Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Software Systems

Lectures Week 8

Basic Software Engineering Techniques

(Modular programming + GNU Tools + Repositories)

Prof. Joseph Vybihal

Computer Science

McGill University

Part A

Modular Programming in C

Readings: chapter 4

Modular Programming

A program composed from multiple objects

Other types:

- Single Text File programming (the trivial case)
- Multiple Text File but Single Source programming

Why is modular programming useful?

- Objects are standalone (opens up engineering possibilities)
- Helpful when working in a team of developers
- Helpful when understanding the concept of “named spaces”

Single Source Multiple Text File

We create our program using many .c files, but we merge them at compile-time into a single document.

```
Bash-prompt $ vi file1.c
Bash-prompt $ vi file2.c
Bash-prompt $ vi file3.c
Bash-prompt $ gcc file1.c
Bash-prompt $ ./a.out
```

Notice:
- Three .c files are created, but
- We only compile the first source file
- There exists a command in the first source file that merges the other source files into itself

Benefits:
- More than one person can work on project.

Drawback:
- Compiling entire project each time.

#include

The #include directive has two formats:

- #include<library>
  - Merge a library into your program
- #include "path/textfile"
  - Merge a text file into your program

```
#include "file2.c"              // loads source file from current directory

#include "/user/jack/file2.c"   // loads from specified directory

#include "../jack/file2.c"      // loads from a relative directory
```

Example

Linkedlist.c

```
#include<stdio.h>
void printList(struct NODE *ptr) {
            …..
}
```

MyMax.c

```
#include<stdlib.h>
int max(int a, int b, int c) {
      int aMax = a;
      aMax = (b>aMax) ? b : aMax;
      aMax = (c>aMax) ? c : aMax;
      return aMax;
}
```

```
#include<stdio.h>
#include "Linkedlist.c"
#include "MyMax.c"

int main() {
    int a,b,c,result;
    scanf("%d %d %d", &a, &b, &c);
    result = max(a,b,c);
    printf("%d\n", result);
}
```

Main.c

```
Bash-prompt $ vi Linkedlist.c
Bash-prompt $ vi MyMax.c
Bash-prompt $ vi Main.c
Bash-prompt $ gcc Main.c
Bash-prompt $ ./a.out
```

Modular Programming

We create our program using many .c files. We compile each source file separately, and merge later.

```
Bash-prompt $ vi file1.c
Bash-prompt $ vi file2.c
Bash-prompt $ vi file3.c
Bash-prompt $ gcc –c file1.c file2.c file3.c
Bash-prompt $ gcc file1.o file2.o file3.o
Bash-prompt $ ./a.out
```

Notice:

- The gcc –c command compiles each source file separately saving each .c file as a compiled .o file. These .o files cannot execute on their own.
- The gcc command by itself merges all the .o files into an a.out executable file.

Benefits:
- More than one person can work on project.
- Only changed files needs to be compiled.
- Each file has its own named space.

Drawback:
- Compiling is more complicated.

Named Space

- A semi-private space.
    - Each .o file's global variables are, by default, local to only the .o file it was **defined** & **compiled** within.
    - In contrast, all function names are public but "hidden"
- Using the extern command a global object's variable can be accessed publicly.

File1.c

```
int x;
void printX() {
   printf("%d\n", x);
}
```

File2.c

```
int main() {
   printf("%d\n", x);            // error
   printX();                     // legal
}
```

```
Bash-prompt $ gcc –c File1.c File2.c
Bash-prompt $ gcc File1.o File2.o
Bash-prompt $ ./a.out
```

Named Space

- Using the extern command a global object's variable can be accessed publicly.

File1.c

```
int x;
void printX() {
    printf("%d\n", x);
}
```

File2.c

```
extern int x;
int main() {
    printf("%d\n", x);          // legal
    printX();                   // legal
}
```

```
Bash-prompt $ gcc –c File1.c File2.c
Bash-prompt $ gcc File1.o File2.o
Bash-prompt $ ./a.out
```

Variable Example

Linkedlist.c

```
#include<stdio.h>
struct NODE *head;
void printList(struct NODE *ptr) {
            .....
}
```

MyMax.c

```
#include<stdlib.h>
int max(int a, int b, int c) {
    int aMax = a;
    aMax = (b>aMax) ? b : aMax;
    aMax = (c>aMax) ? c : aMax;
    return aMax;
}
```

Main.c

```
#include<stdio.h>
extern struct NODE *head;

int main() {
   int a,b,c,result;
   scanf(“%d %d %d”, &a, &b, &c);
   result = max(a,b,c);
   printf(“%d\n”, result);
}
```

```
Bash-prompt $ vi Linkedlist.c
Bash-prompt $ vi MyMax.c
Bash-prompt $ vi Main.c
Bash-prompt $ gcc –c Main.c Linkedlist.c MyMax.c
Bash-prompt $ gcc Main.o Linkedlist.o MyMax.o
Bash-prompt $ ./a.out
```

Function Example

Linkedlist.c

```
#include<stdio.h>
struct NODE *head;
void printList(struct NODE *ptr) {
            …..
}
```

MyMax.c

```
#include<stdlib.h>
int max(int a, int b, int c) {
    int aMax = a;
    aMax = (b>aMax) ? b : aMax;
    aMax = (c>aMax) ? c : aMax;
    return aMax;
}
```

Main.c

```
#include<stdio.h>
extern struct NODE *head;

int main() {
   int a,b,c,result;
   scanf(“%d %d %d”, &a, &b, &c);
   result = max(a,b,c);
   printf(“%d\n”, result);
}
```

```
Bash-prompt $ gcc –c Main.c Linkedlist.c MyMax.c
Bash-prompt $ gcc Main.o Linkedlist.o MyMax.o
Bash-prompt $ ./a.out
```

Notice max() can be invoked from Main.c, but MyMax.c was compiled into MyMax.o and shared with the developer of Main.c. Since .o files are compiled the Main.c developer only knows the name of the methods because that information was told by the MyMax.c developer (ie. “hidden”, requires special knowledge).

The .h File

In the last example we saw:

extern struct NODE *head;

The above command assumes that the definition for NODE exists in both the Linkedlist.c and Main.c files.

- This can be done in two ways:
  - Method 1: write the NODE definition in both files
  - Method 2: write the NODE definition in one file and share

Example

Node.h

```
struct NODE {
    int data;
    struct NODE *next;
};
```

Linkedlist.c

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

struct NODE *head;
void printList(struct NODE *ptr) {
                .....
}
```

Main.c

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

extern struct NODE *head;

int main() {
    int a,b,c,result;
    scanf("%d %d %d", &a, &b, &c);
    result = max(a,b,c);
    printf("%d\n", result);
}
```

This is considered better form because you avoid duplication errors by writing the definition once.

Problem

Abigail, Bethany, and Sebastian want to work together on a telephone book program. A text file contains the names and numbers in the book (assume one name and number per line). The size of the text file is unknown. A user wants to enter the name and see the number.

How can they work together to create this C program (given what we have covered)?

The Pre-processor

Before gcc compiles your code it will pre-process it.

- This means it will make changes to your source file before it compiles it.

Common pre-processor commands used by developers:

- #include ← we have seen this already
- #define
- #ifdef
- #ifndef

(many other exist, beyond scope of the course)

#define

A technique by which a developer can define new terms or commands that are not part of the C language.

Syntax:

- #define TERM EXPRESSION
  - The EXPRESSION will be inserted into the source code everywhere it finds TERM

Examples:

- #define TRUE 1
- #define FALSE 0
- if (x == TRUE) { … }

Using #define as a macro

A variable #define command

Syntax:

- #define TERM(ARG1, ARG2) EXPRESSION
  - The EXPRESSION will be inserted into the source code everywhere it finds TERM
  - ARG1 and ARG2 will be inserted into the EXPRESSION

Example:

- #define MAX(A,B) (A>B)?A:B
- int result = MAX(5,10);

#ifdef and #ifndef

The pre-processor if-statement

Syntax:

- #ifdef TERM
  - If TERM has been previously #define’d then true, else false.
- #ifndef TERM
  - If TERM has not been previously #define’d then true, else false.

General syntax:

```
#ifdef TERM
    // multiple C programming statements go here
#else
    // optional #else part, multiple C programming statements go here
#endif
```

Optional Compiling Example

```
#define FRENCH
// #define ENGLISH
int main() {
    int age;
    #ifdef FRENCH
            printf(“Votre age: “);
    #endif
    #ifdef ENGLISH
            printf(“Your age: “);
    #endif
    scanf(“%d”, &age);
    :
}
```

This actually compiles your program into a French version without any English.

Instead of keeping two versions of your program you can keep one version and just compile into the language you want.

Crash proof .h file example

Node.h

```
#ifndef NODEH
#define NODEH
 struct NODE {
     int data;
     struct NODE *name;
 };
#endif
```

Lib.h

```
#include “Node.h”
 void addNode(struct NODE * ptr) {
                 :
                 :
 }
```

Often the developer who uses a library does not know all the inner workings of the library (or shared file) and can easily cause a situation where they include a file twice. #ifndef can help.

```
#include <stdio.h>
#include “Node.h”
#include “Lib.h”

 void main() {
      :
      :
 }
```

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Part B

GNU Tools

Readings: chapter 4, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/ and https://sourceware.org/binutils/docs/gprof/

What is GNU?

GNU = Gnu is Not Unix

- Open source project to create a Unix-like but free environment
- It contains a Unix-like operating system: HURD kernel
- It contains many development tools

In the next two lectures we will look at development tools

- Some based on GNU
- Others not based on GNU

The make Tool

Modular programming:

- Great for team programming
- Fast way to compile large projects

The GNU make Tool helps us compile complicated projects.

The make Tool can also do other software engineering activities, like backups, etc.

The make Tool

Composed of:

- The **make** program
- The **makefile** text file

Developers define their software engineering rules within the text file named **makefile**.

The **make** program uses that text file.

If your favorite IDE uses the term Project, then inside somewhere is a makefile and a make program.

Basic makefile Text File

```
program: file1.o file2.o
  gcc –o program file1.o file2.o
file1.o: file1.c
  gcc –c file1.c
file2.o: file2.c
  gcc –c file2.c
backup:
  cp *.c /home/project/backup
```

This means:

- gcc –c file1.c file2.c
- gcc –o program file1.o file2.o

It can also mean:

- cp *.c /home/project/backup

Some power exists in selectively executing portions of the makefile.

Basic makefile Text File

```
program: file1.o file2.o
  gcc –o program file1.o file2.o
file1.o: file1.c
  gcc –c file1.c
file2.o: file2.c
  gcc –c file2.c
backup:
  cp *.c /home/project/backup
```

Method 1:

- Bash-prompt $ make

This invokes the make program.

It assumes a makefile exists in the same directory and executes the first line. In our example: program: file1.o file2.o

Notice the recursive definition.

Basic makefile Text File

```
program: file1.o file2.o
  gcc –o program file1.o file2.o
file1.o: file1.c
  gcc –c file1.c
file2.o: file2.c
  gcc –c file2.c
backup:
  cp *.c /home/project/backup
```

Method 2:

- Bash-prompt $ make backup

This invokes the make program and specifically targets a portion of the makefile.

It assumes a makefile exists in the same directory and in our example executes: cp *.c /home/project/backup

Notice it executes like Bash.

Demo

Creating the make file for a two file non-recursive factorial program:

- Main.c with the main function
- Factorial.c with the factorial function

gprof
Optimizing for Speed

Unacceptably Slow

- An application that is unacceptably slow is an application that takes longer to complete an operation than desired.
- Thus, the application needs to be <u>optimized</u>.
  - The code of the application needs to be improved so the application can accomplish the same tasks with less resources.
  - The <u>behaviour</u> of the application should remain unchanged.
- Before optimizing, we need to <u>identify</u> the portion of code that is slow.

Definitions of slow

Big Oh

- Logical speed of the algorithm

Clock Time

- Actual speed as related to computer hardware MHz, RAM, Devices

Advantages and Problems

- Big Oh:
  - Advantage: true speed independent from hardware
  - Problems: long to do on large pieces of code
- Clock:
  - Advantages: can be automated
  - Problems: get better CPU, program runs faster

Profilers

- Profilers are dynamic performance analysis tools
  - As opposed to lint which are static analysis tools.
- They record data as the application executes
- This data is often used to determine which part of an application needs optimization.

Using GPROF

```
$ gcc –pg file.c
```

(creates a gmon.out binary file containing information about the architecture of the program)

```
$ gprof –b a.out gmon.out > textfile
$ gprof -b a.out gmon.out < input > output
```

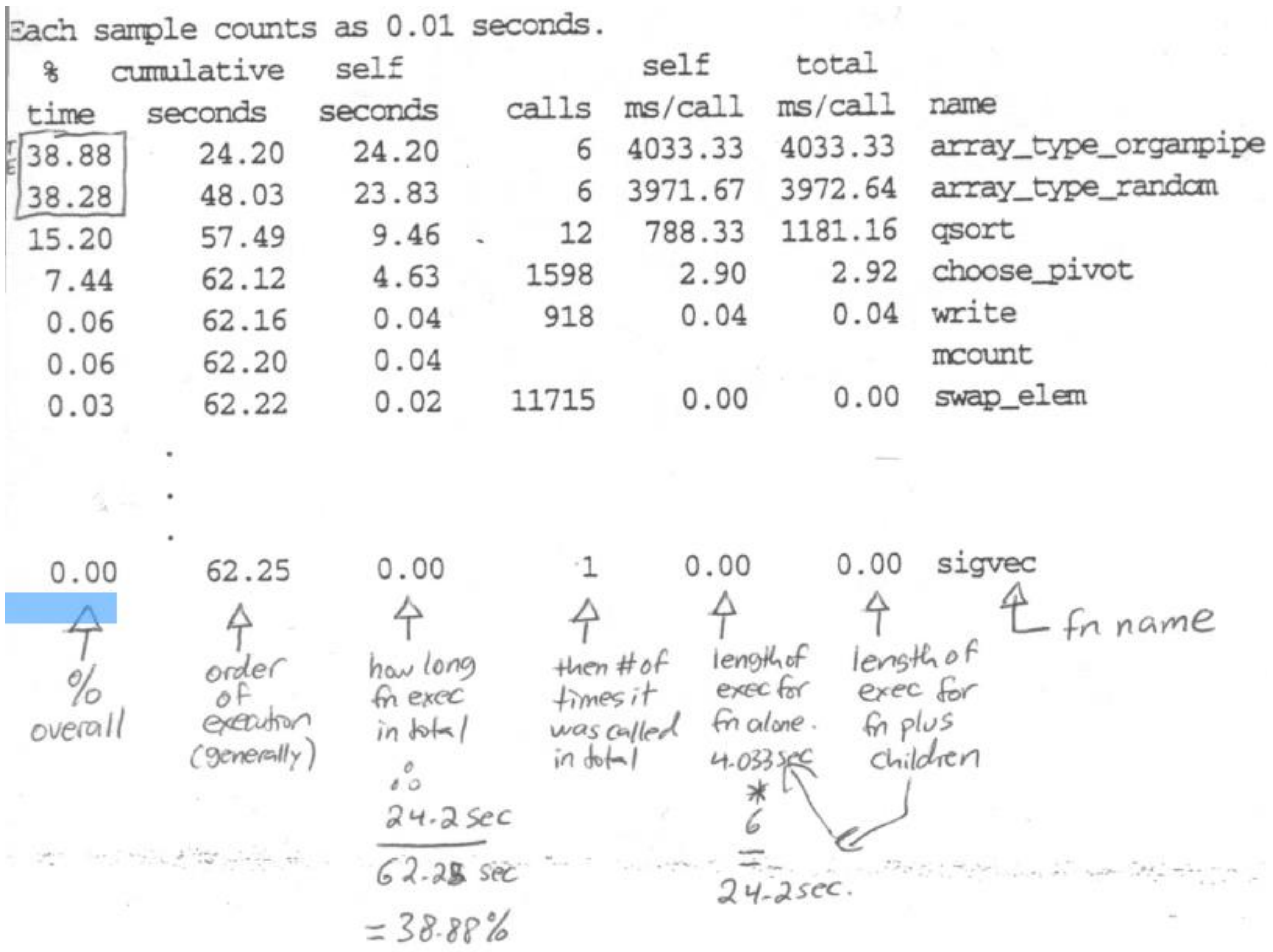
Options:

-b	not verbose
-s	merge all gmon files
-z	table of functions never called

The Flat Profile

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 38.88     24.20    24.20        6  4033.33  4033.33  array_type_organpipe
 38.28     48.03    23.83        6  3971.67  3972.64  array_type_random
 15.20     57.49     9.46       12   788.33  1181.16  qsort
  7.44     62.12     4.63     1598     2.90     2.92  choose_pivot
  0.06     62.16     0.04      918     0.04     0.04  write
  0.06     62.20     0.04                             mcount
  0.03     62.22     0.02    11715     0.00     0.00  swap_elem
              .
              .
              .
  0.00     62.25     0.00        1     0.00     0.00  sigvec
```

% overall

order of execution (generally)

how long fn exec in total
∴
$\frac{24.2 \text{ sec}}{62.25 \text{ sec}} = 38.88\%$

then # of times it was called in total

length of exec for fn alone. 4.033 sec
$* 6 = 24.2$ sec.

length of exec for fn plus children

fn name

% time
This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.

cumulative seconds
This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

self seconds
This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.

calls
This is the total number of times the function was called, blank otherwise.

self ms/call
This represents the average number of milliseconds spent in this function per call.

total ms/call
Average number of milliseconds spent in this function and its descendants per call.

name
This is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds field is sorted.

The Call Graph

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.92                 main [1]
                0.06    0.85       1/1           test [2]
                0.01    0.00       1/1           some_other_test [5]
-----------------------------------------------
                0.06    0.85       1/1           main [1]
[2]     98.9    0.06    0.85       1         test [2]
                0.00    0.85       1/1           another_test [3]
-----------------------------------------------
                0.00    0.85       1/1           test [2]
[3]     92.3    0.00    0.85       1         another_test [3]
                0.85    0.00       1/1           yet_another_test [4]
-----------------------------------------------
                0.85    0.00       1/1           another_test [3]
[4]     92.3    0.85    0.00       1         yet_another_test [4]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[5]      1.1    0.01    0.00       1         some_other_test [5]
-----------------------------------------------
```

Unix
Bash
C
GNU
Systems

The Primary Line

The line that describes the function which the entry is about.

```
index  % time    self  children called     name
[3]       100.0    0.00    0.05       1         report [3]
```

index
Entries are numbered with consecutive integers, for cross-referencing.

% time
This is the percentage of the total time that was spent in this function, including time spent in subroutines called from this function. Cannot be totalled.

self
This is the total amount of time spent in this function. This should be identical to the number printed in the seconds field for this function in the flat profile.

children
This is the total amount of time spent in calls made by this function.

called
This is the number of times the function was called.

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Function's Callers

A function's entry has a line for each function it was called by.

```
index  % time    self  children called     name
...
                  0.00    0.05       1/1          main [2]
[3]      100.0   0.00    0.05       1         report [3]
```

self
An estimate of the amount of time spent when it was called from main.

children
An estimate of the amount of time spent in subroutines when called from main.

called
Two numbers: the number of times was called from main, followed by the total number of non-recursive calls from all its callers.

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Function's Children

A line for each other function that it called.

```
index  % time    self  children called     name
[2]       100.0    0.00    0.05       1         main [2]
                   0.00    0.05       1/1           report [3]
```

self
An estimate of the amount of time spent directly when called from main.

children
An estimate of the amount of time spent in subroutines of report when report was called from main.

called
Two numbers, the number of calls to report from main followed by the total number of non-recursive calls to report.

Demo

- First: non-recursive factorial
- Then: recursive factorial

GDB

Readings: http://www.tutorialspoint.com/gnu_debugger/

Bug

A software bug is an error, flaw, mistake, failure, or fault in a computer program that prevents it from working as intended, producing an incorrect result.

- Bugs can exist at different levels
  - Design
  - Source Code
- Bugs have severities
  - Some bugs produce an incorrect answer
  - Some bugs simply crash an application.
  - Some bugs cause loss of data.
  - Some bugs cause loss of money.
- The worst bugs cause loss of life.

Well known bugs

- Y2K – date overflow
- Ariane 5 Flight 501 – conversion overflow
- MIM-104 Patriot bug – clock drift
- MARS orbiter crash - metric and imperial

Unix
Bash
C
GNU
Systems

Debugging

Debugging is the act of finding the source of a bug and fixing it.

- The hardest part of debugging is finding the problem.
  - This becomes exponentially difficult when the source is very large.
- Just analyzing the code is not always enough to find bugs.
  - You need to run the application.
- Debuggers are tools that help the debugging process.

Debuggers

- Debuggers allow a programmer to run the application in a different mode.
- When running under this different mode ("debug mode"), many new features are available to the programmer.
  - If an application crashes, the programmer can see what line caused the fatal operation. He can also consult the content of the memory.
  - A programmer can analyse an application, line-by-line. At each line, he can consult the content of the memory.
- Most programming languages have a debugger.

To Printf or Not to Printf

- Printf is useful when debugging
  - How can we use it? Suggestions?
- A symbolic debugger can do things printf() can't.
  - halt the program temporarily,
  - list source code,
  - print the datatype of a variable
  - jump to an arbitrary line of code
- You can use a symbolic debugger ON a process that has already crashed and died.

Introduction to GDB

GDB is a debugger which is part of the Free Software Foundation's GNU operating system.

GDB can be used to debug:

- C, C++, Objective-C, Fortran, Java and Assembly programs.

Unix
Bash
C
GNU
Systems

COMP 206 – Joseph Vybihal
Software Systems

Using GDB

First:

- gcc -g -o HelloWorld HelloWorld.c
  - The program is then compiled with additional information such as the source code and symbol table.

This additional information is needed by the debugger.

Starting GDB

Second:

```
gdb HelloWorld
```

The executable

```
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
   Copyright 2003 Free Software Foundation, Inc.
   GDB is free software, covered by the GNU General
   Public License, and you are welcome to change it
   and/or distribute copies of it under certain
   conditions. Type "show copying" to see the
   conditions. There is absolutely no warranty for
   GDB. Type "show warranty" for details. This GDB
   was configured as "i386-redhat-linux-gnu"...

(gdb)
```

The command line

Getting Help

Type Help:

```
(gdb) help
```

You can also get help on a specific command by typing "help" and the name of that command:

```
(gdb) help breakpoints
 Making program stop at certain points.
```

Looking at the Source code

You can use the *list* command:

- List
- List filename
- List linenumber
- List function

```
(gdb) list main.c:10
5
6         int main (int argc, char **argv) {
7
8                 library* mylibrary = createLibrary(20);
9
10                loadLibrary("lib.txt", mylibrary);
11
12                addBookToLibrary(mylibrary, createBook("Lotr", "Tolkien", 300));
13                addBookToLibrary(mylibrary, createBook("Harry_Potter", "Rowing",
    50));
14                addBookToLibrary(mylibrary, createBook("C_Prog", "Kerning", 100));
```

You can change the size of a list using the *set listsize*

GNU Debugger with Core Dump

```
$ gcc -g file.c
$ gdb a.out core-dump
  GDB is a free software and you are welcome
   to distribute copies of it under centerain conditions;
   GDB 4.15.2-96q3; Copywrite 2000 Free Software Foundation,
   Inc.

   Program terminated with signal 7, Emulation trap.
   #0 0x2734 in swap (l=93643, u=93864, strat=1) at file.c:110
   110       x=y;

   (gdb) run
```

The run-time error message (crash)

Some Terminology

```
Bash-prompt $ gdb a.out
(gdb) break 4
(gdb) run                    ← run until end, crash, or breakpoint

int main() {
  int age;

  printf("Enter age: ");
  scanf("%d", &age);                                   Breakpoint   Stop & prompt
                                                                      Step & inspect
  if (age > 17)                                                       Step & inspect
          printf("Welcome\n");                                      Continue
  else
          printf("Try again when you are older\n");

  return 0;
}
```

How to use GDB

- Run you program to see error in Bash
- Use printf to solve problem – not good?
- Take notice to where the problem happened
- Bash-Prompt $ gdb a.out
- (gdb) break before-problem-location
- (gdb) step and print through code to see
- (gdb) change variables, continue or run fn

Step-by-Step

STEP & NEXT

- STEP
  - Step into
  - Go to next line of program. If the next line is a function call then enter into the function.
- NEXT
  - Skip next
  - Go to next line of program. If the next line is a function call then do not go into the function just execute the function in its entirety. Go to the next line after the function call.

Unix
Bash
C
GNU
Systems

GDB Command-line Commands

•Quit	end gdb
•List	show 10 lines
•List n,m	show lines n to m
•List function	show all of a function by name
•Run	run your program
•Run (later ctrl-c)	run, then interrupt program
•Run –b < invals > outvals	redirect input and output to program
•Backtrace	see the run-time stack (call stack)
•Whatis x	show x’s declaration
•Print x	show value stored at x
•Print fn(y)	execution fn with y as parameter
•Print a @ length	show “length” elements of array a

GDB Command-line Commands

•break LINE_NO	interrupt program at line number
•break FUNC_NAME	interrupt program at function call
•break LINE_NO if EXPR	interrupt at line number if expr true
•break FUNC_NAME if EXPR	interrupt at fn call if expr true
•break FILE_NAME:LINE_NO	interrupt at line number is source-file file
•continue	continue program execution after break
•watch EXPR	stop program as soon as expr is true
•set variable NAME = VALUE	change contents of a variable
•ptype NAME	pretty print of structure n
•call fn(y)	execute fn with parameter y

GDB Printing Data and History

•(gdb) whatis p
type = int *

•(gdb) print p
$1 = (int *) 0xf8000000

•(gdb) print *p
$2 = Cannot access memory at address 0xf8000000

•(gdb) print $1-1
$3 = (int *) oxf7fffffc

•(gdb) print *$3
$4 = 0

GDB Breakpoint Management

```
•(gdb) break 17
Breakpoint 1 at 0x2929: file.c, line 17.

•(gdb) break 30 if x == 100
Breakpoint 2 at 0x3550: file.c, line 30.

•(gdb) info breakpoints
Num  Type        Disp Enabled  Address  What
1    breakpoint  Keep    Y     0x2929   in calc at file.c: 17
2    breakpoint  Keep    Y     0x3550   in sum at file.c: 30

•(gdb) delete 1
•(gdb) delete         ← everything!
•(gdb) clear 17       ← any break or watch on line 17
•(gdb) disable n      ← do not delete but turn off
•(gdb) enable n
•(gdb) enable once n  ← turn on for one time
```

Unix
Bash
C
GNU
Systems

Where am I?

At any moment, even after a crash, you can use the *where* command to produce a backtrace. (stacktrace)

```
(gdb) where

#0  createBook (title=0x804a218 "Lotr", author=0x804a110
   "Tolkien",
    pages=300) at book.c:8
#1  0x080487fe in loadLibrary (filename=0x8048aa8 "lib.txt",
    myLibrary=0x804a008) at file.c:20
#2  0x08048567 in main () at main.c:10

(gdb)
```

Demo

GDB of non-recursive factorial debug session

Part C

Code Repositories (GIT)

Readings: https://www.learnenough.com/git-tutorial?gclid=Cj0KEQjw8tbHBRC6rLS024qYjtEBEiQA7wIDeXo_owx61WwsbBiFPTFsgvcLriD526qlAmford4B1IgaAk9j8P8HAQ

Backup

A folder that contains a copy of your file.

Often the copy is older than the original file because as you develop you modify the original file.

```
Bash-prompt $ vi file.c
Bash-prompt $ cp file.c backup
Bash-prompt $ vi file.c
```

If something bad happens to your original file you can always revert back to your backup copy.

Repository

A database containing all the versions of your file (version control system)

This is an improvement over a backup because a repository contains the entire history of all your backup files, also:

- It can be shared
- You can assign permissions
- You can enforce development and deployment rules

Repository History

$$\begin{tikzcd}
\text{File1.c version 1} \arrow[d] & \leftarrow \text{Known as the } \textbf{root} \text{ version} \\
\text{File1.c version 2} \arrow[d] & \\
\text{File1.c version 3} \arrow[d] & \\
\text{File1.c version 4} \arrow[d] & \\
\text{File1.c version 5} & \leftarrow \text{Known as the } \textbf{tip} \text{ version}
\end{tikzcd}$$

The files in your working directory are known as the **current** version

Git

A popular version control system

Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Its current maintainer since 2005 is Junio Hamano.

How to use basic Git

First – create the folder for your project

```
Bash-prompt $ mkdir project
```

Second – put/create the files for the project

```
Bash-prompt $ cd project
Bash-prompt $ cp ../afile          (copy from another dir)
Bash-prompt $ vi anotherfile       (you create a new file)
```

Third – create the Git repository

```
Bash-prompt $ git init
```

How to use basic Git

Fourth – Select the files to put in the repo

```
Bash-prompt $ git add file1 file2
```

Fifth – Now put the selected files into the repo

```
Bash-prompt $ git commit –m “message”
```

- It is very important to add a message describing the changes you made since the last commit. In the future if you will want to undo what you did these messages will help you figure out how far back you may want to go.

Sixth – Done

- This is the basic loop.
- You can now continue working on your files until the next commit, where you will repeat steps 4 and 5.

How to get something back

There are two cases where you may want to undo your work:

- You did a commit but you want to undo it
  - Bash-prompt $ git checkout -f
- You made an error in your original file that you cannot correct, so you want to go back to a previous version
  - Bash-prompt $ git log ← to see commit history, find <SHA>
  - Bash-prompt $ git checkout <SHA>

Which files to commit

There are two cases:

- You were not paying attention to the files you changed and now you would like to commit
  - Bash-prompt $ git diff ← all tip files compared with current
- Your team member worked on the file at the same time as you!!
  - Using Bash:
    - Bash-prompt $ diff yourfile friendfile
    - Bash-prompt $ vi files and copy past, then use git add + git commit
  - Using Git:
    - Bash-prompt $ git diff filename ← compares tip file with current file

Important

Git tries its best to merge different versions of a file automatically by itself.

It will resort to showing you the diff of two or more files when it is having trouble merging.

How to make & use a shared Git

Step 1 – create project folder & cd into it

Step 2:

- Start your own shared Git
  - Bash-prompt $ git init –bare ← makes it more compatible
  - Share your URL with friends
- Join an existing shared Git
  - Bash-prompt $ git clone <url>

The <url> is:

- ssh://user@server/path/folder
- http://user@server/path/folder

Often you will need to ask the system administrator for the URL info/permission.

How to make & use a shared Git

Step 3 – Before you work on any file, make sure that no one else has remotely changed the file

```
Bash-prompt $ git pull <url>
```

Step 4 – Now vi your files

Step 5 – Commit to the shared repo

```
Bash-prompt $ git add <filename>
Bash-prompt $ git commit –m "message"
Bash-prompt $ git push <url>
```

How to work in a team

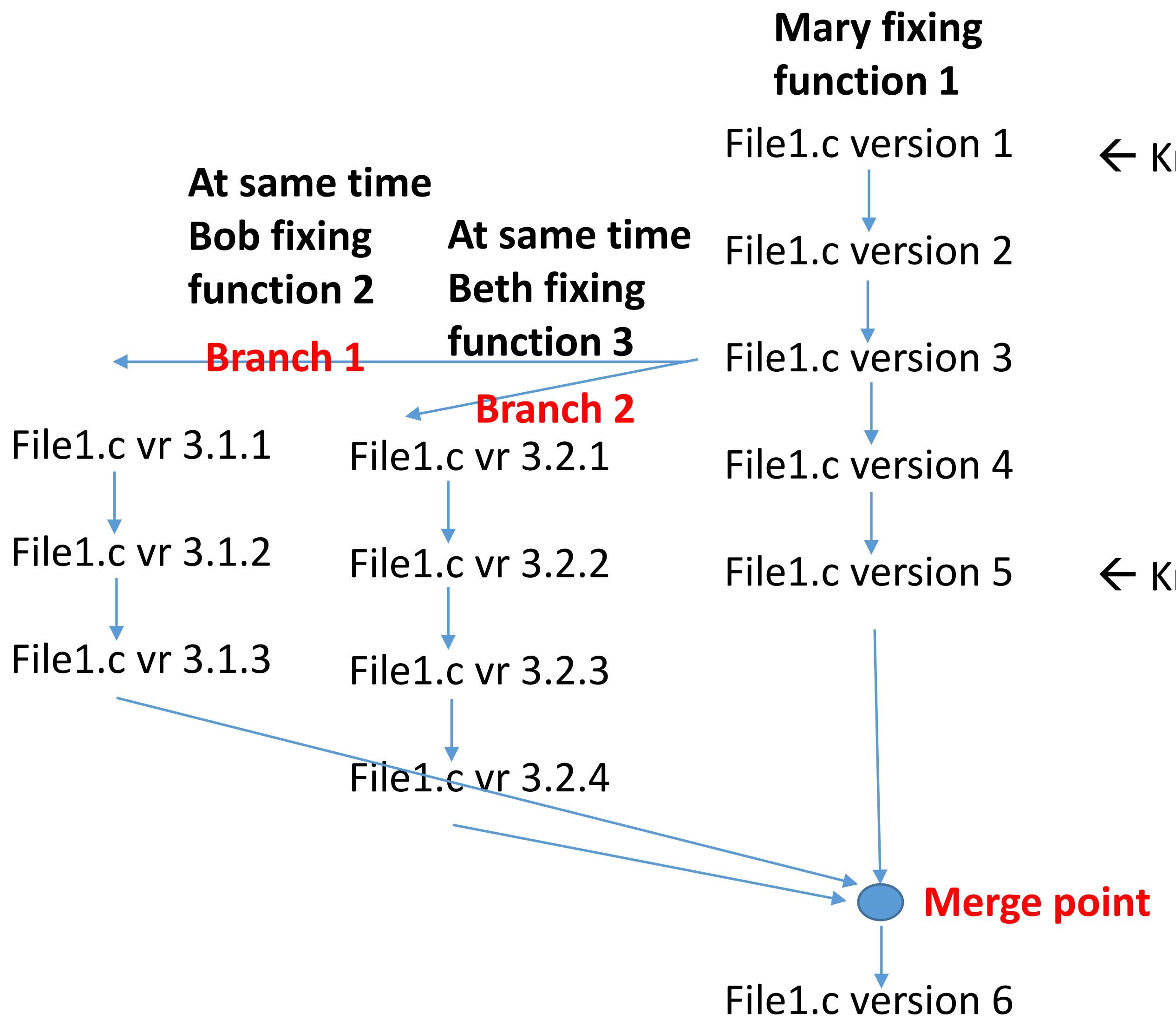
Option 1:

- Assign a source file to a developer.
- No other developer is permitted to edit that file.
- Option 1 can be carried out with the git commands we have already seen.

Option 2:

- Anyone can edit any file.
- Divide the work into branches.
- Option 2 requires us to understand branching.

Repository History



← Known as the **root** version

← Known as the **tip** version

Uses:
- Auto merging, and
- Diff

To sort things out.

How to use branching

Step 1 – See the current branches

Bash-prompt $ git branch

Step 2:

- Join a branch, or
  - Bash-prompt $ git checkout <branch_name>
- Create your own branch
  - Bash-prompt $ git branch <new_branch_name>
  - Bash-prompt $ git checkout <new_branch_name>
  - Or do it in one shot:
    - Bash-prompt $ git checkout –b <new_branch_name>

Step 3 – now whatever you do it will effect only the branch

How to use branching

Step 5 – To exit a branch

```
Bash-prompt $ git checkout master
```

Step 6 – To merge a branch with the master

```
Bash-prompt $ git checkout master
Bash-prompt $ git merge <branch_name>
```

Step 7 – Manager branches

- Delete merged branch
  - Bash-prompt $ git branch -d <branch_name>
- Delete a branch that has not been merged
  - Bash-prompt $ git branch -D <branch_name>

Demo

Using GIT locally

Using GIT with an open source project