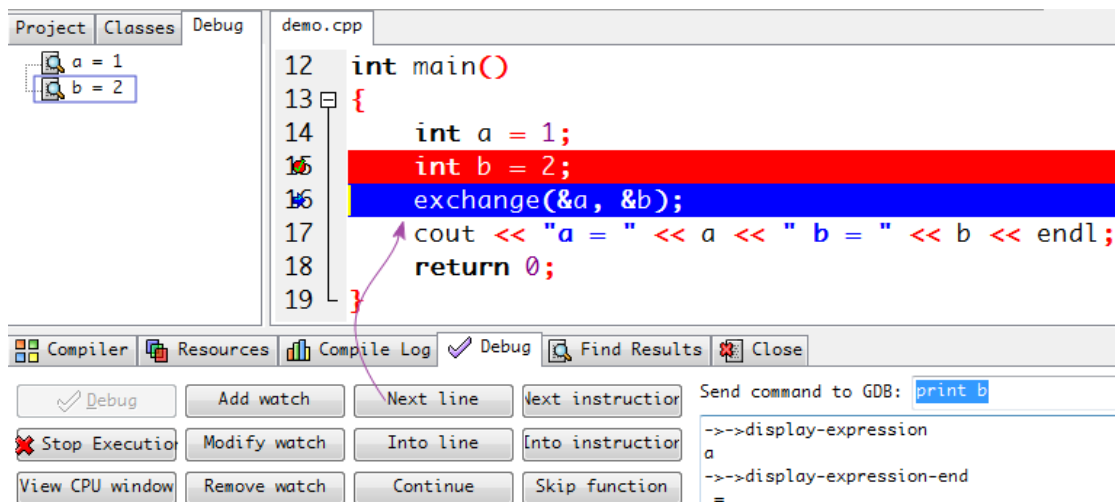
3. 单步执行与单步进入

如果我想仔细查看断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：

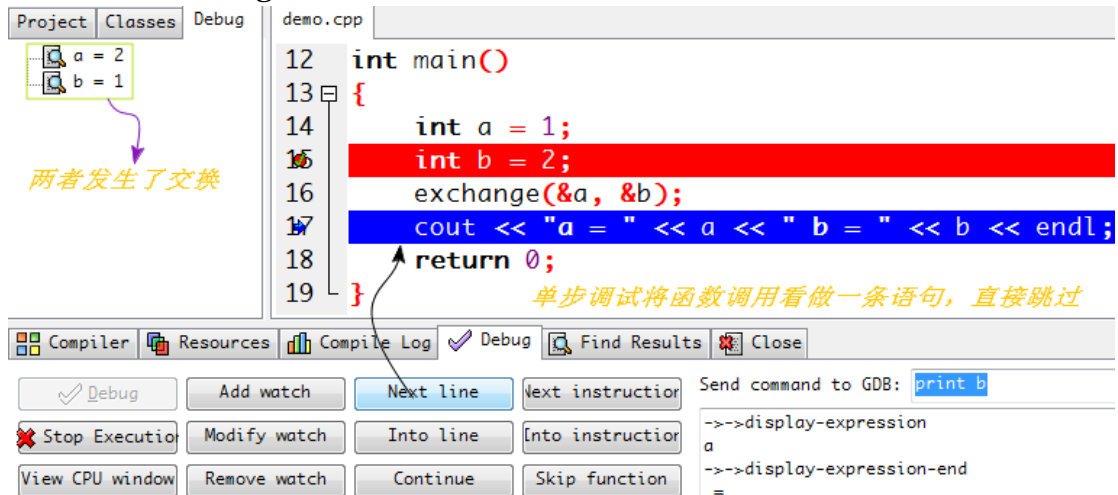


单步执行之后的状态，**b** 的值发生了变化，指示将要运行的代码的蓝色箭头往下移动了一步（这就是所谓的单步）。

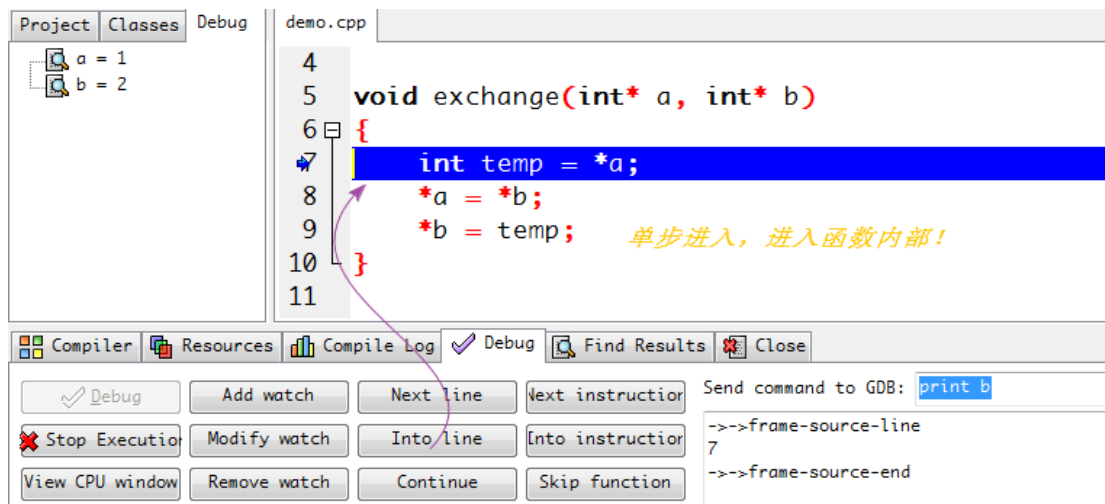


与单步执行非常近似的一个概念叫做单步进入，在没有函数调用的时候，单步执行和单步进入没有什么区别，但是有函数调用的时候区别就来了。

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。

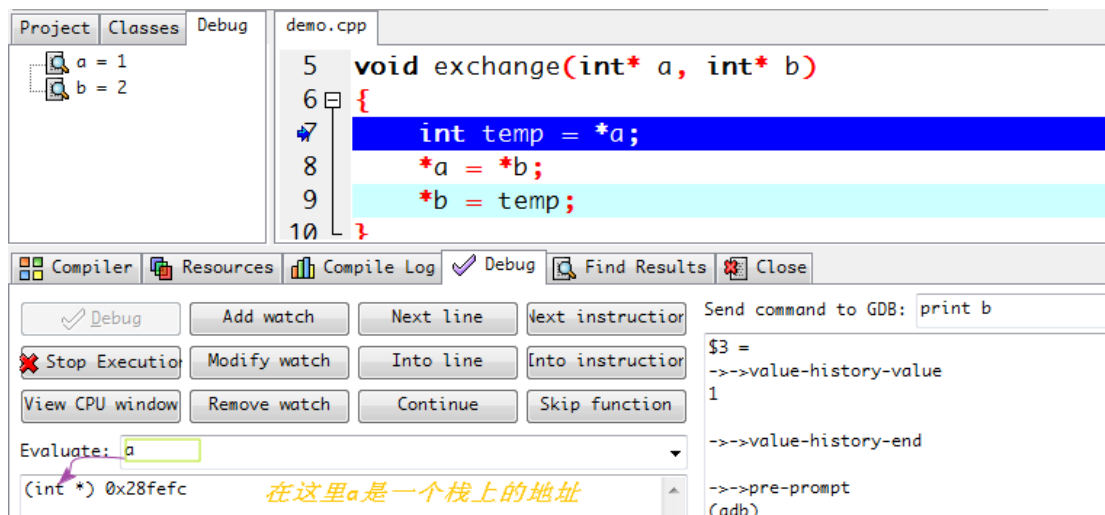
4. 继续执行

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。

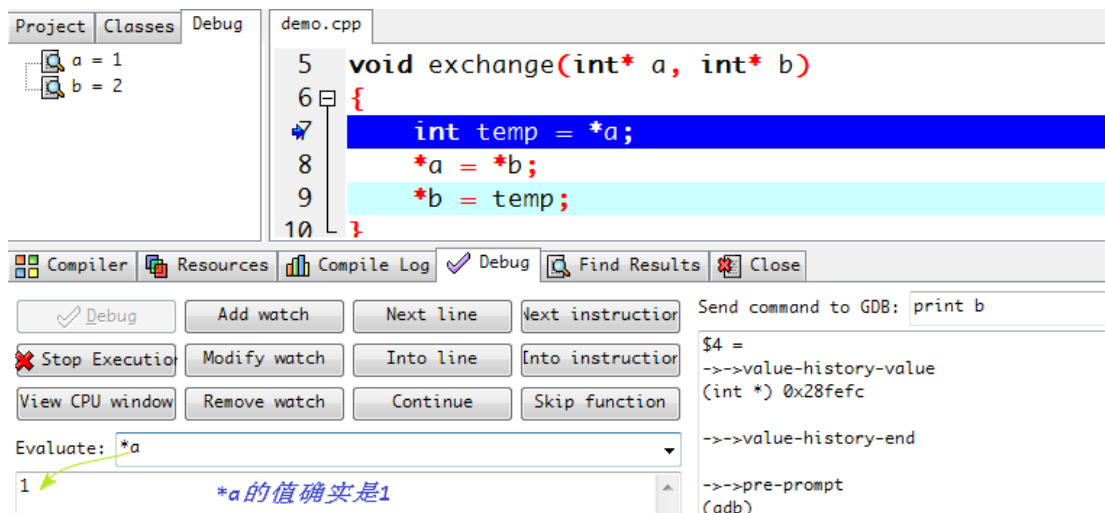
掌握了上面的几个调试技巧, 个人觉得差不多了。

5. 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!

二. 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。

1. 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`。

2. 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：



```
1.234557
请按任意键继续. . .
```

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

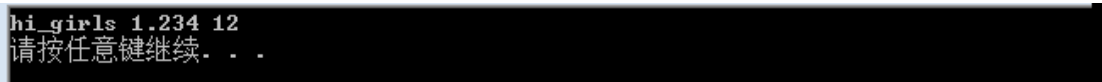
```
#include <stdio.h>
int scanf(const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

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：



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。

三.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。

四. 一些细枝末节的东西

1. 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。

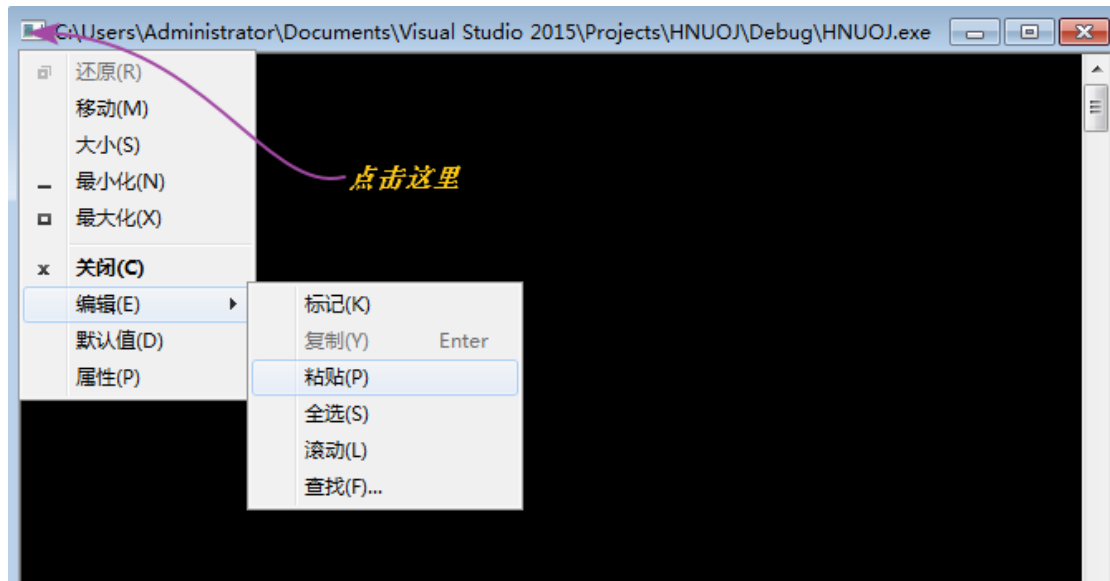
2. 大批量的输入

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。

样例输入

```
10  
C J  
J B  
C B  
B B  
B C  
C C  
C B  
J B  
B C  
J J
```

然后，点击程序左上方的图标，选择编辑->粘贴即可。



3. 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。

4. 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

一.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。

1.gdb 命令

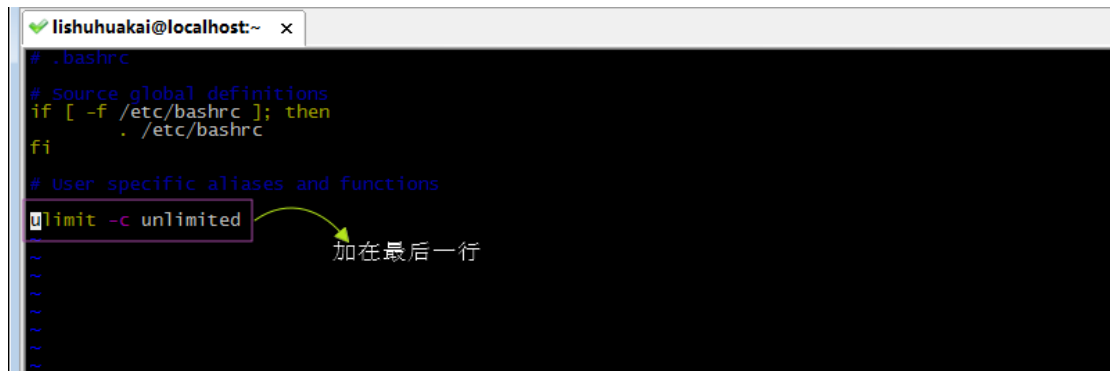
该命令主要用来启动调试。

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

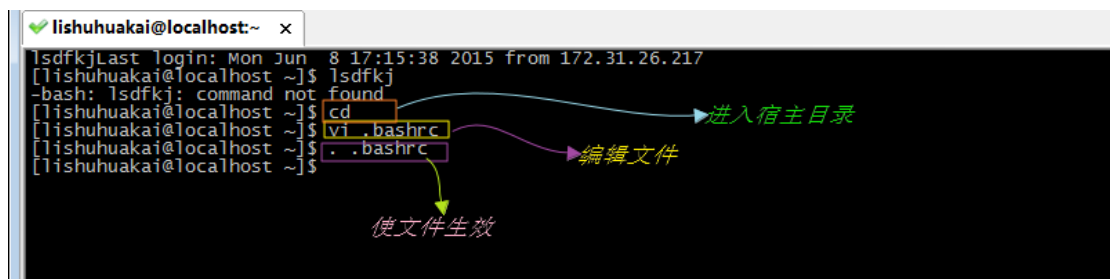
ulimit -c unlimited

修改完 .bashrc 文件后记得 **.bashrc** 让修改生效。



```
lshuhuakai@localhost:~ x
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
# User specific aliases and functions
ulimit -c unlimited
~
~
~
~
~
```

加在最后一行



```
lshuhuakai@localhost:~ x
lsdfkjLast login: Mon Jun  8 17:15:38 2015 from 172.31.26.217
[lshuhuakai@localhost ~]$ lsdfkj
-bash: lsdfkj: command not found
[lshuhuakai@localhost ~]$ cd
[lshuhuakai@localhost ~]$ vi .bashrc
[lshuhuakai@localhost ~]$ . .bashrc
[lshuhuakai@localhost ~]$
```

进入宿主目录
编辑文件
使文件生效

下面是一个没有语法错误，但是存在逻辑错误的代码：

```
lishuhuakai@localhost:~/demo/test x
#include <stdio.h>
void test(void)
{
    int *i = NULL;
    *i = 2;
}

int main(void)
{
    printf("hello world\n");
    test();
    return 0;
}
```

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

```
[lishuhuakai@localhost test]$ gdb -q 1 core.1997
Reading symbols from /home/lishuhuakai/demo/test/1...done.
[New Thread 1997]
Missing separate debuginfo for
Try: yum --disablerepo='*' --enablerepo='*-debug*' install /usr/lib/debug/.build-id/05/14ca88cad3d
af052da205c024
Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `1'.
Program terminated with signal 11, Segmentation fault.
#0 0x080483c4 in test () at test.c:5
5 *p = 2;
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6.i686
(gdb)
```

2.run 命令

```
(gdb) run
No core file now.
Starting program: /home/lishuhuakai/demo/test/1
hello, world

Program received signal SIGSEGV, Segmentation fault.
0x080483c4 in test () at test.c:5
5 *p = 2;
```

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。

3.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。

4. where 命令

`where` 命令，可以显示导致段错误的执行函数处。

```
(gdb) where
#0  0x080483c4 in test () at test.c:5
#1  0x080483e6 in main () at test.c:10
```

```
#0  0x080483c4 in test () at test.c:5
#1  0x080483e6 in main () at test.c:10
```

5. 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。

```
(gdb) list
1  #include <stdio.h>
2  void test(void)
3  {
4      int *p = NULL;
5      *p = 2;
6  }
7  int main()
8  {
9      printf("hello, world\n");
10     test();
11 }
```

```
(gdb) list 1,5
1  #include <stdio.h>
2  void test(void)
3  {
4      int *p = NULL;
5      *p = 2;
(gdb) list 9, 13
9      printf("hello, world\n");
10     test();
11     return 0;
12 }
(gdb)
```

6. 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。

```
lshuhukai@localhost:~/demo/test x
1 #include <stdio.h>
2 void test(void)
3 {
4     int *p = NULL;
5     *p = 2;
6 }
7 int main()
8 {
9     printf("hello, world\n");
10    test();
11    return 0;
12 }
```

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：

```
[lshuhukai@localhost test]$ gdb -q 1
Reading symbols from /home/lshuhukai/demo/test/1...done.
(gdb) break 10
Breakpoint 1 at 0x80483e1: file test.c, line 10.
(gdb) break test
Breakpoint 2 at 0x80483ba: file test.c, line 4.
(gdb) info break
Num   Type             Disp Enb Address          what
1     breakpoint       keep y   0x080483e1 in main at test.c:10
2     breakpoint       keep y   0x080483ba in test at test.c:4
(gdb) delete 1
```

在第10行设置断点
对test函数设置断点
使用info break 可以查看断点的信息
delete + 断点序号 可以删除相应的断点

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。

7.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**。

8.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:

```
[lshuhuakai@localhost test]$ gdb 1
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-56.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/lshuhuakai/demo/test/1...done.
(gdb) break test
Breakpoint 1 at 0x80483ba: file test.c, line 4.
(gdb) run
Starting program: /home/lshuhuakai/demo/test/1
hello, world

Breakpoint 1, test () at test.c:4
4       int *p = NULL;
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6.i686
(gdb) step
5       *p = 2;
(gdb) print p
$1 = (int *) 0x0
(gdb) print *p
Cannot access memory at address 0x0
(gdb) 
```

我们可以看到, **print** 命令确实强大, 方便地输出了变量的值。

9.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。

```
(gdb) whatis p
type = int *
(gdb) whatis *p
type = int
(gdb) 
```

10.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。

```
[lishuhuakai@localhost test]$ gdb -q 1
Reading symbols from /home/lishuhuakai/demo/test/1...done.
(gdb) break test
Breakpoint 1 at 0x80483ba: file test.c, line 4.
(gdb) run
Starting program: /home/lishuhuakai/demo/test/1
hello, world

Breakpoint 1, test () at test.c:4
4       int *p = NULL;
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6.i686
(gdb) return
Make test return now? (y or n) y
#0  main () at test.c:11
11      return 0;
(gdb)
```

开始调试
对test函数设置断点
开始运行
在test函数里直接返回，test函数接下来的部分未被执行，因此不会出错，直接进入main函数，如果test函数返回一个int值，我们输入return 5，那么main函数那头会接收到一个5。

11.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。

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/lishuhuakai/demo/test/1
hello, world

Breakpoint 1, test () at test.c:4
4       int *p = NULL;
(gdb) print p
$1 = (int *) 0x0
(gdb) set p 0x999
Undefined set print command: "0x999". Try "help set print".
(gdb) set variable p 0x999
A syntax error in expression, near `0x999'.
(gdb) print p
$2 = (int *) 0x0
(gdb) setp
Undefined command: "setp". Try "help".
(gdb) step
5       *p = 2;
(gdb) print p
$3 = (int *) 0x0
(gdb) set variable p = 0x999
(gdb) print p
$4 = (int *) 0x999
(gdb)
```

重新运行
之前已经对test函数设置了断点
前后的值进行比较，发现p的值已经改变
设置p的值

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。