Debugging Tutorial

1. Introduction

This tutorial intends to help students get used to using important tools and methods for testing and debugging complex applications efficiently.

2. printf debugging

Using printf function is a simplest form of software debugging method, which is often useful for debugging small and simple programs. Due to its simplicity, this method is surprisingly widely used even in the professional software development. Let us see the following example:

```
#include <stdio.h>

int main(void)
{
    int a, b, c;
    a = -1;
    b = 1;
    c = -1;
    a = b + c;
    b = a + c;
    c = a + b + 1;
    a = b / c;
    b = a * c;
    printf("a b c = %d %d %d\n", a, b, c);

    return 0;
}
```

test.c

The above program crashes by an unknown reason. How can we systematically find the location and reason of this system error? For the first effort, you can simply insert printf functions between every line to see which line causes the application crash. In the following, we put eight printf calls, each of which displays an asterisk symbol. We hope that by counting the number of printed asterisks, we can locate the error. Do you think it is working or not?

Put your answer here:

```c
#include <stdio.h>

int main(void)
{
    int a, b, c;
    printf("*");
    a = -1;
    printf("*");
    b = 1;
    printf("*");
    c = -1;
    printf("*");
    a = b + c;
    printf("*");
    b = a + c;
    printf("*");
    c = a + b + 1;
    printf("*");
    a = b / c;
    printf("*");
    b = a * c;
    printf("a b c = %d %d %d\n", a, b, c);

    return 0;
}
```

If the above approach is not working, why is it and what is your solution for that?

Put your answer here:

※. HINT: Note that calling a printf function do not immediately make the content appear in the screen. Usually, printf calls are buffered, thus the content to be displayed remains in the memory for a time and flushed to the screen at a later time by an event such as meeting a new line character or calling a fflush function.

3. Logging macros

```
#include <stdio.h>

#ifdef DEBUG
#define DBG(fmt, ...) printf("(%s:%d) " fmt "\n", __FILE__, __LINE__,
__VA_ARGS__)
#else
#define DBG(fmt, ...)
#endif

int main(void)
{
    int a, b, c;

    a = -1;
    DBG("a = %d", a);
    b = 1;
    DBG("b = %d", b);
    c = -1;
    DBG("c = %d", c);
    a = b + c;
    DBG("a = %d", a);
    b = a + c;
    DBG("b = %d", b);
    c = a + b + 1;
    DBG("c = %d", c);
    a = b / c;
    DBG("a = %d", a);
    b = a * c;
    printf("a b c = %d %d %d\n", a, b, c);

    return 0;
}
```

In the above example, we use a newly defined macro DBG instead of calling printf functions. It has the following nice features compared to the printf method:

- You can simply turn on and off the debugging messages by adding or removing –DDEBUG compiler option.
- It automatically shows a file name and line number for supporting more efficient debugging.

In the following, we show how to turn on and off our debugging macros.

```
$ gcc test.c -DDEBUG
$ ./a.out
(a.c:16) a = -1
(a.c:18) b = 1
(a.c:20) c = -1
(a.c:22) a = 0
(a.c:24) b = -1
(a.c:26) c = 0
Floating point exception (core dumped)

$ gcc test.c
$ ./a.out
Floating point exception (core dumped)
```

※. Note that turning on and off debugging macros in the above way has some bad effect when the source code is maintained for a long time. Sometimes, during the maintenance, a certain variable can be eliminated. However, the maintainer may not remove the logging code for that variable since the program still compiles well. Then after years, a new maintainer happens to turn on the debugging causing enormous amount of syntax errors. However, the above method is quite well suited for small or middle scale programs.

4. gdb

Gdb is a symbolic debugger. Using it, you can run the program line-by-line and investigate each variable's value or you can even change it while the program is running. In order to use gdb, your program should be compiled with a –g option as in the following:

```
$ gcc test.c -g
```

Then you can start a debugging session as in the following:

```
$ gdb ./a.out
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...done.
(gdb)
```

You can list the program inside the gdb session.

```
(gdb) list
1       #include <stdio.h>
2
3       int main(void)
4       {
5           int a, b, c;
6
7           a = -1;
8           b = 1;
9           c = -1;
10          a = b + c;
(gdb)
```

First, let us set a breakpoint in line 7.

```
(gdb) b 7
Breakpoint 1 at 0x400535: file b.c, line 7.
(gdb)
```

Then start the program. The execution stops at line 7 because we set a breakpoint there.

```
(gdb) run
Starting program: /home/auto/debug/a.out

Breakpoint 1, main () at b.c:7
7           a = -1;
(gdb)
```

You can run the program line-by-line using n command and see the value of each variable using print command. By investigating the value of c, we can see that it will cause a divide-by-zero error. By running

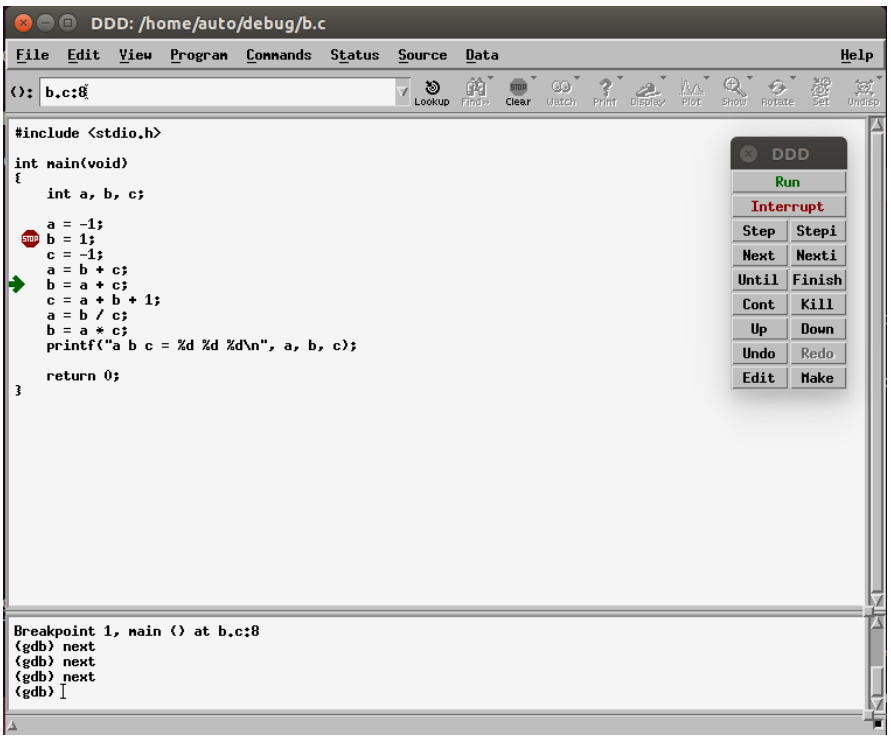
the line 13, gdb reports that an arithmetic exception occurs in that line as expected.

```
(gdb) n
8           b = 1;
(gdb) n
9           c = -1;
(gdb) n
10          a = b + c;
(gdb) n
11          b = a + c;
(gdb) n
12          c = a + b + 1;
(gdb) n
13          a = b / c;
(gdb) print c
$1 = 0
(gdb) n

Program received signal SIGFPE, Arithmetic exception.
0x0000000000400572 in main () at b.c:13
13          a = b / c;
```

You can run ddd instead of gdb, which is a graphical front-end for gdb.

```
$ ddd ./a.out
```



5. Core dump anslysis

The following program has a fault that causes a memory exception so called segmentation fault. Let's see how we can simply locate the error by core dump analysis.

```
#include <stdio.h>
#include <string.h>
void outer_function(void);
void inner_function(void);
void bad_function(void);
int main(void)
{
    outer_function();
    return 0;
}
void outer_function(void)
{
    inner_function();
}
void inner_function(void)
{
    bad_function();
}
void bad_function(void)
{
    char a[100];
    char *p = 0;

    memcpy(p, a, sizeof(a));
}
```

core.c

Core is a file generated when a program abnormally dies. In other words, it is a dead body. We can investigate this dead body, i.e., core, to find the reason of death. First, we have to enable core generation as in the following. Then running the program generates a core file.

```
$ ulimit -c unlimited
$ ./a.out
Floating point exception (core dumped)
$ ls -l core
-rw------- 1 auto auto 253952 Oct 15 23:58 core
```

To analyze the core file, we can simply run gdb with the executable file and core file as its arguments. Then you can locate the error by calling bt command, which shows the back trace of the program at which it crashed.

```
$ gdb ./a.out core
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
...
...
Reading symbols from ./a.out...(no debugging symbols found)...done.
[New LWP 64418]
Core was generated by `./a.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00000000004005aa in bad_function ()
(gdb) bt
#0  0x00000000004005aa in bad_function ()
#1  0x0000000000400581 in inner_function ()
#2  0x0000000000400576 in outer_function ()
#3  0x0000000000400566 in main ()
```

6. Memory debugging

In the following, there is a memory leak caused by allocating a memory and not freeing the allocated memory while the program is running. This kind of memory leak is really difficult to find. Thus we need a special memory debugging tool to locate the memory leak.

```
#include <stdio.h>
#include <stdlib.h>

void bad_function(void);

int main(void)
{
    bad_function();
    return 0;
}

void bad_function(void)
{
    char *p1, *p2, *p3;

    p1 = (char *)malloc(1024 * 204);
    p2 = (char *)malloc(1024 * 204);
    p3 = (char *)malloc(1024 * 204);

    free(p1);
    free(p2);
}
```

memory.c

Below is a simple usage of valgrind, a memory debugger. By calling it with proper options, it tells about the location of memory leak in detail.

```
$ gcc -g memory.c
$ valgrind --tool=memcheck --leak-check=yes ./a.out
==64564== Memcheck, a memory error detector
==64564== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==64564== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright
info
==64564== Command: ./a.out
==64564==
==64564==
==64564== HEAP SUMMARY:
==64564==     in use at exit: 208,896 bytes in 1 blocks
==64564==   total heap usage: 3 allocs, 2 frees, 626,688 bytes allocated
==64564==
==64564== 208,896 bytes in 1 blocks are definitely lost in loss record 1 of 1
==64564==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==64564==    by 0x4005BA: bad_function (memory.c:18)
==64564==    by 0x400585: main (memory.c:8)
==64564==
==64564== LEAK SUMMARY:
==64564==    definitely lost: 208,896 bytes in 1 blocks
==64564==    indirectly lost: 0 bytes in 0 blocks
==64564==      possibly lost: 0 bytes in 0 blocks
==64564==    still reachable: 0 bytes in 0 blocks
==64564==         suppressed: 0 bytes in 0 blocks
==64564==
==64564== For counts of detected and suppressed errors, rerun with: -v
==64564== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```