Recitation 3: Compilation Tools

COP3402 FALL 2015 – ARYA POURTABATABAIE – //2015

FROM EURIPIDES MONTAGNE, FALL 2014

The Compilation Process

The Compilation Process

You probably already know that **gcc** is the dominant compiler on UNIX-like environments, including Eustis.

You probably also know that **gcc -o hello hello.c** will compile "hello.c" into the program "hello".

What you may not know is that it can be used to examine several steps of the compilation process:

Input	Program	Output
Source code	Preprocessor	Expanded source code
Expanded source code	Compiler	Assembly language
Assembly language	Assembler	Object code
Object code	Linker	Executable code
Executable code	Loader	Execution

Simple Test Programs

NO LIBRARIES

```
int test_fun(int x){
  return x + 17;
}
int main(void){
  int x = 1;
  int y;
  y = test_fun(x);
  return 0;
}
```

WITH I/O LIBRARY

```
#include <stdio.h>

int main (void){
  printf ("Hello, world!\n");
  return 0;
}
```

Deconstructed Compilation

Step	Description	Command
Step 1	Preprocessing	`cpp hello.c hello.i`
Step 2	Compilation	`gcc -S hello.i -o hello.s`
Step 3	Assembly	`as hello.s -o hello.o`
Step 4	Linking*	`gcc hello.o`
Step 5	Execution	`./a.out`

*To get the full effect we really should do the linking with `ld`, but it’s far more annoying than just using gcc for this step.

Compilation Process: No-Library Results

Preprocessing

simple.c

```
int test_func(int x){
 return x + 17;
}

int main(void){
 int x = 1;
 int y;
 y = test_func(x);
 return 0;
}
```

simple.i

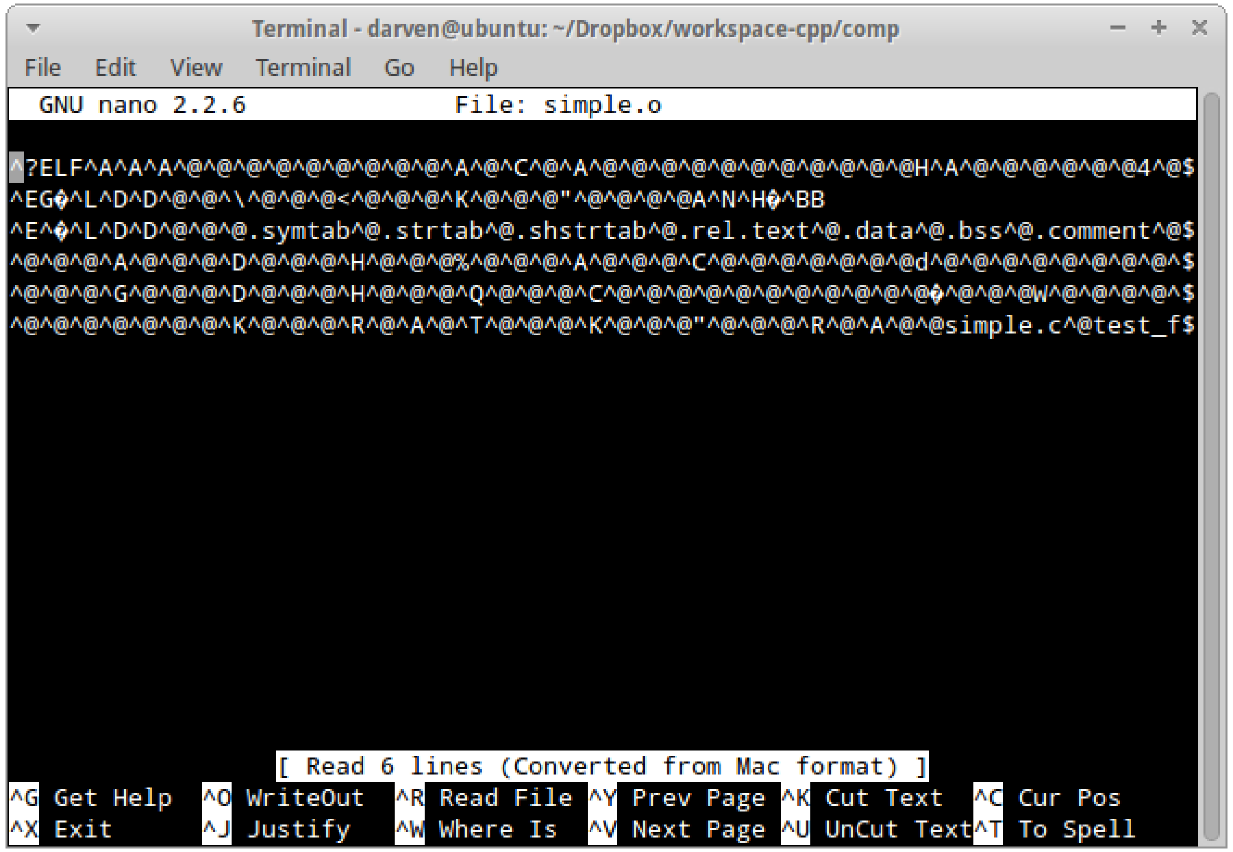
```
# 1 "simple.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "simple.c"
int test_func(int x){
 return x + 17;
}

int main(void){
 int x = 1;
 int y;
 y = test_func(x);
 return 0;
}
```

Compilation – simple.s

```
        .file   "simple.c"
        .text
        .globl  test_func
        .type   test_func, @function
test_func:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        movl    8(%ebp), %eax
        addl    $17, %eax
        popl    %ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   test_func, .-test_func
        .globl  main
        .type   main, @function
```

```
main:
.LFB1:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $20, %esp
        movl    $1, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, (%esp)
        call    test_func
        movl    %eax, -4(%ebp)
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE1:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-
2ubuntu1~12.04) 4.7.3"
        .section .note.GNU-stack,"",@progbits
```

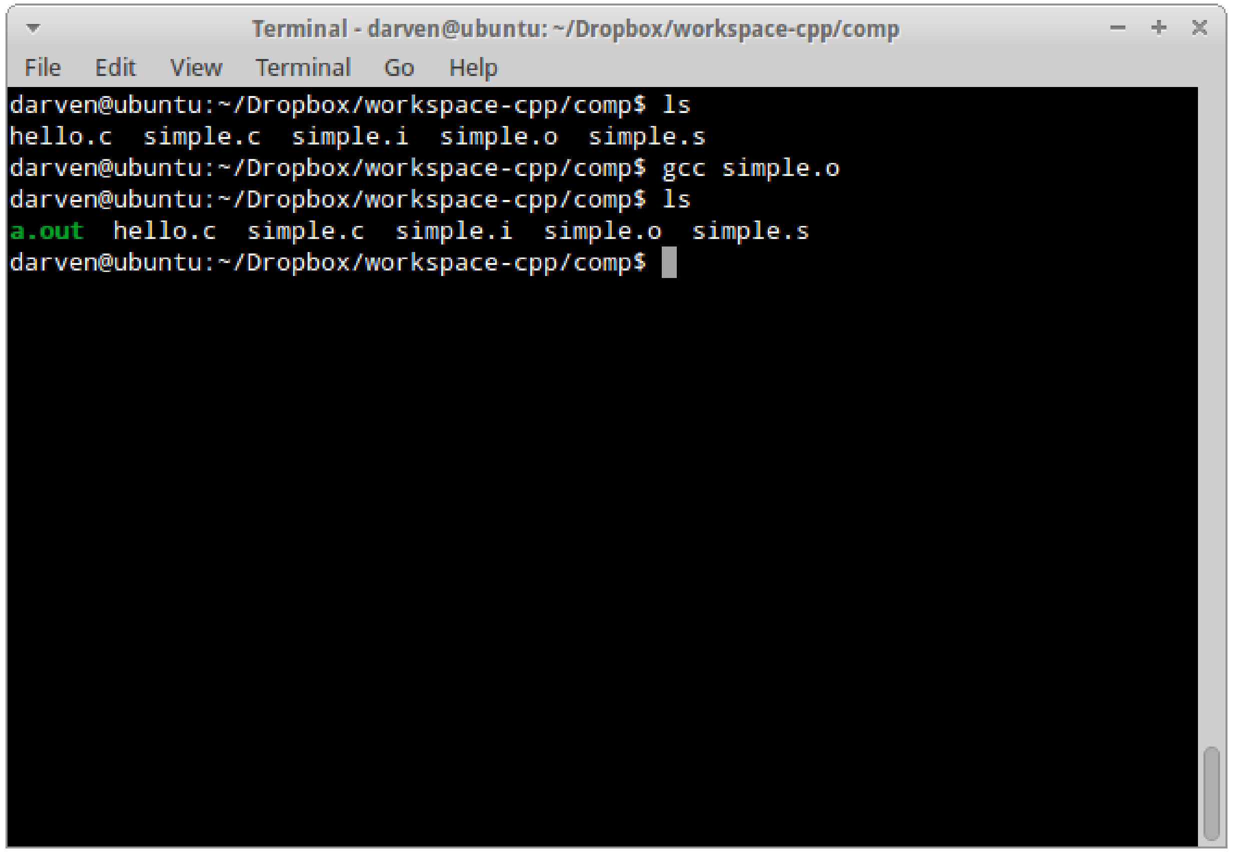
Assembly – simple.o


Terminal - darven@ubuntu: ~/Dropbox/workspace-cpp/comp
File Edit View Terminal Go Help

```
  GNU nano 2.2.6              File: simple.o

^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^A^@^C^@^A^@^@^@^@^@^@^@^@^@^@H^A^@^@^@^@^@^@4^@$
^EG�^L^D^D^@^@^\^@^@^@<^@^@^@^K^@^@^@"^@^@^@^@A^N^H�^BB
^E^�^L^D^D^@^@^@.symtab^@.strtab^@.shstrtab^@.rel.text^@.data^@.bss^@.comment^@$
^@^@^@^A^@^@^@^D^@^@^@^H^@^@^@%^@^@^@^A^@^@^@^C^@^@^@^@^@^@^@d^@^@^@^@^@^@^@^@^$
^@^@^@^G^@^@^@^D^@^@^@^H^@^@^@^Q^@^@^@^C^@^@^@^@^@^@^@^@^@^@^@�^@^@^@W^@^@^@^@^$
^@^@^@^@^@^@^@^K^@^@^@^R^@^A^@^T^@^@^@^K^@^@^@"^@^@^@^R^@^A^@^@simple.c^@test_f$




                     [ Read 6 lines (Converted from Mac format) ]
^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text^T To Spell
```

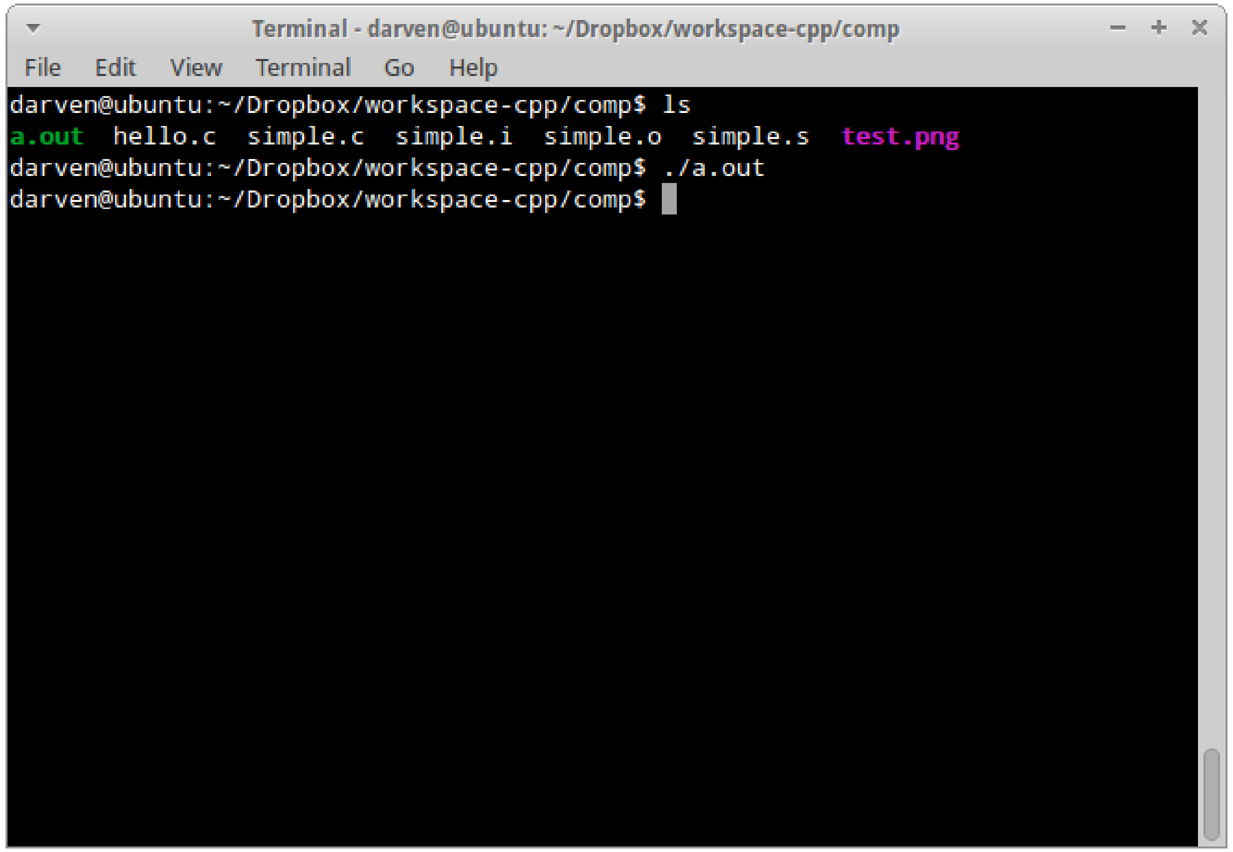
Linking - a.out

(You *can* use `gcc simple.o -o simple` instead, if you want)


Terminal - darven@ubuntu: ~/Dropbox/workspace-cpp/comp
File Edit View Terminal Go Help

```
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ls
hello.c  simple.c  simple.i  simple.o  simple.s
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ gcc simple.o
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ls
a.out  hello.c  simple.c  simple.i  simple.o  simple.s
darven@ubuntu:~/Dropbox/workspace-cpp/comp$
```

Execution - ./a.out


Terminal - darven@ubuntu: ~/Dropbox/workspace-cpp/comp
File Edit View Terminal Go Help

```
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ls
a.out  hello.c  simple.c  simple.i  simple.o  simple.s  test.png
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ./a.out
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ 
```

Compilation Process: With-Library Results

Preprocessing

(We trimmed **hello.i** so it would fit on the slide – it’s *really long*)

hello.c

```
#include <stdio.h>

int main(void){
      printf("Hello world!\n");
      return 0;
}
```

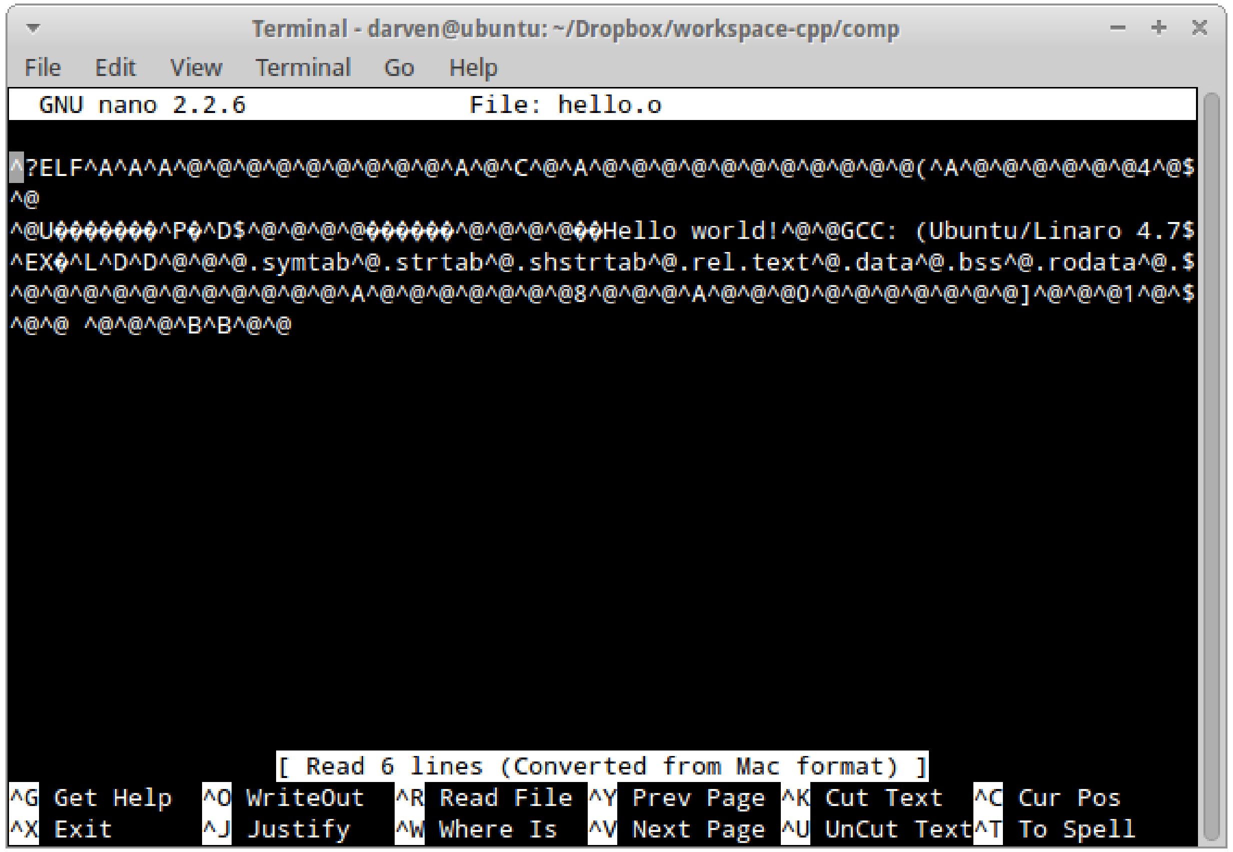
hello.i

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4

. . .

typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
. . .
typedef unsigned int _G_uint16_t __attribute__ ((__mode__
(__HI__)));
typedef unsigned int _G_uint32_t __attribute__ ((__mode__
(__SI__)));

. . .

# 940 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2

int main(void){
 printf("Hello world!\n");
 return 0;
}
```

Compilation – hello.s

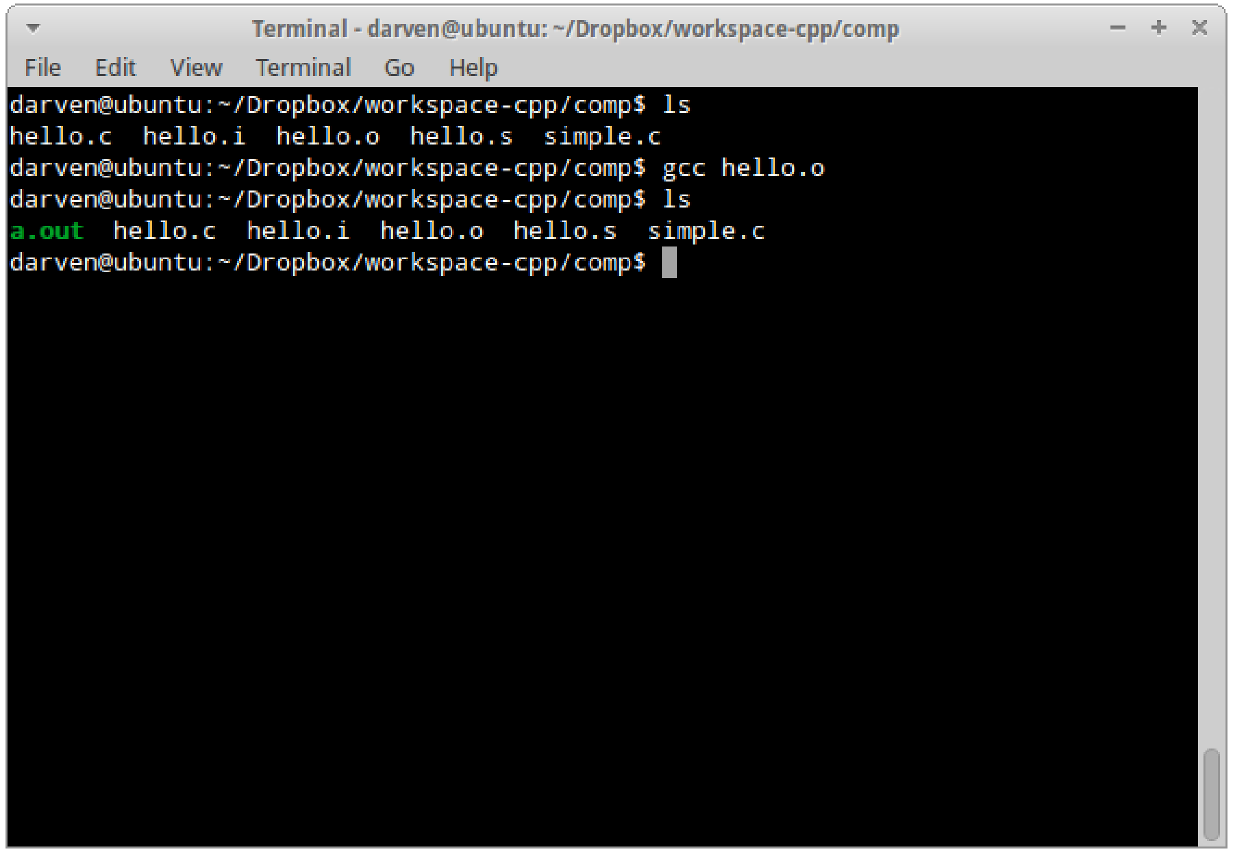
```
        .file   "hello.c"
        .section .rodata
.LC0:
        .string "Hello world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-
2ubuntu1~12.04) 4.7.3"
        .section .note.GNU-stack,"",@progbits
```

Assembly – hello.o


Terminal - darven@ubuntu: ~/Dropbox/workspace-cpp/comp
File Edit View Terminal Go Help

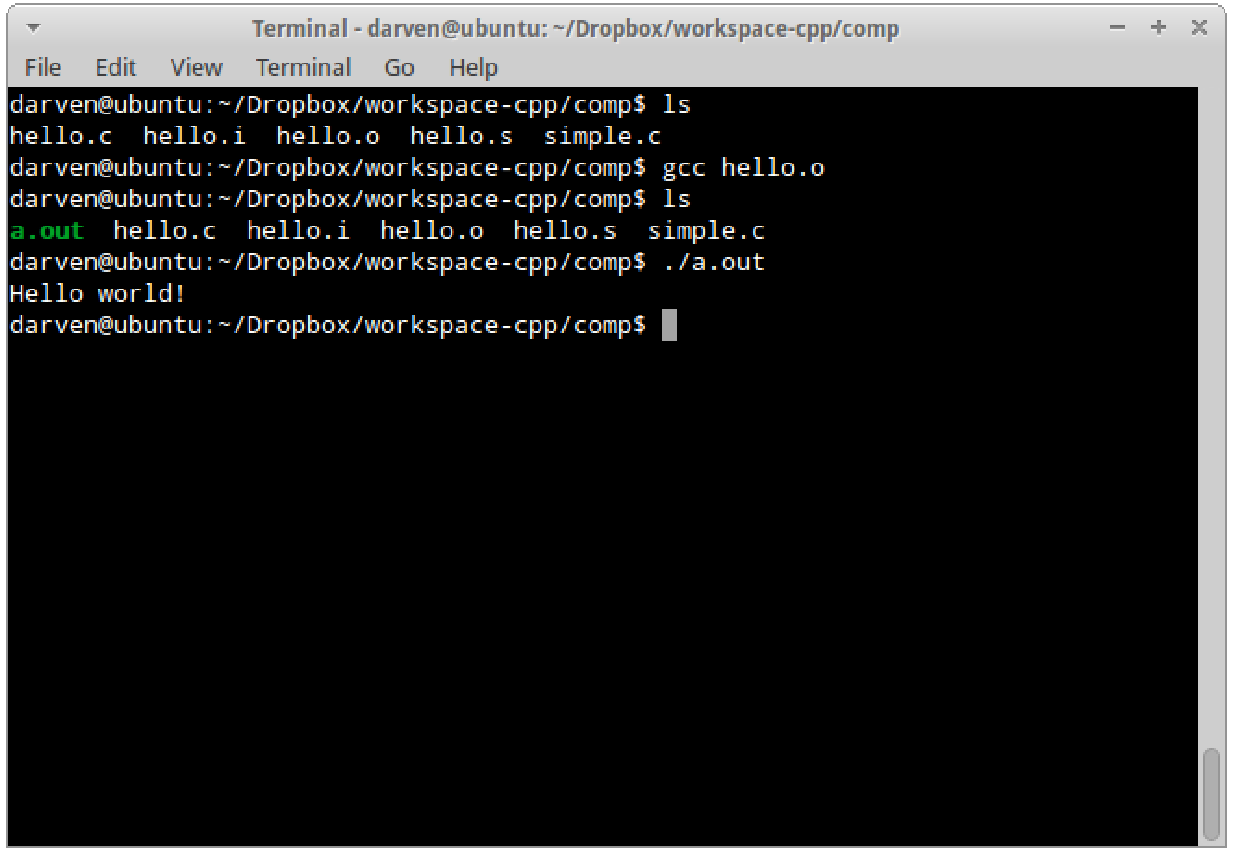
```
 GNU nano 2.2.6                File: hello.o

^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^A^@^C^@^A^@^@^@^@^@^@^@^@^@^@^@(^A^@^@^@^@^@^@4^@$
^@
^@U�������^P�^D$^@^@^@^@������^@^@^@^@��Hello world!^@^@GCC: (Ubuntu/Linaro 4.7$
^EX�^L^D^D^@^@^@.symtab^@.strtab^@.shstrtab^@.rel.text^@.data^@.bss^@.rodata^@.$
^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@8^@^@^@^A^@^@^@0^@^@^@^@^@^@^@]^@^@^@1^@^$
^@^@ ^@^@^@^B^B^@^@

                  [ Read 6 lines (Converted from Mac format) ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Linking - a.out


Terminal - darven@ubuntu: ~/Dropbox/workspace-cpp/comp
File Edit View Terminal Go Help

```
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ls
hello.c  hello.i  hello.o  hello.s  simple.c
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ gcc hello.o
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ls
a.out  hello.c  hello.i  hello.o  hello.s  simple.c
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ 
```

Execution - ./a.out


Terminal - darven@ubuntu: ~/Dropbox/workspace-cpp/comp
File Edit View Terminal Go Help

```
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ls
hello.c  hello.i  hello.o  hello.s  simple.c
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ gcc hello.o
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ls
a.out  hello.c  hello.i  hello.o  hello.s  simple.c
darven@ubuntu:~/Dropbox/workspace-cpp/comp$ ./a.out
Hello world!
darven@ubuntu:~/Dropbox/workspace-cpp/comp$
```

Makefiles

Makefiles

If you’ve worked with *projects* in any sort of integrated development environment, you already know what makefiles are – they’re the UNIX-like approach to exactly the same function.

- Small programs use a single file; it’s easy to just compile them
- Larger programs have many lines of code, and possibly multiple programmers
- Staying with one file is *not* the answer – large files are slow to compile, hard to maintain, and can only be worked on by one person at a time
- We need a way to maintain large numbers of source files in a single project

A **makefile** is a script that contains:

- The structure of a project – what files it contains, and which files depend on others
- Instructions for creating files that need to be created

A Project Structure

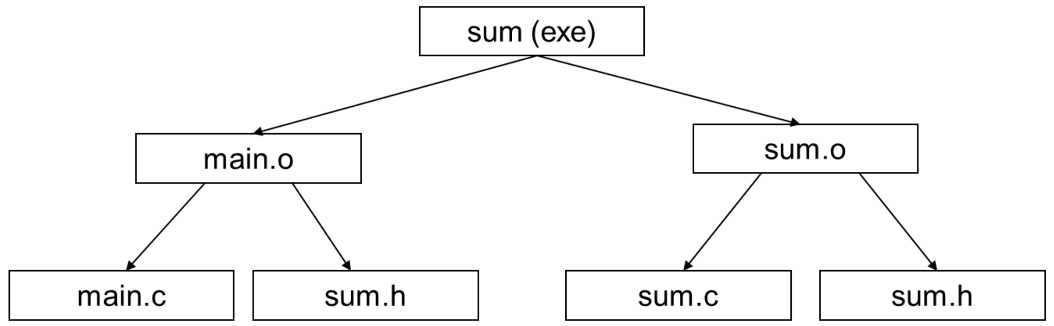
The structure of a project can be viewed as a directed acyclic graph.

- (If it couldn't, then your project couldn't be reliably built)

Let's look at a relatively simple example, with three files:

- **main.c, sum.c**, and **sum.h**

Both main.c and sum.c include sum.h, and we want the executable file to be named **sum**.



The Makefile

The syntax of a rule in a makefile is:

```
      target: dependencies
        action
```

So in this case, we’ve told **make** that:

- sum depends on main.o and sum.o, and to create sum, you should run gcc –o on them.
- main.o depends on main.c and sum.h, and to create main.o, you should run gcc –c on main.c.
- Analogous for sum.o.

```
sum: main.o sum.o
  gcc -o sum main.o sum.o

main.o: main.c sum.h
  gcc -c main.c

sum.o: sum.c sum.h
  gcc -c sum.c
```

What happens when we run **make**

- Make reads the makefile (classically by default, make looks for **Makefile**, with the capital M)
- Make constructs the project dependencies tree
- Unless the makefile or the command line tells it otherwise, make will try to create the target of the *first rule in the file*
- Make walks down the tree to see if there are any sub-targets that need to be recreated
  - A sub-target needs to be recreated if and only if it is *older than one or more of its dependencies*
- As lower-level targets are recreated, higher-level targets that depend on them will usually need to be recreated as well
  - The most typical manifestation of this: if you change anything, no matter what it is, you're probably going to be re-linking
- Dependencies that many targets share will force make to recreate all of those targets if they are changed
  - Change a core header file, and you're going to have to recompile everything

Make In Operation

File	Modified
`sum`	`10:03`
`main.o`	`09:56`
`sum.o`	`09:35`
`main.c`	`10:45`
`sum.c`	`09:14`
`sum.h`	`08:39`

Only main.o needs to be recompiled – sum.c and sum.h are unchanged. However, sum will need to be relinked. So make performs:

```
gcc –c main.c

gcc –o sum main.o sum.o
```

How Not to Write Makefiles

As in the last slide, make will always do the minimum re-compilation necessary – *if* you write the makefile correctly. Do you see what's wrong with the following?

```
prog: main.c sum1.c sum2.c
  gcc –o prog main.c sum1.c sum2.c
```

More on Makefiles

http://www.gnu.org/software/make/manual/make.html is the canonical reference for makefiles – and a good tutorial as well. It explains how to do many other things with makefiles, such as:

- Add variables to account for different builds
- Create generic rules so that, for example, every .o file doesn't need an explicit description of how to compile it from its corresponding .c file

Two words to the wise, however:

- **Never make your makefile more complicated than it needs to be**
- Makefiles are code – if they need to be complicated, **comment them!**