仅通过崩溃地址找出源代码的出错行

作者：老罗

原文出处：http://www.luocong.com/articles/show_article.asp?Article_ID=29

作为程序员，我们平时最担心见到的事情是什么？是内存泄漏？是界面不好看？……错啦！我相信我的看法是不会有人反对的——那就是，程序发生了崩溃！

“该程序执行了非法操作，即将关闭。请与你的软件供应商联系。”，呵呵，这句 M$ 的“名言”，恐怕就是程序员最担心见到的东西了。有的时候，自己的程序在自己的机器上运行得好好的，但是到了别人的机器上就崩溃了；有时自己在编写和测试的过程中就莫名其妙地遇到了非法操作，但是却无法确定到底是源代码中的哪行引起的……是不是很痛苦呢？不要紧，本文可以帮助你走出这种困境，甚至你从此之后可以自豪地要求用户把崩溃地址告诉你，然后你就可以精确地定位到源代码中出错的那行了。（很神奇吧？呵呵。）

首先我必须强调的是，本方法可以在目前市面上任意一款编译器上面使用。但是我只熟悉 M$ 的 VC 和 MASM，因此后面的部分只介绍如何在这两个编译器中实现，请读者自行融会贯通，掌握在别的编译器上使用的方法。

Well，废话说完了，让我们开始！ ：）

首先必须生成程序的 MAP 文件。什么是 MAP 文件？简单地讲，MAP 文件是程序的全局符号、源文件和代码行号信息的唯一的文本表示方法，它可以在任何地方、任何时候使用，不需要有额外的程序进行支持。而且，这是唯一能找出程序崩溃的地方的救星。

好吧，既然 MAP 文件如此神奇，那么我们应该如何生成它呢？在 VC 中，我们可以按下 Alt+F7，打开“Project Settings”选项页，选择 C/C++ 选项卡，并在最下面的 Project Options 里面输入：/Zd，然后要选择 Link 选项卡，在最下面的 Project Options 里面输入：/mapinfo:lines 和 /map:PROJECT_NAME.map。最后按下 F7 来编译生成 EXE 可执行文件和 MAP 文件。

在 MASM 中，我们要设置编译和连接参数，我通常是这样做的：

```
rc %1.rc
ml /c /coff /Zd %1.asm
```

link /subsystem:windows /mapinfo:exports /mapinfo:lines /map:%1.map %1.obj %1.res

把它保存成 makem.bat ，就可以在命令行输入 makem filename 来编译生成 EXE 可执行文件和 MAP 文件了。

在此我先解释一下加入的参数的含义：

/Zd 表示在编译的时候生成行信息
/map[:filename] 表示生成 MAP 文件的路径和文件名
/mapinfo:lines 表示生成 MAP 文件时，加入行信息
/mapinfo:exports 表示生成 MAP 文件时，加入 exported functions （如果生成的是 DLL 文件，这个选项就要加上）

OK，通过上面的步骤，我们已经得到了 MAP 文件，那么我们该如何利用它呢？

让我们从简单的实例入手，请打开你的 VC ，新建这样一个文件：

```
01
//*****************************************************************
02 //程序名称：演示如何通过崩溃地址找出源代码的出错行
03 //作者：罗聪
04 //日期：2003-2-7
05 //出处：http://www.luocong.com（老罗的缤纷天地）
06 //本程序会产生“除0错误”，以至于会弹出“非法操作”对话框。
07 //“除0错误”只会在 Debug 版本下产生，本程序为了演示而尽量简化。
08 //注意事项：如欲转载，请保持本程序的完整，并注明：
09 //转载自“老罗的缤纷天地”（http://www.luocong.com）
10
//*****************************************************************
11
12 void Crash(void)
13 {
14 int i = 1;
```

```
15 int j = 0;
16 i /= j;
17 }
18
19 void main(void)
20 {
21 Crash();
22 }
```

很显然本程序有“除0错误”，在 Debug 方式下编译的话，运行时肯定会产生“非法操作”。好，让我们运行它，果然，“非法操作”对话框出现了，这时我们点击“详细信息”按钮，记录下产生崩溃的地址——在我的机器上是 0x0040104a 。

再看看它的 MAP 文件：（由于文件内容太长，中间没用的部分我进行了省略）

```
CrashDemo

Timestamp is 3e430a76 (Fri Feb 07 09:23:02 2003)

Preferred load address is 00400000

Start Length Name Class
0001:00000000 0000de04H .text CODE
0001:0000de04 0001000cH .textbss CODE
0002:00000000 00001346H .rdata DATA
0002:00001346 00000000H .edata DATA
0003:00000000 00000104H .CRT$XCA DATA
0003:00000104 00000104H .CRT$XCZ DATA
0003:00000208 00000104H .CRT$XIA DATA
0003:0000030c 00000109H .CRT$XIC DATA
0003:00000418 00000104H .CRT$XIZ DATA
0003:0000051c 00000104H .CRT$XPA DATA
0003:00000620 00000104H .CRT$XPX DATA
0003:00000724 00000104H .CRT$XPZ DATA
0003:00000828 00000104H .CRT$XTA DATA
```

```
0003:0000092c 00000104H .CRT$XTZ DATA
0003:00000a30 00000b93H .data DATA
0003:000015c4 00001974H .bss DATA
0004:00000000 00000014H .idata$2 DATA
0004:00000014 00000014H .idata$3 DATA
0004:00000028 00000110H .idata$4 DATA
0004:00000138 00000110H .idata$5 DATA
0004:00000248 000004afH .idata$6 DATA

     Address Publics by Value Rva+Base Lib:Object

     0001:00000020 ?Crash@@YAXXZ 00401020 f CrashDemo.obj
0001:00000070 _main 00401070 f CrashDemo.obj
0004:00000000          __IMPORT_DESCRIPTOR_KERNEL32          00424000
kernel32:KERNEL32.dll
0004:00000014 __NULL_IMPORT_DESCRIPTOR 00424014 kernel32:KERNEL32.dll
0004:00000138 __imp__GetCommandLineA@0 00424138 kernel32:KERNEL32.dll
0004:0000013c __imp__GetVersion@0 0042413c kernel32:KERNEL32.dll
0004:00000140 __imp__ExitProcess@4 00424140 kernel32:KERNEL32.dll
0004:00000144 __imp__DebugBreak@0 00424144 kernel32:KERNEL32.dll
0004:00000148 __imp__GetStdHandle@4 00424148 kernel32:KERNEL32.dll
0004:0000014c __imp__WriteFile@20 0042414c kernel32:KERNEL32.dll
0004:00000150           __imp__InterlockedDecrement@4           00424150
kernel32:KERNEL32.dll
0004:00000154            __imp__OutputDebugStringA@4            00424154
kernel32:KERNEL32.dll
0004:00000158 __imp__GetProcAddress@8 00424158 kernel32:KERNEL32.dll
0004:0000015c __imp__LoadLibraryA@4 0042415c kernel32:KERNEL32.dll
0004:00000160           __imp__InterlockedIncrement@4           00424160
kernel32:KERNEL32.dll
0004:00000164          __imp__GetModuleFileNameA@12          00424164
kernel32:KERNEL32.dll
0004:00000168 __imp__TerminateProcess@8 00424168 kernel32:KERNEL32.dll
0004:0000016c __imp__GetCurrentProcess@0 0042416c kernel32:KERNEL32.dll
0004:00000170         __imp__UnhandledExceptionFilter@4         00424170
```

kernel32:KERNEL32.dll
0004:00000174 __imp__FreeEnvironmentStringsA@4 00424174
kernel32:KERNEL32.dll
0004:00000178 __imp__FreeEnvironmentStringsW@4 00424178
kernel32:KERNEL32.dll
0004:0000017c __imp__WideCharToMultiByte@32 0042417c
kernel32:KERNEL32.dll
0004:00000180 __imp__GetEnvironmentStrings@0 00424180
kernel32:KERNEL32.dll
0004:00000184 __imp__GetEnvironmentStringsW@0 00424184
kernel32:KERNEL32.dll
0004:00000188 __imp__SetHandleCount@4 00424188 kernel32:KERNEL32.dll
0004:0000018c __imp__GetFileType@4 0042418c kernel32:KERNEL32.dll
0004:00000190 __imp__GetStartupInfoA@4 00424190 kernel32:KERNEL32.dll
0004:00000194 __imp__HeapDestroy@4 00424194 kernel32:KERNEL32.dll
0004:00000198 __imp__HeapCreate@12 00424198 kernel32:KERNEL32.dll
0004:0000019c __imp__HeapFree@12 0042419c kernel32:KERNEL32.dll
0004:000001a0 __imp__VirtualFree@12 004241a0 kernel32:KERNEL32.dll
0004:000001a4 __imp__RtlUnwind@16 004241a4 kernel32:KERNEL32.dll
0004:000001a8 __imp__GetLastError@0 004241a8 kernel32:KERNEL32.dll
0004:000001ac __imp__SetConsoleCtrlHandler@8 004241ac
kernel32:KERNEL32.dll
0004:000001b0 __imp__IsBadWritePtr@8 004241b0 kernel32:KERNEL32.dll
0004:000001b4 __imp__IsBadReadPtr@8 004241b4 kernel32:KERNEL32.dll
0004:000001b8 __imp__HeapValidate@12 004241b8 kernel32:KERNEL32.dll
0004:000001bc __imp__GetCPInfo@8 004241bc kernel32:KERNEL32.dll
0004:000001c0 __imp__GetACP@0 004241c0 kernel32:KERNEL32.dll
0004:000001c4 __imp__GetOEMCP@0 004241c4 kernel32:KERNEL32.dll
0004:000001c8 __imp__HeapAlloc@12 004241c8 kernel32:KERNEL32.dll
0004:000001cc __imp__VirtualAlloc@16 004241cc kernel32:KERNEL32.dll
0004:000001d0 __imp__HeapReAlloc@16 004241d0 kernel32:KERNEL32.dll
0004:000001d4 __imp__MultiByteToWideChar@24 004241d4
kernel32:KERNEL32.dll
0004:000001d8 __imp__LCMapStringA@24 004241d8 kernel32:KERNEL32.dll

0004:000001dc __imp__LCMapStringW@24 004241dc kernel32:KERNEL32.dll
0004:000001e0 __imp__GetStringTypeA@20 004241e0 kernel32:KERNEL32.dll
0004:000001e4 __imp__GetStringTypeW@16 004241e4 kernel32:KERNEL32.dll
0004:000001e8 __imp__SetFilePointer@16 004241e8 kernel32:KERNEL32.dll
0004:000001ec __imp__SetStdHandle@8 004241ec kernel32:KERNEL32.dll
0004:000001f0 __imp__FlushFileBuffers@4 004241f0 kernel32:KERNEL32.dll
0004:000001f4 __imp__CloseHandle@4 004241f4 kernel32:KERNEL32.dll
0004:000001f8 \177KERNEL32_NULL_THUNK_DATA 004241f8 kernel32:KERNEL32.dll

entry point at 0001:000000f0

Line numbers for .\Debug\CrashDemo.obj(d:\msdev\myprojects\crashdemo\crashdemo.cpp) segment .text

13 0001:00000020 14 0001:00000038 15 0001:0000003f 16 0001:00000046 17 0001:00000050 20 0001:00000070 21 0001:00000088 22 0001:0000008d

如果仔细浏览 Rva+Base 这栏，你会发现第一个比崩溃地址 0x0040104a 大的函数地址是 0x00401070 ，所以在 0x00401070 这个地址之前的那个入口就是产生崩溃的函数，也就是这行：

0001:00000020 ?Crash@@YAXXZ 00401020 f CrashDemo.obj

因此，发生崩溃的函数就是 ?Crash@@YAXXZ ，所有以问号开头的函数名称都是 C++ 修饰的名称。在我们的源程序中，也就是 Crash() 这个子函数。

OK，现在我们轻而易举地便知道了发生崩溃的函数名称，你是不是很兴奋呢？呵呵，先别忙，接下来，更厉害的招数要出场了。

请注意 MAP 文件的最后部分——代码行信息（Line numbers information），它是以这样的形式显示的：

13 0001:00000020

第一个数字代表在源代码中的代码行号，第二个数是该代码行在所属的代码段中的偏移量。

如果要查找代码行号，需要使用下面的公式做一些十六进制的减法运算：

崩溃行偏移 = 崩溃地址（Crash Address） - 基地址（ImageBase Address） - 0x1000

为什么要这样做呢？细心的朋友可能会留意到 Rva+Base 这栏了，我们得到的崩溃地址都是由 偏移地址（Rva）+ 基地址（Base） 得来的，所以在计算行号的时候要把基地址减去，一般情况下，基地址的值是 0x00400000 。另外，由于一般的 PE 文件的代码段都是从 0x1000 偏移开始的，所以也必须减去 0x1000 。

好了，明白了这点，我们就可以来进行小学减法计算了：

崩溃行偏移 = 0x0040104a - 0x00400000 - 0x1000 = 0x4a

如果浏览 MAP 文件的代码行信息，会看到不超过计算结果，但却最接近的数是 CrashDemo.cpp 文件中的：

16 0001:00000046

也就是在源代码中的第 16 行，让我们来看看源代码：

16 i /= j;

哈！！！果然就是第 16 行啊！

兴奋吗？我也一样！ ：）

方法已经介绍完了，从今以后，我们就可以精确地定位到源代码中的崩溃行，而且只要编译器可以生成 MAP 文件（包括 VC、MASM、VB、BCB、Delphi……），本方法都是适用的。我们时常抱怨 M$ 的产品如何如何差，但其实 M$ 还是有意无意间提供了很多有价值的信息给我们的，只是我们往往不懂得怎么利用而已……相信这样一来，你就可以更为从容地面对“非法操作”提示了。你甚至可以要求用户提供崩溃的地址，然后就可以坐在家中舒舒服服地找到出错的那行，并进行修正。

是不是很爽呢？ ：）

对“仅通过崩溃地址找出源代码的出错行”一文的补充与改进

作者：上海伟功通信 roc

读了老罗的“仅通过崩溃地址找出源代码的出错行”(下称"罗文")一文后，感觉该文还是可以学到不少东西的。不过文中尚存在有些说法不妥，以及有些操作太繁琐的地方 。为此，本人在学习了此文后，在多次实验实践基础上，把该文中的一些内容进行补充与改进，希望对大家调试程序，尤其是 release 版本的程序有帮助 。欢迎各位朋友批评指正。

一、该方法适用的范围

在windows程序中造成程序崩溃的原因很多，而文中所述的方法仅适用与:由一条语句当即引起的程序崩溃。如原文中举的除数为零的崩溃例子。而笔者在实际工作中碰到更多的情况是:指针指向一非法地址 ，然后对指针的内容进行了，读或写的操作。例如：

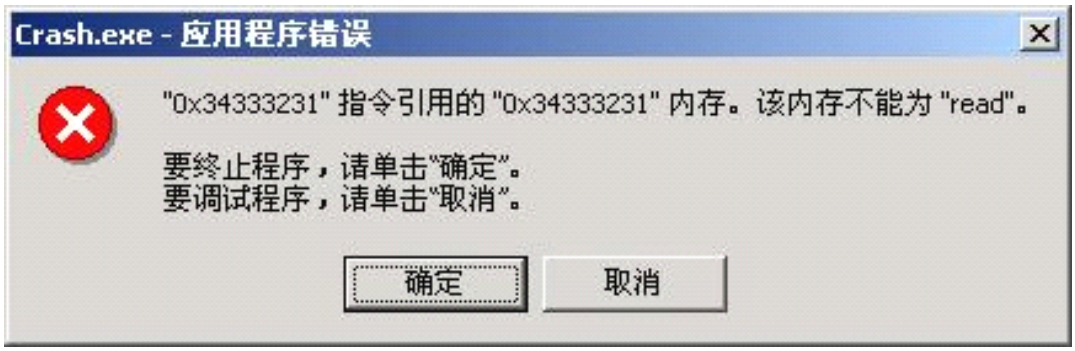
```
void Crash1()
{
 char * p =(char*)100;
 *p=100;
}
```

这些原因造成的崩溃，无论是 debug 版本，还是 release 版本的程序，使用该方法都可找到造成崩溃的函数或子程序中的语句行，具体方法的下面还会补充说明。 另外，实践中另一种常见的造成程序崩溃的原因:函数或子程序中局部变量数组越界付值，造成函数或子程序返回地址遭覆盖，从而造成函数或子程序返回时崩溃。例如:

```
#include
void Crash2();
int main(int argc,char* argv[])
{
        Crash2();
        return 0;
}

void Crash2()
{
        char p[1];
        strcpy(p,"0123456789");
}
```

在 vc 中编译运行此程序的 release 版本，会跳出如下的出错提示框。

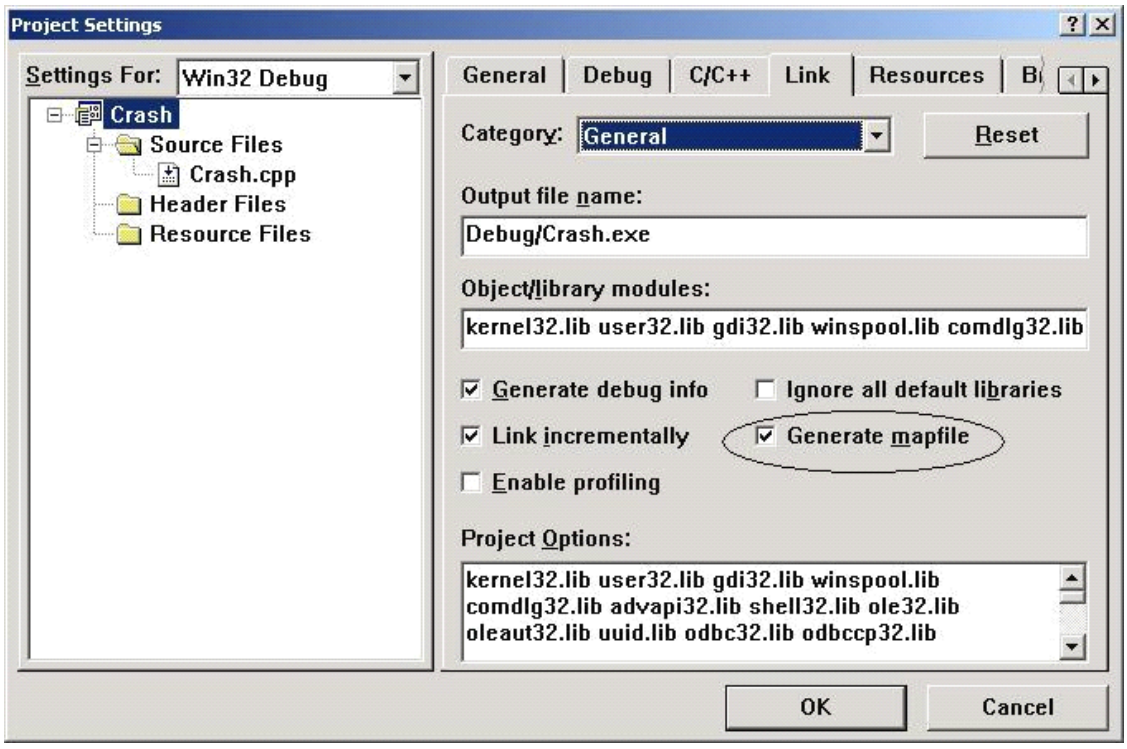


图一 上面例子运行结果

这里显示的崩溃地址为 :0x34333231。这种由前面语句造成的崩溃根源，在后续程序中方才显现出来的情况，显然用该文所述的方法就无能为力了。不过在此例中多少还有些蛛丝马迹可寻找到崩溃的原因:函数 Crash2 中的局部数组 p 只有一个字节大小 ，显然拷贝"0123456789"这个字符串会把超出长度的字符串拷贝到数组 p 的后面，即 *(p+1)="1"， *(p+2)="2"，*(p+3)="3"，*(p+4)=4。。。。。。而字符"1"的 ASC 码的值为 0x31，"2"为 0x32，"3"为 0x33，"4"为 0x34。。。。。，由于 intel 的 cpu 中 int 型数据是低字节保存在低地址中 ，所以保存字符串"1234"的内存，显示为一个 4 字节的 int 型数时就是 0x34333231。显然拷贝"0123456789"这个字符串时，"1234"这几个字符把函数 Crash2 的返回地址给覆盖 ，从而造成程序崩溃。对于类似的这种造成程序崩溃的错误朋友们还有其他方法排错的话，欢迎一起交流讨论。

二、设置编译产生 map 文件的方法

该文中产生 map 文件的方法是手工添加编译参数来产生 map 文件。其实在 vc6 的 IDE 中有产生 map 文件的配置选项的。操作如下:先点击菜单"Project"->"Settings。。。"，弹出的属性页中选中"Link"页 ，确保在"category"中选中"General"，最后选中 "Generate mapfile"的可选项。若要在在 map 文件中显示 Line numbers 的信息的话 ，还需在 project options 中加入 /mapinfo:lines 。Line numbers 信息对于"罗文"所用的方法来定位出错源代码行很重要 ，但笔者后面会介绍更加好的方法来定位出错代码行，那种方法不需要 Line numbers 信息。



图二 设置产生 MAP 文件

三、定位崩溃语句位置的方法

"罗文"所述的定位方法中，找到产生崩溃的函数位置的方法是正确的，即在 map 文件列出的每个函数的起始地址中，最近的且不大于崩溃地址的地址即为包含崩溃语句的函数的地址。但之后的再进一步的定位出错语句行的方法不是最妥当，因为那种方法前提是，假设基地址的值是 0x00400000，以及一般的 PE 文件的代码段都是从 0x1000 偏移开始的。虽然这种情况很普遍，但在 vc 中还是可以基地址设置为其他数，比如设置为 0x00500000，这时仍旧套用

崩溃行偏移 = 崩溃地址 - 0x00400000 - 0x1000

的公式显然无法找到崩溃行偏移。 其实上述公式若改为

崩溃行偏移 = 崩溃地址 - 崩溃函数绝对地址 + 函数相对偏移

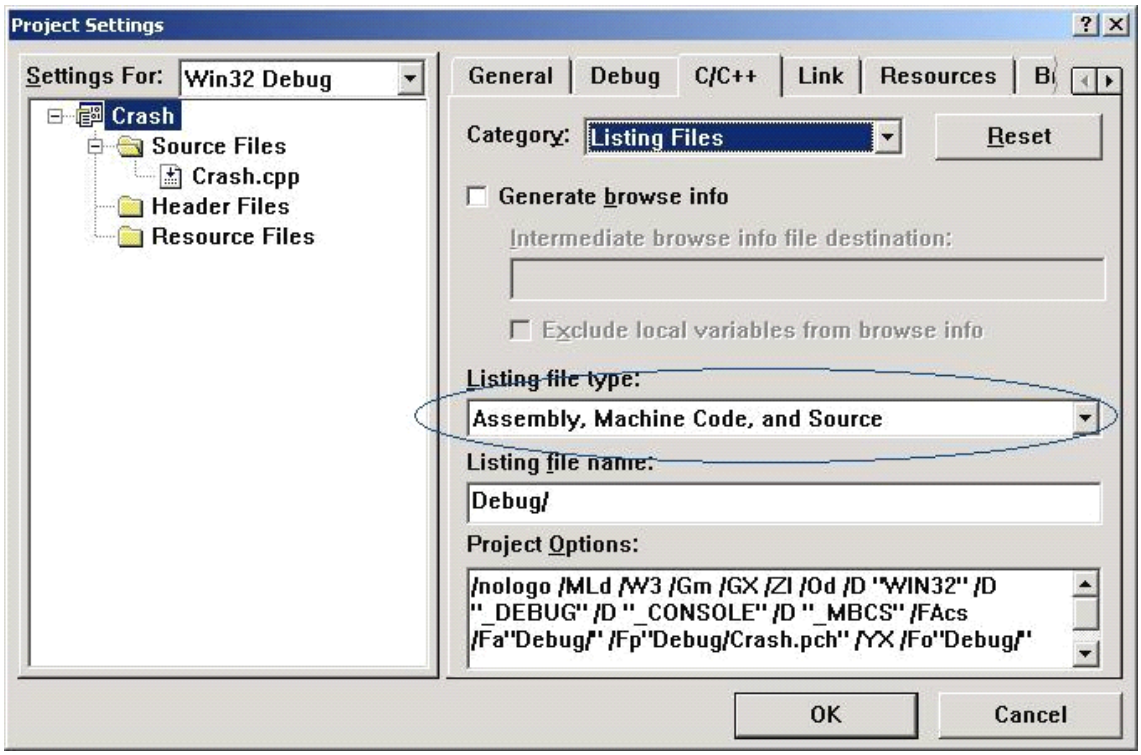
即可通用了。仍以"罗文"中的例子为例:"罗文"中提到的在其崩溃程序的对应 map 文件中，崩溃函数的编译结果为

0001:00000020 ?Crash@@YAXXZ 00401020 f CrashDemo。obj

对与上述结果，在使用我的公式时 ，"崩溃函数绝对地址"指 00401020， 函数相对偏移指 00000020， 当崩溃地址= 0x0040104a 时， 则 崩溃行偏移 = 崩溃地址 - 崩溃函数起始地址+函数相对偏移 = 0x0040104a - 0x00401020 + 0x00000020= 0x4a，结果与"罗文"计算结果相同。但这个公式更通用。

四、更好的定位崩溃语句位置的方法。

其实除了依靠 map 文件中的 Line numbers 信息最终定位出错语句行外，在 vc6 中我们还可以通过编译程序产生的对应的汇编语句，二进制码，以及对应 c/c++语句为一体的"cod"文件来定位出错语句行 。先介绍一下产生这种包含了三种信息的"cod"文件的设置方法:先点击菜单"Project"->"Settings。。。"，弹出的属性页中选中"C/C++"页 ，然后在"Category"中选则"Listing Files"，再在"Listing file type"的组合框中选择"Assembly，Machine code， and source"。接下去再通过一个具体的例子来说明这种方法的具体操作。



图三 设置产生"cod"文件

准备步骤 1)产生崩溃的程序如下:

```
01 //**********************************************************************
02 //文件名称：crash。cpp
03 //作用:     演示通过崩溃地址找出源代码的出错行新方法
04 //作者：    伟功通信 roc
05 //日期：    2005-5-16
06//**********************************************************************
```

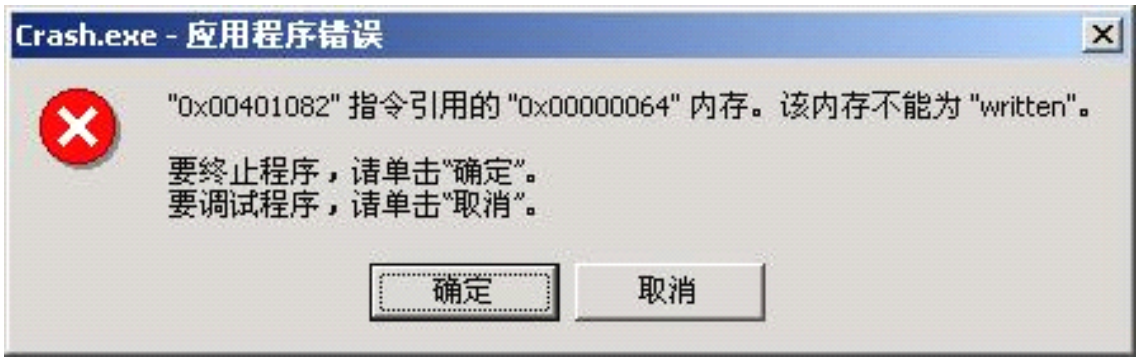
```
07 void Crash1();
08 int main(int argc,char* argv[])
09 {
10         Crash1();
11         return 0;
12 }
13
14 void Crash1()
15 {
16    char * p =(char*)100;
17    *p=100;
18 }
```

准备步骤 2)按本文所述设置产生 map 文件(不需要产生 Line numbers 信息)。

准备步骤 3)按本文所述设置产生 cod 文件。

准备步骤 4)编译。这里以 debug 版本为例(若是 release 版本需要将编译选项改为不进行任何优化的选项，否则上述代码会因为优化时看作废代码而不被编译，从而看不到崩溃的结果)，编译后产生一个"exe"文件 ，一个"map"文件，一个"cod"文件。

运行此程序，产生如下如下崩溃提示:



图四 上面例子运行结果

排错步骤 1)定位崩溃函数。可以查询 map 文件获得。我的机器编译产生的 map 文件的部分如下:

```
 Crash

 Timestamp is 42881a01 (Mon May 16 11:56:49 2005)
```

Preferred load address is 00400000

Start Length Name Class
0001:00000000 0000ddf1H .text CODE
0001:0000ddf1 0001000fH .textbss CODE
0002:00000000 00001346H .rdata DATA
0002:00001346 00000000H .edata DATA
0003:00000000 00000104H .CRT$XCA DATA
0003:00000104 00000104H .CRT$XCZ DATA
0003:00000208 00000104H .CRT$XIA DATA
0003:0000030c 00000109H .CRT$XIC DATA
0003:00000418 00000104H .CRT$XIZ DATA
0003:0000051c 00000104H .CRT$XPA DATA
0003:00000620 00000104H .CRT$XPX DATA
0003:00000724 00000104H .CRT$XPZ DATA
0003:00000828 00000104H .CRT$XTA DATA
0003:0000092c 00000104H .CRT$XTZ DATA
0003:00000a30 00000b93H .data DATA
0003:000015c4 00001974H .bss DATA
0004:00000000 00000014H .idata$2 DATA
0004:00000014 00000014H .idata$3 DATA
0004:00000028 00000110H .idata$4 DATA
0004:00000138 00000110H .idata$5 DATA
0004:00000248 000004afH .idata$6 DATA

Address Publics by Value Rva+Base Lib:Object

0001:00000020 _main 00401020 f Crash.obj
0001:00000060 ?Crash1@@YAXXZ 00401060 f Crash.obj
0001:000000a0 __chkesp 004010a0 f LIBCD:chkesp.obj
0001:000000e0 _mainCRTStartup 004010e0 f LIBCD:crt0.obj
0001:00000210 __amsg_exit 00401210 f LIBCD:crt0.obj
0001:00000270 __CrtDbgBreak 00401270 f LIBCD:dbgrpt.obj
...

对于崩溃地址 0x00401082 而言，小于此地址中最接近的地址(Rva+Base 中的地址)为 00401060，其对应的函数名为?Crash1@@YAXXZ，由于所有以问号开头的函数名称都是 C++修饰的名称 ，"@@YAXXZ"则为区别重载函数而加的后缀，所以?Crash1@@YAXXZ 就是我们的源程序中，Crash1() 这个函数。

排错步骤 2)定位出错行。打开编译生成的"cod"文件，我机器上生成的文件内容如下:

```
        TITLE   E:\Crash\Crash。cpp
        .386P
include listing.inc
if @Version gt 510
.model FLAT
else
_TEXT   SEGMENT PARA USE32 PUBLIC "CODE"
_TEXT   ENDS
_DATA   SEGMENT DWORD USE32 PUBLIC "DATA"
_DATA   ENDS
CONST   SEGMENT DWORD USE32 PUBLIC "CONST"
CONST   ENDS
_BSS    SEGMENT DWORD USE32 PUBLIC "BSS"
_BSS    ENDS
$$SYMBOLS       SEGMENT BYTE USE32 "DEBSYM"
$$SYMBOLS       ENDS
$$TYPES         SEGMENT BYTE USE32 "DEBTYP"
$$TYPES         ENDS
_TLS    SEGMENT DWORD USE32 PUBLIC "TLS"
_TLS    ENDS
;       COMDAT _main
_TEXT   SEGMENT PARA USE32 PUBLIC "CODE"
_TEXT   ENDS
;       COMDAT ?Crash1@@YAXXZ
_TEXT   SEGMENT PARA USE32 PUBLIC "CODE"
_TEXT   ENDS
FLAT    GROUP _DATA,   CONST, _BSS
        ASSUME  CS: FLAT,DS: FLAT, SS: FLAT
endif
PUBLIC          ?Crash1@@YAXXZ                                  ; Crash1
```

```
PUBLIC          _main
EXTRN __chkesp:NEAR
;       COMDAT _main
_TEXT   SEGMENT
_main   PROC NEAR                                       ; COMDAT

; 9      : {

   00000        55              push    ebp
   00001        8b ec           mov     ebp，  esp
   00003        83 ec 40  sub   esp, 64                 ; 00000040H
   00006        53              push    ebx
   00007        56              push    esi
   00008        57              push    edi
   00009        8d 7d c0  lea   edi, DWORD PTR [ebp-64]
   0000c        b9 10 00 00 00  mov     ecx，  16                       ; 00000010H
   00011        b8 cc cc cc cc  mov     eax，  -858993460               ; ccccccccH
   00016        f3 ab           rep stosd

; 10     :       Crash1();

   00018        e8 00 00 00 00  call    ?Crash1@@YAXXZ          ; Crash1

; 11     :       return 0;

   0001d        33 c0           xor     eax，  eax

; 12     : }

   0001f5f              pop     edi
   00020        5e              pop     esi
   00021        5b              pop     ebx
   00022        83 c4 40        add     esp, 64                 ; 00000040H
   00025        3b ec           cmp     ebp, esp
   00027        e8 00 00 00 00  call    __chkesp
```

```
   0002c    8b e5          mov    esp, ebp
   0002e    5d             pop    ebp
   0002fc3                 ret    0
_main    ENDP
_TEXT    ENDS
;        COMDAT ?Crash1@@YAXXZ
_TEXT    SEGMENT
_p$ = -4
?Crash1@@YAXXZ PROC NEAR                                  ; Crash1, COMDAT

; 15     : {

   00000    55             push   ebp
   00001    8b ec          mov    ebp, esp
   00003    83 ec 44  sub  esp, 68                  ; 00000044H
   00006    53             push   ebx
   00007    56             push   esi
   00008    57             push   edi
   00009    8d 7d bc  lea  edi, DWORD PTR [ebp-68]
   0000c    b9 11 00 00 00 mov    ecx, 17                  ; 00000011H
   00011    b8 cc cc cc cc mov    eax, -858993460                  ; ccccccccH
   00016    f3 ab          rep stosd

; 16     :    char * p =(char*)100;

   00018    c7 45 fc 64 00
            00 00          mov    DWORD PTR _p$[ebp], 100     ; 00000064H

; 17     :    *p=100;

   0001f8b 45 fc  mov     eax, DWORD PTR _p$[ebp]
   00022    c6 00 64       mov    BYTE PTR [eax], 100   ; 00000064H

; 18     : }

   00025    5f             pop    edi
```

```
   00026        5e                pop     esi
   00027        5b                pop     ebx
   00028        8b e5             mov     esp, ebp
   0002a        5d                pop     ebp
   0002b        c3                ret     0
?Crash1@@YAXXZ ENDP                                    ; Crash1
_TEXT  ENDS
END
```

其中

```
?Crash1@@YAXXZ PROC NEAR                               ; Crash1, COMDAT
```

为 Crash1 汇编代码的起始行。产生崩溃的代码便在其后的某个位置。接下去的一行为:

```
; 15     : {
```

冒号后的"{"表示源文件中的语句，冒号前的"15"表示该语句在源文件中的行数。 这之后显示该语句汇编后的偏移地址，二进制码，汇编代码。如

```
00000    55                push     ebp
```

其中"0000"表示相对于函数开始地址后的偏移， "55"为编译后的机器代码， " push ebp"为汇编代码。从"cod"文件中我们可以看出，一条(c/c++)语句通常需要编译成数条汇编语句 。此外有些汇编语句太长则会分两行显示如:

```
00018    c7 45 fc 64 00
         00 00             mov      DWORD PTR _p$[ebp], 100     ; 00000064H
```

其中"0018"表示相对偏移，在 debug 版本中，这个数据为相对于函数起始地址的偏移(此时每个函数第一条语句相对偏移为 0000)；release 版本中为相对于代码段第一条语句的偏移(即代码段第一条语句相对偏移为 0000，而以后的每个函数第一条语句相对偏移就不为 0000 了)。"c7 45 fc 64 00 00 00 "为编译后的机器代码 ，"mov DWORD PTR _p$[ebp]， 100"为汇编代码， 汇编语言中";"后的内容为注释，所以";00000064H"，是个注释这里用来说明 100 转换成 16 进制时为"00000064H"。

接下去，我们开始来定位产生崩溃的语句。

第一步，计算崩溃地址相对于崩溃函数的偏移，在本例中已经知道了崩溃语句的地址(0x00401082)，和对应函数的起始地址(0x00401060)，所以崩溃地址相对函数起始地址的偏移就很容易计算了:

崩溃偏移地址 = 崩溃语句地址 - 崩溃函数的起始地址 = 0x00401082 - 0x00401060 = 0x22。

第二步，计算出错的汇编语句在cod文件中的相对偏移。我们可以看到函数Crash1()在cod文件中的相对偏移地址为0000，则

崩溃语句在 cod 文件中的相对偏移 = 崩溃函数在 cod 文件中相对偏移 + 崩溃偏移地址 = 0x0000 + 0x22 = 0x22

第三步，我们看 Crash1 函数偏移 0x22 除的代码是什么?结果如下

```
00022  c6 00 64              mov      BYTE PTR [eax], 100    ; 00000064H
```

这句汇编语句表示将 100 这个数保存到寄存器 eax 所指的内存单元中去，保存空间大小为 1 个字节(byte)。程序正是执行这条命令时产生了崩溃，显然这里 eax 中的为一个非法地址 ，所以程序崩溃了!

第四步，再查看该汇编语句在其前面几行的其对应的源代码，结果如下:

```
; 17     :    *p=100;
```

其中 17 表示该语句位于源文件中第 17 行，而“*p=100;”这正是源文件中产生崩溃的语句。

至此我们仅从崩溃地址就查找出了造成崩溃的源代码语句和该语句所在源文件中的确切位置，甚至查找到了造成崩溃的编译后的确切汇编代码!

怎么样，是不是感觉更爽啊?

五、小节

1、新方法同样要注意可以适用的范围，即程序由一条语句当即引起的崩溃。另外我不知道除了 VC6 外，是否还有其他的编译器能够产生类似的"cod"文件。

2、我们可以通过比较 新方法产生的 debug 和 releae 版本的"cod"文件，查找那些仅 release 版本(或 debug 版本)有另一个版本没有的 bug(或其他性状)。例如"罗文"中所举的那个用例 ，只要打开 release 版本的"cod"文件，就明白了为啥 debug 版本会产生崩溃而 release 版本却没有:原来 release 版本中产生崩溃的语句其实根本都没有编译 。同样本例中的 release 版本要看到崩溃的效果，需要将编译选项改为为不优化的配置。