Compilation, Make, and Makefiles

C Program Compilation

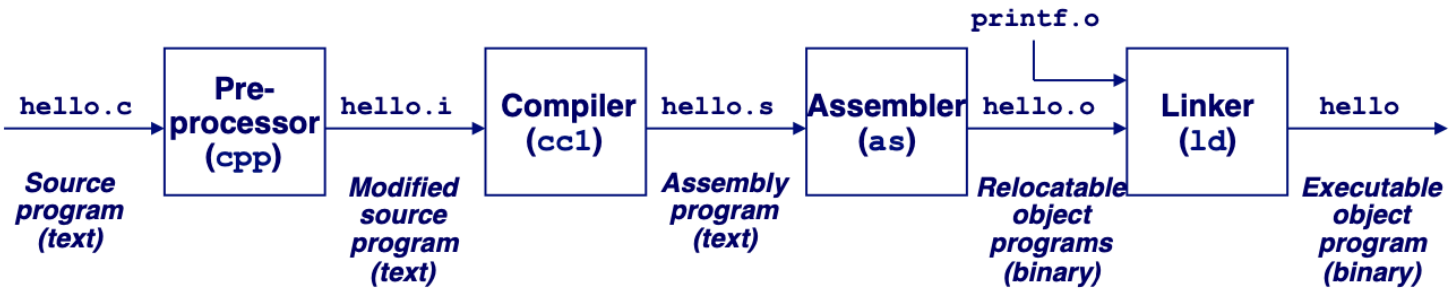
- Compile
  - gcc HelloWorld.c -o HelloWorld
  - Results in binary executable named HelloWorld
- Running it:
  - ./hello
- What if our programs become complex and there are multiple .c files for our program?

C Program Compilation

- To compile a program divded across multiple source files we can simply
  - gcc file1.c file2.c file3.c ….. fileN.c -o program
- This would work, but it may take a while
- If there is change in one of the files, all files need to be recompiled
- When dealing with large code bases, this won't be great

More Efficient Compiling

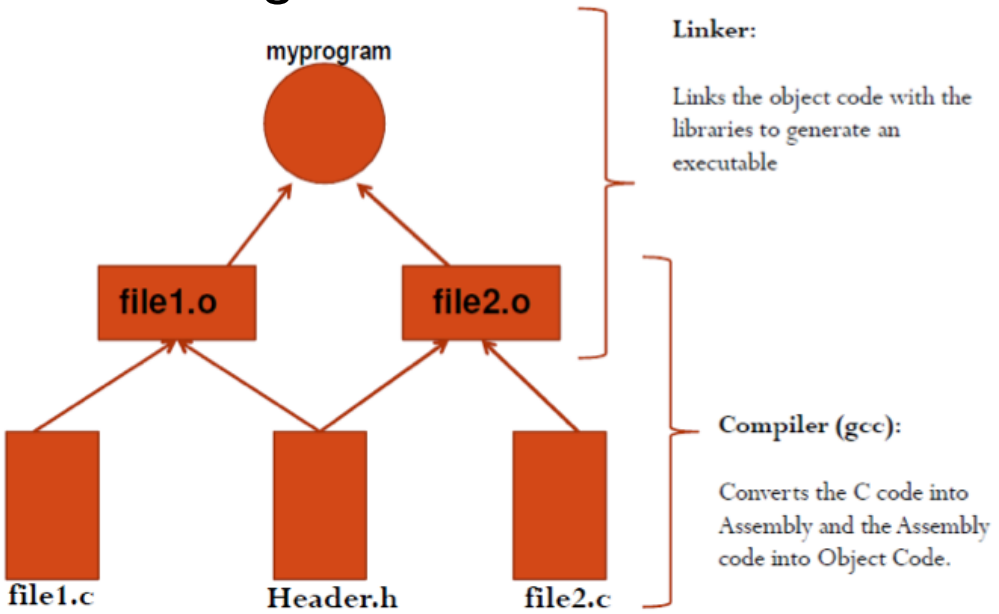
- Recall the compilation progress



- .c files are compiled and assembled into objects then linked to created the final executable object
- We can generate object files in isolation and link them together

More Efficient Compiling

- Consider the following:



- Generate file1.o and file2.o then link them
  - gcc -c file1.c (This will generate file1.o)
  - gcc -c file2 .c (This will generate file2.o)
  - gcc -o program file1.o file2.o (Link and generate executable named "program")
- If we change one file, we only need to compile that one file's object file (.o) and link it with the existing object files
- This would save processing time every time we want to compile

Make

- Make is a utility and is used to help compile programs
- Helps automates compilation process for large c projects
- It is good practice to use as you develop bigger and more complex code

Makefile

- In order to use Make you need to have a makefile
- A makefile is file that consists of various "rules" that consists of commands to run
- Format for rules:

```
<rulename>: <dependencies>
        <commands to run>
```

Note: command lines after the rule must be tabbed, no spaces or else make will complain

- Example:

```
all: program.c
        gcc -o program program.c
```

- If we run *make <rulename>*, make
  - makes sure the dependencies are met
  - if some dependencies are not met, then run rules to generate those dependencies if possible
  - Once dependencies are met then run the commands

Simple Makefile Example

- Example makefile

```
all: program.c
	gcc -o program program.c

clean:
	rm -rf program
```

- Typically there are two common rules that you should define
  1. all rule - main rule that's ran when running just "make"
  2. clean rule - rule to clean up all generated files
- For the example when we run "make", Make
  1. Looks if program.c exists
  2. If it does, run command "gcc -o program program.c"
- For the example when we run "make clean"
  1. Runs "rm -rf program" which deletes the program executable

More Typical Makefile Example

- Example makefile for a program with multiple files:

```
all: program

program: file1.o file2.o main.o
	gcc -o program file1.o file2.o main.o

clean:
	rm -rf file1.o file2.o program
```

- Where are the rules to generate the object files (.o)?
- Luckily Make automatically finds the corresponding .c files for a .o file and generates the .o file to resolve object file dependencies
- Make will also NOT recompile an object file if it has been previously compiled and the corresponding source code (.c file) has not changed
  - Saves us time by compiling only the code that changed

- Learn More:
  - https://www.gnu.org/software/make/manual/make.html
- Try and make your own makefiles for your own programs