Module 4 – Strings and Debugging Techniques in C

Tutorial Questions

Objectives
To develop your understanding of strings in C, and to review some debugging techniques.

Activities

1. Explain the meaning of each of the following declarations:
   a. char *a[12];
   b. char *d[4] = {"north", "south", "east", "west");

2. Declare an array of strings whose initial values are "red", "green" and "blue".

3. Assume the following declarations:

```
char *token, *endPtr, *result;
char line[12];
result = fgets(line,sizeof(line),stdin);
```

   a. What is the value of *result* if the user entered Control-D?
   b. What is/are the contents of *line* if the user just pressed 'enter'?
   c. What each of the following inputs, what would be the last character and the second last character in *line* if the user entered the input and pressed 'enter'?
      - abcdefghijkl
      - abcdefghij

4. Bugs, Pitfalls, and C : Problem #1

   Make a list of the things that could go wrong with this C program segment. How would you correct them ?

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

int main(void)
{
   const char * day[] = { "Sunday", "Monday", "Tuesday",
         "Wednesday", "Thursday", "Friday", "Saturday" };
   int dayNum;

   printf("Please enter a day number (0-6): ");
   scanf("%d", &dayNum);
   printf("That day is %s\n", day[dayNum]);
   return EXIT_SUCCESS;
}
```

5. Bugs, Pitfalls, and C : Problem #2

   Make a list of the things that could go wrong with this C program segment. How would you correct them?

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

```
#include <stdlib.h>

int main(void)
{
   char name[10];
   char colour[10];

   printf("Please enter your name: ");
   scanf("%s", name);
   printf"Please enter your favourite colour: ");
   scanf("%s", colour);
   printf("Thank you, %s. ", name);

   if (!strcmp(colour, "Red"))
         printf("Red is a good colour.\n");
   else printf("%s is an okay colour.\n");

   return EXIT_SUCCESS;
}
```

6. Make a list of techniques for locating the source of bugs in C programs.

7. Using `splint` to identify errors

`splint` is a program verifier or checker. It is generally fussier than gcc which is a good thing, not a bad thing. There are many occasions where gcc or `splint` warning messages can lead you to the source of non-syntactic errors in your programs.

Use `splint` as follows: `splint file.c`.

Where `file.c` may be any .c file name.

Once you are satisfied that `splint` has told you has much as you can understand and act on, use gcc to attempt to compile your program. Then test and debug as usual.

At first the use of `splint` may seem superfluous or merely a waste of time, but with practise you will come to learn what the warning/error messages mean, and be able to remove many errors from your source before wasting time trying to track them down after compilation.

8. Using gdb to debug programs

The purpose of a debugger such as GDB is to allow you to see what is going on ``inside'' another program while it exe- cutes-or what another program was doing at the moment it crashed.

C programs to be debugged with GDB must be compiled with the -g compiler flag.

eg. `gcc -g -ansi -Wall -pedantic ins-bug.c`

Enter and compile, with -g, the following "buggy program", whilst you may be able to identify all errors by inspection in this case, use this example to learn a little of what gdb can do.

We have uploaded a gdb “cheat sheet” into the course material for this week (gdb.txt). Feel free to print this out and look up commands as you work through this exercise or any of your own debugging.

```
/*
 * This program has deliberate errors.
 * Use gdb to assist you locating them.
 * Please note that you are unwise to
 * simply copy and paste from a pdf into a
 * text editor as some hidden characters that
 * impact on the compiler may be inserted.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{

     int i;
     int *intArray;
     char string[50];
     for (i=0; i<=10; i++)
     {
          intArray[i] = rand() % 10;
     }
     printf("'''string'' is %i characters long\n",
     strlen(string));
     strcpy(string, "This is a medium sized string");
     printf("'''string'' is %i characters long\n",
     strlen(string));
     return EXIT_SUCCESS;
}
```

Here is an example of using gdb. Follow these steps, and attempt to rid the program of all errors.

```
[e70949@csitprdap02 ~]$ gcc -ansi -Wall -pedantic ins-bug.c -o ins-bug -g
ins-bug.c: In function ‘main’:
ins-bug.c:23: warning: implicit declaration of function ‘strlen’
ins-bug.c:23: warning: incompatible implicit declaration of built-in
function ‘strlen’
ins-bug.c:23: warning: format ‘%i’ expects type ‘int’, but argument 2 has
type ‘long unsigned int’
ins-bug.c:24: warning: implicit declaration of function ‘strcpy’
ins-bug.c:24: warning: incompatible implicit declaration of built-in
function ‘strcpy’
ins-bug.c:25: warning: format ‘%i’ expects type ‘int’, but argument 2 has
type ‘long unsigned int’
ins-bug.c:21: warning: ‘intArray’ may be used uninitialized in this
function
ins-bug.c:23: warning: ‘string’ is used uninitialized in this function
[e70949@csitprdap02 ~]$ ./ins-bug
Segmentation fault (core dumped)
[e70949@csitprdap02 ~]$ gdb ins-bug
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
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 "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/el9/E70949/ins-bug...done.
(gdb) list
10    #include <stdio.h>
11    #include <stdlib.h>
12
13    int main()
14    {
15
16        int i;
17        int *intArray;
18        char *string;
19        for (i=0; i<=10; i++)
(gdb)
20        {
21            intArray[i] = rand() % 10;
22        }
23        printf("'string' is %i characters long\n", strlen(string));
24        strcpy(string, "This is a medium sized string");
25        printf("'string' is %i characters long\n", strlen(string));
26        return EXIT_SUCCESS;
27    }
(gdb) run
Starting program: /home/el9/E70949/ins-bug

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005dd in main () at ins-bug.c:21
21            intArray[i] = rand() % 10;
(gdb) display i
1: i = 0
(gdb) display intArray[i]
Disabling display 2 to avoid infinite recursion.
2: intArray[i] = Cannot access memory at address 0x0
(gdb) quit
A debugging session is active.

      Inferior 1 [process 782] will be killed.
```

Quit anyway? (y or n) y

9. Valgrind

It is a common misconception that valgrind is only useful for detecting memory leaks. In fact, valgrind is a collection of many debugging tools and other tools that help you as a programmer. In this course, we barely dip our toes in the water.

Note that you need to have compiled your program with the -g flag to get full information from valgrind.

Let's start with the same source code we started with in the previous exercise. Run valgrind as follows:

```
valgrind ./exe [where exe is the name of your executable]
```

You will get output similar to the following:

```
==30639== Use of uninitialised value of size 8
==30639==    at 0x40070D: main (in /home/el9/E70949/bad)
==30639==
==30639== Invalid write of size 4
==30639==    at 0x40070D: main (in /home/el9/E70949/bad)
==30639==  Address 0x0 is not stack'd, malloc'd or (recently)
free'd
```

Something you need to think about here is “what is typically 8 bytes in size?” “what is typically 4 bytes in size?” “How can I relate this information to what I see in my source code?”

We can actually get even more information from valgrind. This time, run the following command:

```
valgrind –track-origins=yes ./exe
```

In this case you will get information about when the memory was initially allocated. In a program this small it probably is not that useful but in a larger program, it will certainly help you track down where uninitialised values are being used.

Please ensure you read the output from valgrind very carefully. Valgrind is actually rather detailed in its output and if you read the output carefully from top to bottom many debugging tasks become rather trivial. However, if you don't bother reading the output carefully, you may miss some rather obvious things to check.