Data and File Structures Laboratory

Tools: Gcov, Cscope, Ctags, and Makefiles

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

August, 2018

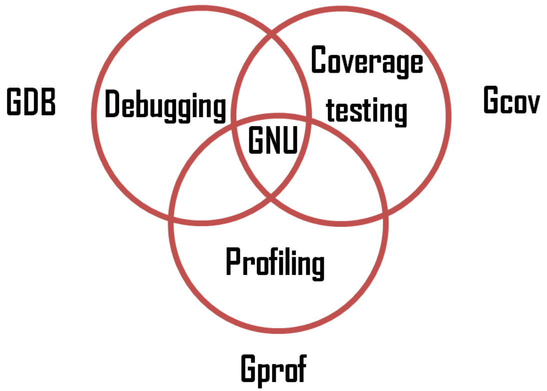
Gcov

Gcov stands for `GNU Coverage Testing Tool`

- It is a source code coverage analysis and statement-by-statement profiling tool.
- It explores the frequency of execution for each line of a source code during a run.
- It can find the portions of a source code which are not used (or not exercised in testing).
- It (in combination with Gprof) can speed up a program by concentrating on specific lines of the source code.
- Gcov currently supports the following languages: Fortran, C.

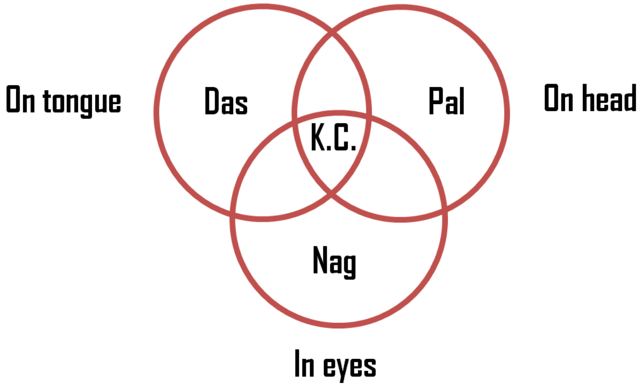
Note: Lcov is a graphical front-end for Gcov.

Managing the source code



Note: A lot more is there from GNU.

Managing water – A little fun!!!



Running a program with Gcov

- Compile a program to allow the compiler to prepare an instrumented executable (with coverage details)
  `$gcc -Wall -fprofile-arcs -ftest-coverage -o progx progx.c` (progx.gcno is created)
- Execute the program
  `$./progx` (progx.gcda is created)
- Run the program with Gcov
  `$gcov progx.c` (progx.c.gcov is created)

1. Gcov produces an annotated version of the original source file containing the frequency of executions at different levels.

Note: The options '-fprofile-arcs' and '-ftest-coverage' ask the compiler to generate additional information needed by Gcov and include additional code in the object files for generating the extra profiling information, respectively.

Running a program with Gcov – An example

```
#include<stdio.h>
int main(){
    int i;
    for(i=1; i<=10; i++){
        if(i%3 == 0)
            printf("%d is divisible by 3\n", i);
        if(i%11 == 0)
            printf("%d is divisible by 11\n", i);
    }
    return 0;
}
```

Running a program with Gcov – An example

```
#include<stdio.h>
int main(){
    int i;
    for(i=1; i<=10; i++){
        if(i%3 == 0)
            printf("%d is divisible by 3\n", i);
        if(i%11 == 0)
            printf("%d is divisible by 11\n", i);
    }
    return 0;
}
```

1. `$ gcc -Wall -fprofile-arcs -ftest-coverage -o progx progx.c`
2. `$./progx`
3. `gcov progx.c`

```
File ‘progx.c’
Lines executed:85.71% of 7
Creating ‘progx.c.gcov’
```

Running a program with Gcov – An example

Contents of the file `progx.c.gcov`:

```
    -:    0:Source:progx.c
    -:    0:Graph:progx.gcno
    -:    0:Data:progx.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include<stdio.h>
    1:    2:int main(){
    -:    3:    int i;
   11:    4:    for(i=1; i<=10; i++){
   10:    5:        if(i%3 == 0)
    3:    6:            printf("%d is divisible by 3\n", i);
   10:    7:        if(i%11 == 0)
#####:    8:            printf("%d is divisible by 11\n", i);
    -:    9:    }
    1:   10:    return 0;
    -:   11:}
```

Running Gcov with different options

- Run the program to include branch frequencies (in percentage) and show the summary on standard output
  ```
  $gcov -b progx.c
  ```
- Run the program to include branch frequencies (in count)
  ```
  $gcov -c progx.c
  ```
- Run the program to include output summaries for each function in addition to the file level summary
  ```
  $gcov -f progx.c
  ```
- Run the program to display the progress on standard output
  ```
  $gcov -d progx.c
  ```

Note: There are plenty of other options like '-a' (for all blocks), '-i' (for intermediate format), '-j' (for human readable format), '-k' (for using colors), '-m' (for demangled function names), etc.

Running Gcov with different options – An example

```
#include<stdio.h>
int main(){
    int i, sum = 0;
    for(i=0; i<10; i++)
        sum += i;
    if(sum != 45)
        printf("Failed ...\n");
    else
        printf("Succeed ...\n");
}
```

Running Gcov with different options – An example

```
#include<stdio.h>
int main(){
    int i, sum = 0;
    for(i=0; i<10; i++)
        sum += i;
    if(sum != 45)
        printf("Failed ...\n");
    else
        printf("Succeed ...\n");
}
```

1. `gcov -b progx.c`

```
   File 'progx.c'
   Lines executed:85.71% of 7
   Branches executed:100.00% of 4
   Taken at least once:75.00% of 4
   Calls executed:50.00% of 2
   Creating 'progx.c.gcov'
```

Running Gcov with different options – An example

Contents of the file `progx.c.gcov`:

```
    ...
    -:    1:#include<stdio.h>
function main called 1 returned 100% blocks executed 88%
    1:    2:int main(){
    1:    3:    int i, sum = 0;
   11:    4:    for(i=0; i<10; i++)
branch  0 taken 91%
branch  1 taken 9% (fallthrough)
   10:    5:        sum += i;
    1:    6:    if(sum != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
#####:    7:        printf("Failed ...\n");
call    0 never executed
    -:    8:    else
    1:    9:        printf("Succeed ...\n");
call    0 returned 100%
    -:   10:}
```

Cscope

Cscope is a developer's tool for browsing the source code. It can be used to find many different things present in the source files.

- Find a C symbol
- Find a global definition
- Find functions called by a function
- Find functions calling a function
- Find a text string
- Change a text string
- Find a egrep pattern
- Find a file
- Find files including a file
- Find assignments to a symbol

Using Cscope

- Open a particular source file in Cscope
  ```
  $cscope progx.c
  ```
- Search in the entire current directory
  ```
  $cscope -R
  ```
- Exit from the environment
  `$^d` (Press ctrl+d)

Note: Use the shortcut keys *tab* and *upper/lower arrows* to navigate within the Cscope environment.

Cscope – An example

```
#include<stdio.h>
void copyFile(FILE *, FILE *); // Prototype declaration
void main(int argc, char *argv[]){
    FILE *fptr;
    while(--argc > 0){
        if((fptr = fopen(*++argv, "r")) == NULL){
            printf("The file %s is not found\n", *argv);
        else{
            copyFile(fptr, stdout); // Characterwise file copying
            fclose(fptr);
        }
    }
}
void copyFile(FILE *inputFile, FILE *outputFile){
    int ch;
    while((ch = fgetc(inputFile)) != EOF)
        putc(ch, outputFile);
}
```

Cscope – Find a C symbol

```
C symbol: FILE

  File    Function                Line
0 progx.c <global>                  2 void copyFile(FILE *, FILE *);
1 stdio.h <global>                 48 typedef struct _IO_FILE FILE;
2 progx.c main                      4 FILE *fptr;
3 progx.c main                     14 void copyFile(FILE *inputFile, FILE
                                      *outputFile){
4 stdio.h __USING_NAMESPACE_STD    53 __USING_NAMESPACE_STD(FILE)
5 stdio.h renameat                195 extern FILE *tmpfile (void ) __wur;

* Lines 1-7 of 86, 80 more - press the space bar to display more *
```

Find this C symbol: FILE (Press Enter)

Cscope – Find a global definition

```
Could not find the global definition: ch
```

Find this global definition: ch (Press Enter)

Cscope – Find functions called by a function

```
Functions called by this function: main

  File     Function Line
0 progx.c fopen     6 if((fptr = fopen(*++argv, "r")) == NULL){
1 progx.c printf    7 printf("The file %s is not found\n", *argv);
2 progx.c copyFile  9 copyFile(fptr, stdout);
3 progx.c fclose   10 fclose(fptr);
4 progx.c copyFile 14 void copyFile(FILE *inputFile, FILE *outputFile){
5 progx.c fgetc    16 while ((ch = fgetc(inputFile)) != EOF)
6 progx.c putc     17 putc(ch, outputFile);
```

Find functions called by this function: main (Press Enter)

Cscope – Find functions calling a function

```
Functions calling this function: copyFile

  File     Function Line
0 progx.c main       9 copyFile(fptr, stdout);
1 progx.c main      14 void copyFile(FILE *inputFile, FILE *outputFile){
```

Find functions calling this function: copyFile (Press Enter)

Cscope – Find a text string

```
Text string: std

  File        Line
0 prog3.c       1 #include<stdio.h>
1 prog3.c       9 copyFile(fptr, stdout); // Characterwise file copying
2 stdio.h       1 /* Define ISO C stdio on top of C++ iostreams.
3 stdio.h      20 * ISO C99 Standard: 7.19 Input/output <stdio.h>
4 stdio.h      33 # include <stddef.h>
5 stdio.h      83 #  include <stdarg.h>
6 stdio.h     164 #include <bits/stdio_lim.h>
7 stdio.h     168 extern struct _IO_FILE *stdin;  /* Standard input stream. */

* Lines 1-9 of 62, 54 more - press the space bar to display more *
```

Find this text string: std (Press Enter)

Cscope – Change a text string

```
Change "fptr" to "fp"

  File    Line
0 prog3.c  4 FILE *fptr;
1 prog3.c  9 copyFile(fptr, stdout); // Characterwise file copying
2 prog3.c 10 fclose(fptr);
```

Change this text string: fptr (Press Enter) To: fp (Press Enter) Select lines to change (press the ? key for help): 2 (Press Enter)

Cscope – Find a egrep pattern

```
Egrep pattern: (in|out)putFile

  File    Line
0 prog3.c 14 void copyFile(FILE *inputFile, FILE *outputFile){
1 prog3.c 16 while((ch = fgetc(inputFile)) != EOF)
2 prog3.c 17 putc(ch, outputFile);
```

Find this egrep pattern: (in—out)putFile (Press Enter)

Cscope – Find a file

```
File: stdio.h

  File
0 /usr/include/stdio.h
```

Find this file: stdio.h (Press Enter)

Cscope – Find files #including a file

```
Files #including this file: stdio.h

  File    Line
0 prog3.c   1 #include <stdio.h>
1 prog3.c   1 #include <stdio.h>
2 stdio.h 933 #include <bits/stdio.h>
3 wchar.h  36 #include <stdio.h>
```

Find files including this file: stdio.h (Press Enter) (Press Enter)

Cscope – Find assignments to a symbol

```
Assignments to this symbol: ch

  File    Line
0 prog3.c 16 while ((ch = fgetc(inputFile)) != EOF)
```

Find assignments to this symbol: ch (Press Enter)

Ctags

Ctags is a tool that makes it easy to navigate between the files in a large source code project. It provides some of the features like jumping from the current source file to definitions of functions and structures in other files.

Ctags does not create lookup tags for local variables and references by default, because tools such as ctags and cscope are used for large codebases, where if local variables are tagged, it will create a confusing list of lookups.

- Run a program with Ctags
  `$ctags progx.c` (The tags file is created)
- Search in the entire current directory
  `$ctags -R` (The tags file is created)
- Exit from the environment
  `$^d` (Press ctrl+d)

Ctags – An example

```
!_TAG_FILE_FORMAT       2       /extended format; --format=1 will not append ;
" to lines/
!_TAG_FILE_SORTED       1       /0=unsorted, 1=sorted, 2=foldcase/
!_TAG_PROGRAM_AUTHOR    Darren Hiebert  /dhiebert@users.sourceforge.net/
!_TAG_PROGRAM_NAME      Exuberant Ctags //
!_TAG_PROGRAM_URL       http://ctags.sourceforge.net    /official site/
!_TAG_PROGRAM_VERSION   5.9~svn20110310 //
main    progx.c /^void main(int argc, char *argv[]){$/;"        f
...
```

Makefiles

If a source file is recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable.

The makefiles are specially formatted text files to organize code compilation. It tells *make* how to compile and link a program.

Note: Purpose of the *make* utility is to determine automatically which pieces of a large program need to be recompiled, and issue the necessary commands to recompile them.

Makefiles – An example

Contents of the file `progx.c`:

```
#include<addMake.h>
int main(){
    // call a function in another file
    displayMake();
    return 0;
}
```

Contents of the file `addFunction.c`:

```
#include<stdio.h>
#include<addMake.h>
int displayMake(void){
    printf("Using makefiles ...");
    return 0;
}
```

Contents of the file `addMake.h`:

```
// Function prototype declaration
int displayMake(void);
```

Makefiles – An example

Option 1 for compilation: Without creating any library (trivial)

```
$gcc -o progx progx.c addFunction.c
```

Option 2 for compilation: By creating static library (treating object files separately)

```
$gcc -c addFunction.c -o addFunction.o
$gcc -c progx.c -o progx.o
$gcc -o progx main.o addFunction.o
```

Option 3 for compilation: Using Makefiles

Let's see!!!

Makefiles – An example

Contents of `makefile (Example 1)`:

```
progx: progx.c addFunction.c
	gcc -o progx progx.c addFunction.c -I.
```

Note: There must be a *tab* at the beginning of any command (e.g., *gcc* here).

Makefiles – An example

Contents of `makefile (Example 2)`:

```
CC=gcc
CFLAGS=-I.

progx: progx.o addFunction.o
     $(CC) -o progx progx.o addFunction.o
```

Note: The macro *CC* defines the C compiler to use, and *CFLAGS* defines the list of flags to pass to the compilation command.

Makefiles – An example

Contents of `makefile (Example 3)`:

```
CC=gcc
CFLAGS=-I.
DEPS = addMake.h

%.o: %.c $(DEPS)
$(CC) -c -o $@ $< $(CFLAGS)
progx: progx.o addFunction.o
$(CC) -o progx progx.o addFunction.o
```

Note: The macro *DEPS* defines all the set of *.h* files on which the *.c* files depend on.

Makefiles – An example

Contents of `makefile (Example 4)`:

```
CC=gcc
CFLAGS=-I.
DEPS = addMake.h
OBJ = progx.o addFunction.o

%.o: %.c $(DEPS)
$(CC) -c -o $@ $< $(CFLAGS)
progx: $(OBJ)
$(CC) -o $@ $^ $(CFLAGS)
```

Note: The macro *OBJ* defines all the object files used.

Makefiles – An example

Contents of `makefile (Example 5)`:

```
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)
ODIR=obj
LDIR =../lib
LIBS=-lm
_DEPS = addMake.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
_OBJ = addMake.o addFunction.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))

$(ODIR)/%.o: %.c $(DEPS)
$(CC) -c -o $@ $< $(CFLAGS)
hellomake: $(OBJ)
$(CC) -o $@ $^ $(CFLAGS) $(LIBS)
.PHONY: clean
clean:
rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

Makefiles – PHONY (dummy) targets

- Useful to “just” run commands
- Not a file, BUT
- `make` runs commands to “build” target

Example:

```
.PHONY: clean
clean:
    $(RM) *.o *~
```

Makefiles – Some common variables

`AR`	Archive-maintaining program (default: `ar`)
`CC`	Program for compiling C programs (default: `cc`)
`CXX`	Program for compiling C++ programs (default: `g++`)
`RM`	Command to remove a file (default: `rm -f`)

Makefiles – Some common flags

`ARFLAGS`	Flags to give the ar program (default: `rv`)
`CFLAGS`	Extra flags to give to the C compiler
`CXXFLAGS`	Extra flags to give to the C++ compiler
`LDFLAGS`	Extra flags to give to compilers while invoking the linker

Makefiles – Special symbols

`$@`	name of the target
`$?`	names of the changed dependencies
`$^`	names of all dependencies
`$<`	name of first dependency

Problems – Day 13

1. Prepare Makefiles for the codebase provided in Test.zip.