$Id: lab5c-gdb-valgrind.mm,v 1.67 2012-11-05 15:12:46-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Labs-cmps012m/lab5c-gdb-valgrind

This lab will introduce you to **gdb**, the Gnu debugger associated with **gcc**, and **valgrind**, which can be used to track uninitialized variables, memory leak, and dangling pointers. Uninitialized variables are variables which are declared but whose value is used before being assigned to. Memory leak occurs when memory is not freed when no longer needed; C does not have a garbage collector. A dangling pointer points at storage that has been freed and should no longer be accessed.

Before beginning this lab, study some of the tutorials in **Tutorials/gdb-tutorials**. There are links in this directory to **gdb-tutorial-handout.pdf** and **gdb-tutorial-ohio.html**, which are fairly short, but also a link to **gdb-tutorial-rms**, which is much longer and more detailed.

1. Detailed steps

In this lab, you will follow some detailed steps. For each step, submit the files listed. After you have submitted the necessary files, verify that they are all in the submit directory by using **ls**. As in a previous lab, the command

grep Submit: *.tt

will summarize the files you need to submit. Most of the commands will be interactive, so make use of the **script** command to capture command line output. A terminal session can be captured with

script *filename*

where *filename* is the file into which you want your session captured. Be sure not to use anything other than line mode commands in this file, and examine it after to verify this. Specifically, never use an editor inside a terminal running **script**.

For this lab, make sure **/afs/cats.ucsc.edu/courses/cmps012b-wm/bin** is in your **$PATH**, and that the environment variable **$SHELL** is **/bin/bash**. If you use **tcsh**, this can be accomplished for the current session with

setenv SHELL /bin/bash

Following are the items for this lab. Capture the output in the file specified at the end of each part.

(01) A couple of uninitialized variables. For convenience, a script **mk** (Figure 1) has been provided to avoid the need for a **Makefile** in this lab only. It contains compilation instructions. Start with the program **uninit.c**. (Figure 2). Use the following commands.

Submit: **part01.script**

mk uninit.c	Run the script to lint and compile.
valgrind uninit	Check for uninitialized variables.
echo $?	What is the exit status ? **bash** will capture the crash.
pstatus 139	Print the meaning of the crash.
exit	Get out of **script**.

(02) Now look into your program with **gdb**, capturing your session into **part02.script**

Submit: **part02.script**

gdb uninit	Start **gdb**.
run	Run the program.
where	Ask where in the program it crashed.
list	Look at a few lines around where it crashed.
print foo	Print the values of some variables.
print pointer	
print *pointer	
print argv[0]	
quit	Quit **gdb**. Answer yes to the subprocess question.

(03) Now step through the program a line at a time.

Submit: `part03.script`

`gdb uninit`	
`break main`	Set a breakpoint at the beginning of the main function.
`run`	Note that it stops at the breakpoint.
`print foo`	Note that the value is some number, but there is no way to figure out why it has this value.
`step`	Step one instruction.
`print pointer`	
`step`	Step one more instruction. Note that it crashes at this point.
`quit`	

(04) Now let use look at `malloc`, similar to Java's `new;` and `delete`, which releases storage. C does not have a garbage collector. Start with the program `malloc.c` (Figure 3).

Submit: `part04.script`

`valgrind malloc`	Note that there are a couple of `malloc`'s but only one `free` later. So one block was leaked.
`gdb malloc`	
`break main`	Set a breakpoint in the main function.
`run`	
`print ptr`	
`print *ptr`	Bad memory access because `ptr` is not initialized.
`step`	
`print ptr`	Now `ptr` has a value obtained from the heap.
`print *ptr`	But the value it points it is uninitialized. If it is 0, that is just a coincidence.
`step`	
`step`	
`step`	
`print ptr`	
`print *ptr`	Now it points at initialized storage.
`step`	The storage is freed by the call to `free`.
`step`	
`step`	
`step`	The reference to `__libc_start_main` is the startup function called by the operating system to set up the environment and call main.
`quit`	

(05) Examine `list1.c` (Figure 4). Compile it with `mk list1.c` and look at the errors and warnings you see. Capture the output from this compilation and submit it. Read the man page `malloc`(3) to see what header file was not included.

Submit: `part05.script`

(06) Copy `list1.c` to `list2.c` and fix the missing header file. Capture the output.

Submit: `part06.script`

Submit: `list2.c`

`mk list2.c`	Note the warning from `lint`.
`valgrind list2 foo bar`	Note the complaints from `valgrind`. It complains about memory leak, but also about invalid access to memory.
`gdb list2`	
`run foo bar`	Note how arguments are given to a program, on the `run` not on the invocation of `gdb`.
`where`	
`list`	Does not list the lines around the point of the crash.
`list list2.c:23`	We can select the particular set of lines to list.
`print head`	Not in the current stack frame. Note that we have called several library functions, as shown by `where`.
`bt`	A backtrace is another way of looking at the stack.
`up`	
`up`	
`up`	We unwind the stack three levels here.
`print head`	Now we are in the correct frame.
`print *head`	
`print *(head->word)`	We can use more complicated C expressions.
`print head->link->link->link`	
`print *(head->link->link->link)`	

(07) Run `list2` again, showing values in `argv`.

Submit: `part07.script`

`gdb list2`	
`break main`	
`run foo bar`	Run the program with two command line arguments, namely `foo` and `bar`.
`print argc`	
`print argv`	
`print argv[0]`	`argv[0]` is always the name of the program being run.
`print argv[1]`	
`print argv[2]`	
`print argv[3]`	`argv[argc]` is always the null pointer, represented as `0x0` in C.
`print argv[4]`	After `argv` is the default environment which you can display using the `env` or `printenv` command.
`print argv[5]`	
`print argv[6]`	

(08) Copy `list2.c` to `list3.c` and use `valgrind` and `dbx` as appropriate so that you can track down and fix all of the memory faults. Ignore memory leak for now.

Submit: `part08.script`

Submit: `list3.c`

`valgrind --leak-check=full \` `list3 foo bar baz qux`	Run `valgrind` with the option `--leak-check=full` to verify that your program in fact has no problems except leaks.

(09) Copy `list3.c` to `list4.c`. Eliminate memory leak by using `free` to release all allocated storage.

Submit: `part09.script`

Submit: **`list4.c`**

`valgrind --leak-check=full \` **`list4 foo bar baz qux`**	Verify that your program now works with no memory faults and no memory leak.
`echo $?`	Make sure the exit status is **`EXIT_SUCCESS`**.

(10) Write a program called **`environ.c`** which will declare the external variable
`extern char **environ;`
and write a loop iterating over that array, printing each element per line. The stopping condition is meeting a null pointer, as there is no variable indicating how large the array is.

Submit: **`part10.script`**

Submit: **`environ.c`**

`./environ`	Print out all your environment variables.

2. What to submit

Submit the 14 files mentioned above. Verify the submit by looking in **`/afs/cats.ucsc.edu/class/cmps012b-wm.w11/lab5`**. If you are doing pair programming, submit the required files as well.

3. Debugging with `ddd`

An alternative to **`gdb`** is **`ddd`**, which is a GUI wrapper around **`gdb`**. It is not part of this lab and there is nothing to submit from using **`ddd`**, but you might want to explore it. For example:

(1) Start with: **`ddd uninit &`**. The ampersand (**`&`**) at the end of the line causes the program to be run in the background. If you are using a terminal without X11 forwarding this will not work.

(2) In the **`gdb`** window, type: **`break main`**. Note that a stop sign appears in the code.

(3) Then type: **`run`**. An arrow shows the breakpoint.

(4) Click on ***Step*** several times.

(5) You may also use print statements in the **`gdb`** window to examine variables.

```
#!/bin/sh -x
# $Id: mk,v 1.4 2012-10-25 18:13:13-07 - - $
#
# This script takes the names of C source files and compiles them
# into executable images.  Each source must be a complete program.
#
for CSOURCE in $*
do
   cid + $CSOURCE
   checksource $CSOURCE
   EXECBIN=`echo $CSOURCE | sed 's/\.c$//'`
   gcc -g -O0 -Wall -Wextra -std=gnu99 $CSOURCE -o $EXECBIN -lm
done
```

Figure 1. **`code/mk`**

```c
// $Id: uninit.c,v 1.1 2011-02-01 17:55:43-08 - - $

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

int main (int argc, char **argv) {
   int foo;
   printf ("%d\n", foo);
   int *pointer;
   printf ("%d\n", *pointer);
}
```

Figure 2. `code/uninit.c`

```c
// $Id: malloc.c,v 1.1 2011-02-01 18:35:38-08 - - $

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

typedef struct node *node_ref;
struct node {
   int value;
   node_ref link;
};

int main (int argc, char **argv) {
   node_ref ptr = malloc (sizeof (struct node));
   ptr = malloc (sizeof (struct node));
   ptr->value = 6;
   ptr->link = NULL;
   printf ("%p-> {%d, %p}\n", ptr, ptr->value, ptr->link);
   free (ptr);
   return EXIT_SUCCESS;
}
```

Figure 3. `code/malloc.c`

```
// $Id: list1.c,v 1.1 2011-02-01 18:51:19-08 - - $

#include <assert.h>
#include <stdio.h>

typedef struct node *node_ref;
struct node {
   char *word;
   node_ref link;
};

int main (int argc, char **argv) {
   node_ref head;
   for (int argi = 0; argi < 5; ++argi) {
      node_ref node = malloc (sizeof (node_ref));
      assert (node != NULL);
      node->word = argv[argi];
      node->link = head;
      head = node;
   }
   for (node_ref curr = head; curr->link != NULL; curr = curr->link) {
      printf ("%p->node {word=%p->[%s], link=%p}\n",
              curr, curr->word, curr->word, curr->link);
   }
   return 9;
}
```

Figure 4. `code/list1.c`