Debuggers

A debugger gives you control of program execution:

- normal execution (`run`, `cont`)
- stop at a certain point (`break`)
- one statement at a time (`step`, `next`)
- examine program state (`print`)

The gdb debugger

gdb – a line-based interactive debugger.
ddd – an X-windows-based interface for gdb.
To use programs with gdb (or ddd), they must be compiled with the gcc compiler.
gdb (ddd) takes two arguments:

```
% gdb executable
```

E.g.

```
% gdb a.out [code]
```

(The core argument is used if you have a core image because the operating system terminated the program)

gdb sessions

A *session* with gdb is a sequence of commands to control and observe the executable.

```
$ gcc -Wall -g -o prog prog.c
$ gdb prog
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licen
This is free software: you are free to change and redistribute i
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
(gdb) break f
Breakpoint 1 at 0x1082c: file prog.c, line 32.
(gdb) run
```

gdb sessions

```
Starting program: ..../prog
Enter a b c: 1 2 3
Breakpoint 1, f (i=1, j=2) at prog.c:32
32                  a = i + j;
(gdb) next
33                  b = i*i + j*j;
(gdb) next
34                  return a*b;
(gdb) print a
$1 = 3
(gdb) print b
$2 = 5
(gdb) cont
...
```

Basic gdb commands

- `quit` – quits from gdb
- `help [CMD]` – on-line help

Gives information about `CMD` command.

- `run ARGS` – run the program

`ARGS` are whatever you normally use, e.g.

```
% ./prog < data
```

is achieved by:

```
(gdb) run < data
```

gdb status commands

- `where` – stack trace

Find which function the program was executing when it crashed.
Stack may also have references to system error-handling functions.

- `up [N]` – move "up" the stack

Allows you to skip to "scope" of a particular function in stack.

- `list [LINE]` — show code

Displays five lines either side of current statement.

- `print EXPR` – display expression values

`EXPR` may use (current values of) variables.
Special expression `aat1` shows all of the array `a`.

gdb execution commands

- `break [PROC|LINE]` - set break-point

On entry to function `PROC` (or reaching line `LINE`), stop execution and return control to gdb.

- `next` - single step (over functions)

Execute next statement; if statement is a function call, execute entire function body.

- `step` - single step (into functions)

Execute next statement; if statement is a function call, go to first statement in function body.

For more details see gdb's on-line help.

Version Control

Any large, useful software system ...

- will undergo many changes in its lifetime
- multiple programmers making changes
- who may work on the code concurrently and independently

The process of code change needs to be managed so that

- changes produce "consistent" versions of the system
- many programmers can easily work simultaneously
- we can roll back to earlier version if needed
- documentation of when, who, & why changes made
- multiple versions of system can be distributed, tested, merged

Version Control

Consider the following simple scenario:

- a software system contains a source code file `x.c`
- system is worked on by several teams of programmers
- Ann in Sydney adds a new feature in her copy of `x.c`
- Bob in Singapore fixes a bug in his copy of `x.c`

Ultimately, we need to ensure that

- all changes are properly recorded (when, who, why)
- both the new feature and the bug fix are in the next release
- if we later find bugs in old release, they can be fixed

Git

- distributed version control system - multiple repositories, no "master"
- every user has their own repository
- created by Linux Torvalds for Linux kernel
- not better than competitors but better supported/more widely used (e.g. github/bitbucket)
- at first stick with a small subset of commands
- substantial time investment to learn to use Git's full power

Creating a git Repository

- Create repository **git init**
- Copy exiting repository **git clone**

Git uses the sub-directory `.git` to store a local repository.
Inside `.git` is stored all versions of all files under version control (in clever efficient way).
If intereteste dtsartby reading about how git uses SHA-1 hashes.

Tracking a Project with Git

- Project must be in single directory tree.
- Usually don't want to track all files in directory tree
- Don't track binaries, derived files, temporary files, large static files
- Use **.gitignore** files to indicate files never want to track
- Use **git add** *file* to indicate you want to track *file*
- Careful: **git add** *directory* will every file in *file* and sub-directories

Git Commit

- A git commit is a snapshot of all the files in the project.
- Can return the project to this state using **git checkout**
- Beware if you accidentally add a file with confidential info to git - need to remove it from all commits.
- **git add** copies file to staging area for next commit
- **git commit -a** if you want commit current versions of all files being tracked

Git Push/Pull

- git **push** adds commits from your repository to a remote repository
- git **remote** lets you give names to other repositories
- git **pull** adds commits from a remote repository to your repository

Example: making Git Repository Public via Github

Github popular repo hosting site (see competitors e.g. bitbucket)
Github student accounts free for small number of repos
Github and competitors also let you setup collaborators, wiki, web pages, issue tracking
Web access to git repo e.g. https://github.com/mirrors/linux

Example: making Git Repository Public via Github

Its a week after the COMP1511 assignment was due and you want to publish your code to the world.
Create github account - assume you choose *ilove1511* as your login
Create a repository - assume you choose *my_code* for the repo name
Add your ssh key (`.ssh/id_rsa.pub`) to github (Account Settings - SSH Public Keys - Add another public key)

```
cd ~/ass2
git remote add origin git@github.com:ilove1511/my_code.git
git push -u origin master
```

Building Software Systems

Software systems need to be built / re-built

- during the development phase (change,compile,test,repeat)
- if distributed in source code form (assists portability)

Simple example: systems implemented as multi-module C programs:

- multiple object files (`.o` files) and libraries (`libxyz.a`)
- compiler that only recompiles what it's told to

Multi-module C Programs

Large C programs are implemented as a collection of `.c` and `.h` files.

A pair of `a.c` and `a.h` generally defines a *module*:

- `a.c` contains operations related to one particular kind of data/object
- `a.h` exports definitions for types/operations defined in `a.c`

Why partition a program into modules at all?

- as a principle of good design
- to share development work in a large project
- to create re-usable libraries

Example Multi-module C Program

Imagine a large game program with

- an internal model of the game world (places, objects, actors)
- code to manage graphics (mapping the world to the screen)

The graphics code would ...

- typically be separated from the world code (software eng)
- need to use the world operations (to render current scene)

Then there would be a main program to ...

- accept user commands, modify world, update display

Example Multi-module C Program

Building Large Programs

Building the example program is relatively simple:

```
$ gcc -c -g -Wall world.c
$ gcc -c -g -Wall graphics.c
$ gcc -c -g -Wall main.c
$ gcc -Wall -o game main.o world.o graphics.o
```

For larger systems, building is either

- inefficient (recompile everything after any change)
- error-prone (recompile just what's changed + dependents)
  - module relationships easy to overlook
    (e.g. `graphics.c` depends on a `typedef` in `world.h`)
  - you may not know when a module changes
    (e.g. you work on `graphics.c`, others work on `world.c`)

make

Classical tool to build software dating to '70s.
Many variants and alternatives but still useful & widely used.
More recent tools heavily used for particular languages.
E.g for Java maven or ant very popular.

make

`make` allows youto

- document intra-module dependencies
- automatically track of changes

`make` works from a file called `Makefile` (or `makefile`)
A `Makefile` contains a sequence of rules like:

```
target : source1 source2 ...
        commands to create target from sources
```

Beware: *each command is preceded by a single* tab *char.*
Take care using cut-and-paste with`Makefiles`

Dependencies

The `make` command is based on the notion of *dependencies*.
Each rule in a `Makefile` describes:

- dependencies between each target and its sources
- commands to build the target from its sources

`Make` decides that a target needs to be rebuilt if

- it is older than any of its sources (based on file modification times)

Example Makefile #1

A `Makefile` for the earlier example program:

```
CC=gcc
CFLAGS=-Wall -std-c11
    game : main.o graphics.o world.o
            $(CC) $(CFLAGS) -o game main.o graphics.o worl

    main.o : main.c graphics.h world.h
            $(CC) $(CFLAGS) -c main.c

    graphics.o : graphics.c world.h
            $(CC) $(CFLAGS) -c -g -Wall graphics.c

    world.o : world.c
            $(CC) $(CFLAGS) -c -g -Wall world.c
```

How make Works

The make command behaves as:

```
make(target, sources, command):
    # Stage 1
    FOR each S in sources DO
        rebuild S if it needs rebuilding
    END
    # Stage 2
    IF (no sources OR
        any source is newer than target) THEN
       run command to rebuild target
    END
```