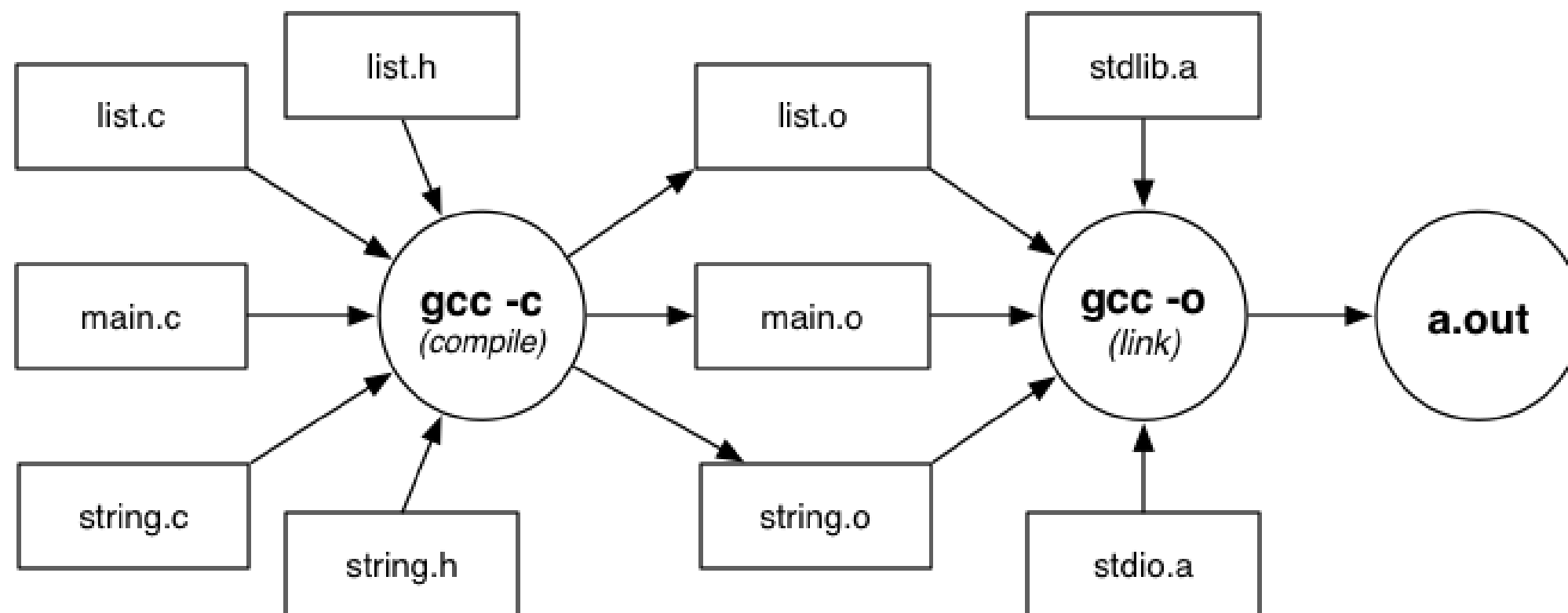


Compilation, Debugging and Makefiles

Computing 2 17x1

THE C COMPILER (GCC)

- applies source-to-source transformation (pre-processor)
- compiles *source code* to produce *object files*
- links object files and *libraries* to produce *executables*



COMPILATION AND LINKING WITH GCC

- `gcc -c list.c`
 - Produces `list.o` from `list.c` and `list.h`
- `gcc -c string.c`
 - Produces `string.o` from `string.c` and `string.h`
- `gcc -c main.c`
 - Produces `main.o` from `main.c`, `list.h`, `string.h`
- `gcc -o a.out main.o string.o list.o`
 - Links `main.o`, `string.o`, `list.o` and libraries to create an executable called `a.out`

GCC OPTIONS

- `gcc -v`
 - Displays detailed information about the exact sequence of commands used to compile and link a program
- `ulimit -v 4096`
 - Produces `string.o` from `string.c` and `string.h`
- `gcc -c main.c`
 - Produces `main.o` from `main.c`, `list.h`, `string.h`
- `gcc -o a.out main.o string.o list.o`
 - Links `main.o`, `string.o`, `list.o` and libraries to create an executable called `a.out`

DEBUGGING

- Initial versions of programs always have errors
- Symptoms of errors
 - Program quits with fatal error (eg segfault)
 - Program runs forever (infinite loop)
 - Program does not produce expected results
- Errors can be caused by
 - Misunderstanding programming language constructs
 - Misunderstanding the problem
 - Incorrect logic
 - Carelessness (uninitialized, off-by-one, pointers)

DEBUGGING

- Debugging: process of
 - Finding the location/s of incorrect code
 - Fixing incorrect code that causes error
- Debuggers: software tools that
 - Assist in the process of debugging
 - By allowing detailed observation of execution state
- Critical part of debugging
 - Narrowing focus to small region of large code/state

DEBUGGING

- Testing can help debugging
 - Test cases for boundary conditions (eg. Empty list)
 - Sequence of tests revealing
 - Trigger points .. ok before, fails after
 - Patterns of behaviour ... eg. Always one more than expected
- Use deduction to identify/explain patterns.
- In general: run more tests before resorting to debugger

GDB: THE GNU DEBUGGER

- **gdb** provides facilities to
 - Control execution of program
 - Step by step execution, breakpoints
 - View intermediate state of program
 - Values stored in program variables
- Plain **gdb** uses a command-line interface
- **ddd** provides a GUI wrapper around gdb.
- Must be compiled with `—gdwarf-2` option

BASIC GDB COMMANDS

- **quit**: quits from gdb
- **help** [CMD] : on-line help
- **run** ARGS: run the program
 - ARGS are whatever you normally use eg.
 - \$ xyz < data
 - Would be run in gdb like
 - (gdb) run < data

BASIC GDB 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 down the stack
 - Allows you to skip to scope of a particular function
- **list** [LINE]: show code
 - Displays five lines either side of current statement
- **print** EXPR: display expression values
 - EXPR may use (current values of) variables

GDB EXECUTION COMMANDS

- **break** [FUNC | LINE] : set break-point
 - Stop execution and return control to gdb on entry to function FUNC or on reaching line LINE
- **next**: single step (over functions)
 - execute next statement
 - if the statement is a function call, execute the whole function
- **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

EXERCISE: USING GDB TO FIND ERRORS

- What is the output of running this code?

```
int foo (int *p);  
int main (void)  
{   int *p = 0; /* null pointer */  
    return foo (p);  
}  
int foo (int *p)  {  
    int y = *p; return y;  
}
```

MAKEFILES

- **Make** is a software configuration tool that
 - specifies dependencies between software components
 - controls compilation when source code is updated
 - produces "minimal required recompilation" on update
- In fact, it can be used for any task which involves
 - multiple inter-dependent files
 - need to produce some files from others

MAKEFILES...

- **make** is driven by dependencies given in a **Makefile**

- A *dependency* specifies

target : source₁ source₂ ...

commands to build target from sources

- e.g.

```
eval: eval.o tokens.o stack1.o
```

```
gcc -o eval eval.o tokens.o stack1.o
```

- Rule: *target* is rebuilt if older than any *source_i*

EXAMPLE MAKEFILE

```
game : main.o list.o string.o
```

```
    gcc -o game main.o list.o string.o -lm
```

```
main.o : main.c list.h string.h
```

```
    gcc -Wall -Werror -O -c main.c
```

```
list.o : list.c list.h
```

```
    gcc -Wall -Werror -O -c list.c
```

```
string.o : string.c
```

```
    gcc -Wall -Werror -O -c string.c
```

```
clean :
```

```
    rm -f *.o core
```

```
clobber : clean
```

```
    rm -f game
```

HOW MAKE WORKS

- The make command behaves as:

- make(target):

 - Find makefile rule for the target

 - for each S in Sources { make(S) }

 - if (no sources OR any source is newer than target){

 - perform Action to rebuild target

 - }

EXAMPLE MAKEFILE REVISITED

```
CC = gcc
```

```
CFLAGS = -Wall -Werror -O
```

```
game : main.o list.o string.o
```

```
    $(CC) -o game main.o list.o string.o
```

```
main.o : main.c list.h string.h
```

```
    $(CC) $(CFLAGS) -c main.c
```

```
list.o : list.c list.h
```

```
    $(CC) $(CFLAGS) -c list.c
```

Etc...

RUNNING MAKE

- To build the first target in the makefile just type
- **make**
- If make arguments are targets, build just those targets:
- **make world.o**
- **make clean**
- **make clobber**
- The -n option instructs make
 - to tell what it would do to create targets
 - but don't execute any of the commands