Lab 04 – UNIX Tools / C Programming

Objectives

- Learning a Unix tool - make
- Using GNU debugger – gdb
- Detecting memory leaks with valgrind
- Explore different C compiler (gcc) options

Compiler Options

You can compile your C program with various levels of optimization turned on (*e.g.*, *-O, -O3, -Ofast*). Here are some useful/popular compiler and optimization options:

The most basic form: `gcc hello.c` executes the complete compilation process and outputs an executable with name a.out

Use option –o: `gcc hello.c -o hello` produces an output file with name 'hello'.

Use option –Wall: `gcc -Wall hello.c -o hello` enables all the warnings in GCC.

Use option –E: `gcc -E hello.c > hello.i` produces the output of preprocessing stage

Use option –S: `gcc -S hello.c > hello.S` produces only the assembly code

Use option –C: `gcc -C hello.c` produces only the compiled code (without linking)

Use option –O: `gcc -O hello.c` sets the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

Syntax

```
$ gcc -Olevel [options] [source files] [object files] [-o output file]
```

Ref: https://www.rapidtables.com/code/linux/gcc/gcc-o.html

Using GNU debugger – gdb

gdb is an extremely useful tool when you have to debug your program for runtime (e.g., segmentation faults, core dumps, array out-of-bound exceptions) and logical errors. To debug C programs using gdb, you have to compile your programs with the -g option to instruct the compile to generate the executable with source-level debug information. Once your program is compiled with the -g option, then you can use gdb to debug your program as shown below:

```
gdb myexefile
gdb -tui myexefile
```

The -tui option displays the source code and gdb command prompt in a split screen (this is the recommended option for new users of gdb). The table below provides some of the commonly used *gdb* commands, the corresponding action performed by these commands, and an example.

gdb command	**Description**	**Example**

file *executable*	specifies the program to execute (if you did not provide it as an argument to gdb)	file a.out
break *[file:]function*	set breakpoint at function (in file)	break main b myfunc
break *line*	set breakpoint at line	break 15 b 15
run *[args]*	execute the program with optional command-line arguments	run run 10 20
set *args*	set command-line arguments	set args 10 20
bt	display the program stack (backtrace)	bt
print *expr*	print the value of the expression	print i p i
continue	continue program execution	continue c
next	execute next program line and step over any function calls in the line	next n
step	execute next program line and step into any function calls in the line	step s
list	display source lines where it is currently stopped (or lines after the last list command)	list l
list *[file]:function*	display source lines from the beginning of the function (in file)	list myfunc l myfunc
list *line*	display source lines around the line	list 15 l 15
where	display where error occurred	where
help *[command]*	display information on using gdb or display information on gdb *command*	help help break
quit	Exit gdb	quit

When you get a segmentation fault, if you compile your program with the -g option and run the program through gdb (using: *gdb -tui myexefile*), you will be able to immediately identify the line that is causing the segmentation fault.

You can find a detailed tutorial on using gdb at: http://beej.us/guide/bggdb/

Detecting memory leaks with valgrind

When you are using dynamic memory allocation, a common problem one encounters is memory leaks and deallocation errors. valgrind is a set of tools that support memory debugging and code profiling on Linux systems (see valgrind documentation for more details on the various tools supported by valgrind). Some of the common uses of valgrind (specifically, memcheck tool) are as follows:

- detect memory leaks
- detect invalid use of pointers
- detect uninitialized variables
- detect improper use of freeing memory

Note that valgrind will NOT be able to perform bound checking on static arrays.

To use valgrind, compile the program with -g option, and run your program with command-line arguments using valgrind as follows:

```
valgrind ./a.out 10
```

The output will display any errors and provide recommendations on additional options to run valgrind again. A detailed explanation of each message can be found at: http://valgrind.org/docs/manual/mc-manual.html#mc-manual.errormsgs.

A simple quick start guide to use valgrind is available at: http://valgrind.org/docs/manual/quick-start.html

Using Makefile – make

make is a utility that is used to automatically detect which program need to be recompiled while working on a large number of source programs and will recompile only those programs that have been modified. The *make* utility uses a *Makefile* to describe the rules for determining the dependencies between the various programs and the compiler and compiler options to use for compiling the programs. In case of C programs, an executable is created from object files (.o files) and object files are created from source files. Source files are often divided into header files (.h files) and actual source files (.c files).

Exercise Makefile – make

Create Make file to compile and execute main.c, hello.c and factorial.c

Below is a Makefile sample that you can use for reference.

```
# Makefile Example

# Variable Declaration
CC = gcc
CFLAGS = -c -Wall
OBJECT = main.o factorial.o hello.o
PROGRAM = output
DEP = functions.h

# Rules
all: $(PROGRAM)

main.o: main.c $(DEP)
        $(CC) $(CFLAGS) main.c

factorial.o: factorial.c $(DEP)
        $(CC) $(CFLAGS) factorial.c

hello.o: hello.c $(DEP)
        $(CC) $(CFLAGS) hello.c

$(PROGRAM): $(OBJECT) $(DEP)
        $(CC) $(OBJECT) -o $(PROGRAM)

exec:
        ./output

clean:
        rm -rf output
        rm -rf *.o

#End of file
```

After creating your Makefile, execute below commands:

1. make
2. make exec
3. Commit some changes in either of the c files (hello.c or main.c) and execute make command again
4. make clean

Note: Create Makefile in same folder/directory as your c code. Also make command, by default looks for Makefile, if you used a different name, run this: "*make -f yourfile.name*".