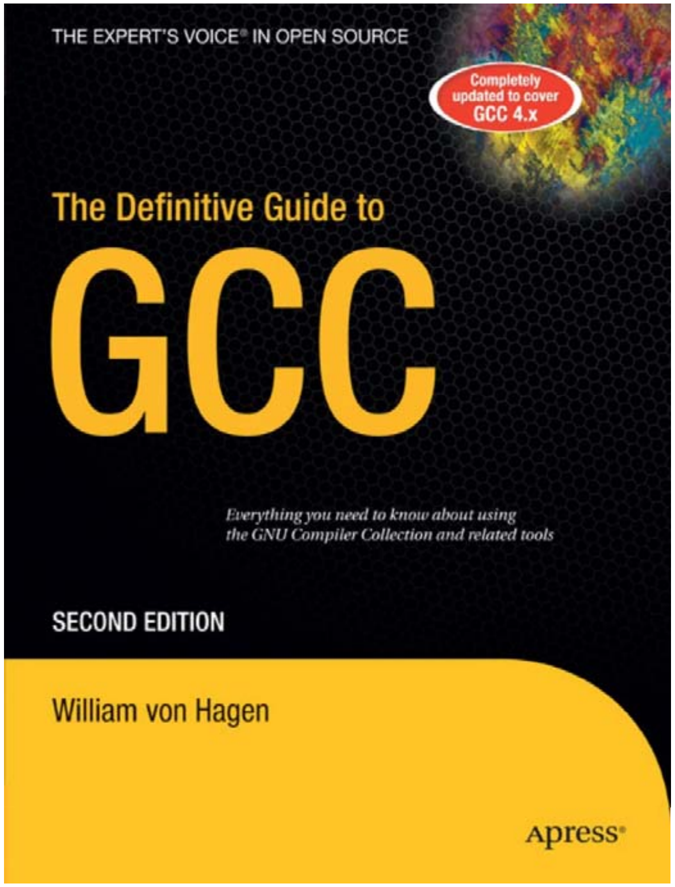
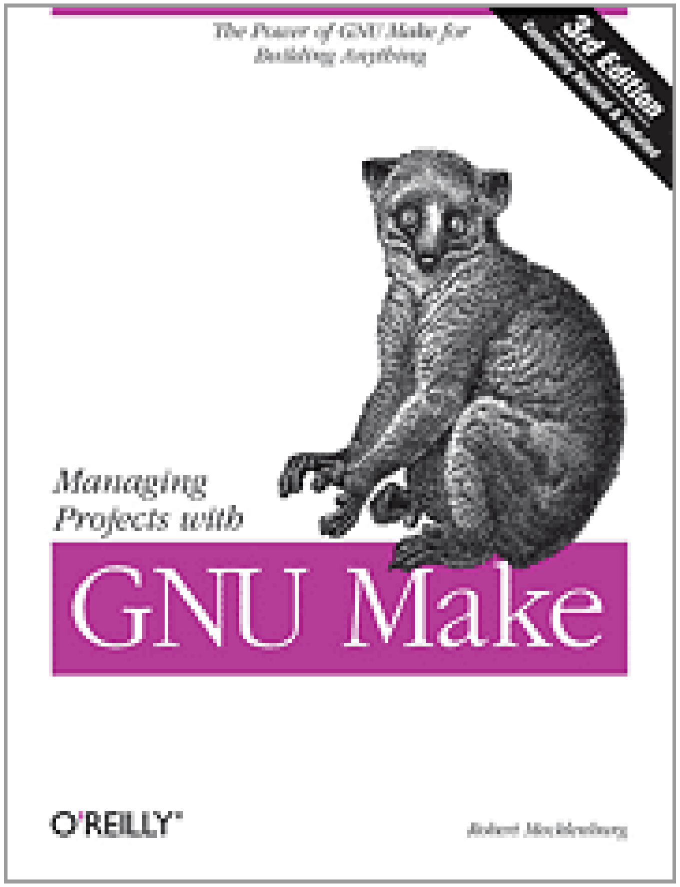
Compilation and Makefiles


THE EXPERT'S VOICE® IN OPEN SOURCE
Completely updated to cover GCC 4.x
The Definitive Guide to
GCC
Everything you need to know about using the GNU Compiler Collection and related tools
SECOND EDITION
William von Hagen
Apress®



Lecture Outline

- What is gcc?
  - What are the different *switches* available?

- What is a *Makefile*?

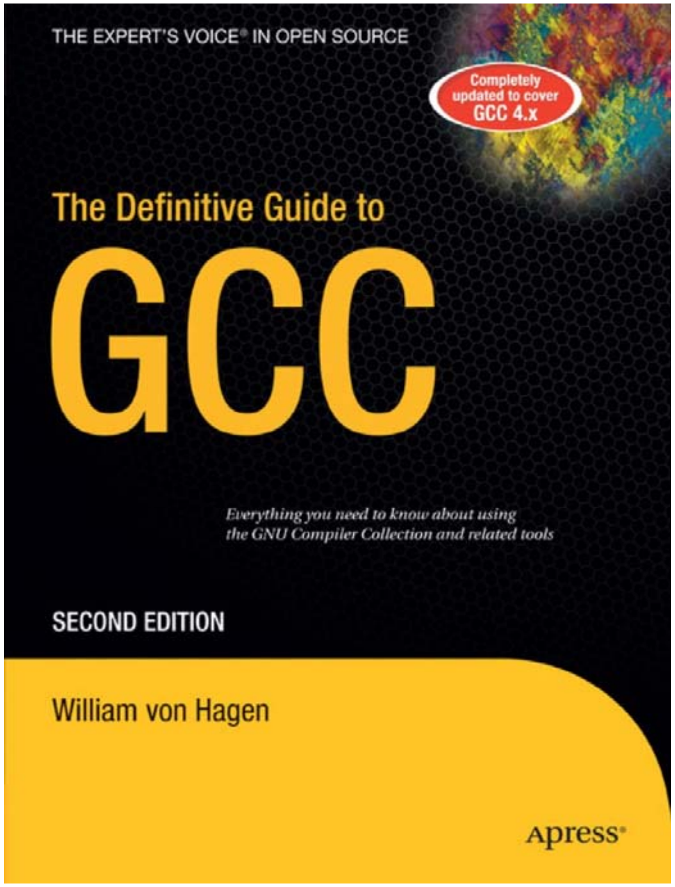
- Exercise:
  - Take an existing file, split it into multiple separate programs, and write a Makefile to simplify compilation of the files

gcc is not a single program

<u>gcc is a conglomeration of programs that work together to make an executable file</u>

We need to know what it does and how we can better control it.

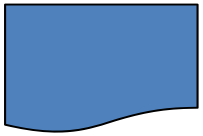
In particular, we would like to be able to make multiple, separate programs which can be combined into one executable


THE EXPERT'S VOICE® IN OPEN SOURCE
Completely updated to cover GCC 4.x
The Definitive Guide to
GCC
Everything you need to know about using the GNU Compiler Collection and related tools
SECOND EDITION
William von Hagen
Apress®

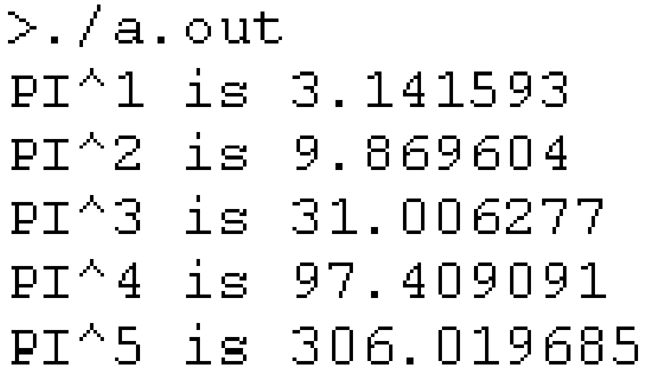
What does gcc do?

code.c

```
#include <stdio.h>
#include <math.h>

#define NumIter 5

int main()
{
    int i;
    for (i=1; i<=NumIter; i++)
        printf(“PI^%d is %f\n”, i, pow(M_PI, i));
}
```

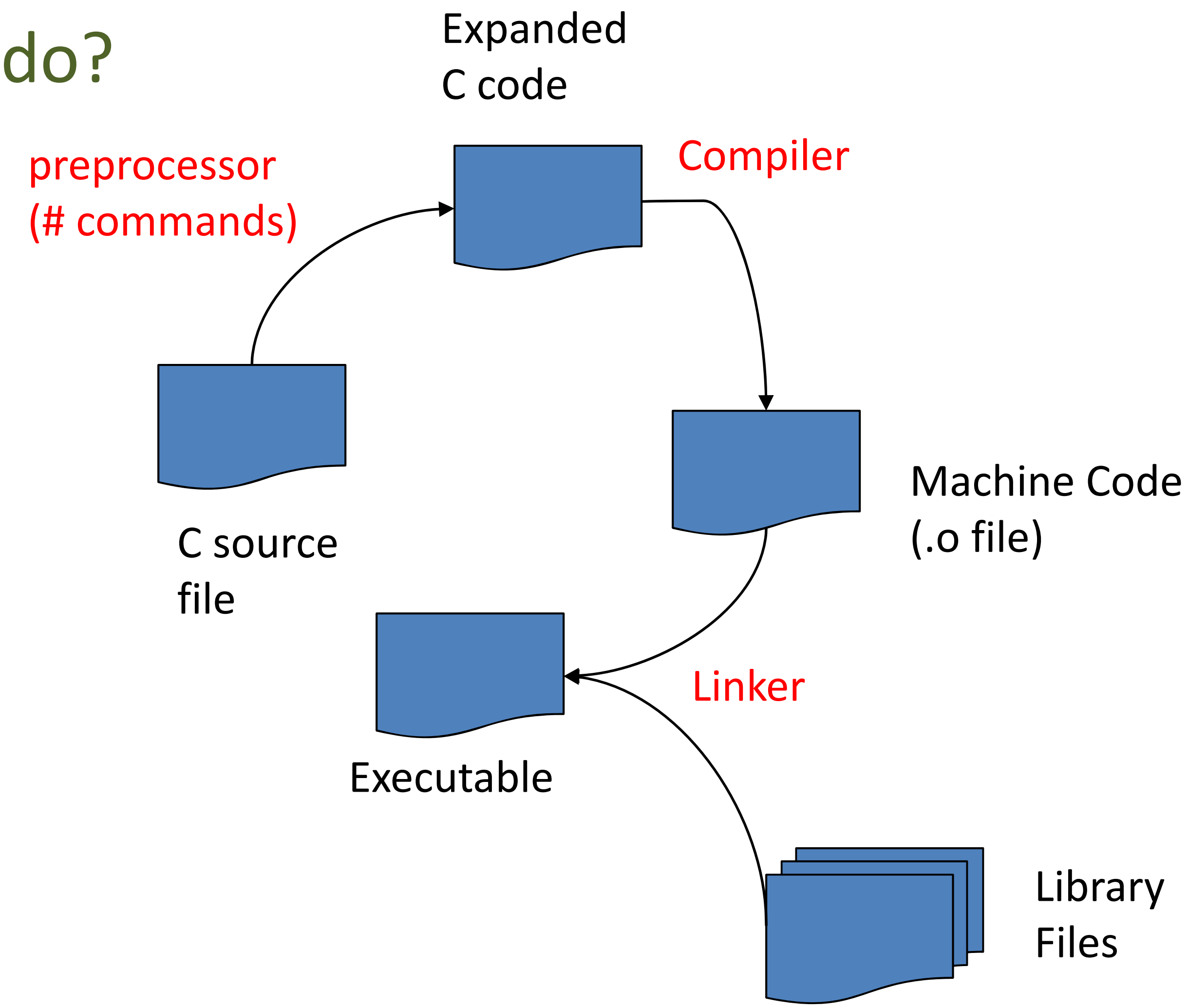


C source file



```
>./a.out
PI^1 is 3.141593
PI^2 is 9.869604
PI^3 is 31.006277
PI^4 is 97.409091
PI^5 is 306.019685
```

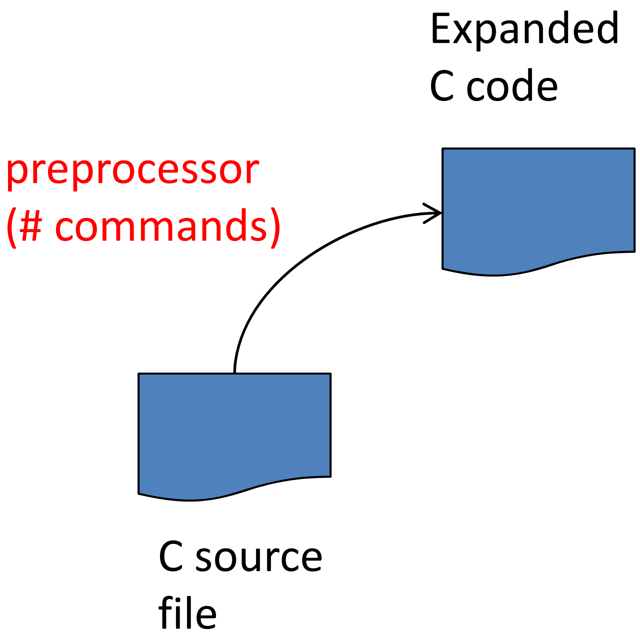
What does gcc do?



Some gcc options

- Preprocessing (**`gcc –E code.c > code.i`**)
  - Removes preprocessor directives (commands that start with #)
  - Produces **`code.i`** *Don't use directly*
- Compiling (**`gcc –o code.o –c code.i`**)
  - Converts source code to machine language with unresolved directives
  - Produces the **`code.o`** binary
- Linking (**`gcc –lm –o code code.o`**)
  - Creates machine language exectutable
  - Produces the **code** binary by linking with the math library (-lm)

C Preprocessor, cpp


Expanded C code

preprocessor (# commands)

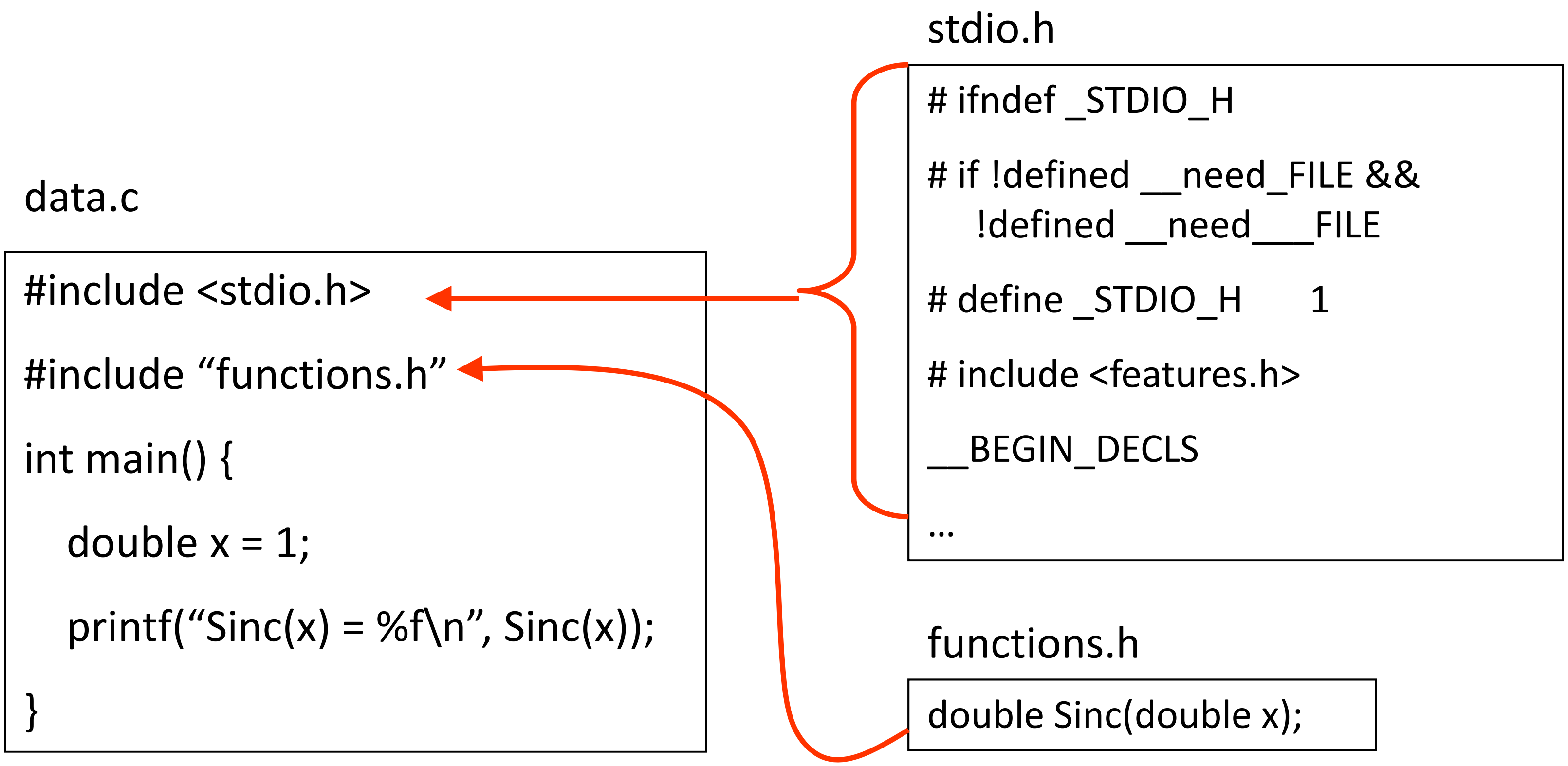
C source file

- Processes commands that are preceded with # symbols and expands the code
  - #include
  - #define
  - #ifdef and #ifndef

#include

Include *header files* that contain declarations necessary for compiling code

- the declarations provide the proper types, but not the actual definitions (the actual code)



data.c

```
#include <stdio.h>

#include "functions.h"

int main() {

   double x = 1;

   printf("Sinc(x) = %f\n", Sinc(x));

}
```

stdio.h

```
# ifndef _STDIO_H

# if !defined __need_FILE &&
     !defined __need___FILE

# define _STDIO_H      1

# include <features.h>

__BEGIN_DECLS

…
```

functions.h

```
double Sinc(double x);
```

```
gcc –E data.c > data.i
```

data.i is like an “expanded” C code in which the #include command is “substituted” with text from the files

Example

display.h

```
void DisplayMe(int n);
```

main.c

```
#include “display.h”

int main()
{
    DisplayMe(12);
}
```

Header Files

We’ve been using .h files to define things from the C libraries:
stdio.h, stdlib.h, stdbool.h, etc.

These are called *Header Files*.

We’re going to create our own .h files to share information between .c files.

#define

#defines a preprocessor variable

- every place the variable occurs, the definition will be substituted as code

Syntax: #define var_name var_definition

- Examples:

  #define DEBUG 1

  #define PI 3.1415926535

preproc.c

```
#include <stdio.h>
#define PI 3.14157265
#define square(x) (x*x)

int main() {
   printf("Value of PI is %f\n", PI);
   printf("Square of 3.5 is %f\n", square(3.5));
}
```

```
gcc –E preproc.c > preproc.i
```

preproc.i

```
# 1 “preproc.c"
# 1 "<built-in>"
# 1 "<command-line>“
…
typedef unsigned int size_t;
…
int main() {
   printf("Value of PI is %f\n", 3.14157265);
   printf("Square of 3.5 is %f\n", (3.5*3.5));
}
```

#ifdef, #ifndef, #if

Surround code that *might* be included. Include for compilation when desired

```
#ifdef DEBUG
….
#endif
#ifndef DEBUG
…
#endif
```

Common use

```
    if(num == 1)
    {
        /* This is the only time we actually move a disk */
        DisplayTower(tower);

#if 0
        printf("Press return");
        fgets(cmd, sizeof(cmd), stdin);
#endif

        MoveDisk(tower, fm, to);
        return;
    }
```

This is a handy way to *turn off* a block of code without removing it. It often works better than trying to comment out the block, which may have comments already in it.

Header Files

hanoi.h:

```
#define NumDisks 6
#define NumPins 3
```

I have two .c files in the same program. Both need to know NumDisks and NumPins. So, I put that information in a header file: hanoi.h

hanoi.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include “hanoi.h”

bool CheckDone(int tower[NumPins][NumDisks]);
```

display.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include “hanoi.h”

void DisplayTower(int tower[NumPins][NumDisks])
{
…
}
```

Example

hanoi.h:

```
#define NumDisks 6
#define NumPins 3
```

hanoi.c:

```
#include “hanoi.h”

bool CheckDone(int tower[NumPins][NumDisks]);

#if 0
bool DebugStuff(int x, int b[NumPins]);
#endif
```

What would happen?

hanoi.h:

```
#define NumDisks 6
#define NumPins 3
```

solve.h:

```
#include “hanoi.h”

void Autosolve(int
tower[NumPins][NumDisks]);
```

hanoi.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include “solve.h”
#include “hanoi.h”

bool CheckDone(int tower[NumPins][NumDisks]);
```

Include Guards

hanoi.h:

```
#ifndef HANOI_H
#define HANOI_H

#define NumDisks 6
#define NumPins 3

#endif
```

solve.h:

```
#ifndef SOLVE_H
#define SOLVE_H

#include “hanoi.h”

void Autosolve(int
tower[NumPins][NumDisks]);

#endif
```

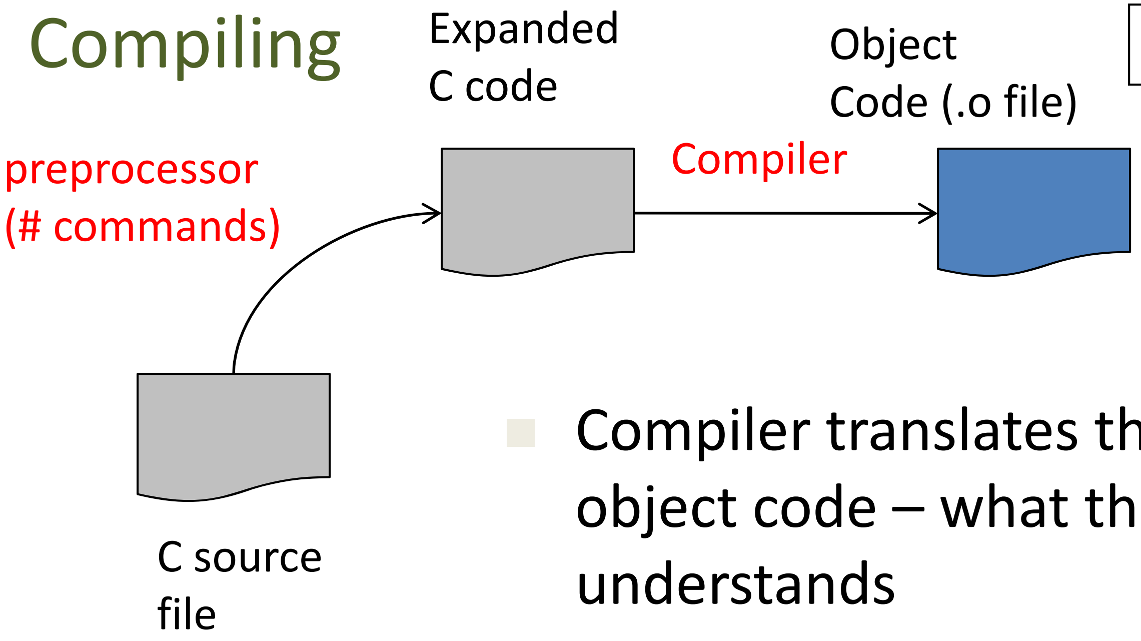
hanoi.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include “solve.h”
#include “hanoi.h”

bool CheckDone(int tower[NumPins][NumDisks]);
```

Include Guards: Conditional compilation code that protects a section of code in a header from being compiled more than once.

Compiling

```
gcc –c code.c
```



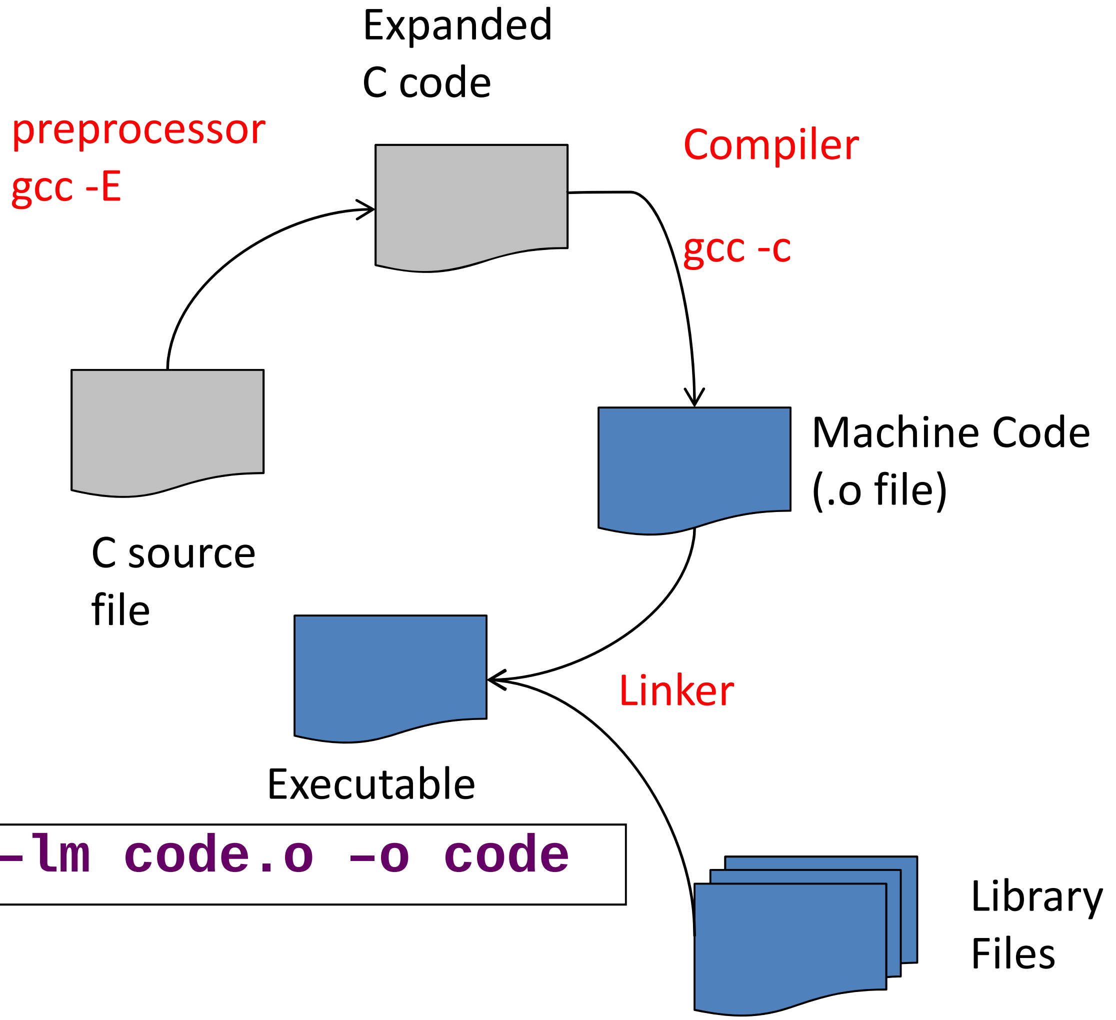
- Compiler translates the C source code into object code – what the machine actually understands
  - Machine-specific
- Each line represents either a piece of data or a machine-level instruction
  - To create the assembly code:

```
gcc -c preproc.c
```

Linking

- Object file *may* not be directly executable
  - Missing some parts
  - Still has some names to be resolved
- The linker (ld) takes multiple object files (.o) and puts them together into one executable file
  - Resolves references to calls from one file to another
- Linking is important as it allows multiple, separate files to be brought together

Linking



```
gcc -lm code.o -o code
```

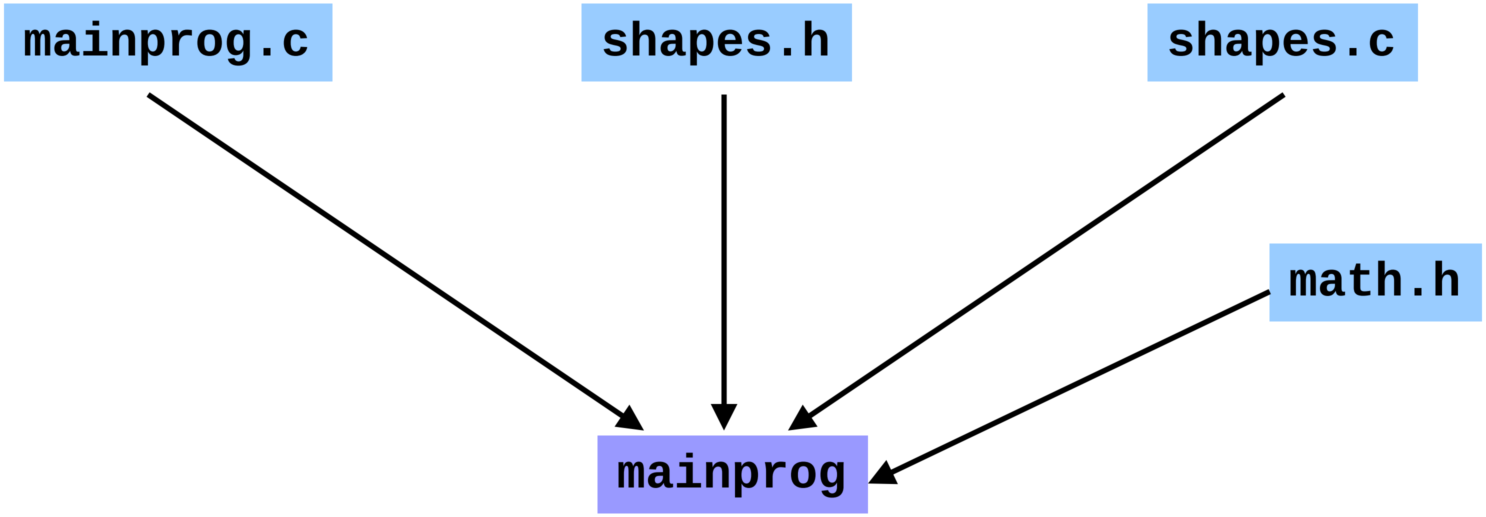
Separate compilation

- For large, complex programs, it is a good idea to break your files into separate .c and .h files

- Can debug each separately, then reuse

- Can distribute pieces to other so they can incorporate into their code without changing their code (just link it in)

Example (Separate Compilation)



mainprog.c

shapes.h

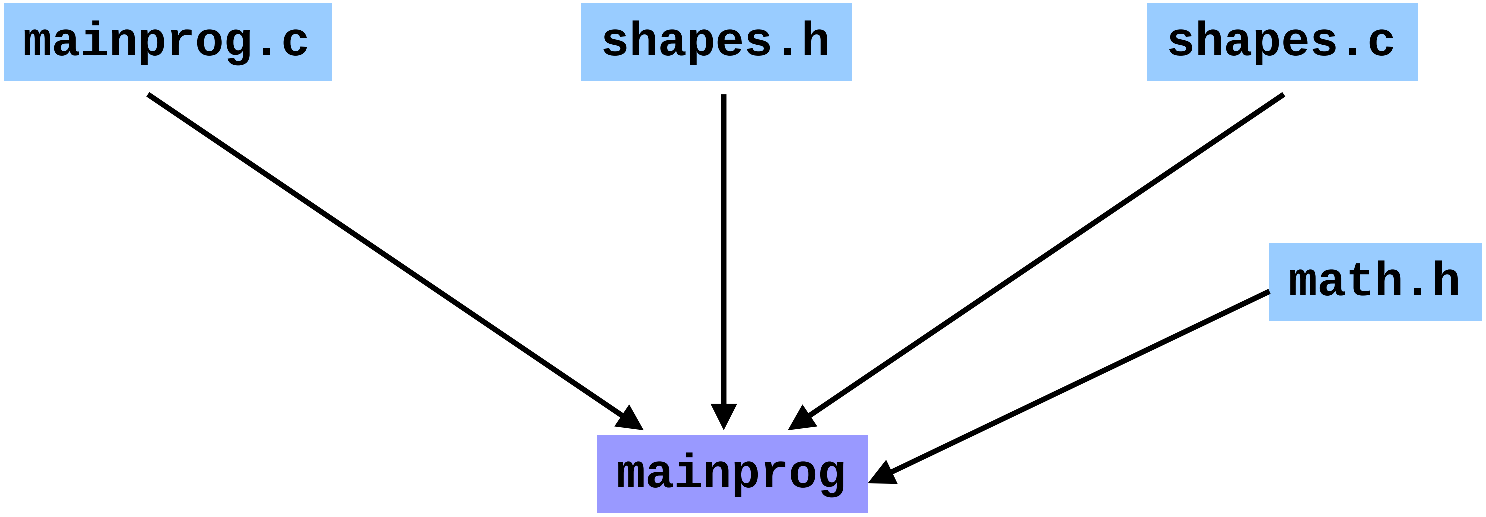
shapes.c

math.h

mainprog

```
gcc -c shapes.c
gcc –c mainprog.c
gcc –lm mainprog.o shapes.o –o mainprog
```

Another way (Compile all at once)



mainprog.c

shapes.h

shapes.c

math.h

mainprog

```
gcc –lm shapes.c mainprog.c –o mainprog
```

Controlling compilation/linking better

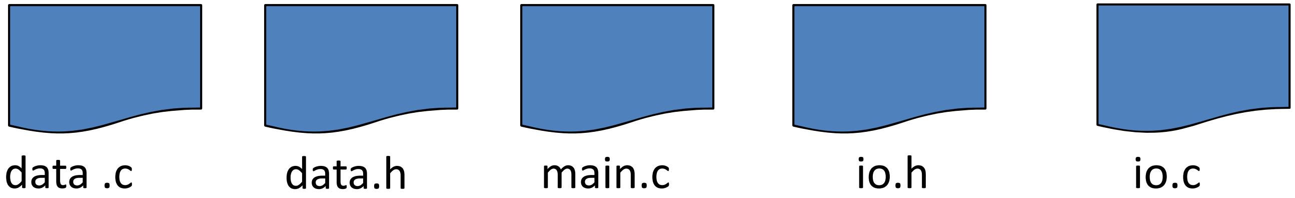
- We need a better way to control compilation
  - too much detail to type each time
- We can save time by only updating (compiling) the parts that need updating
  - if a .o file is OK (unchanged source), leave it. If a .c file is changed, the .o file needs to be regenerated then everything relinked

Makefiles

- Suppose you have a program, which is made up of these 5 source files:

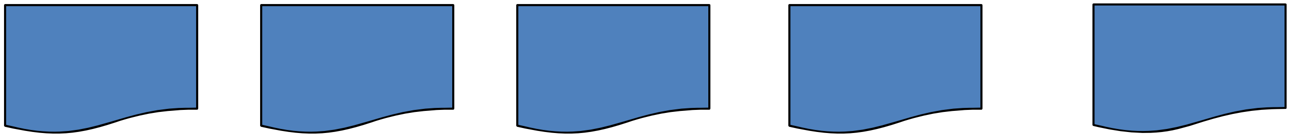
main.c	(include data.h, io.h)
data.c	(include data.h)
data.h	
io.c	(include io.h)
io.h	

How to create the executable?



```
gcc -c data.c       → data.o
gcc -c io.c         → io.o
gcc -c main.c       → main.o
gcc main.o io.o data.o –o executable
```

How to create the executable?



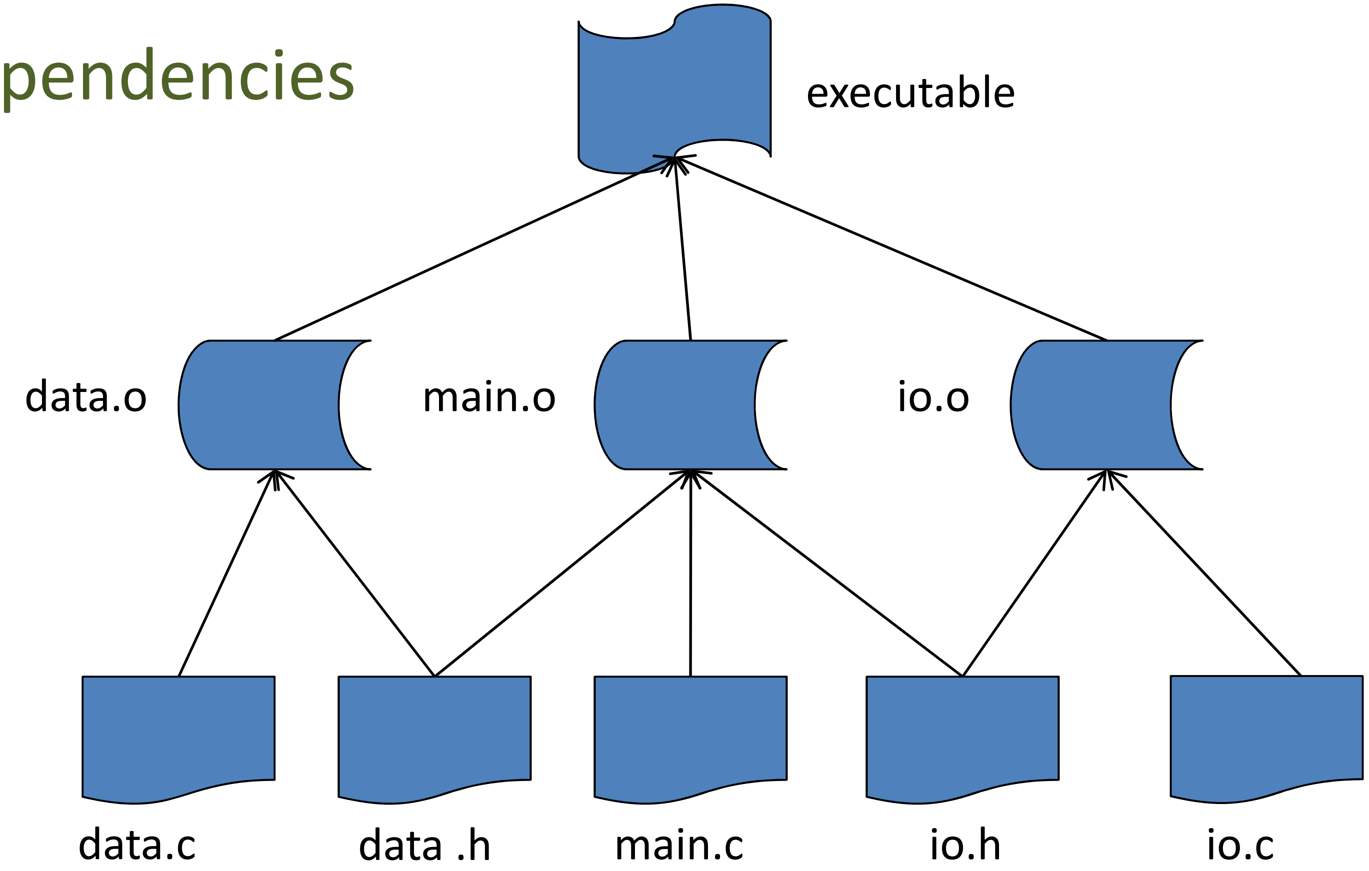
data .c data.h main.c io.h io.c

```
gcc -c data.c
gcc -c io.c
gcc -c main.c
gcc main.o io.o data.o –o executable
```

What if you modify data.h?

Do you need to re-run gcc -c on io.c and main.c?

Dependencies

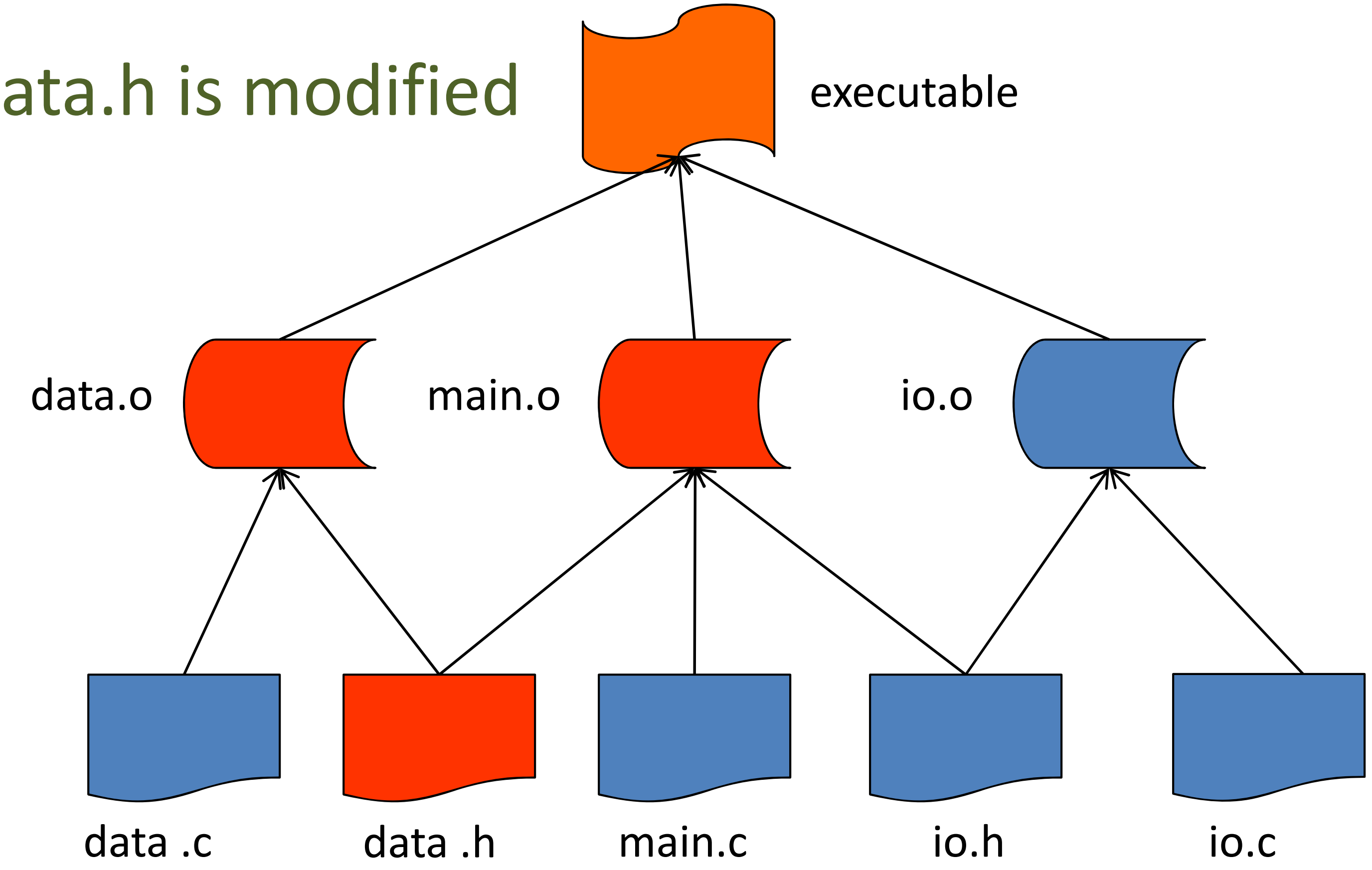


What is a Makefile?

- A Makefile is used with the ***make*** utility to determine which portions of a program to compile.

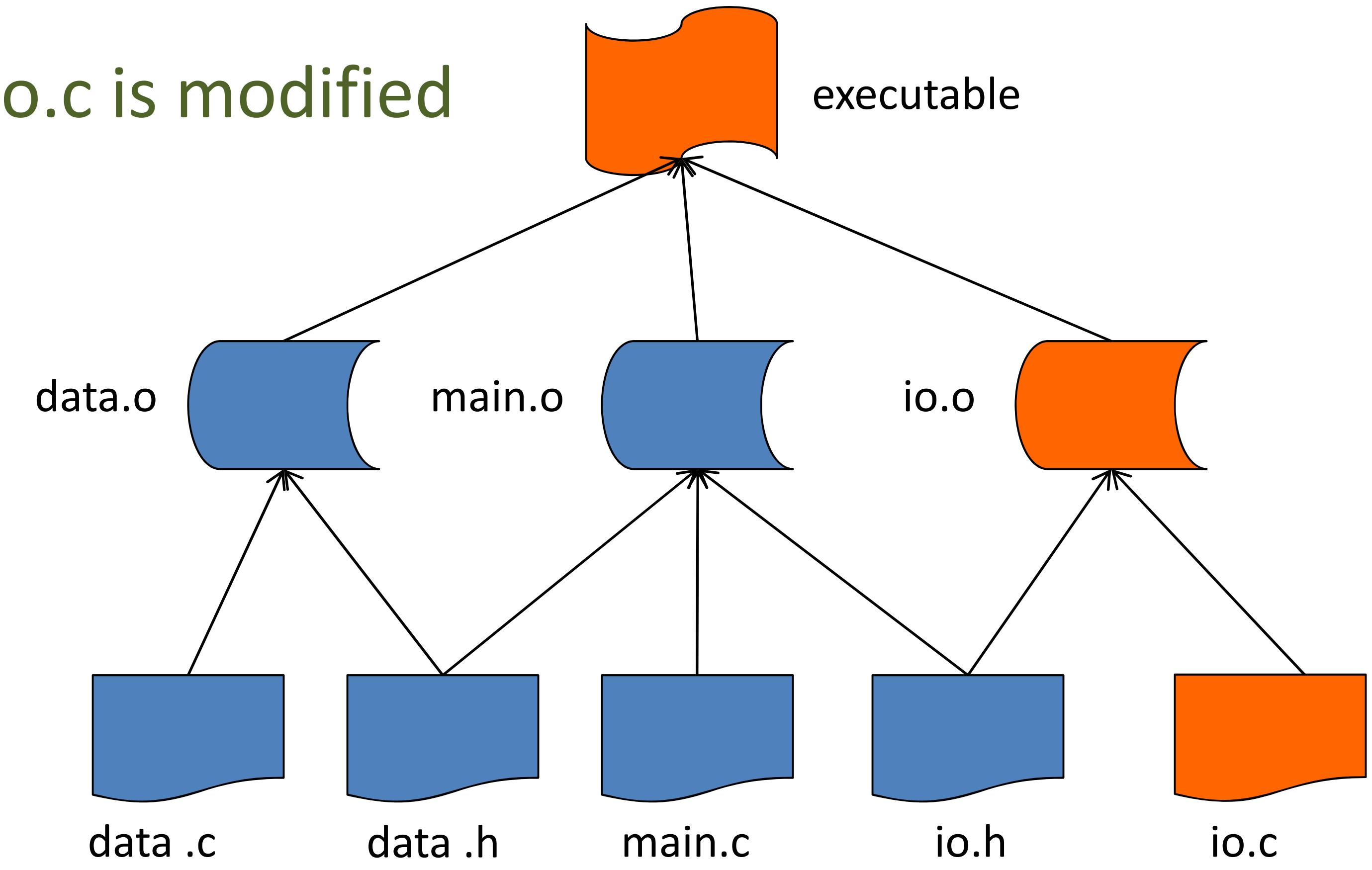
- It is basically a script that guides the make utility to choose the appropriate program files that are to be compiled and linked together

If data.h is modified



No need to re-create data.o and main.o

If io.c is modified



No need to re-create data.o and main.o

hanoi.h:

```
#ifndef HANOI_H
#define HANOI_H

#define NumDisks 6
#define NumPins 3

#endif
```

solve.h:

```
#ifndef SOLVE_H
#define SOLVE_H

#include “hanoi.h”

void Autosolve(int
tower[NumPins][NumDisks]);

#endif
```

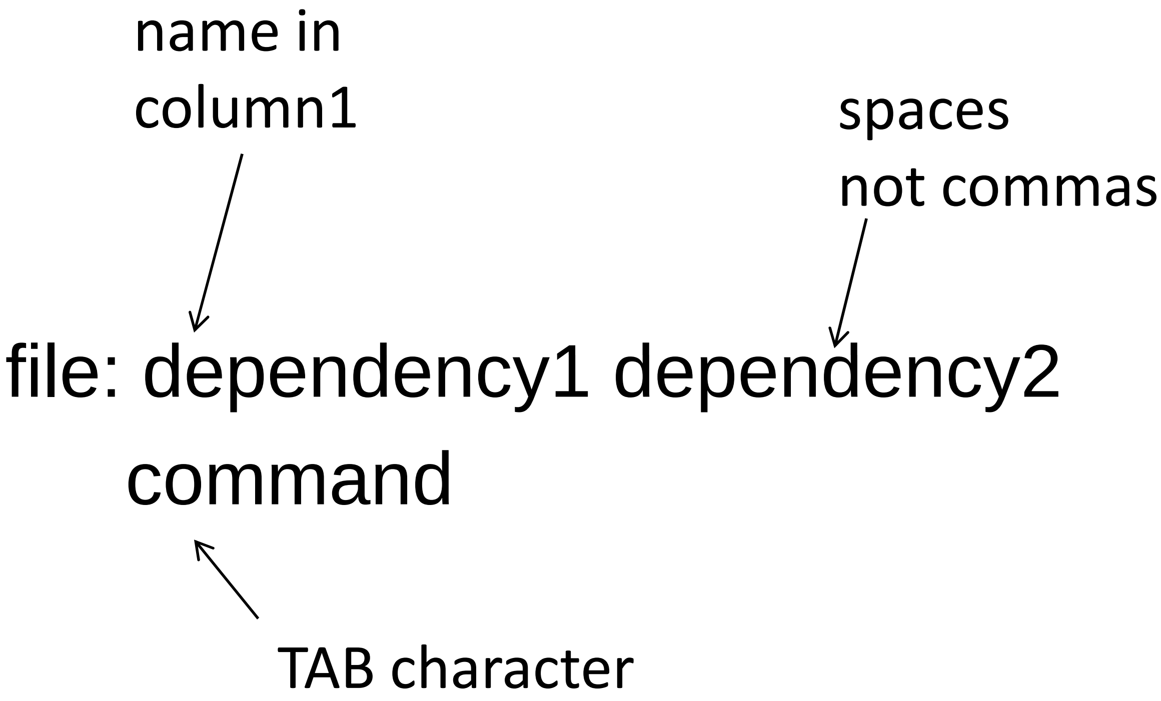
hanoi.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include “solve.h”
#include “hanoi.h”

bool CheckDone(int tower[NumPins][NumDisks]);
```

display.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include “hanoi.h”

bool DisplayTower(int tower[NumPins][NumDisks]);
```

What is in a Makefile?



name in column1

spaces not commas

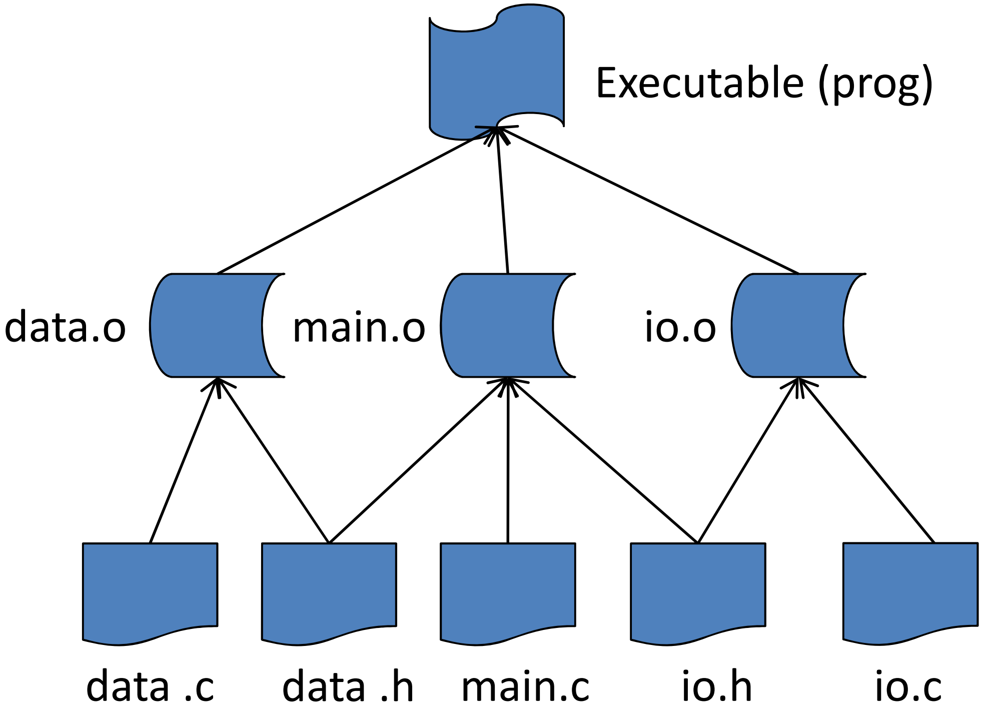
```
file: dependency1 dependency2
	command
```

TAB character

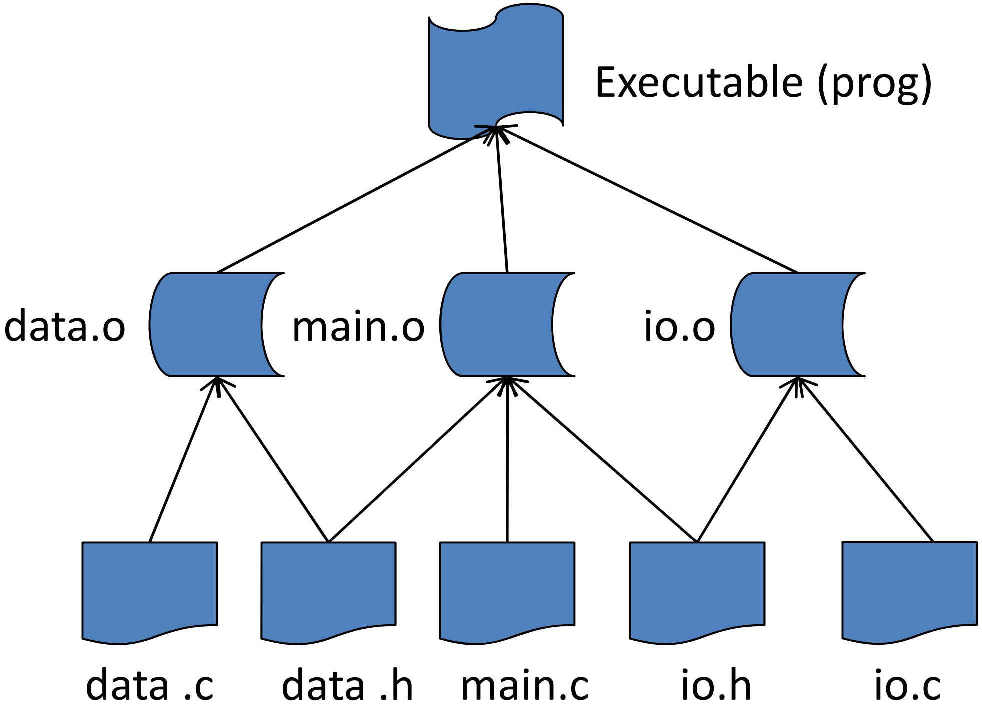
What it means

- The first line indicates what a file depends on. That is, if any of the dependencies change, then that file must be updated
  - What updating means is defined in the command listed below the dependency
- Don’t forget the TAB character to begin the second line

Dependency graph



Creating a Makefile

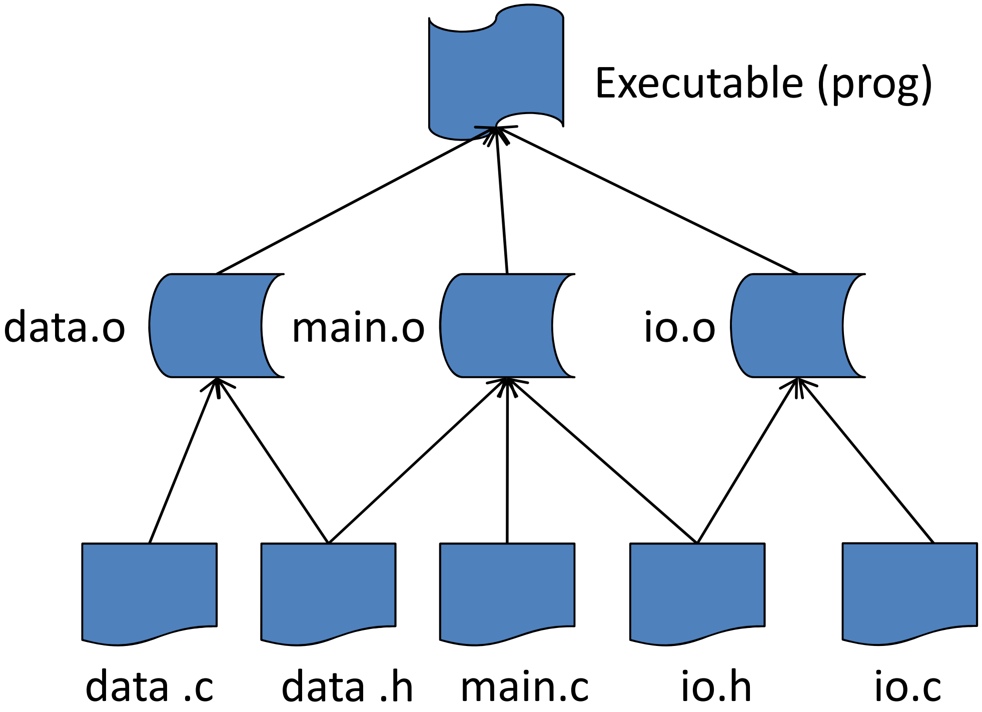


Makefile:

```
prog:

data.o:

main.o:

io.o:
```

List the object (*.o) and executable files that make up the program

Creating a Makefile

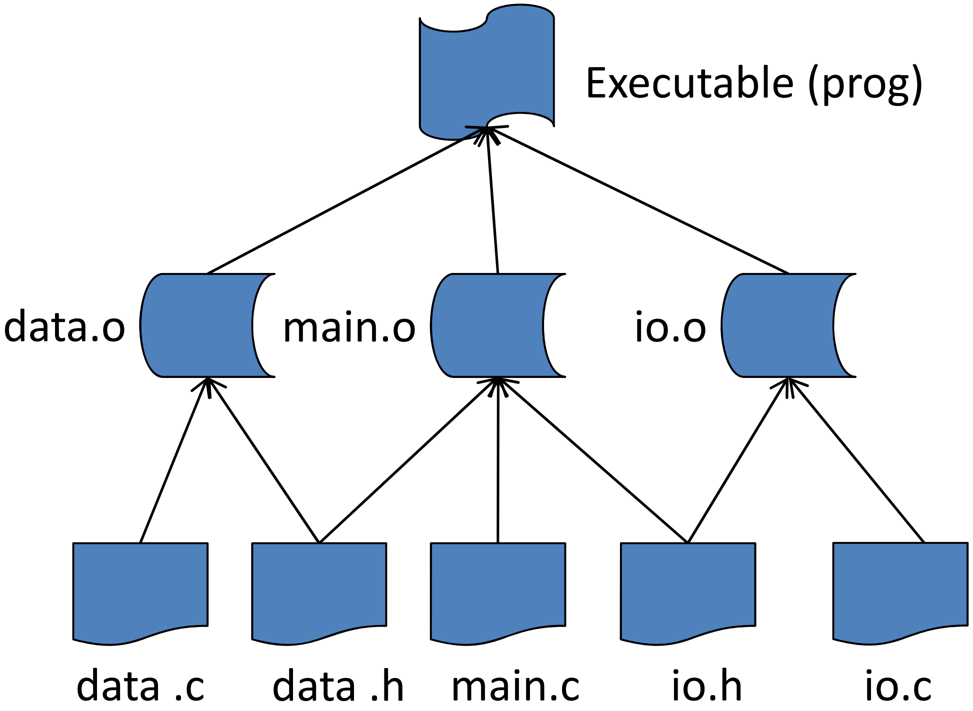


Makefile:

```
prog:  data.o main.o io.o
        gcc –o prog data.o main.o io.o

data.o: data.h data.c
        gcc –c data.c

main.o: data.h io.h main.c
        gcc –c main.c

io.o: io.h io.c
        gcc –c io.c
```

First line specifies the dependencies for each object and executable files

Creating a Makefile



Makefile:

```
prog:  data.o main.o io.o
         gcc –o prog data.o main.o io.o

data.o: data.h data.c
         gcc –c data.c

main.o: data.h io.h main.c
         gcc –c main.c

io.o: io.h io.c
         gcc –c io.c
```

Second line contains the command to execute if any of the dependence files are modified

Additional makefile “targets”

```
clean:
	rm -rf *.o example22
```

Command “make clean” calls the clean command

In this case, clears the .o files and the executable

Usage

- make
  - To create the executable
- make linkedList.o
  - do what has to be done to make that .o file, then stop
- make clean
  - run the clean command