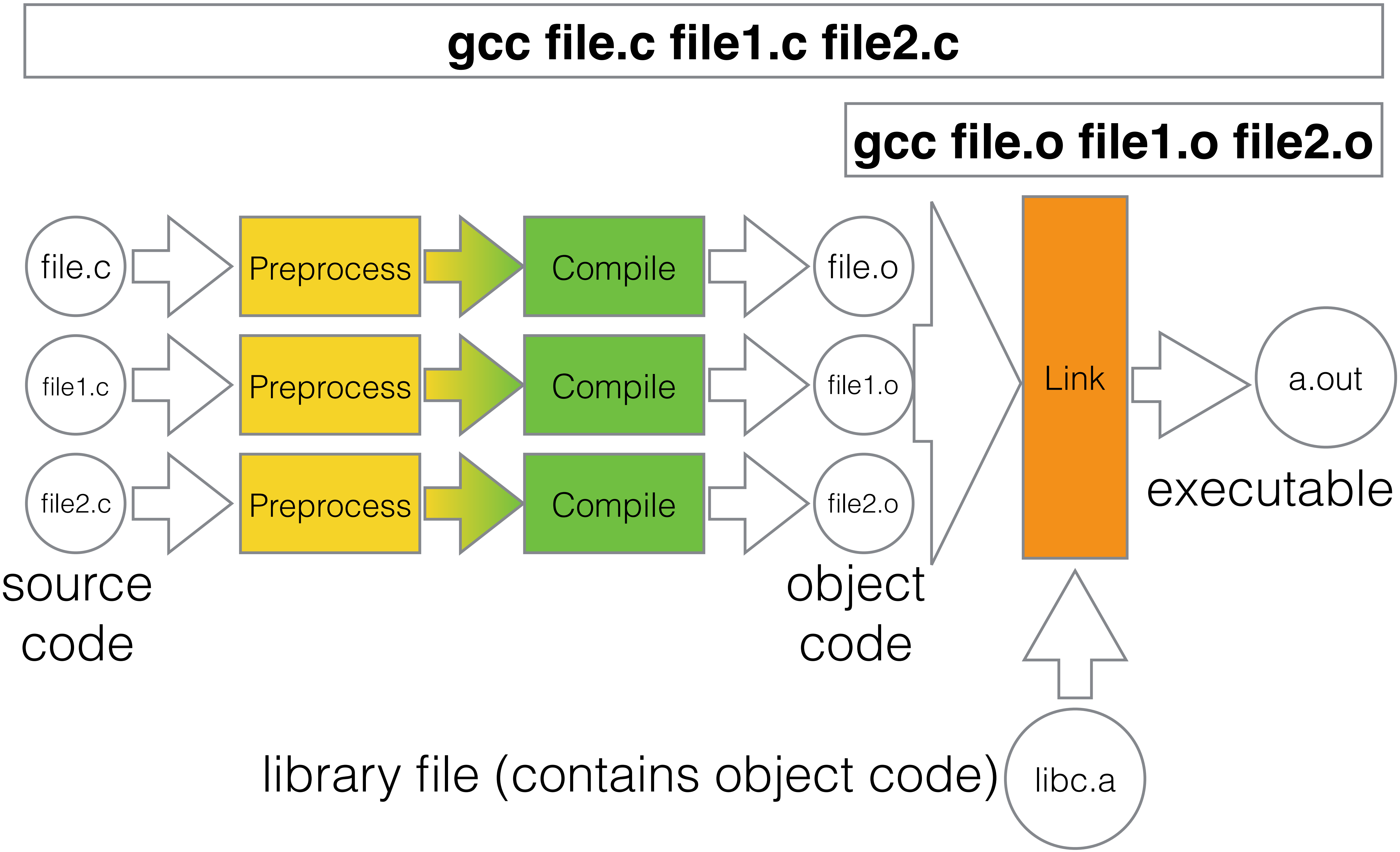
The C Build Process

(P2T: C Lecture 6)

The C Build Process



The Preprocessor

- The preprocessor “prepares” the initial source code for the compiler.
- The output is *also* C source code.
- The preprocessor performs text manipulation on the input source
  - replace text with other text
  - insert or remove text (based on a test)
  - include text from another file at this point
- All *preprocessor directives* start with a #

#define

- `#define` is the simplest preprocessor directive.
- It takes a single word "macro", and replaces it with the provided text, everywhere in the rest of the file.
- (Text inside `""` is not replaced.)

```
#define APPLE DELICIOUS FRUIT

This is an APPLE, not a SAPPLE.
"Quoted APPLE text."
```

preprocess

```
This is an DELICIOUS FRUIT, not a SAPPLE.
"Quoted APPLE text."
```

Doing this with numeric literals is one way to set global constants, like π.

```
#define PI 3.14159
```

"Predefined" macros

- C language provides some macros which will automatically be replaced with useful values by the preprocessor.
- `__LINE__` will be replaced with the line number it appears on.
- `__FILE__` is the name of the file being processed.
- `__DATE__` and `__TIME__` are replaced appropriately.

```
printf("We are on line %d in file %s.\n", __LINE__, __FILE__);
```

```
printf("We are on line %d in file %s.\n", 23, "myfile.c");
```

#ifdef ... #else ... #endif

- User defined macros can also be used to control if a piece of text is kept in, or removed from, the source code, using `#ifdef` constructs.

```
#define APPLE

#ifdef APPLE
We know about apple.
#else
We don't know about apple.
#endif
```

APPLE is now `#defined`, albeit to an empty value.

or

```
#define APPLE

#ifndef APPLE
We don't know about apple.
#else
We know about apple.
#endif
```

preprocess

```
We know about apple.
```

preprocess

If APPLE wasn't `#defined`, what would the resulting text be?

#include

- **`#include`** copies all the text from the named file, and inserts it into the text at the point of the directive. (*The preprocessor will then process the included text, if it also contains directives.*)

```
#include "mytext"

This is also some BANANA
```

mytext

```
This is the text in mytext.
It is not very interesting.
#define BANANA text.
```

preprocess

```
This is the text in mytext.
It's not very interesting.

This is also some text.
```

`#define` in mytext makes this change here

#include <> versus ""

- If you specify the filename in an `#include` like `<stdio.h>` then the preprocessor will look for the file in some standard system locations.

- If you want to refer to a file that's in the same directory as your file (for example), then use `"myfile"`. (You can give a path if you want.)

Looks for stdio.h file in system locations for important header files. (`/usr/include/` etc)

```
#include <stdio.h>

#include "myheader.h"
```

Looks for myheader.h file in local directory path (and other "local" locations).

Common definitions

- In order to allow code in one (source) file to know about functions and values in other (source) files, we need a way to add:
  - function prototypes for functions defined in other files
  - common constants etc
- We can then use those functions in our other files (and let the linker join up all the code at the end).

Header files

- C calls a file which exists just to be `#include`d to provide common declarations a "header file".
- Conventionally, they have the `.h` suffix (but they contain normal C code).
- For every `.c` file you write which contains code you want to reference in other `.c` files:
    - Make a `.h` file for the prototypes etc you want to reference elsewhere.

#include guards

- It's an error in C to declare a function (or any other name) more than once in the same file.

- If our definition is in a header, how do we stop the header being included more than once?

file.h

```
#ifndef FILE_H
#define FILE_H

(text of header)

#endif
```

The first time this is `#include`d, the `#define` activates, and defines FILE_H. The *second* time we include it, the `#ifndef` skips the entire file!

Modern compilers also support putting
`#pragma once`
at the top of a header file, for same effect. (This is *not* universally supported!)

Preprocessing example

example.c

```c
#include <stdio.h>
#include "function.h"

int main(void){

	printf("The result of function f is: %f\n", f(27));

	return 0;
}
```

What happens if we don't include function.h?
(What does the warning mean?)

function.h

```c
//declaration of f (multiplies number by 2)
double f(double);
```

function.c

```c
double f(double a) {
	return a*=2;
}
```

```
gcc -E example.c | less
```

(what is the prototype for printf?)

The Compiler

- The compiler takes C source code and translates it into actual machine code.
- Source code is broken down into syntactic units ("tokens"), and the relationships between them.
- Instructions are then converted to machine code, with "symbols" attached to named items for reference.
- The resulting machine code, annotated with symbols, and prepended with a list of all the symbols in the file, is called "object code".

Optimisation

- As with natural languages, naïve compilation (translating “word for word” into machine code) doesn’t always give the best result.
- We can ask the compiler to spend more time and effort to produce more efficient, or faster, or shorter, representations - *optimisations*.
- We specify these with the `-O` flag, specifying a value from `0` (no optimisations) through `3` (all optimisations).
- In general, `-O2` is a good balance between compilation time and code performance.
- In all cases, the logic (within the C Standard definitions) is preserved - the implementation may just be changed.

Compilation example

example.c

```
#include <stdio.h>
#include "function.h"

int main(void){

        printf("The result of function f is: %d\n", f(27));

        return 0;
}
```

function.h

```
//declaration of f (multiplies number by 2)
int f(int);
```

function.c

```
int f(int a) {
        return a*=2;
}
```

```
gcc -c example.c
gcc -c function.c

nm example.o
nm function.o
```

The Linker

- The Linker is responsible for joining up the object code from the Compiler, so that all symbols are mapped to their representation.
- Firstly, it takes all of the object code files it is given (one for each C source file), and turns them into one large file, with "consolidated" symbol index at the top.
- Before object code is linked, function calls are simply a note of the symbol that corresponds to the function they want to call.
- The linker looks up the symbol in the symbol index and replaces it with a call to the function code with the corresponding symbol.
- This also happens for variable names in the file scope in each file.

Libraries

- One way to provide a set of useful functions to others is to package them all up into a "library".

- This is just a specially packaged set of object code (with an index to make it easy to find things in it).

- Libraries have names ending in `.a` or `.so` (the difference is in how and when they are linked).

- We can ask the linker to link to a library called *libname.a* (or *libname.so*) with the option `-lname`

- In this case, the linker will also match symbols in our code with symbols in the library mentioned.

The C Standard Library

- One library is used in (almost) all C code you will ever write: `libc.a` (or `libc.so`).
- The "C Standard Library" contains useful functions for a large number of potential applications, including I/O etc.
- It's so large that the prototypes for its functions are split into multiple header files, grouped by topic.
    - `stdio.h` provides prototypes for the I/O functions in libc
    - `string.h` provides prototypes for the string functions in libc
- As libc is so commonly used, the linker will link it without even being asked.

An exception is the (floating point) mathematics part of libc. For historical reasons, this is often stored in a separate library libm.a / libm.so , which *does* need explicitly linked.

Library locations

- Just as the preprocessor looks in "standard locations" for header files quoted in `< >`, the linker looks in standard locations for libraries.

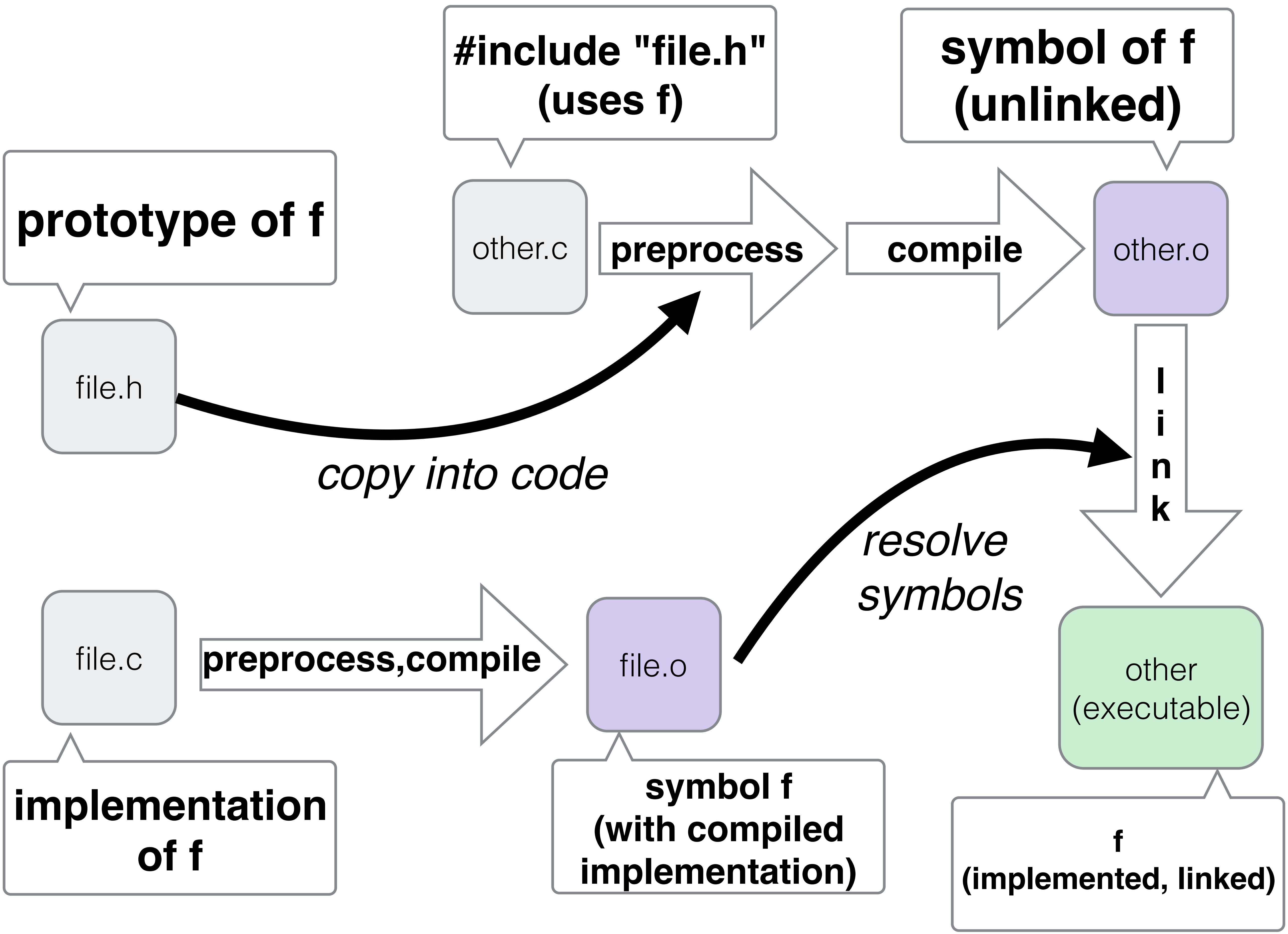
- To add another location to the list to be searched, use `-L/path/to/new/location`

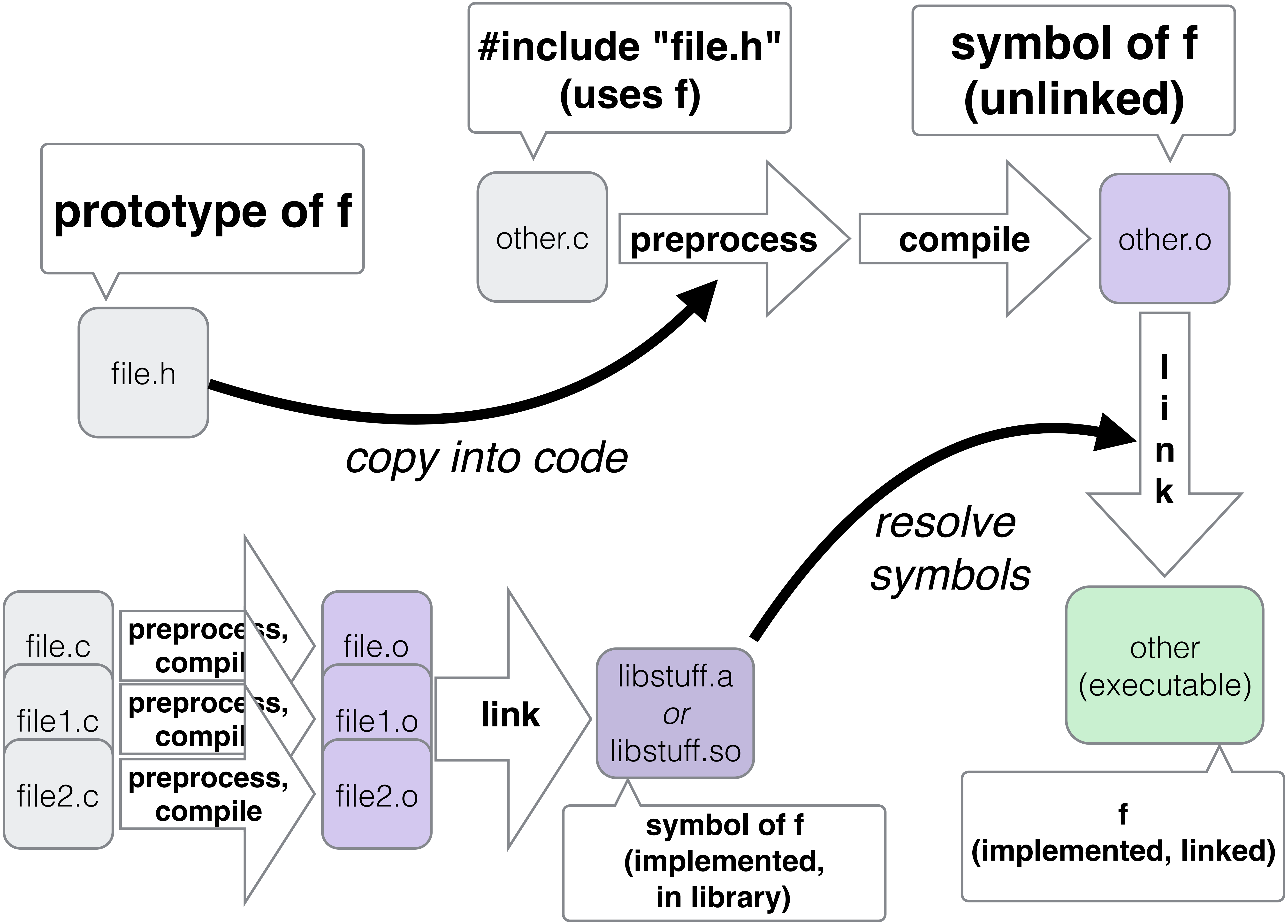
  For example, to link a library with path

  /home/myaccount/lib/libmylibrary.a

  use

  `-L/home/myaccount/lib/ -lmylibrary`


#include "file.h"
(uses f)
symbol of f
(unlinked)
prototype of f
other.c
preprocess
compile
other.o
file.h
l
i
n
k
copy into code
resolve
symbols
file.c
preprocess,compile
file.o
other
(executable)
implementation
of f
symbol f
(with compiled
implementation)
f
(implemented, linked)



#include "file.h" (uses f)
symbol of f (unlinked)
prototype of f
other.c
preprocess
compile
other.o
file.h
copy into code
l i n k
resolve symbols
file.c
file1.c
file2.c
preprocess, compile
preprocess, compile
preprocess, compile
file.o
file1.o
file2.o
link
libstuff.a or libstuff.so
symbol of f (implemented, in library)
other (executable)
f (implemented, linked)

Libraries and Linking example

example.c

```
#include <stdio.h>
#include "function.h"

int main(void){

        printf("The result of function f is: %d\n", f(27));

        return 0;
}
```

function.h

```
//declaration of f (multiplies number by 2
int f(int);
```

function.c

```
int f(int a) {
        return a*=2;
}
```

`gcc example.c function.c`

or (if .o files exist already)

`gcc example.o function.o`

`nm a.out`

(where does printf symbol get resolved?)

`ldd a.out`

Libraries and Linking example

example.c

```
#include <stdio.h>
#include "function.h"

int main(void){

        printf("The result of function f is: %d\n", f(27));

        return 0;
}
```

replace with `#include "function2.h"`

Need to explicitly link libm.a/libm.so
(name is always the bit between "lib" and .a/.so)

```
gcc example.c function2.c -lm

ldd a.out
```

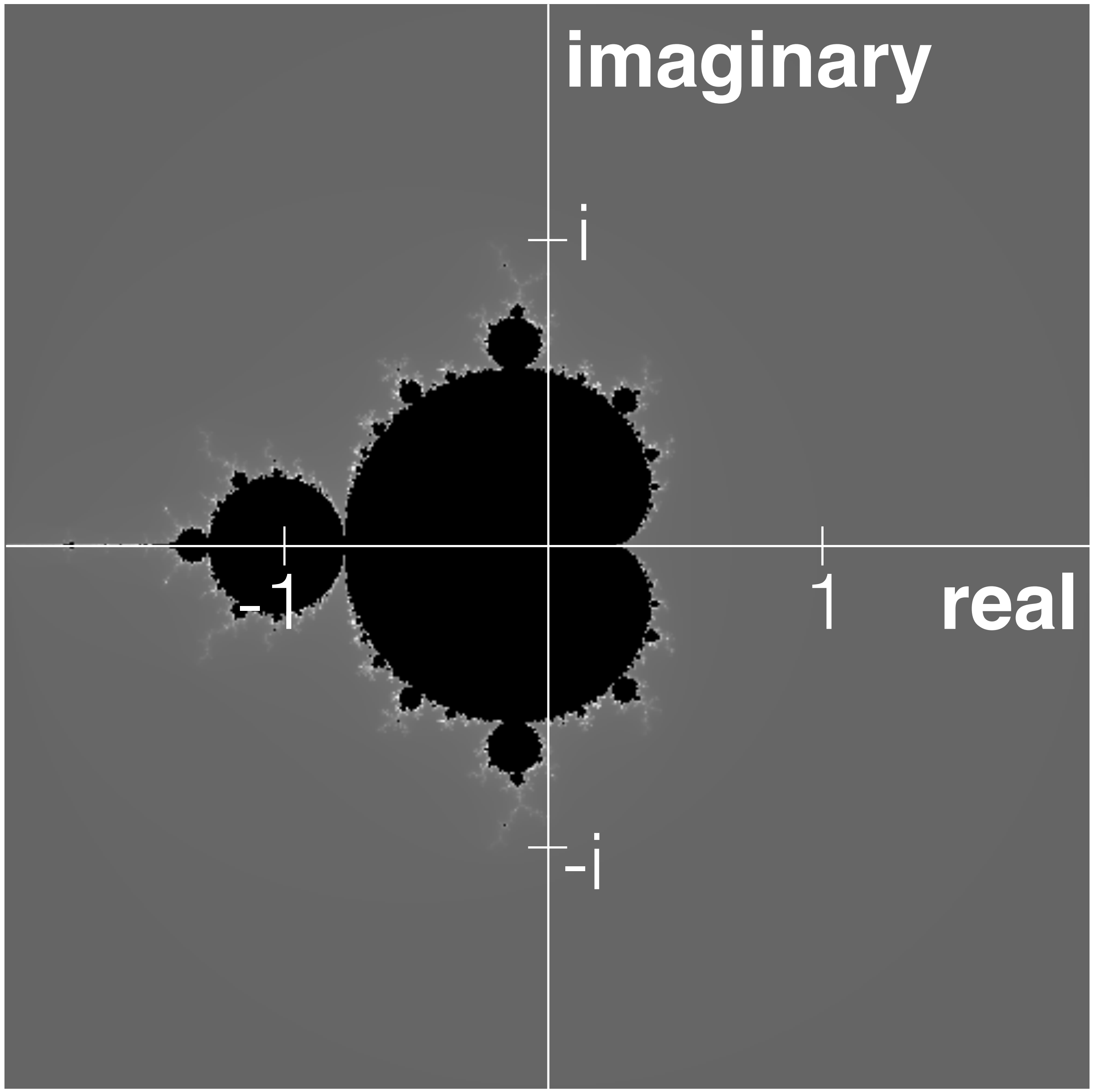
function2.h

```
//function returns cosine of argument
double f(double);
```

function2.c

```
#include <math.h>

double f(double a){
        //cos function declared in math.h, implemented in libm.a
        return cos(a);
}
```



The Mandelbrot set is generated by:

for each point in the complex plane, c

start with $z_0 = 0$
repeat:
$z_{n+1}=z_n^2+c$

if z_n tends to infinity, colour grey based on how fast it does so.

if it doesn't, colour black (is in the set)

This example is not examinable.

mandel.c

```
#include <stdio.h>
#include <complex.h>
#include <math.h>
#include <png.h>
#include <stdint.h>
#include "initpng.h"

#define DIM 500

uint8_t mandel_kernel(double complex c);

int main(void) {

        uint8_t array[DIM][DIM];
        double complex c;

        for(int i=0; i<DIM;i++){
                for (int j=0;j<DIM;j++){
                        //align on range -2..2 in x and y
                        c = (4*(double)i/DIM)-2 + ((4*(double)j/DIM)-2)*I;
                        array[i][j] = mandel_kernel(c);
                }
        }

        FILE *fp = fopen("mandel.png", "wb");

        png_structp png = initpng(fp,DIM);

        uint8_t * rows[DIM];
        for(int i=0;i<DIM;i++) {
                rows[i] = (uint8_t *)array[i];
        }

        png_write_image(png, rows);
        png_write_end(png, NULL);

        fclose(fp);

}
```

complex.h **header for complex numbers**
math.h **for complex math functions**
png.h **for writing png image**
stdint.h **for "exactly 8 bit wide" ints**
initpng.h **for own code to initialise a png**

exactly 8 bit wide integer

floating point complex number!

get "rows" of array, as ptrs

mandelbrot set calculation

```
uint8_t mandel_kernel(double complex c) {
        uint8_t i = 100;
        double complex z = 0 + 0*I;
        while(cabs(z)<2.0 && i>0){
                z = cpow(z,2) + c;
                i++;
        }
        return i;
}
```

cpow raises z to the power 2 (efficiently)
cabs gives the absolute value of z

```
gcc mandel.c initpng.o -lm -lpng
```

link to libm.so (for math)
link to libpng.so (for png creation)

Further reading

- You can read more about the preprocessor (and more features which are not covered in the course), here:

- https://gcc.gnu.org/onlinedocs/cpp/

- You can read a bit more about compiler optimisation here:

- http://moodle2.gla.ac.uk/mod/page/view.php?id=149957

References

- The documentation for the (GNU) implementation of the C Standard Library:
- https://www.gnu.org/software/libc/manual/
- (We have not covered a lot of the contents!)