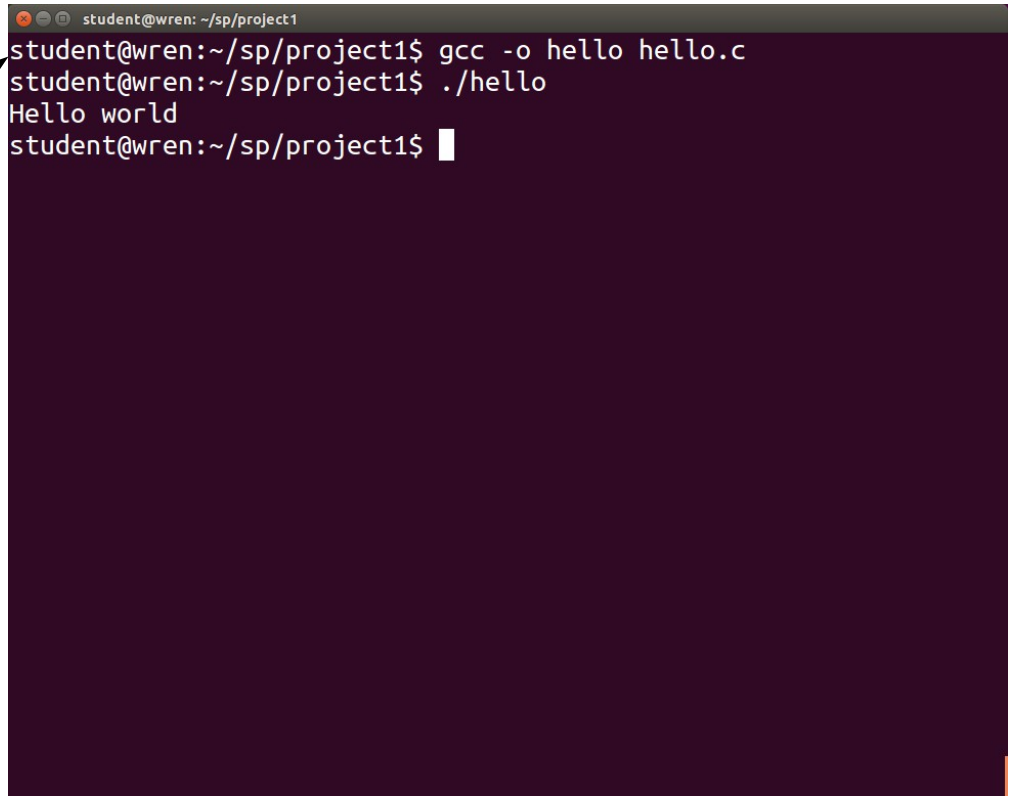
# Week 10 Lecture 2

Compilation and Make

# Compilation

- We use gcc to compile.
  - Gnu open source
- Here it combines
  - Preprocessing
  - Compilation
  - Assembly
  - Linking



# Hello.c

- Preprocessor
- C

student@wren: ~/sp/project1

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        printf("Hello world\n");
        return 0;
}
hello.c (END)
```

# Preprocessing

- C is unusual in that it include a preprocessor.
  - The preprocessor commands start with a '#'
    - The commands change the text of the file before compilation
  - #include <stdio.h> is a preprocessor command.
  - It is replaced with the content of the file stdio.h
    - Contains the function templates for the standard C I/O files.
  - Modern languages do not require functions templates to appear before their use
    - The compiler looks for all of the functions before compiling
    - #include is not found in modern languages.

# Pre-processing 2

- Another pre-processor directive “#define”
  - This directive replaces the name being defined with the string following it.
  - For example, #define THREE 3 will replace every string in the file with the numeral 3.
  - By conventions all elements defined this way are printed in all-caps
  - Modern languages do no have macros because macros can be hard to debug

# Preprocess output: hello.i

- Defines variables

```
char *_IO_save_base;
char *_IO_backup_base;
char *_IO_save_end;

struct _IO_marker *_markers;

struct _IO_FILE *_chain;

int _fileno;
```

- Defines functions

```
extern FILE *fopen (const char *__restrict __filename,
      const char *__restrict __modes) ;




extern FILE *freopen (const char *__restrict __filename,
        const char *__restrict __modes,
        FILE *__restrict __stream) ;
# 295 "/usr/include/stdio.h" 3 4
```

# Compiling

- Compiling transforms C to assembly language
- Assembly language is a symbolic representation of machine language.
- Modern languages translate
  - directly into machine language because compilers optimize machine language better than humans.
  - Into an intermediate interpreted language so the compiled code can run on many machines.
    - An interpreter is a virtual machine
      - I.e., a simulated machine

# Assembly Language “hello”

- Items starting with '.' instruct assembler
- Items ending with ':' are labels
- Instructions are pushq, movq, subq, movl, movq, call, and leave

student@wren: ~/sp/project1

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
:
```

# Assembly

- The assembler translates the assembly language to machine language.
- Output is a very long string of binary digits.
  - Some of these commands refer to programs in other files.
- It is difficult to display binary files.

# Linker/Loader

- The linker adds the code from libraries so that the program can run.
- The loader instructs the computer on how to put the binary files in memory so the program will run.

# Where are the Libraries

- cpp -v produces a list of directories searched.
  - Must specify libraries for files specified by quotation marks.
  - Standard libraries are surrounded by “< >”

```
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/4.9/include
 /usr/local/include
 /usr/lib/gcc/x86_64-linux-gnu/4.9/include-fixed
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
```

# Why this is useful

- Files that have been compiled need not be compiled again unless they change.
- Compilation takes a long time, so not compiling files that do not change can save hours of compile time.
- However, keeping track of when files have been compiled is complicated.

# Make

- Make is a program that defines how to build a programs.
  - Target dates are checked and actions taken only if they are out of date.
- Also allows multiple different activities on the same program.

# Make example

- Targets
  - Can be made
- Dependencies
  - Need to be up to date
- Actions
  - Indicate how to make targets up to date

student@wren: ~/sp/project1

```
all: hello

hello: hello.o
        gcc -o hello hello.o

hello.o: hello.c
        gcc -c hello.c

clean:
        rm hello.o hello
Makefile (END)
```

# Make example 2

- make = make all
- make clean: removes .o and executable
- Touch updates file date
- Forcing new make
- Not made otherwise

```
student@wren:~/sp/project1$ make
gcc -c hello.c
gcc -o hello hello.o
student@wren:~/sp/project1$ ls
hello  hello.c  hello.o  Makefile
student@wren:~/sp/project1$ make clean
rm hello.o hello
student@wren:~/sp/project1$ ls
hello.c  Makefile
student@wren:~/sp/project1$ make all
gcc -c hello.c
gcc -o hello hello.o
student@wren:~/sp/project1$ ls
hello  hello.c  hello.o  Makefile
student@wren:~/sp/project1$ touch hello.c
student@wren:~/sp/project1$ make
gcc -c hello.c
gcc -o hello hello.o
student@wren:~/sp/project1$ make
make: Nothing to be done for 'all'.
student@wren:~/sp/project1$
```

# Make Syntax

- Target separated from dependencies by ':'
- Actions on new lines which must start with a tab character
  - N.B. A common error is to put spaces in instead of a tab. They look the same.

```
dep1: [dep-1 dep-2 ...]
<tab>[command1
<tab>command2
<tab>......]
```

# Make for Calculator (variables)

```
# This is a simple make Macros that contains the flags for the compiler.
CFLAGS  = -Wall -std=c11 -g -rdynamic

# This make instruction lets the program know that clean does not have
# any dependencies
.PHONY: clean

# Here is a simple Make Macro that defines the name of the app we are
# making.
APP = calc

# Object files are designated by a .o, we delete them when we want to
# start again.
OBJS =  calc.o main.o

# Test files are in the test directory
TEST_OBJS = test/test.o test/test_main.o
```

# Make for Calculator (all)

```
# There are two standard Targets your Makefile should probably have:
# "all" and "clean", because they are often command-line Goals.  Also,
# these are both typically Artificial Targets, because they don't
# typically correspond to real files named "all" or "clean".

# The rule for "all" is used to incrementally build your system.  It
# does this by expressing a dependency on the results of that system,
# which in turn have their own rules and dependencies.
all : $(APP)
```

# Make for Calculator (clean)

```
# Here is a simple Rule (used for "cleaning" your build environment).
# It has a Target named "clean" (left of the colon ":" on the first
# line), no Dependencies (right of the colon), and two Commands
# (indented by tabs on the lines that follow).  The space before the
# colon is not required but added here for clarity.

# Make clean removes all of the files that make creates and all of the
# emacs backup files.
clean :
	rm -f $(REBUILDABLES) $(BACKUPS) *~ **/*~
```

# Make for Calculator (calc)

```
# Here is a Rule that uses some built-in Make Macros in its command:
# $@ expands to the rule's target, in this case "calc".  $^ expands to
# the rule's dependencies, in this case the two files main.o and
# calc.o
$(APP) : $(OBJS)
        gcc $(CFLAGS) -o $@ $^
```

# Make for Calculator (test)

```
# This rule says to make a test, which we will call calc_test, you
# will need the TEST_OBJS (test_main and test, both of which are in
# the test directory). We also need to include the cunit library
# (lcunit).
test : $(TEST_OBJS) calc.o
        gcc $(CFLAGS) -o calc_test $^ -lcunit
```

# Make for Calculator (.o)

```
# This rule indicates that any time a file with a .o extension is
# older than the thing with a .c extension, recompile it. It also says
# to make recompile by calling the gcc compile the .c file using the
# CFLAGS and to put the output in the .o file.
%.o : %.c
        gcc $(CFLAGS) -o $@ -c $<
```

# Make for Calculator (Dependency)

```
# These are Dependency Rules, which are rules without any command.
# Dependency Rules indicate that if any file to the right of the colon
# changes, the target to the left of the colon should be considered
# out-of-date.  The commands for making an out-of-date target
# up-to-date may be found elsewhere (in this case, by the Pattern Rule
# above).  Dependency Rules are often used to capture header file
# dependencies.

# These say that calc.o also depends on calc.h and that test
calc.o : calc.h
test.o : test.h
```

# Gcc Tutorial

- https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html