Systems Software

Week 2: Software Tools (Make, GDB, DDD)

Notes By: Jonathan McCarthy

Overview

- Make
- Make and makefile examples
- GDB
- GDB Tools
- ddd
- gprof
- cvs
- File Hierarchy and I/O

Reference

- The main slide content for the Make and Makefile has been sourced from the gnu.org website.
- The level of detail in the GNU documentation is very complete, for more information please read this resource
- https://www.gnu.org/software/make/manual/make.html

GNU make

- Make was implemented by Richard Stallman and Roland McGrath.
- The GNU make utility is used to automate which parts of a large program may need to be recompiled.
- Make can facilitate the creation of larger programs from separate sources.
- Make can automate tasks for compilation, cleaning, debugging and organising outputs.

Make Usage

- The make command allows for a program to be recompiled is a very specific sequence. It uses a makefile to detail the commands for compiling and updating each file.
- The main executable is made up of object files. The object files are compiled from C source files.
- The makefile contains the sequence for creating the main executable, the make command runs the makefile
- Command line arguments can be provided to make to specify which files should be recompiled

Introduction to makefiles

- A makefile will be created to describe the process needed to recompile a program.
- Most often, the makefile tells make how to compile and link a program.

What a Rule Looks Like

- A simple makefile consists of “rules” with the following shape:
- *target* … : *prerequisites* …
- *command*
- …
- …

Makefile:: *target*

- A *target* is usually the name of a file that is generated by a program. (the output file)
- An example of a target would be executable or object files.
- A target can also be the name of an action to carry out, such as ‘clean’.

Makefile:: *prerequisite*

- A *prerequisite* is a file that is used as input to create the target. A target often depends on several files.

Makefile:: *recipe*

- A *recipe* is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line.
- **Note:** you need to put a tab character at the beginning of every recipe line!!!!
- Usually a recipe is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies a recipe for the target need not have prerequisites. For example, the rule containing the delete command associated with the target ‘clean’ does not have prerequisites.

Makefile:: *rule*

- A *rule*, then, explains how and when to remake certain files which are the targets of the particular rule. make carries out the recipe on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action. See Writing Rules.
- A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look very complicated, but all fit the pattern more or less.

Simple Examples

- Simple Message
- Area Example
- Refactor Makefile

Simple Program 1

newMessage.c ~/Documents/Apps/makeExa...

```c
#include <stdio.h>

void sayHello(char* message) {
   printf("\nMessage: %s\n", message);
}
```

program1.c ~/Documents/Apps/makeExa...

```c
#include <stdio.h>
//#include "newMessage.h"

int main( int argc, char *argv[] )  {

   printf("Hello from program 1");
   char s[50] = "\nI live in program_1";
   sayHello(s);
}
```

newMessage.h ~/Documents/Apps/makeExa...

```c
#ifndef NEWMESSAGE_H_
#define NEWMESSAGE_H_

void sayHello(char* message);

#endif // AREARECTANGLE_H
```

jmccarthy@debianJMC2017: ~/Documents/Apps/makeExample

```
jmccarthy:makeExample$ gcc newMessage.c program1.c -o prog
jmccarthy:makeExample$ ./prog
Hello from program 1
Message:
I live in program_1
jmccarthy:makeExample$
```

Make and Makefile for Sample Program 1

Open ▾ | makefile ~/Documents/Apps/makeExample | Save

```
myprog : program1.o newMessage.o
        gcc -o myprog program1.o newMessage.o

newMessage.o : newMessage.c
        gcc -c newMessage.c

program1.o : program1.c newMessage.h
        gcc -c program1.c

clean:
        rm myprog newMessage.o program1.o
```

Makefile ▾ Tab Width: 8 ▾ Ln 6, Col 1 ▾ INS

Make and Makefile for Sample Program 1

jmccarthy@debianJMC2017: ~

File Edit View Search Terminal Help

```
jmccarthy:makeExample$ make clean
rm myprog newMessage.o program1.o
jmccarthy:makeExample$ make
gcc -c program1.c
gcc -c newMessage.c
gcc -o myprog program1.o newMessage.o
jmccarthy:makeExample$ ./myprog
Hello from program 1
Message:
I live in program_1
jmccarthy:makeExample$
```

Sample Program 2

areaOfCircle.c
~/Documents/Apps/makeExample2

```c
#include<stdio.h>

float areaOfCircle(float radius) {
   float area;

   //printf("\nEnter the radius of Circle : ");
   //scanf("%d", &radius);

   area = 3.14 * radius * radius;
   //printf("\nArea of Circle : %f", area);

   return area;
}
```

C Tab Width: 8 Ln 10, Col 44 IN

areaOfCircle.h
~/Documents/Apps/makeExa

```c
#ifndef AREAOFCIRCLE_H_
#define AREAOFCIRCLE_H_

int areaOfCircle(int length);

#endif //  AREAOFCIRCLE_H_
```

Sample Program 2

Open ▾ *areaOfRectangle.c ~/Documents/Apps/makeExample2 Save

```c
int areaOfRectangle(int length, int bredth) {
    int area;

    area = length * bredth;

    return area;
}
```

Open ▾ areaOfRectangle.h ~/Documents/Apps/makeExample2 Save

```c
#ifndef AREAOFRECTANGLE_H_
#define AREAOFRECTANGLE_H_

int areaOfRectangle(int length, int bredth);

#endif //  AREAOFRECTANGLE_H_
```

Sample Program 2

Open areaOfSquare.c ~/Documents/Apps/makeExample2

```c
#include "areaOfRectangle.h"

int areaOfSquare(int length) {
    int area;

    area = areaOfRectangle(length, length);

    return area;
}
```

Open areaOfSquare.h ~/Documents/Apps/makeExample2

```c
#ifndef AREAOFSQUARE_H_
#define AREAOFSQUARE_H_

int areaOfSquare(int length);

#endif //  AREAOFSQUARE_H_
```

Sample Program 2

Open ▾ | area_of_triangle.c ~/Documents/Apps/makeExample2 | Save

```c
#include <math.h>

double area_of_triangle( double a, double b, double c )
{
   double s, area;

   s = (a+b+c)/2;

   area = sqrt(s*(s-a)*(s-b)*(s-c));

   return area;
}
```

Open ▾ | *area_of_triangle.h ~/Documents/Apps/makeExample2 | Save

```c
#ifndef AREAOFTRIANGLE_H_
#define AREAOFTRIANGLE_H_

area_of_triangle( double a, double b, double c );

#endif //  AREAOFTRIANGLE_H_
```

Sample Program 2

Open

main.c
~/Documents/Apps/makeExample2

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


int main( int argc, char *argv[] )  {
   int aor;

   aor = areaOfRectangle(10,80);
   printf("Area of Recctangle: %d", aor);
}
```

Sample Program 2

Open ▾ | makefile ~/Documents/Apps/makeExample2 | Save

```
myprog : main.o areaOfRectangle.o areaOfSquare.o areaOfCircle.o area_of_triangle.o
	gcc -o areaProg main.c areaOfRectangle.o areaOfSquare.o areaOfCircle.o
area_of_triangle.o -lm

main.o : main.c areaOfRectangle.h areaOfSquare.h areaOfCircle.h area_of_triangle.h
	gcc -c main.c

areaOfRectangle.o : areaOfRectangle.c
	gcc -c areaOfRectangle.c

areaOfSquare.o  :areaOfSquare.c areaOfRectangle.h
	gcc -c areaOfSquare.c

areaOfCircle.o : areaOfCircle.c
	gcc -c areaOfCircle.c

area_of_triangle.o : area_of_triangle.c
	gcc -c area_of_triangle.c -lm

clean:
	rm areaProg main.o areaOfRectangle.o areaOfSquare.o areaOfCircle.o area_of_triangle.o
```

Refactor Makefile

Open ▾ makefile ~/Documents/Apps/makeExample3 Save

```
CC=gcc
objects = main.o areaOfRectangle.o areaOfSquare.o areaOfCircle.o area_of_triangle.o
headers = areaOfRectangle.h areaOfSquare.h areaOfCircle.h area_of_triangle.h

myprog : $(objects)
        $(CC) -o areaProg $(objects) -lm

main.o : main.c $(headers)
        $(CC) -c main.c

areaOfRectangle.o : areaOfRectangle.c
        $(CC) -c areaOfRectangle.c

areaOfSquare.o  :areaOfSquare.c areaOfRectangle.h
        $(CC) -c areaOfSquare.c

areaOfCircle.o : areaOfCircle.c
        $(CC) -c areaOfCircle.c

area_of_triangle.o : area_of_triangle.c
        $(CC) -c area_of_triangle.c -lm

clean:
        rm areaProg $(objects)
```

GDB

- GDB is the GNU's Project debugger
- GDB gives an insight into what is actually happening within a program while it is running.
- There are 4 specific tasks GDB can help with to try identify bugs:
  - Start your program, specifying anything that might affect its behavior.
  - Make your program stop on specified conditions.
  - Examine what has happened, when your program has stopped.
  - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Source: https://www.sourceware.org/gdb/

GDB

- GDB is mainly used to debug programs written in C or C++.
- GDB is free software, protected by the gnu General Public License (GPL).

Using GDB – Compiling Programs

- To let GDB be able to read all that information line by line from the symbol table, we need to compile it a bit differently.
- A **Debugging Symbol Table** maps instructions in the compiled binary program to their corresponding variable, function, or line in the source code.
- Symbol tables may be embedded into the program or stored as a separate file. So if you plan to debug your program, then it is required to create a symbol table which will have the required information to debug the program.
- Use the –g flag when compiling your programs with GCC

Installing GDB in Debian

- su
- apt-get update
- apt-get install gdb

Using GDB – Run Program

jmccarthy@debianJMC2017: ~/Documents/Apps/gdbExample1

File Edit View Search Terminal Help

```
jmccarthy@debianJMC2017:~/Documents/Apps/gdbExample1$ ls
hello  hello.c
jmccarthy@debianJMC2017:~/Documents/Apps/gdbExample1$ gdb hello
```

- To run the program in GDB

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...(no debugging symbols found)...done.
(gdb) run
```

Using GDB – Run Program

- If the program takes parameters:

```
For help, type "help".
Type "apropos word" to search for commands
Reading symbols from hello...(no debugging
(gdb) run param1 param2 param3
```

- Running Program:

```
(gdb) run
Starting program: /home/jmccarthy/Documents/Apps/gdbExample1/hello
Enter your age:
```

Using GDB – Commands

- To stop the program: **kill**
- To start the program again: **run**
- To quit GDB: **quit**

- **There are a lot of GDB commands that we won't cover in the slides, have a look at the help pages.**

GDB:: Lifeline of program execution

- With GDB it is possible to:
  - Stop the program
  - See where the program has stopped
  - View contents of variables
  - Set variables
  - Call functions
  - Etc…

GDB:: View the operation of program

- Ctrl C will stop the program
- The continue keyword restarts the program
- For a stopped program, list shows us where the program currently is
- To step through the code use breakpoints with the next and step commands (we sill cover breakpoints later in the slides)

GDB:: View content of a variable

- To view the content of a variable use the print command
  - print age
- Example

Open ▾ | hello.c ~/Documents/Apps/gdbExam... | Save

```
#include <stdio.h>

int main() {

   int age;

   printf("Enter your age:");

   scanf("%d", &age);

}
```

```
gcc -g hello.c -o hello
gdb hello
```

```
Reading symbols from hello...done.
(gdb) break 10
Breakpoint 1 at 0x4005a3: file hello.c, line 10.
(gdb)
```

```
(gdb) run
Starting program: /home/jmccarthy/Documen
Enter your age:12

Breakpoint 1, main () at hello.c:11
11      }
(gdb) print age
$1 = 12
(gdb)
```

GDB:: Changing the value of a variable

- It is possible to change the value of a variable in GDB.
- If the program is not operating as expected, it is possible to set a variable.
- Eg.
- set age = 21
- print age

GDB:: Calling functions

- To call a function in a C use the following:
- **call displayResults()**
- Use the finish to get the function to complete its actions and return a value (if any)

GDB:: Breakpoints

- A breakpoint can be added to stop the program executing at a particular point in the program. (ie a particular line number)
- It is also possible to stop the program at a specific function call.
- When the program has stopped it possible to values variables are holding, examine the stack and step through the program.

GDB:: Breakpoint:: line

- Enter GDB for the program in question
- Add a breakpoint using the following syntax:
  - **break 12**
  - This will add a breakpoint to line 12
- When the program is run it will stop at each breakpoint in the program
- If there are multiple files in the program, you must specify the filename when setting the breakpoint
  - **break hello.c:12**

GDB:: Breakpoint:: function

- Enter GDB for the program in question
- Add a function breakpoint using the following syntax:
  - **break myfunction**
  - This will add a breakpoint to line 12
- When the program is run it will stop at each breakpoint in the program

GDB:: Temporary Breakpoints

- A temporary breakpoint stops once, then the breakpoint is disposed of.
- Syntax:
  - **tbreak 12**
- The standard rules for the general breakpoints apply!!

GDB:: View breakpoints

- To get a list of all breakpoints:
- **info breakpoints**

GDB:: Breakpoints:: remove

- To remove a breakpoint use the following syntax:
- **disable 12**

GDB:: Breakpoints:: skip

- If you wish to skip a breakpoint, use the following syntax:
- **ignore 12 1**
- This will ignore the breakpoint on line 12 for one iteration.

GDB:: Watchpoints

- A watchpoint can be added to a variable.
- If we have a variable named age
  - **watch age**
- Read watchpoint: **rwatch age**
- Rear/Write watchpoint: **awatch age**
- The standard breakpoint command work with watchpoints:
  - **info, disable**

GDB:: Backtrace

- The backtrace command can be used to examine the stack to see the the stack frames that control program flow.
- The stack frames are used to tell a function where to return to after a function is called.
- To examine a stack frame: **info frame**
- The **backtrace** command gives a list of the stack frames
- The frame command can be used to swith between stack frames
  - **frame 12**

DDD – Data Display Debugger

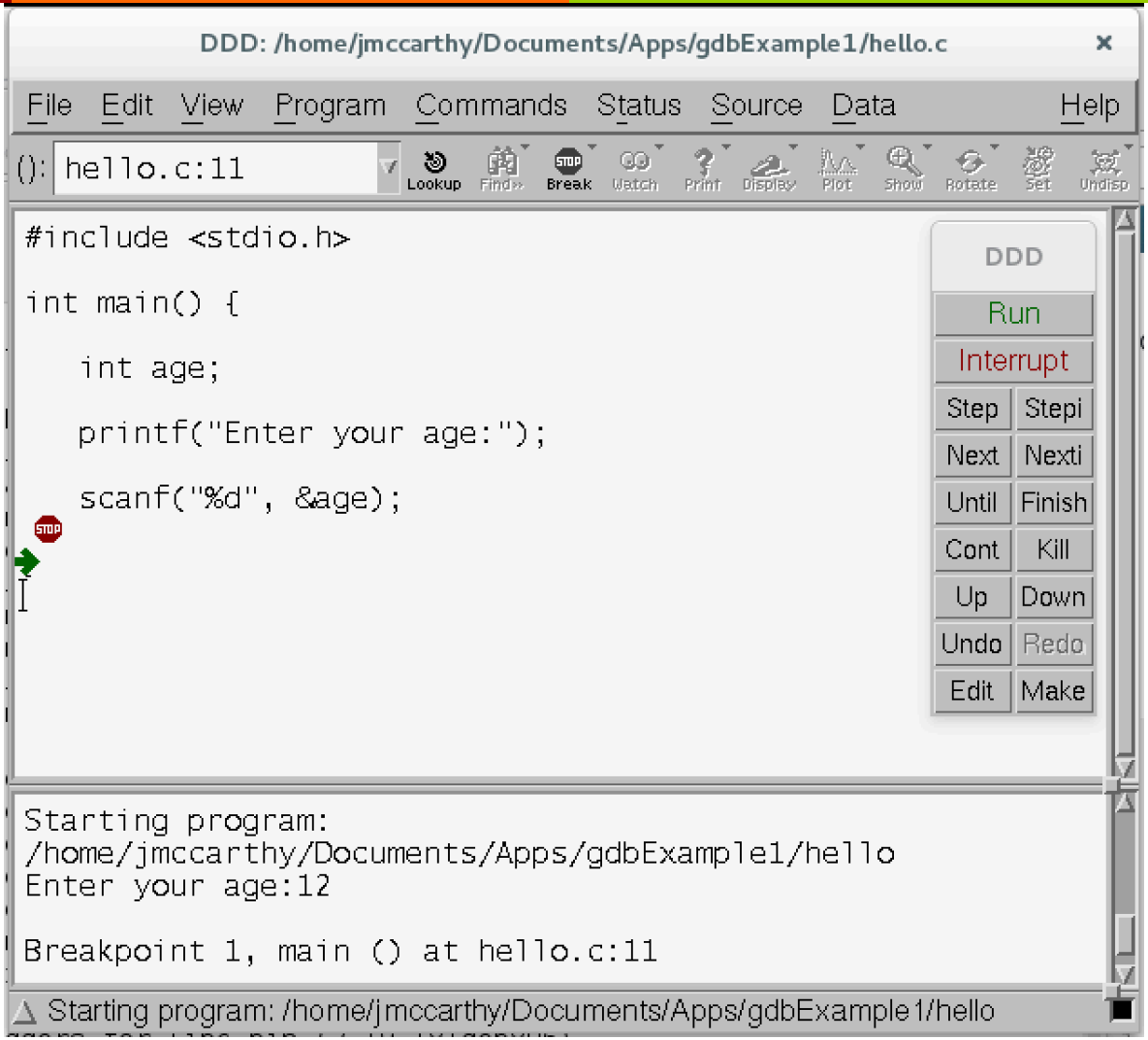
- DDD is a GUI for debugging programs
- The default debugger for DDD is GDB
- The standards operation of GDB is the exact same, the only benefit is this is a GUI application instead of the command line

- **More info:**
- https://www.gnu.org/software/ddd/manual/pdf/ddd.pdf

Installing DDD in Debian

- su
- apt-get update
- apt-get install ddd

DDD - GUI


DDD: /home/jmccarthy/Documents/Apps/gdbExample1/hello.c
File Edit View Program Commands Status Source Data Help
(): hello.c:11
Lookup Find>> Break Watch Print Display Plot Show Rotate Set Undisp
#include <stdio.h>
int main() {
int age;
printf("Enter your age:");
scanf("%d", &age);
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
Starting program:
/home/jmccarthy/Documents/Apps/gdbExample1/hello
Enter your age:12
Breakpoint 1, main () at hello.c:11
Starting program: /home/jmccarthy/Documents/Apps/gdbExample1/hello

Additional Resources

- GDB Documentation:
  - https://www.sourceware.org/gdb/documentation/
  - https://www.sourceware.org/gdb/
- DDD
  - https://www.gnu.org/software/ddd/manual/pdf/ddd.pdf
- Make
  - https://www.gnu.org/software/make/manual/make.html

Questions