

# CSO-Recitation 02

CSCI-UA 0201-007

R02: GCC & Makefiles & Test

# Today's Topics

- Compiling with gcc
- Makefiles
- Testing code

# Reminder

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

# Compiling

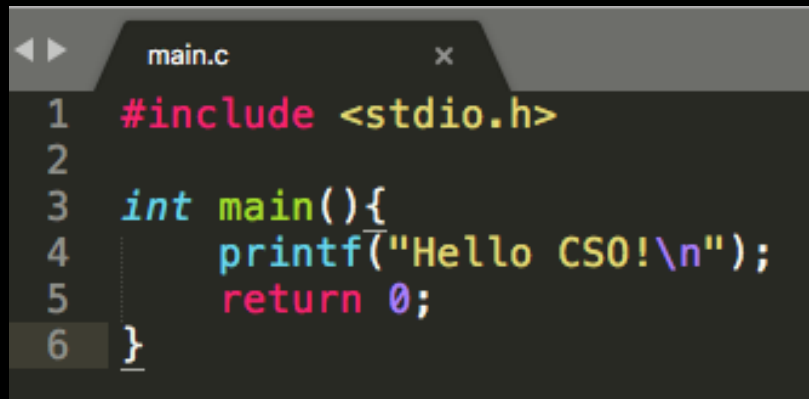
The basics of GCC

# 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



```
1 #include <stdio.h>
2
3 int main(){
4     printf("Hello CS0!\n");
5     return 0;
6 }
```

Source code

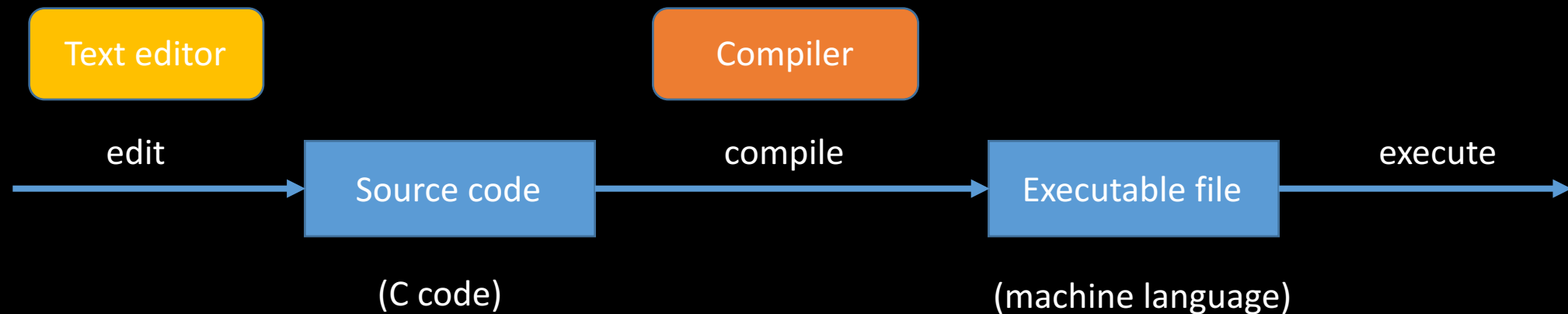


Machine code  
(binary instructions)

# 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?

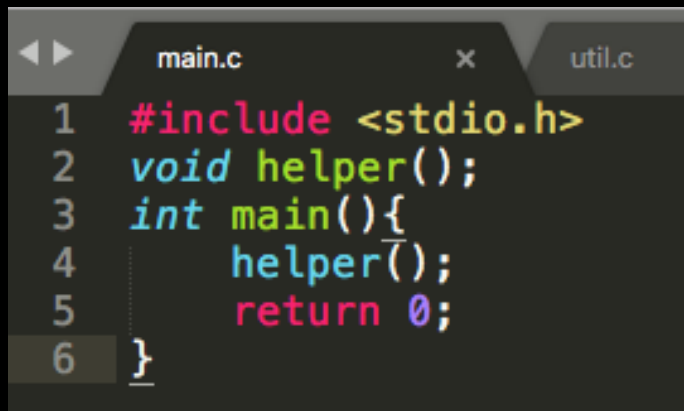
- Consider a simple C program:

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

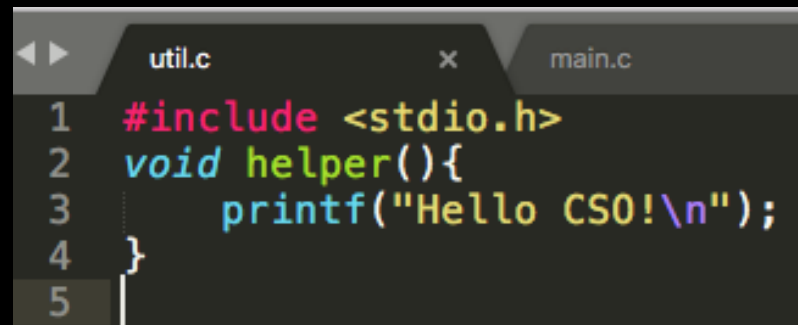
- To run this program, we must first compile it
  - Can use gcc: `gcc main.c` will produce a file called `a.out`
  - We can run a.out by issuing `./a.out`
  - You can choose the name of executable with `-o`, as in `gcc main.c -o myprogram`

# How do you use a compiler?

- You can also compile more than one file

A screenshot of a code editor window with two tabs: 'main.c' and 'util.c'. The 'main.c' tab is active, showing the following C code:

```
1 #include <stdio.h>
2 void helper();
3 int main(){
4     helper();
5     return 0;
6 }
```

A screenshot of a code editor window with two tabs: 'util.c' and 'main.c'. The 'util.c' tab is active, showing the following C code:

```
1 #include <stdio.h>
2 void helper(){
3     printf("Hello CS0!\n");
4 }
5 |
```

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

# A Problem

- For large projects, recompiling everything can be slow
- To avoid this, we can compile files separately
  - Must compile without linking using the `-c` flag
  - Ordinarily, the compiler compiles each source file into object code, and then links them and deletes the intermediate object code
- `gcc -c main.c util.c`
  - Will create `main.o` and `util.o`
  - Then we can add to `main.c` and `util.c` and not have to recompile the other
  - We can later do link by running `gcc main.o util.o -o myprogram`

# Compilation and linking

- Compiling isn't quite the same as creating an executable file!
- Instead, creating an executable is a multistage process divided into two components: **compilation** and **linking**
- **Compilation:**
  - Refers to the processing of source code files and the creation of an **'object' file**
  - Merely products the machine language instructions
- **Linking:**
  - Refers to the creation of a single executable file from multiple object files

# Compiling

- .c is the source code
- .o extension is for object files
  - is a binary file generated by compilation unit
- executable files should have no extension
  - By default a.out, which has the .out extension

# A new problem

- Now we need to keep track of when we have to recompile.

# Make

A helpful build automation tool

# Why do we need Make?

- Even a small project is unbearable to compile with gcc alone
- But in the real world, things are much worse!
  - The Linux kernel has over 45,000 files of C code!
    - So it uses Makefiles... almost 2700 of them
- Make will also know when we need to recompile different sources



# What does Make do?

- **Make** builds projects for us, keeping track of when it needs to recompile or not
  - When **make** recompiles, each changed C source file must be recompiled
  - If any source file has been recompiled, all the object files must be linked together to produce the new executable file
- We tell make about the dependencies in our code using a **Makefile**
- Then, by issuing the command **make** we can build our project, and Make will only compile what it has to

# What is a Makefile?

- Makefile consists of a number of 'rules':

```
target ... : dependencies ...  
command  
...
```

- Rules specify:
  - A target, which is usually the name of a file that is generated by a program
    - Targets include `main.o` or `myprogram`
  - Dependencies, which are files that are used as input to create the target
    - `main.o` needs `main.c`
    - `myprogram` needs `main.o` and `util.o`
  - Commands, which are actions that `make` carries out
    - `gcc -c main.c -o main.o`

# What is a Makefile?

- Rules look like this:

```
myprogram: main.o util.o
```

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

- 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
- However, the rule that specifies commands for the target need not have dependencies, for example “clean”
  - *make clean*

# What is a Makefile?

- Rules look like this:

```
myprogram: main.o util.o
```

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

- A bad Makefile for this little project is:

```
myprogram: main.c util.c
```

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

- Why is that bad?

# A better 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
```

# 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 name of the rule
  - `$$` is the list of dependencies
- Example:

```
%.o: %.c
```

```
gcc -c $$ -o $@
```

# 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
  - Wikipedia lists more than 50 for the C programming language
  - You don't need a framework, though
- 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