CSO-Recitation 02

CSCI-UA 0201-007

R02: GCC & Makefiles & Test

Today’s Topics

- Mini quiz in last week
- Compiling with gcc
- Makefiles
- Testing code

Mini quiz - Q1

- Facebook has 2.7 billion users. If it is to use an unsigned int as user-id, what's the smallest sized int can it use?
- A. 1-byte B. 2-byte C. 3-byte D. 4-byte E. 8-byte

Mini quiz - Q1

- Answer: D. 4-byte
- 3 bytes = 24 bits, range: 0 ~ 2^24 - 1
- 4 bytes = 32 bits, range: 0 ~ 2^32 - 1
- 2^24 - 1 < 2.7 * 10^9 < 2^32 – 1

Mini quiz - Q2

- Which of the following signed 1-byte int (in binary format) is the smallest?
- A. 00000000 B. 10000001 C. 11111111
- D. 00000001 E. 10000011 F. 01111110

Mini quiz - Q2

- Answer: **B**. 10000001
- For signed int, convert it into a decimal number:
  - For i-th bit, (i-th bit) * (+/- 2^i).
  - highest bit: *(-2^i), others: *(+2^i)
- Smallest:
  - Highest bit is 1
  - For other bits(positive), pick the smaller one

Mini quiz - Q3

- Convert bit pattern `10111110` to hex notation. You must prefix your answer with `0x`.

Mini quiz - Q3

- Answer: 0xbe
- 10111110
- 1011 1110
- 1011 = b, 1110 = e

Mini quiz - Q4

- Which of the following 1-byte **unsigned** subtraction operation will overflow?
- A. 0xff – 0x0f B. 0x0f – 0xff C. 0x01 – 0x0f D. 0x0f - 0x01

Mini quiz - Q4

- Answer: B. 0x0f – 0xff, C. 0x01 – 0x0f
- Overflow: when the result is out of the range of the representation
- 8-bit unsigned range: 0 ~ 2 ^ 8 – 1
- For unsigned operation:
  - Case 1: when the result is negative
  - Case 2: when the result is positive but too large (> 2^8 – 1 in this question)

Mini quiz - Q5

- Which of the following 1-byte **signed** addition operation will overflow?
- A. 0xff + 0xfe B. 0x1f + 0xff C. 0x71 + 0x70
- D. 0x05 + 0xfe E. 0x80 + 0x8f

Mini quiz - Q5

- Answer: C. 0x71 + 0x70, E. 0x80 + 0x8f
- Overflow: when the result is out of the range of the representation
- 8-bit signed range: - 2^7 ~ 2^7 – 1
- For signed operation:
  - Case 1: adding two positive numbers, but the MSB of the result is 1 (negative)
  - Case 2: adding two negative numbers, but the MSB of the result is 0 (positive)
    - Case 1 & Case 2: for adding numbers with the same signs, overflow <=> MSB is incorrect
  - Note: overflow wouldn't happen if you add a negative number and a positive number.
  - Why?

Mini quiz - Q6

- If x has bit pattern `0xffffffff`, what's the value of x?
- A. -1, if x is signed int
- B. -1, if x is unsigned int
- C. 2^32 – 1, if x is unsigned int
- D. 2^31 – 1, if x is unsigned int

Mini quiz - Q6

- Answer: A. -1, if x is signed int. C. 2^32 – 1, if x is unsigned int

Mini quiz - Q7

- What's the bit pattern (2's complement) of 32-bit signed integer -130 in hex format? (Please prefix your answer with 0x)

Mini quiz - Q7

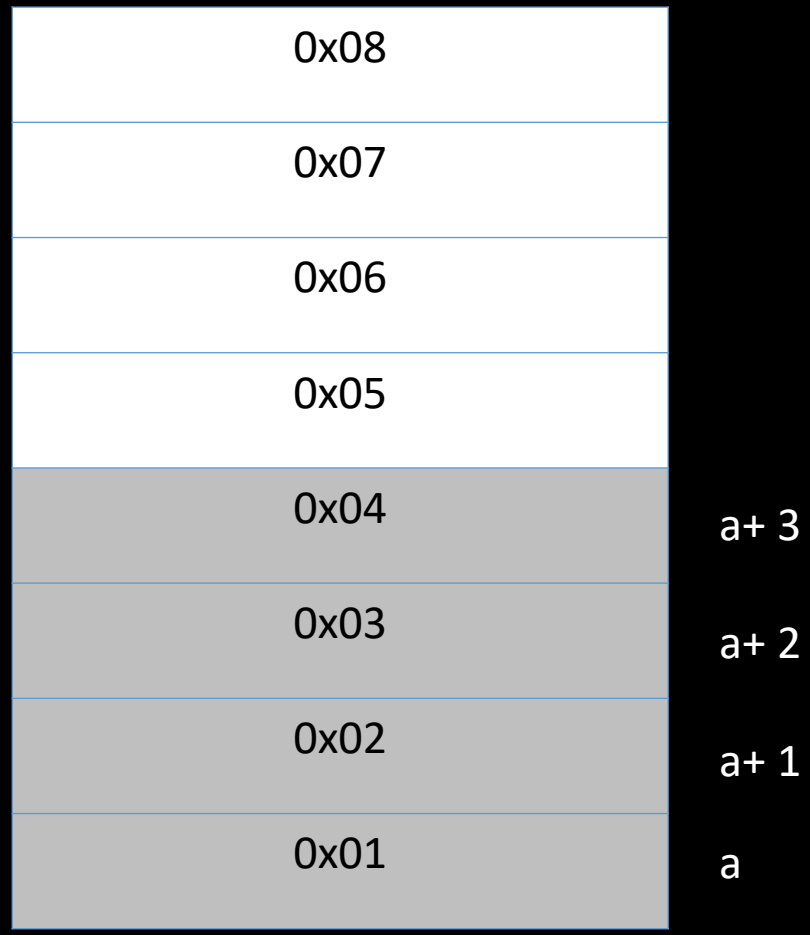
- Answer: 0xffffff7e
- 130 = 16 * 8 + 2: 0000 0000 0000 …. 1000 0010
- 0000 0000 0000 …. 1000 0010
- -> (flip) 1111 1111 1111 …. 0111 1101
- -> (+1) 1111 1111 1111 …. 0111 1110 = 0xffffff7e

Mini quiz - Q8

- Suppose the byte values stored at memory address a, a+1, a+2, a+3, a+4, a+5, a+6, a+7 are 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 respectively. If a Little-Endian processor is to load a 4-byte integer from memory at address a into a 4-byte register, what's the 4-byte register value after the load? (Please write your answer in hex, and prefix it with 0x)

Mini quiz - Q8

- Answer: 0x04030201



Little endian: 0x04030201
Big endian: 0x01020304

Reminder

- Your second weekly mini-quiz
  - Gradescope
  - Due Friday 9pm EST

Compiling

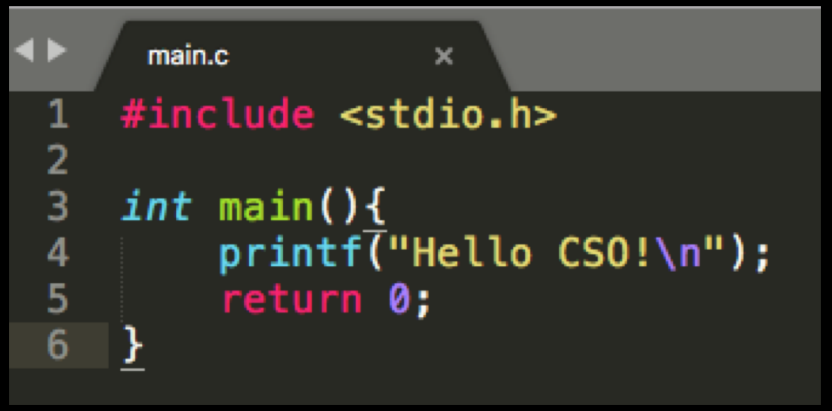
The basics of GCC

GCC

- GCC (upper case) refers to the GNU Compiler Collection
  - This is an open source compiler suite which include compilers for C, C++, Objective C, Fortran, Ada, Go and Java
- gcc (lower case) is the C compiler in the GNU Compiler Collection

What is a compiler?

- C code is for people, not computers
  - In fact, high level languages in general are for people
  - Computer processors only “understand” binary instructions



Source code

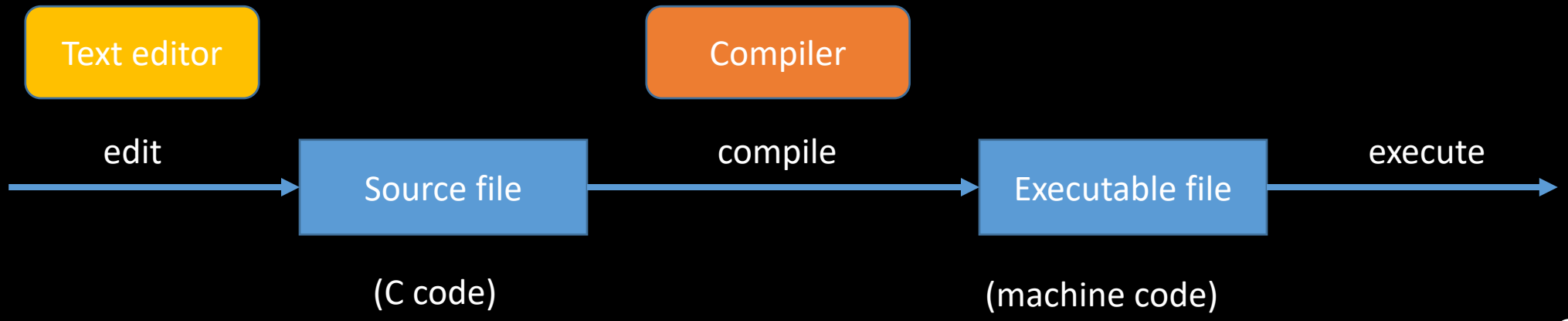
Source file: the file containing source code (aka all ".c" files)

Machine code
(binary instructions)

Executable file: the file containing machine code

What is a compiler?

- C code is for people, not computers
  - In fact, high level languages in general are for people
  - Computer processors only “understand” binary instructions
- A compiler translates code between languages
  - In our cases, it translates from C (the source language) to machine code (the target language)



What is a compiler?

- C code is for people, not computers
  - In fact, high level languages in general are for people
  - Computer processors only “understand” binary instructions
- A compiler translates code between languages
  - In our cases, it translates from C (the source language) to machine code (the target language)
- An alternative way to do things is to have a program read the code and execute commands
  - Such a program is called an interpreter
  - Python is an example of a language that uses an interpreter

How do you use a compiler?

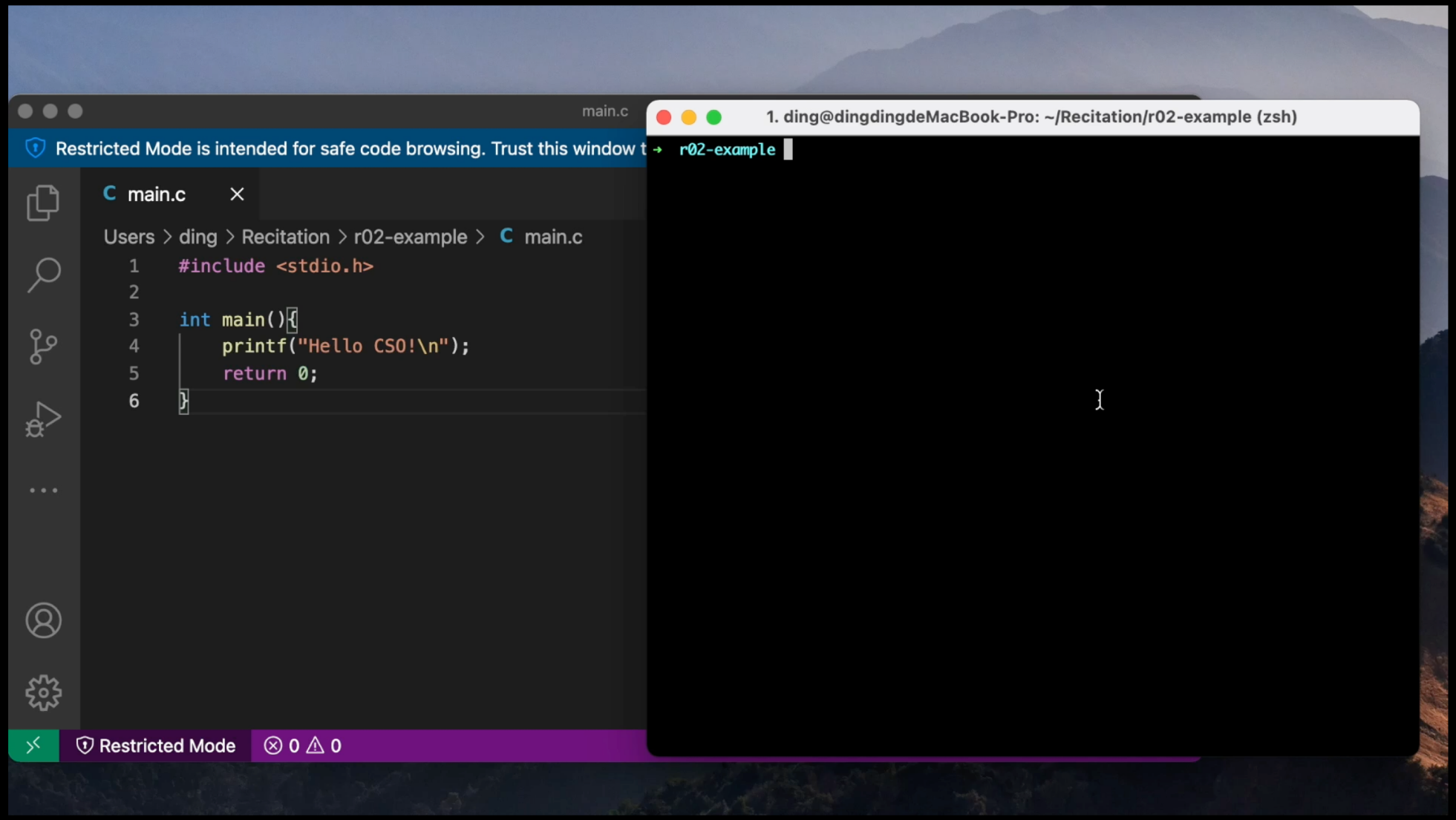
- Consider a simple C program:

main.c

```
#include <stdio.h>
int main( ){
        printf("Hello CSO!\n");
        return 0;
}
```

- To run this program, we must first compile it
  - Can use gcc: gcc main.c -o myprogram
  - A file named myprogram is generated
  - You can run it with ./myprogram

How do you use a compiler?



How do you use a compiler?

- Sometimes you may want to spread the code into multiple file (e.g., the code is too large; or you want to divide it by functionalities)

main.c

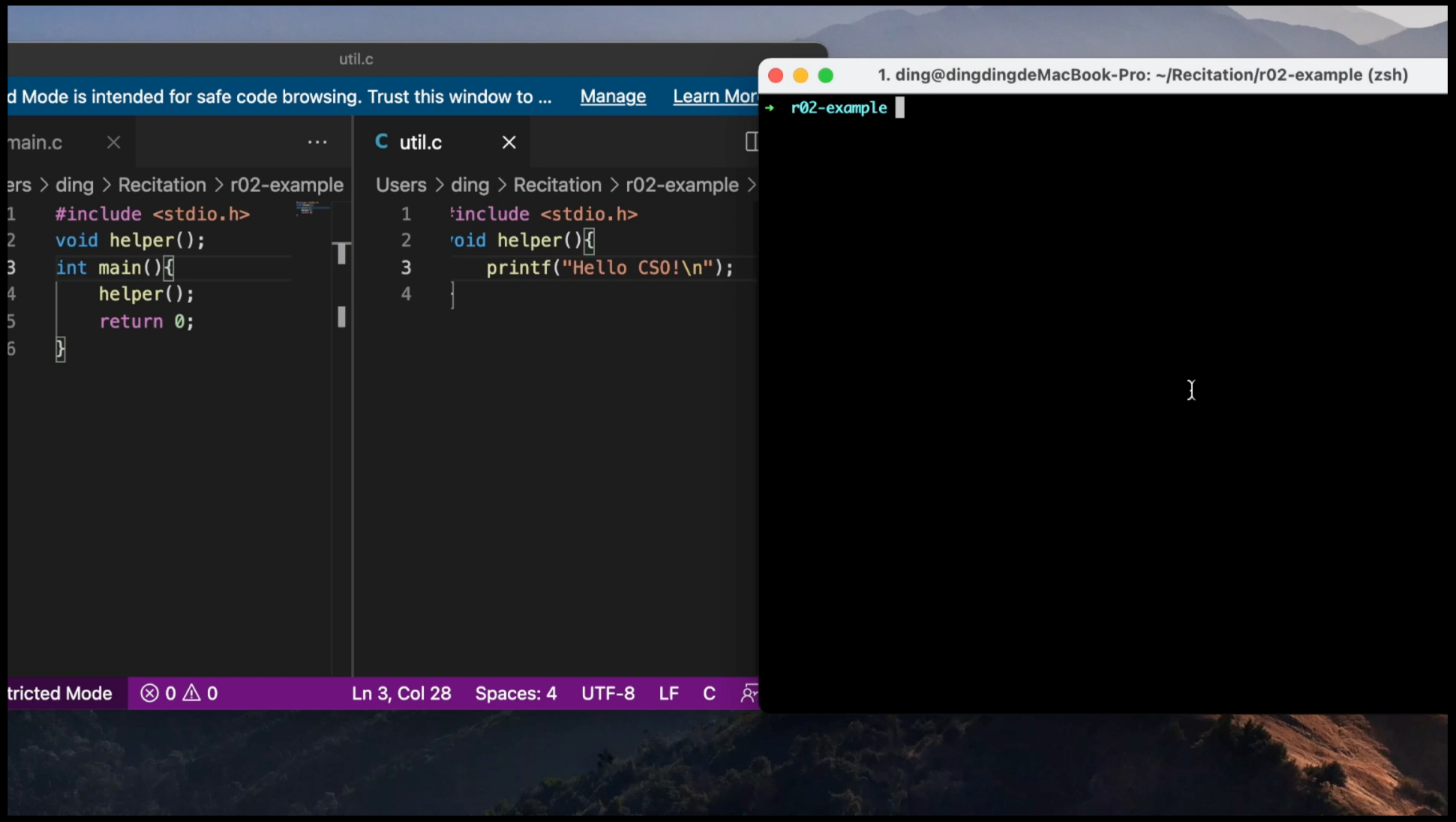
```c
#include <stdio.h>
void helper();
int main(){
    helper();
    return 0;
}
```

util.c

```c
#include <stdio.h>
void helper(){
    printf("Hello CSO!\n");
}
```

- To compile this, we can simply specify both files
  - gcc main.c util.c -o myprogram

How do you use a compiler?

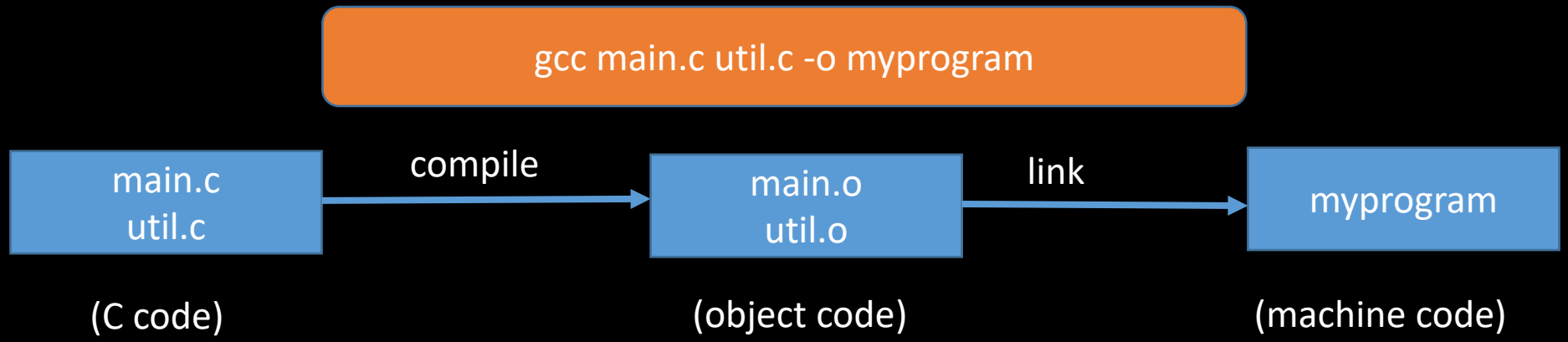


A Problem

- gcc main.c util.c -o myprogram will process every source file
- Even if we only change main.c, util.c is also processed
- Problematic for large project (thousands of files), as reprocessing every file can be slow
- Can we only re-process the changed one?
- Yes! We can make the use of the object files (".o" files)

What's inside gcc main.c util.c -o myprogram?

- Roughly two steps: compilation and linking
- Compilation: For each source file (aka compilation unit), gcc creates a intermediate object file
- Linking: creates a single executable file from multiple object files
- Note: you won't see object files ordinarily as they are automatically deleted after linking



What's inside gcc main.c util.c -o myprogram?

- We can run both steps separatly
- To only run compilation: use the -c flag to stop before linking. It also preserve the intermediate object files
  - gcc -c main.c
    - will create main.o
  - gcc -c util.c
    - will create util.o
- To only run linking:
  - gcc main.o util.o -o myprogram
    - will link main.o and util.o and create the executable file

Solution to the Problem

- The problem: changing one file requires recompiling all other unchanged files, which are wasteful and slow
- The cure:
- use -c to create main.o and util.o
- Every time main.c is changed, recompile it with gcc -c main.c and not have to recompile the other
  - Same when util.c is changed
- We can later do link by running gcc main.o util.o -o myprogram

A new problem

- Now we need to manually keep track of when and what files we have to recompile.
- Too much trouble, and error-prone

Make

A helpful build automation tool

What does Make do?

- Make builds (i.e. compiles) projects for us, keeping track of when it needs to recompile or not
- We create a file named Makefile and write down a set of rules stating what to track
  - e.g., issue gcc -c main.c to generate main.o when main.c is changed
- Then, by issuing the command make we can build our project,
  - next time you issue make, only the changed files will be recompiled

How do we specify rules in Makefile?

- Makefile consists of a number of ‘rules’, each of which looks like:

  target ... : *dependencies* ...

  command

- Target is usually the name of a file generated by the compiler
  - e.g., main.o, util.o, myprogram
- Dependencies are files that are used as input to create the target
  - main.o needs main.c
  - myprogram needs main.o and util.o
- Commands are actions that will be carried out
  - gcc -c main.c -o main.o

How do we specify rules in Makefile?

- Makefile consists of a number of ‘rules’, each of which looks like:

  target ... : *dependencies* ...

  command

- It specifies how to build target:
  - If target is already built (i.e. file existed) and up-to-date (i.e. modified later than all the dependency files), no actions are carried out
  - Otherwise, build each dependency first and then issue command
- To build target, issue make target

How do we specify rules in Makefile?

- An exmple :

```
myprogram: main.o util.o
        gcc main.o util.o -o myprogram
```

- It specifies the rule to build myprogram:
  - Build main.o and util.o first
  - Then issue "gcc …"
- Similary we have rules for main.o and util.o:

```
main.o: main.c
    gcc -c main.c -o main.o
util.o: util.c
    gcc -c util.c -o util.o
```

How do we specify rules in Makefile?

- Issue make myprogram to build myprogram
- Try issuing it twice. You'll find that no actions are taken in the second run
- Try changing main.c and issue make myprogram. You'll find that util.c is not compiled

How do we specify rules in Makefile?

target ... : *dependencies* ...
<*TAB*>*command*

- Attention:
- There must be no space before the target, and there must be a tab before every command for that rule
- Running the *make* command builds the first target by default
- A handy and commonly seen rule

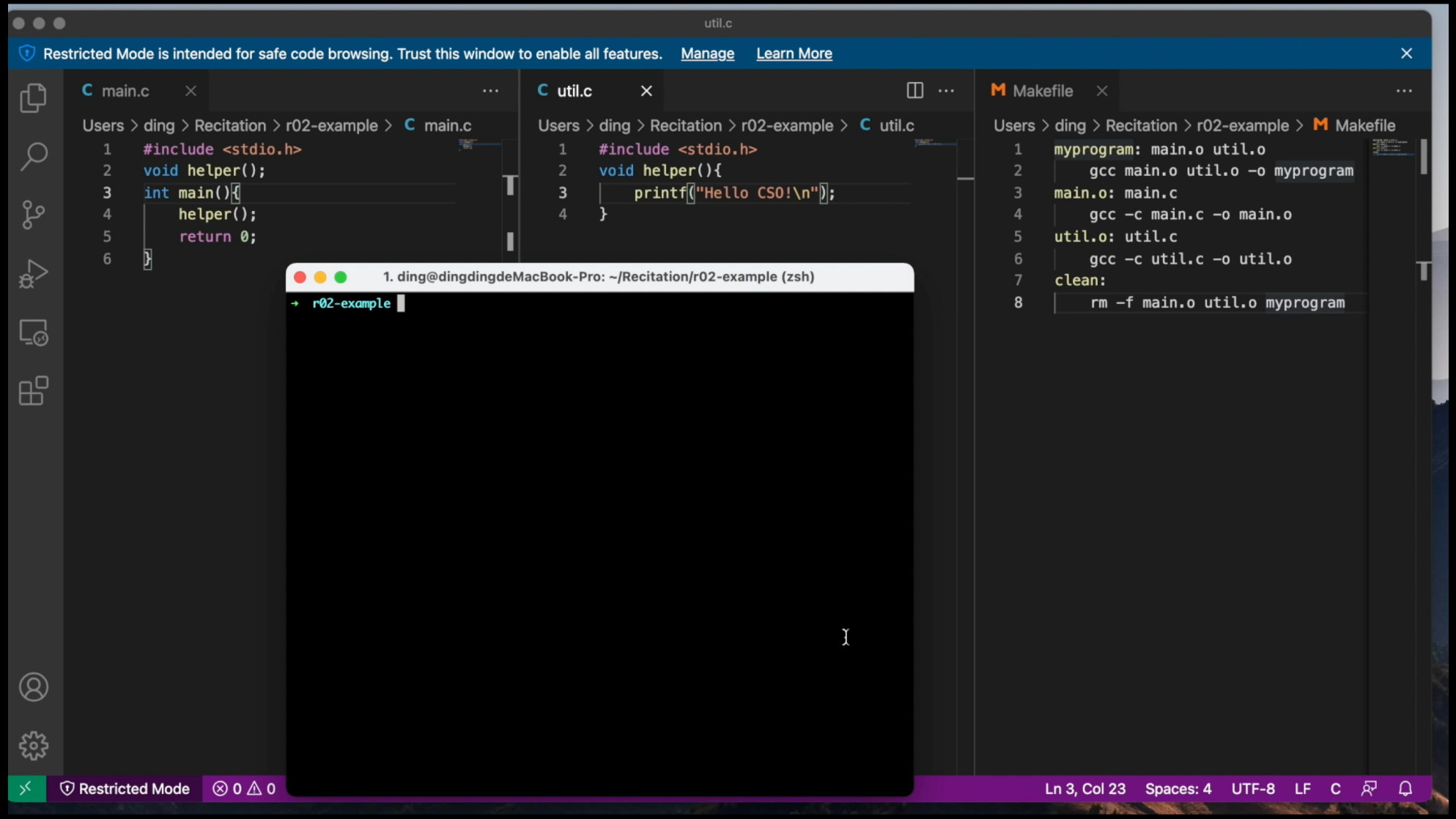
```
clean:
    rm -f main.o util.o myprogram
```

  - make clean is identical to "rm …" (but shorter) which removes all the files generated by the compiler

The overall Makefile

```
myprogram: main.o util.o
	gcc main.o util.o -o myprogram
main.o: main.c
	gcc -c main.c -o main.o
util.o: util.c
	gcc -c util.c -o util.o
clean:
	rm -f main.o util.o myprogram
```

The overall Makefile



Quiz

- A bad Makefile for this little project is:

```
myprogram: main.c util.c
        gcc main.c util.c -o myprogram
```

- Why is that bad?

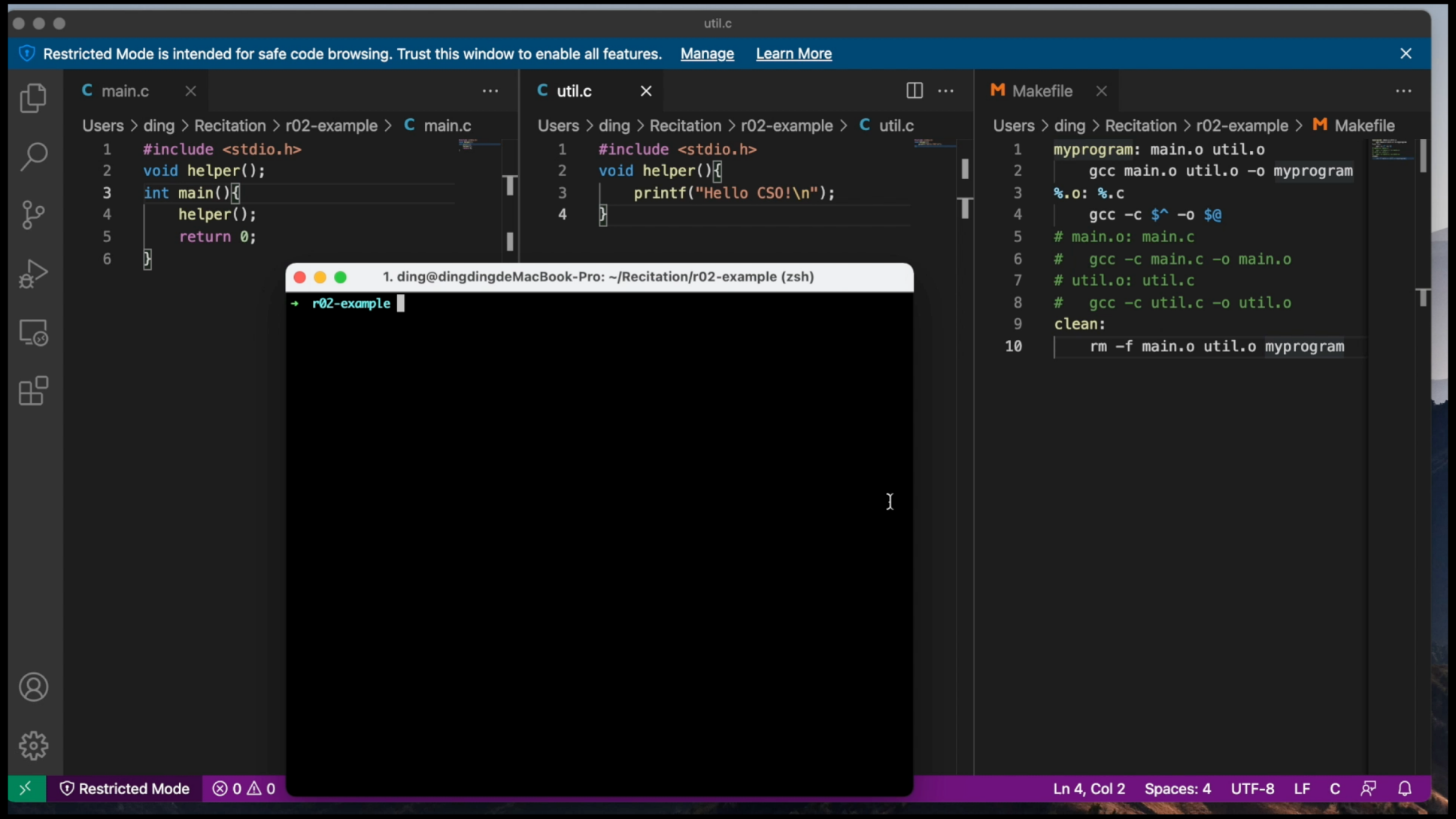
That still seems bad for the 45,000 linux files..

- That's right, and there are better ways of using Makefiles - this is just what you absolutely positively need to know
- Make also supports pattern matching with the percent sign %
  - %.c means all .c files
- Make has "automatic variables"
  - Variables whose meaning within a rule depends on context
  - $@ is the target name that you are building for this rule
  - $^ is the list of dependencies
- Example:

```
%.o: %.c
        gcc -c $^ -o $@
```


```
main.o: main.c
        gcc -c main.c -o main.o
util.o: util.c
        gcc -c util.c -o util.o
```

That still seems bad for the 45,000 linux files..



An exercise

#TODO: Create a makefile for this project

#The name of the executable must be test

#The source code files involved are main.c and util.c

#*make clean* should remove test and any .o files

Testing

Making sure your code does what you think it does

Why test code?

- You need to know that your code works
- You need to know when you broke your own code by changing something
- Many projects actually have more test code than production code
  - An extreme example is SQLite, a popular database program
    - 138,900 lines of C code for production
    - 91,946,200 lines of test code

How do you test code?

- A common way is to write tests for individual units of code, such as functions
- There are many frameworks written to help developers write test cases
- You can write your own tests
  - Think of edge cases that might make your code failed
  - Write a program that calls your code with different inputs and checks that the output is what you'd expect
    - You can use assert to have your program die if something goes wrong
    - assert(1+1==2) will crash if 1+1 is not 2, but be fine otherwise