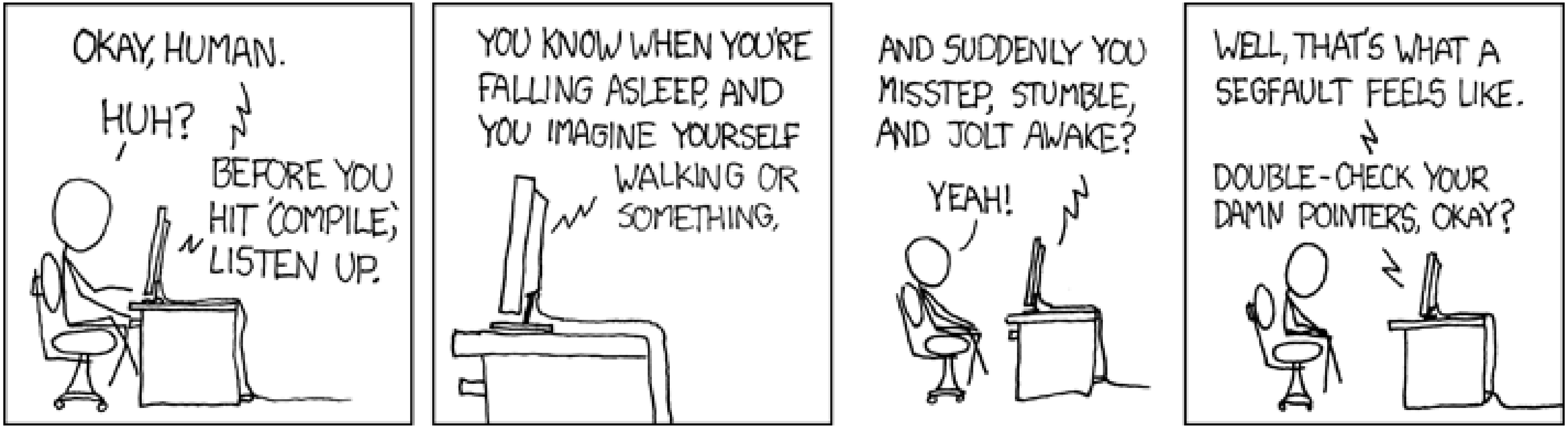
P2T 2019 – C Lecture 6
The C Build Process


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Dr Gordon Stewart

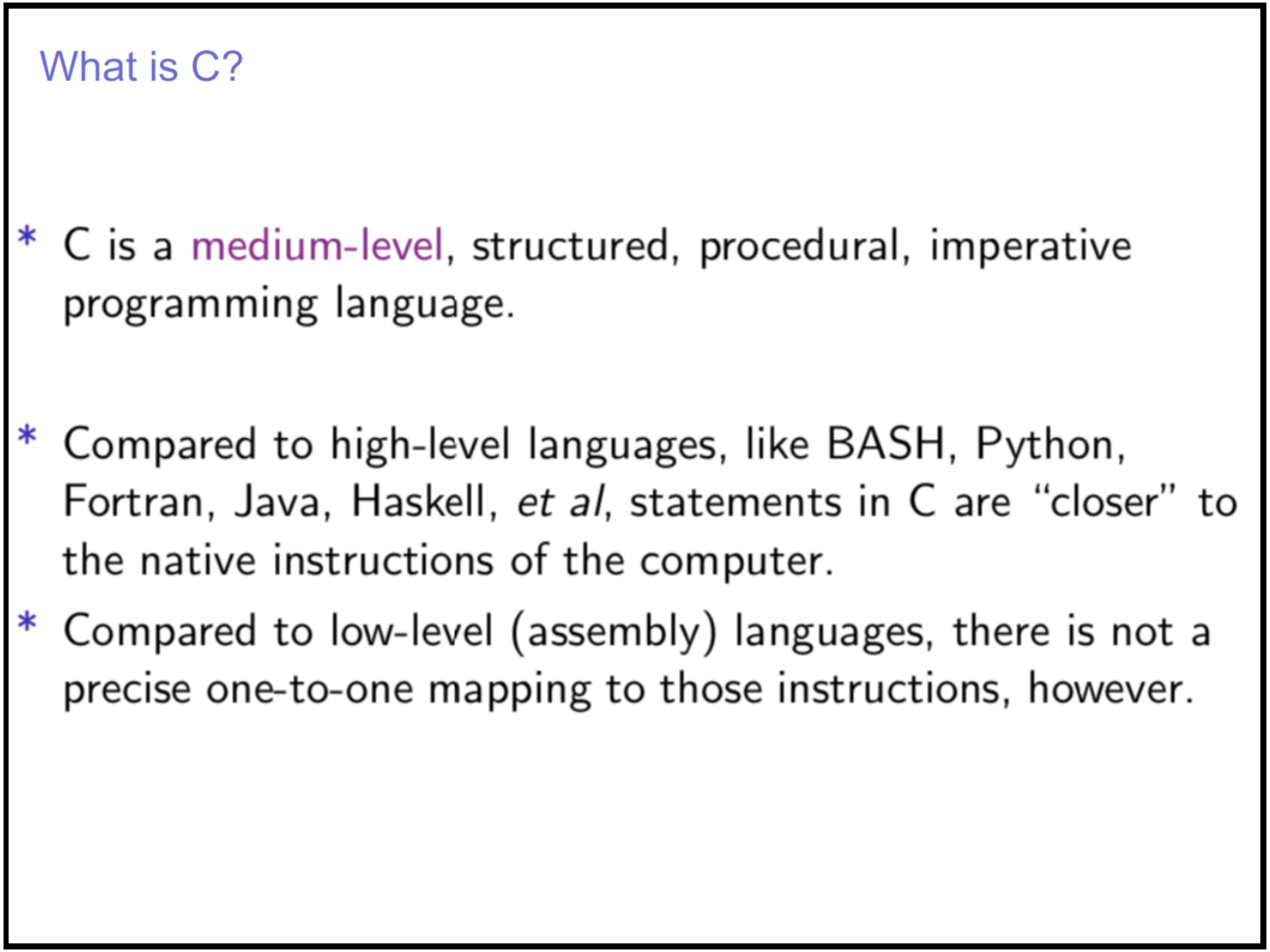
Room 427, Kelvin Building

gordon.stewart@glasgow.ac.uk / Extension 6439

Source material: Prof David Britton / Dr Sam Skipsey

https://xkcd.com/371/

Introduction

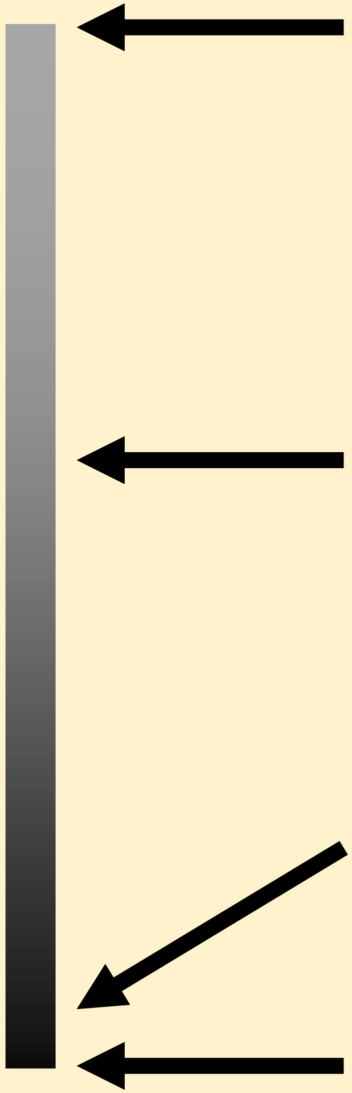


What is C?

* C is a medium-level, structured, procedural, imperative programming language.

* Compared to high-level languages, like BASH, Python, Fortran, Java, Haskell, *et al*, statements in C are “closer” to the native instructions of the computer.
* Compared to low-level (assembly) languages, there is not a precise one-to-one mapping to those instructions, however.

[Background]



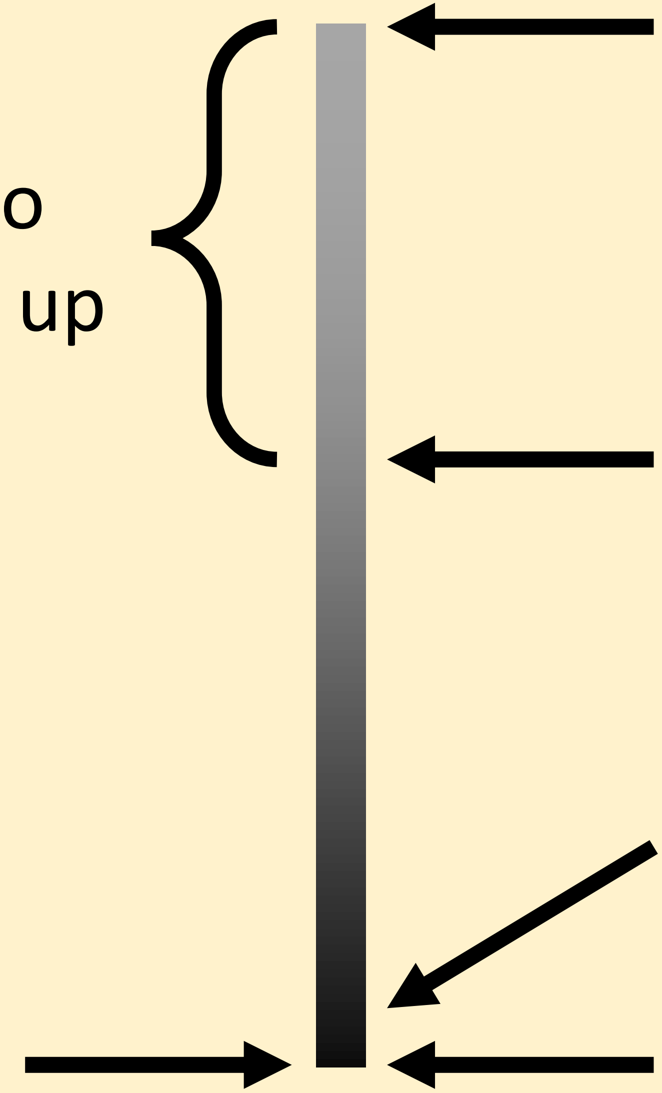
High-level languages: Python, Java

Medium-level languages: C

Low-level languages: Assembly

Machine code

[Background]



High-level languages: Python, Java

We like to program up here

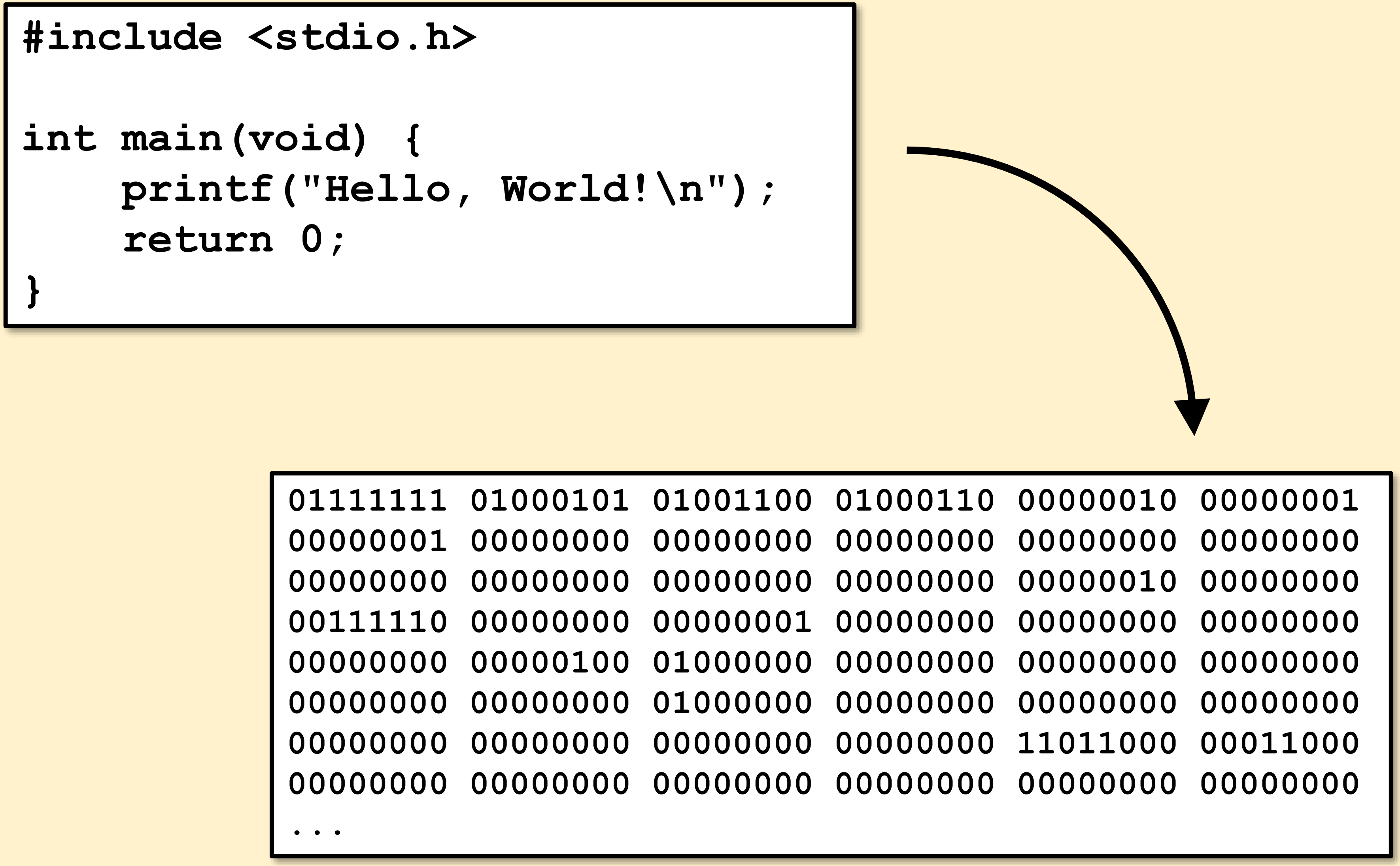
Medium-level languages: C

Low-level languages: Assembly

The CPU runs this

Machine code

[Background]



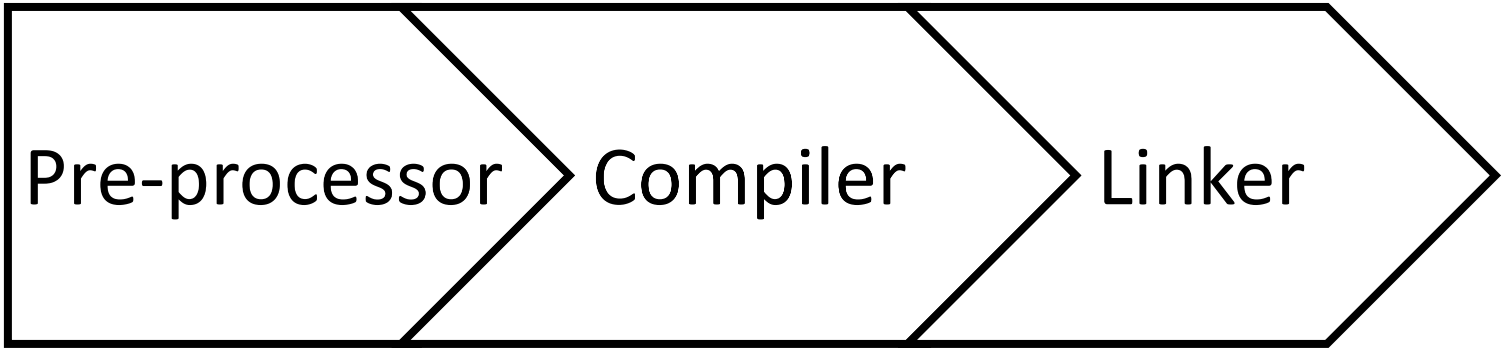
```
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

```
01111111 01000101 01001100 01000110 00000010 00000001
00000001 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000010 00000000
00111110 00000000 00000001 00000000 00000000 00000000
00000000 00000100 01000000 00000000 00000000 00000000
00000000 00000000 01000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 11011000 00011000
00000000 00000000 00000000 00000000 00000000 00000000
...
```

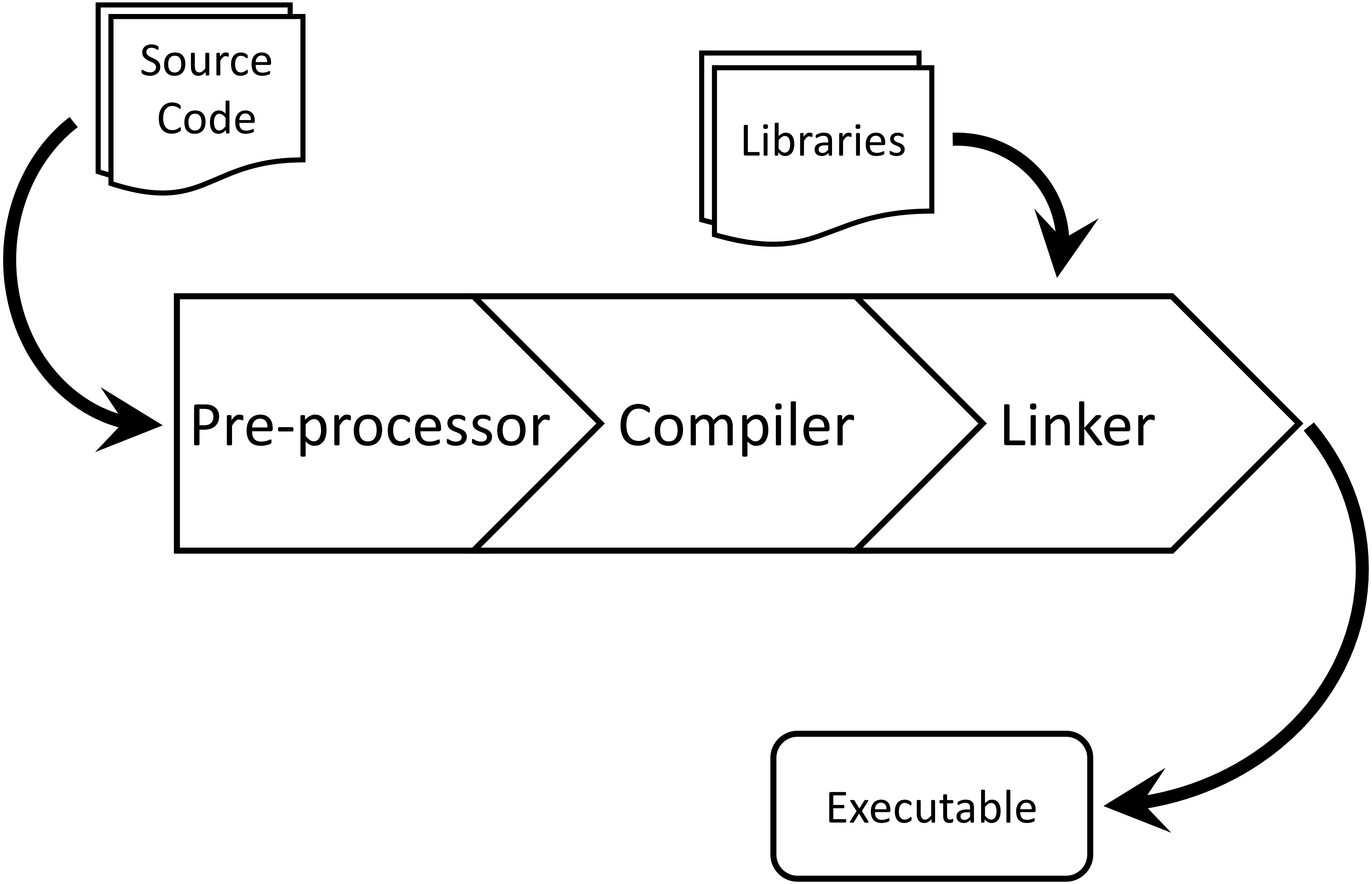
[Background: `gcc`]

- The GNU Compiler Collection
- Introduced in 1987
- **`gcc`** is actually a wrapper for various other applications
- Now supports multiple programming languages: C, C++, Fortran, Java, Ada, Go…
- Supports various target processor architectures: x86, x86-64, ARM, PowerPC, Motorola 68000…
- Other compilers are available, but the overall way in which they work is the same

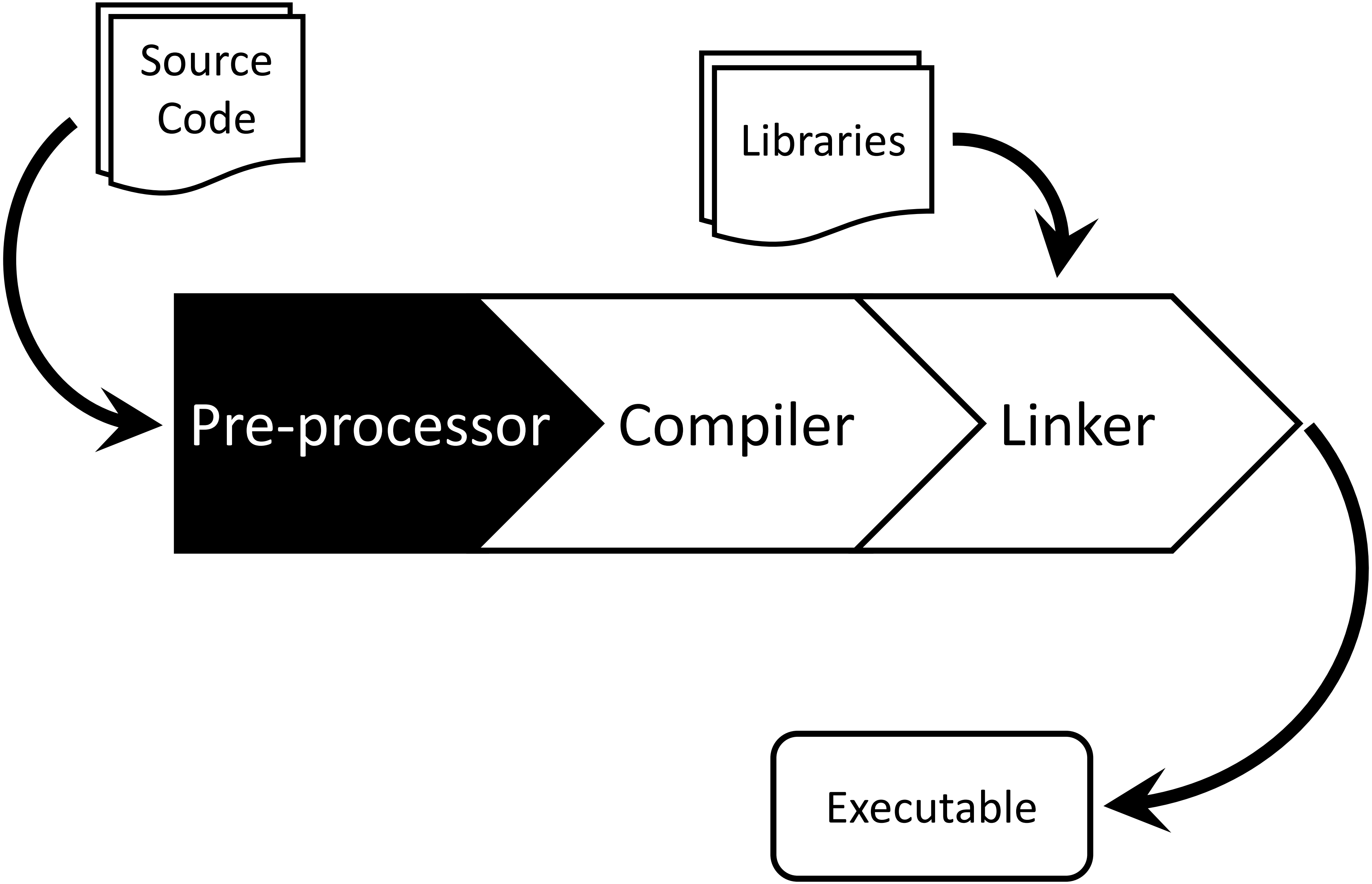
The C Build Process



The C Build Process



The C Build Process



The Pre-processor

- The pre-processor prepares source code for the compiler

- It performs text manipulation on the source code:

  - Replace some text with other text
  - Insert or remove text, based on the result of a conditional test
  - Include text from another file

- Pre-processor output is also C source code

- Pre-processor commands (called **directives**) start with **#**:

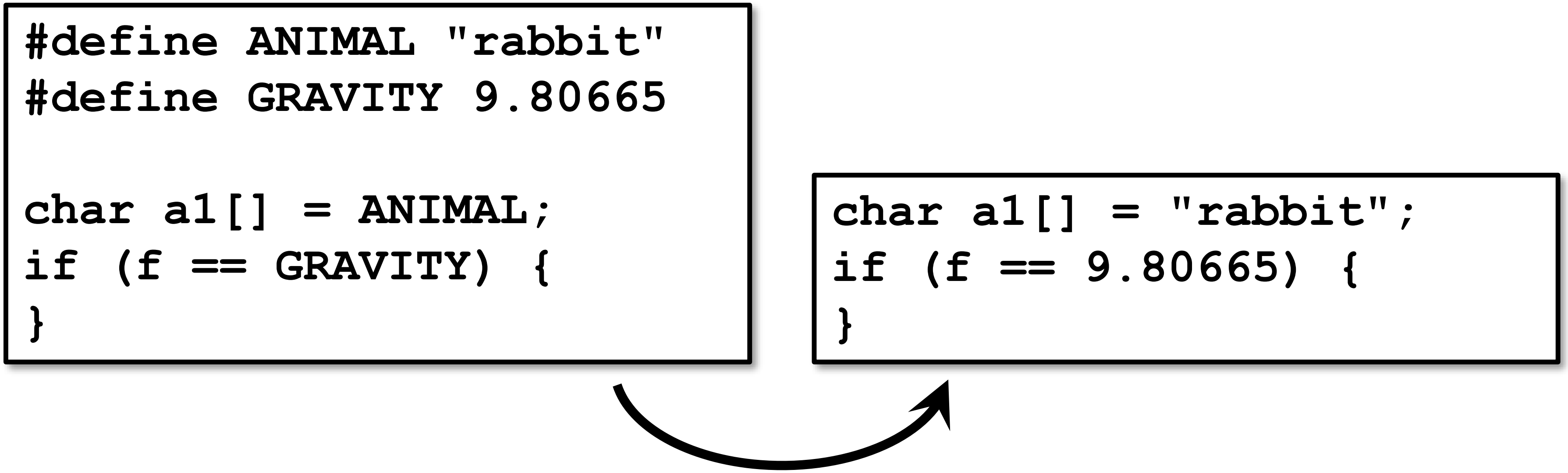
```
#include <stdio.h>
```

Pre-processor: `#define`

- **`#define`** replaces text with something else
    - Similar to “find and replace” in a word processor

- Replaces a single word **macro** with the provided text throughout the rest of the file (text within `"   "` is not replaced)

- This is one way to set constants



```c
#define ANIMAL "rabbit"
#define GRAVITY 9.80665

char a1[] = ANIMAL;
if (f == GRAVITY) {
}
```

```c
char a1[] = "rabbit";
if (f == 9.80665) {
}
```

Pre-processor: `#define`

- C provides some macros which are automatically replaced with useful values

- `__LINE__` will be replaced with the number of the line on which it appears

- `__FILE__` will be replaced with the name of the file in which it appears

- `__DATE__` and `__TIME__` will be replaced appropriately

```
printf("This is line %d of file %s", __LINE__, __FILE__);
```

↓

```
printf("This is line %d of file %s", 123, "filename.c");
```

[Background: cpp]

- To run just the pre-processor: **`cpp FILENAME`**
- Example: **`cpp define.c`**

```
#define ANIMAL "rabbit"
#define ANSWER 42

int main(void) {
    char animal1[] = ANIMAL;   // This will change...
    char animal2[] = "ANIMAL"; // ...but this won't

    int number = 5;
    if (number == ANSWER) {
        // Do something...
    }

    printf("This is line %d of file %s, last saved on %s at %s\n",
__LINE__, __FILE__, __DATE__, __TIME__);

    return 0;
}
```

Pre-processor: **`#ifdef`**

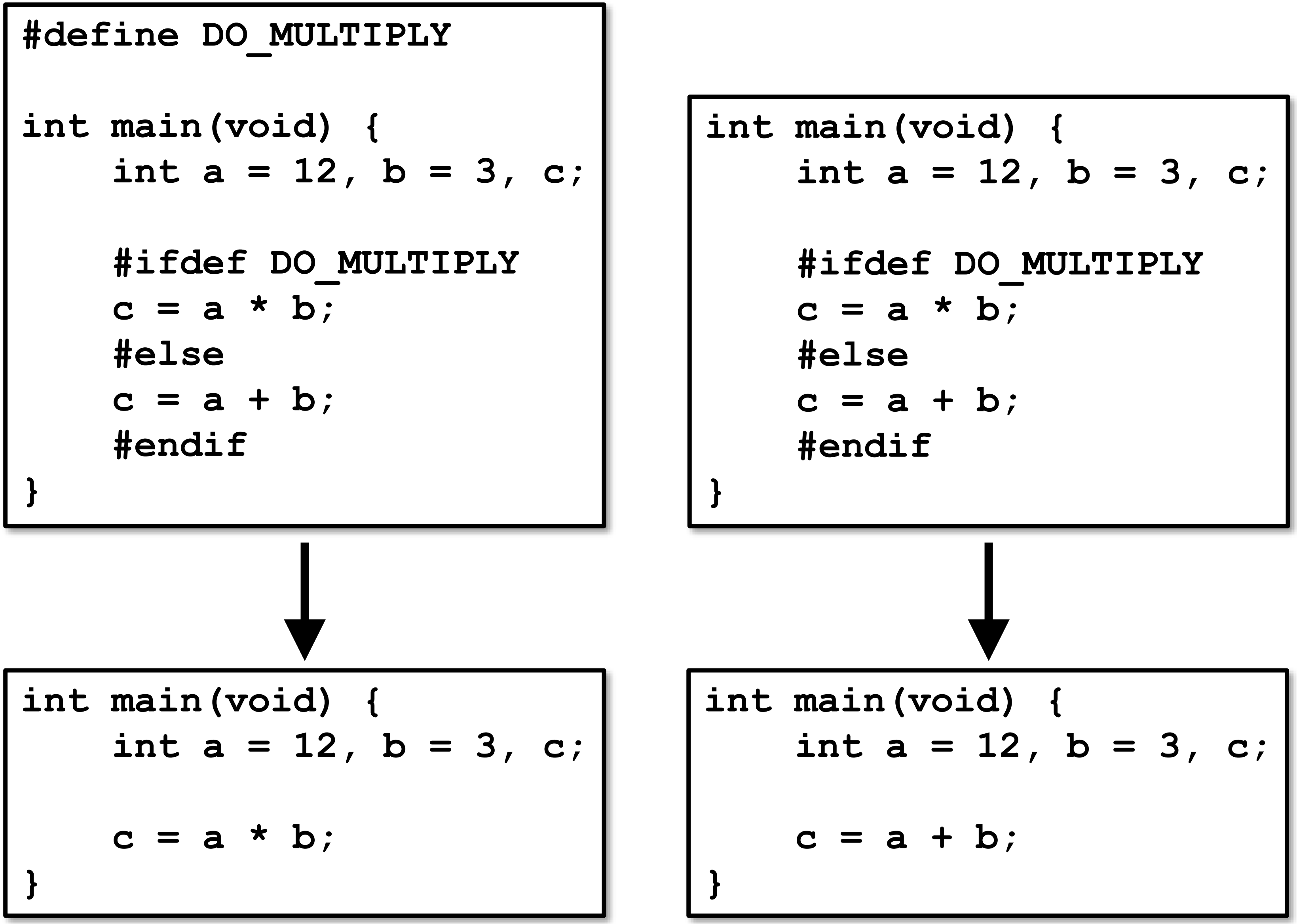
- User-defined macros can also be used to control whether a piece of text is included in the source code

- Use **`#ifdef`** ("if defined") … **`#else`** … **`#endif`**

```
#ifdef SOME_MACRO_NAME
  This text will be included if
  SOME_MACRO_NAME is defined
#else
  This text will be included if
  SOME_MACRO_NAME is not defined
#endif
```

- Can negate test by using **`#ifndef`** ("if not defined")

Pre-processor: `#ifdef`



```
#define DO_MULTIPLY

int main(void) {
    int a = 12, b = 3, c;

    #ifdef DO_MULTIPLY
    c = a * b;
    #else
    c = a + b;
    #endif
}
```

```
int main(void) {
    int a = 12, b = 3, c;

    #ifdef DO_MULTIPLY
    c = a * b;
    #else
    c = a + b;
    #endif
}
```

```
int main(void) {
    int a = 12, b = 3, c;

    c = a * b;
}
```

```
int main(void) {
    int a = 12, b = 3, c;

    c = a + b;
}
```

[Background: cpp]

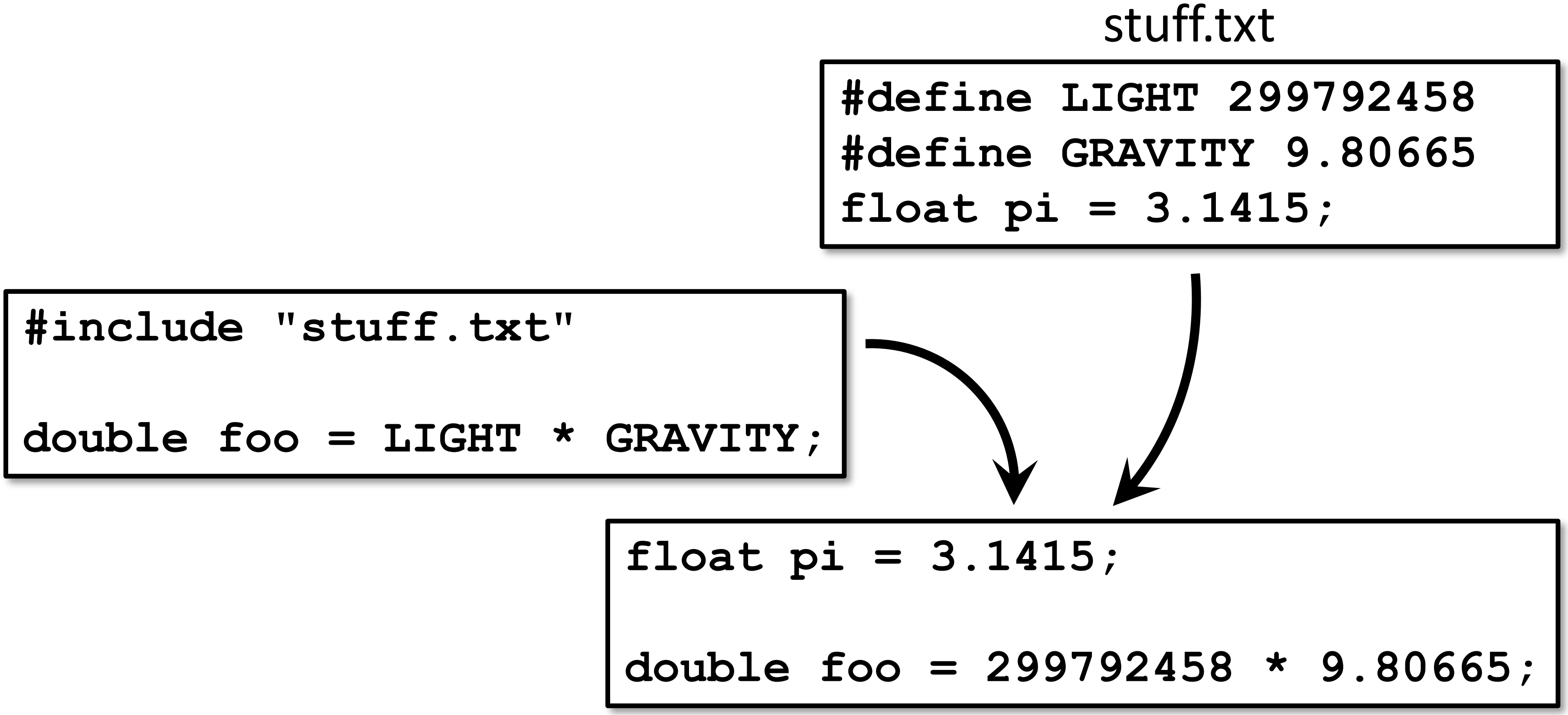
```
cpp ifdef.c
```

```
// This is not a complete code sample!

int main(void) {
    int a, b;

    #ifdef USE_TEST_VALUES
    a = 3;
    b = 4;
    #else
    a = getUserInput(); // Some made-up function to get user input
    b = getUserInput();
    #endif

    printf("%d * %d = %d\n", a, b, a * b);

    return 0;
}
```

Pre-processor: `#include`

- **`#include`** copies all text from the named file and inserts it in place of the directive

- Directives in the included text will also be processed



stuff.txt

```
#define LIGHT 299792458
#define GRAVITY 9.80665
float pi = 3.1415;
```

```
#include "stuff.txt"

double foo = LIGHT * GRAVITY;
```

```
float pi = 3.1415;

double foo = 299792458 * 9.80665;
```

Pre-processor: `#include`

- To search for a file in standard system locations, enclose the filename in angled brackets:

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

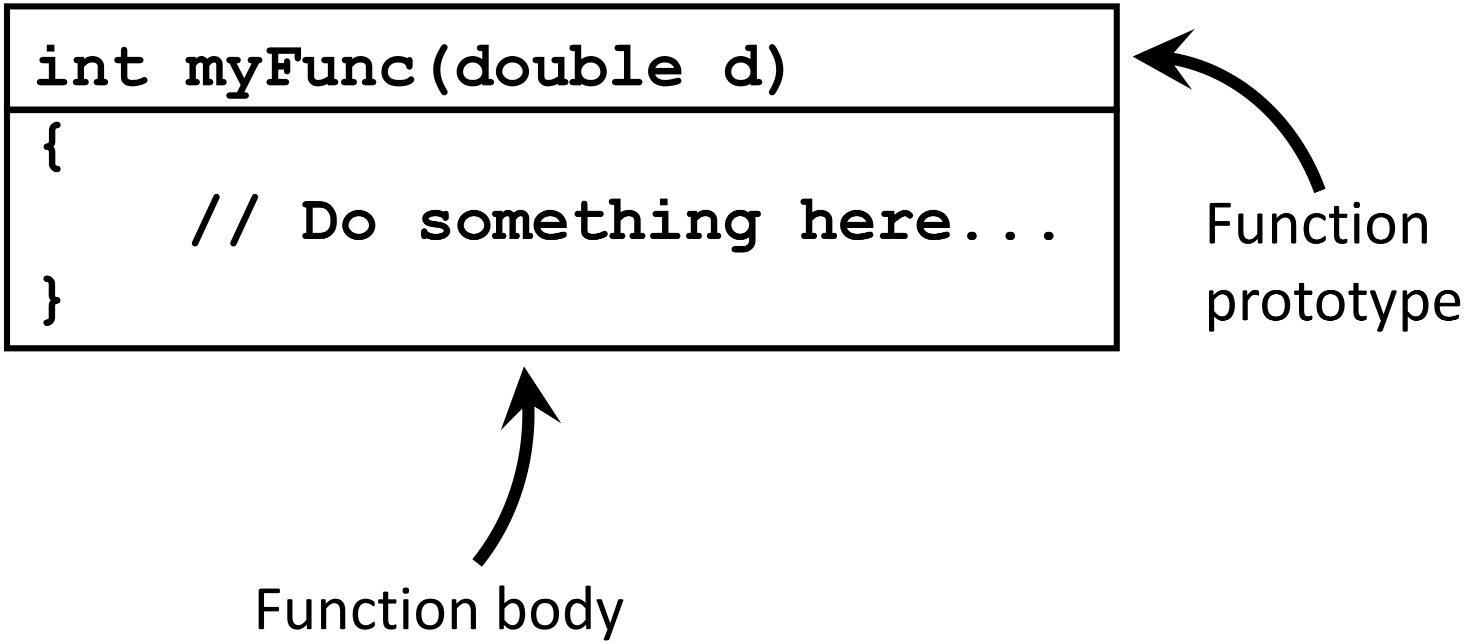
- To first search for a file locally, enclose the filename in inverted commas (you can include a path if you want):

  ```
  #include "myheader.h"
  #include "headers/useful_stuff.h"
  ```

Functions revision (C Lecture 4)

- Before we can use a function, we have to declare it

- A function declaration has two parts: the **function prototype** declares its type signature and name, while the **function body** contains the statements to execute when the function is called



```
int myFunc(double d)
{
    // Do something here...
}
```

Function prototype

Function body

Functions revision (C Lecture 4)

- A function prototype does not have to be followed immediately by a function body

- However, the prototype for a given function much appear in the file before the function is used

- A later declaration of the function body must then be provided (often in a different file)

- The early function prototype is called a **forward declaration**

Forward declaration

```
int f(int a);


int main(void) {
    return f(5);
}


int f(int a) {
    int c = a + 1;
    /* more here */
    return c;
}
```

Functions revision (C Lecture 4)

- Function prototypes are most commonly used to describe the name and type of functions which are defined in a different source file

- This allows the compiler to allocate sufficient memory for the function before it knows exactly how the function is implemented

Common definitions

- To allow code in one source file to use functions and values from another source file, we need a way to add:

  - Function prototypes for functions defined in other files
  - Common constants, etc.

- We can then use a function in lots of different files, and let the **linker** join up all the code at the end

[Background: cpp]

```
cpp hw.c
```

```
#include <stdio.h>


int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

Header files

- A **header file** contains common declarations, and exists purely to be used in **`#include`** directives

- Header files contain normal C source code, and by convention have the **`.h`** file extension

  - **`stdio.h`**, **`stdlib.h`**, **`math.h`**

- For every **`.c`** file you write which contains code you want to share, make a **`.h`** file with the function prototypes, etc., which you want to use elsewhere

Header files: `#include` guards

- You cannot declare a function (or any name) in C more than once in the same file

- If our definition is in a header file, how can we stop this header being included more than once?

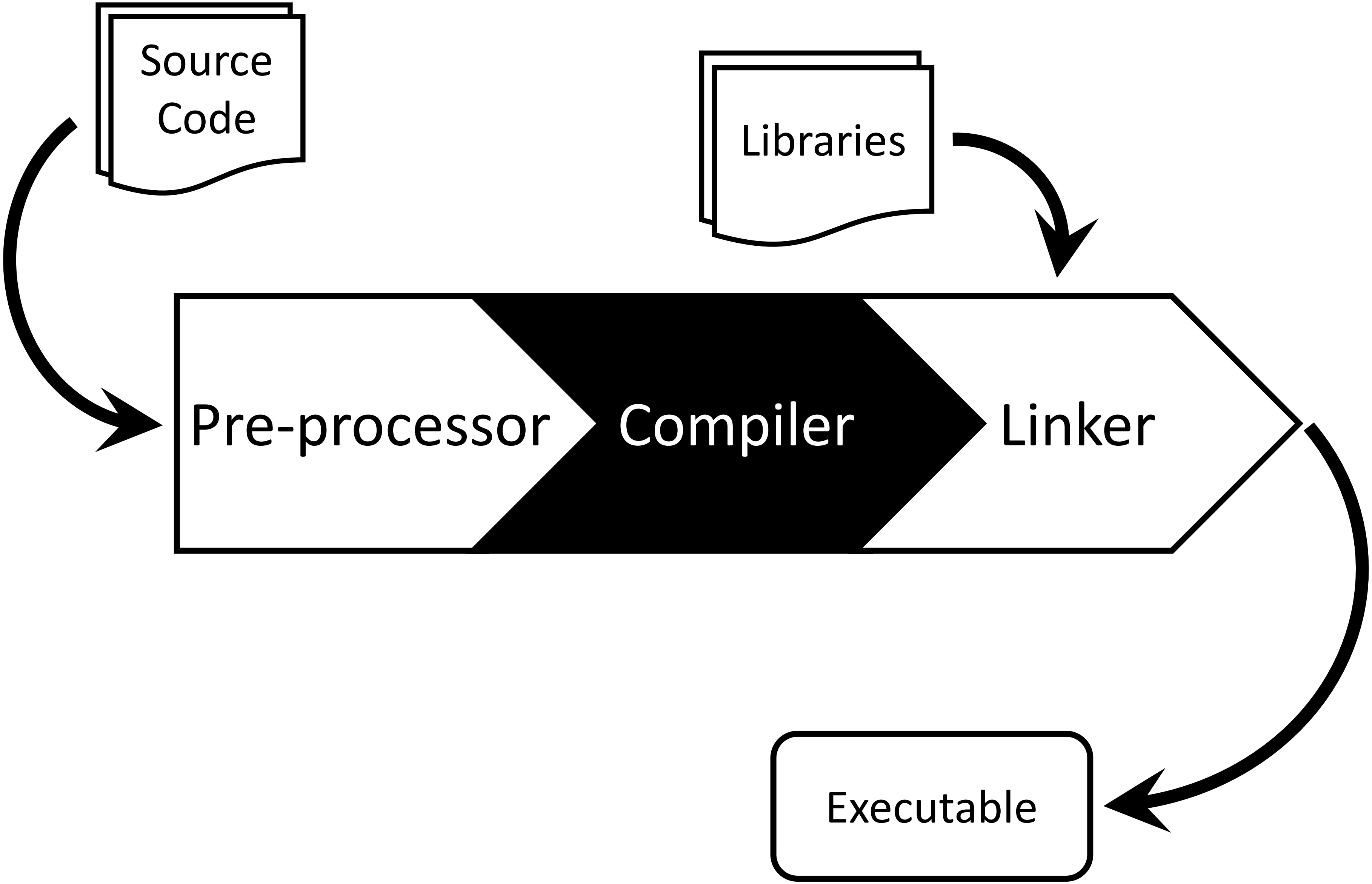
```
#ifndef MY_HEADER
#define MY_HEADER

Other definitions go here

#endif
```

- During the first time this header is included, **`MY_HEADER`** is defined; on subsequent occasions, the header is skipped as **`MY_HEADER`** is already defined

[Header files example]

```
/usr/include/stdio.h

gcc -E example.c

gcc -Wall example.c circles.c
```

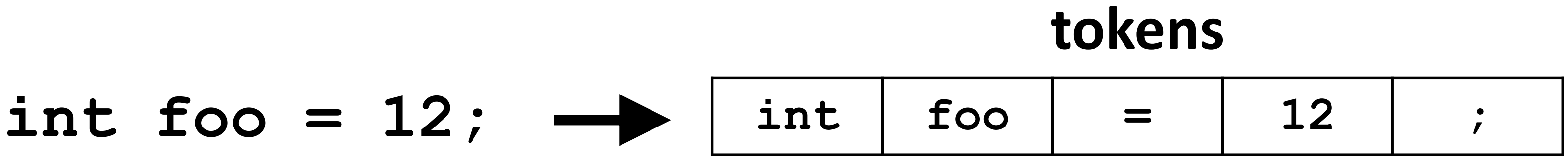
The C Build Process



The Compiler

- The **compiler** takes C source code and translates it into machine code

- Source code is broken down into syntactic units called **tokens**



tokens

```
int foo = 12;
```

→

`int`	`foo`	`=`	`12`	`;`

- Tokens are then parsed to construct an abstract representation of the program

The Compiler

- This representation is converted (via assembly code) to **machine code**
- Items such as file-scope variables and functions are labelled with a name called a **symbol**
- This machine code, along with a table of all symbols used in the file, is called the **object code** (with file extension **`.o`**)
- These symbols are used later to join up bits of code from different sources
- Can compile just object code using **`-c`** argument with **`gcc`**:

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

[Object code example]

```
gcc -c example.c

gcc –c circles.c

nm example.o

nm circles.o
```

The Compiler: Optimisation

- Simple compilation does not always produce the “best” (e.g. fastest, smallest, most efficient) result

- We can tell the compiler to spend more time **optimising** the code it produces, which might include reordering instructions

- Enable optimisations using the `-O` argument:

```
gcc -O2 mycode.c
```

- Enabling optimisations can slow down compilation: use `-O2` for a good balance between compilation time and code performance

- Logic is preserved, but underlying implementation may change

The Compiler: Optimisation

Flag	`gcc` interprets this to mean…
`-O0`	I like my code as it is – don't touch it. Compile it as quickly as you can. (Turn off optimisation.)
`-O1`	Make my code run faster. Don't think too hard.
`-O2`	Make my code run even faster. Reorder bits if you like, but don't make the executable bigger.
`-O3`	Make my code run really fast. Make the executable bigger if you like. Take as long as you need to think about it.
`-Os`	Make my code really small.
`-Ofast`	Make my code run as fast as you can. Don't worry about standards compliance – assume things about floating-point numbers which aren't necessarily true. ***(Proceed with extreme caution!)***

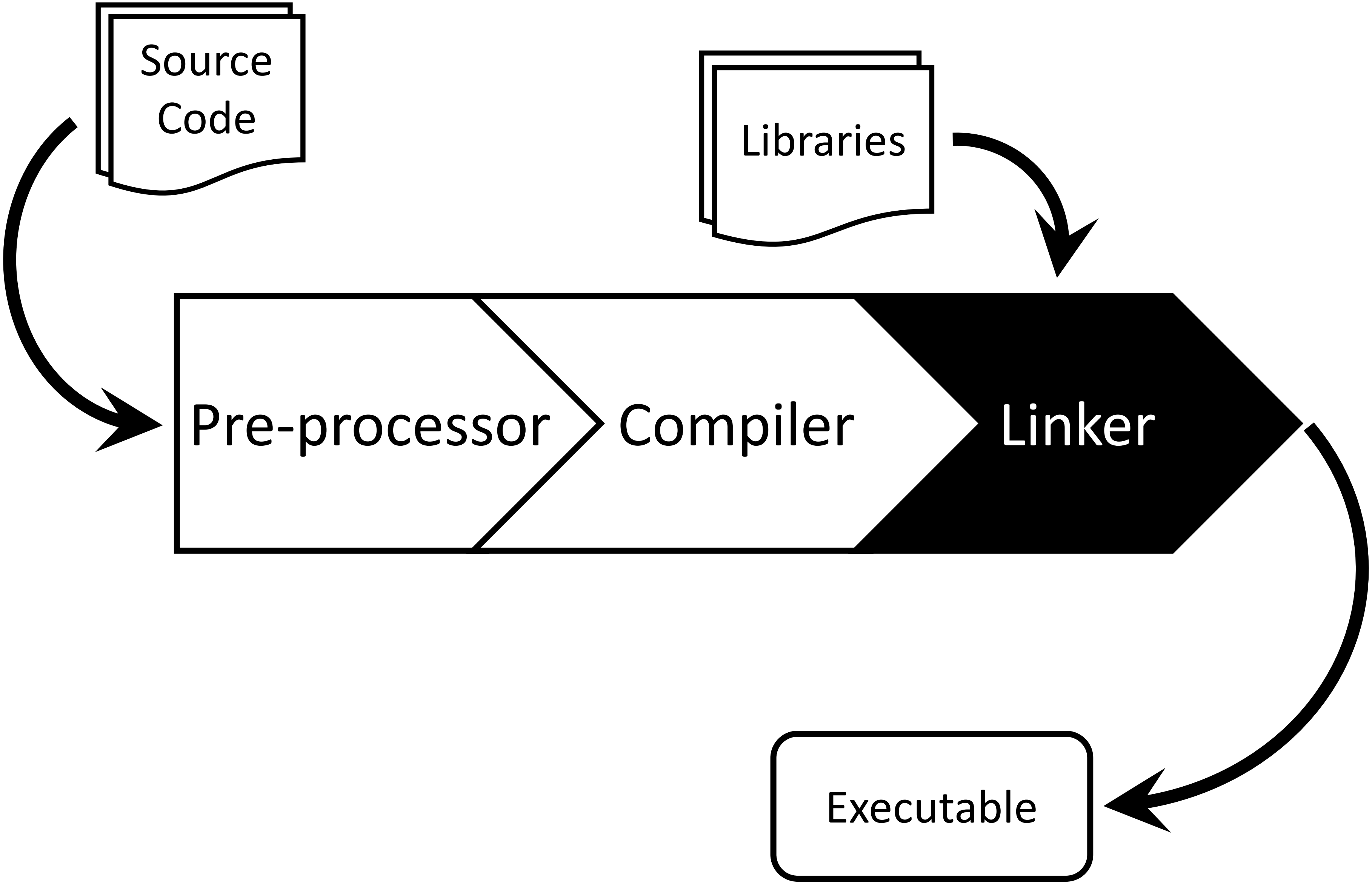
`–O2` recommended for most situations

[Optimisation example]

```
gcc -S [-O2] opt.c
```

```
int x, y = 12;


int main(void)
{
    x = y + 3;
    y = 0;

    return 0;
}
```

The C Build Process



Source Code

Libraries

Pre-processor

Compiler

Linker

Executable

The Linker

- The **linker** joins up all the object code from the compiler, mapping symbols to their corresponding representation

- First, all object code is merged into a single file with a consolidated **symbol index** at the top

- Before object code is linked, function calls are simply a note of the symbol corresponding to the function to be called

- The linker looks up the symbol in the symbol index, and uses this information to replace it with a call to the matching function

- A similar process happens for variables with file scope

The Linker

- You can link object code using a similar command to full compilation:

```
gcc –o OUTPUT_NAME file1.o file2.o file3.o
```

Libraries

- We can link other people’s code with ours, provided we know what symbols are in it
- One way to provide a set of useful functions to others is to package them up into a **library**
- One or more header (**`.h`**) files will be provided, and we can **`#include`** these to use the function prototypes within them
- Libraries have names ending **`.a`** or **`.so`** (the difference relates to how and when the code is linked)
- We can link with a library called **`NAME`** using the **`-lNAME`** option

libc: The C Standard Library

- One library used in almost all C code is the C Standard Library: **libc**

- This contains useful functions for many applications

- It's so large that its function prototypes are split into multiple header files:

  - **stdio.h** for input and output functions
  - **string.h** for string-handling functions

- **gcc** links with **libc** by default…

  - …with the exception of the floating-point maths part (**libm**), which needs to be linked separately (using **-lm**)

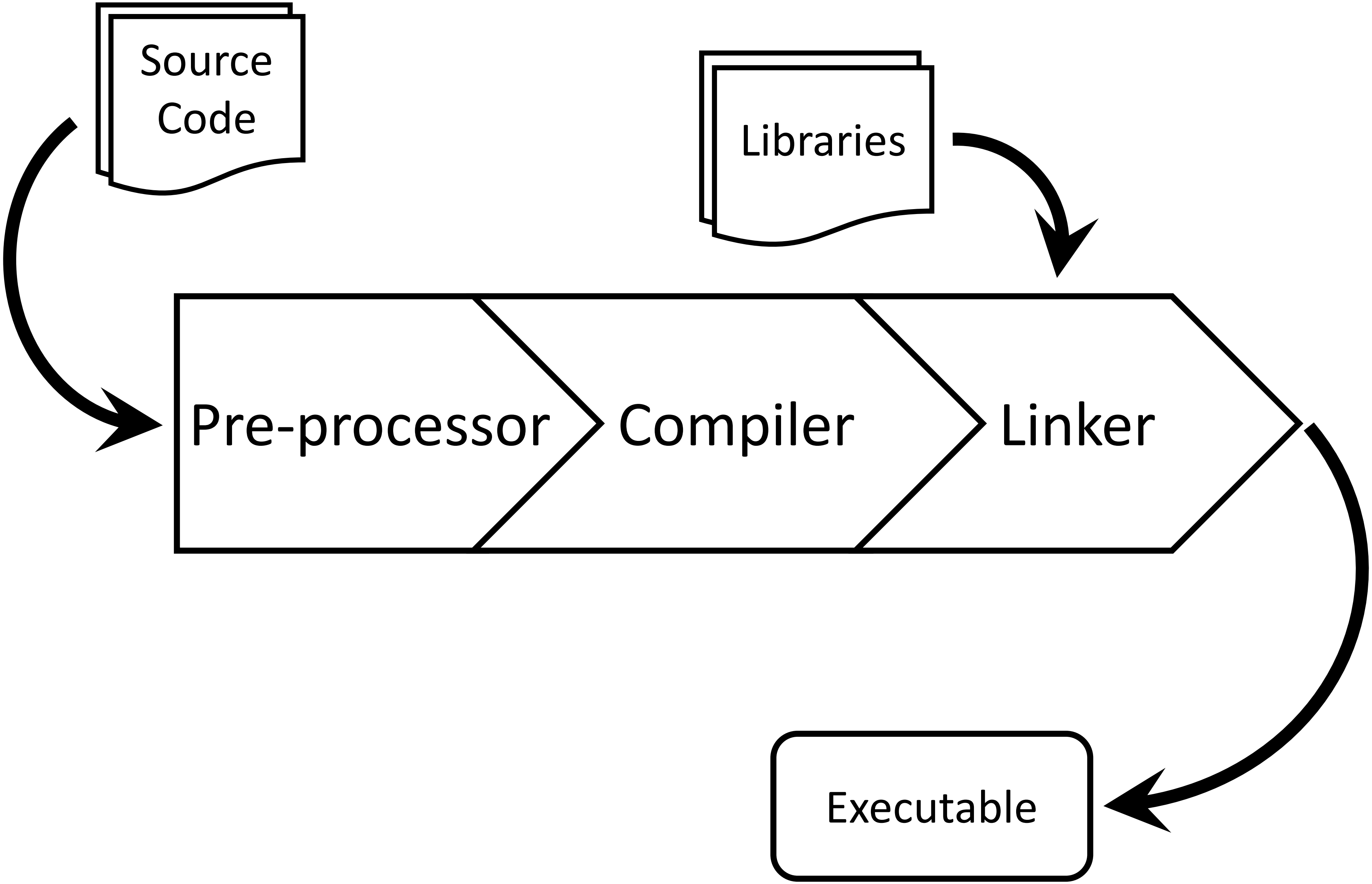
Library locations

- The linker will search in standard locations for libraries

- You can add another location using the -L argument:

```
gcc -L/path/to/library -lnameoflibrary mycode.c
```

The C Build Process



Source Code

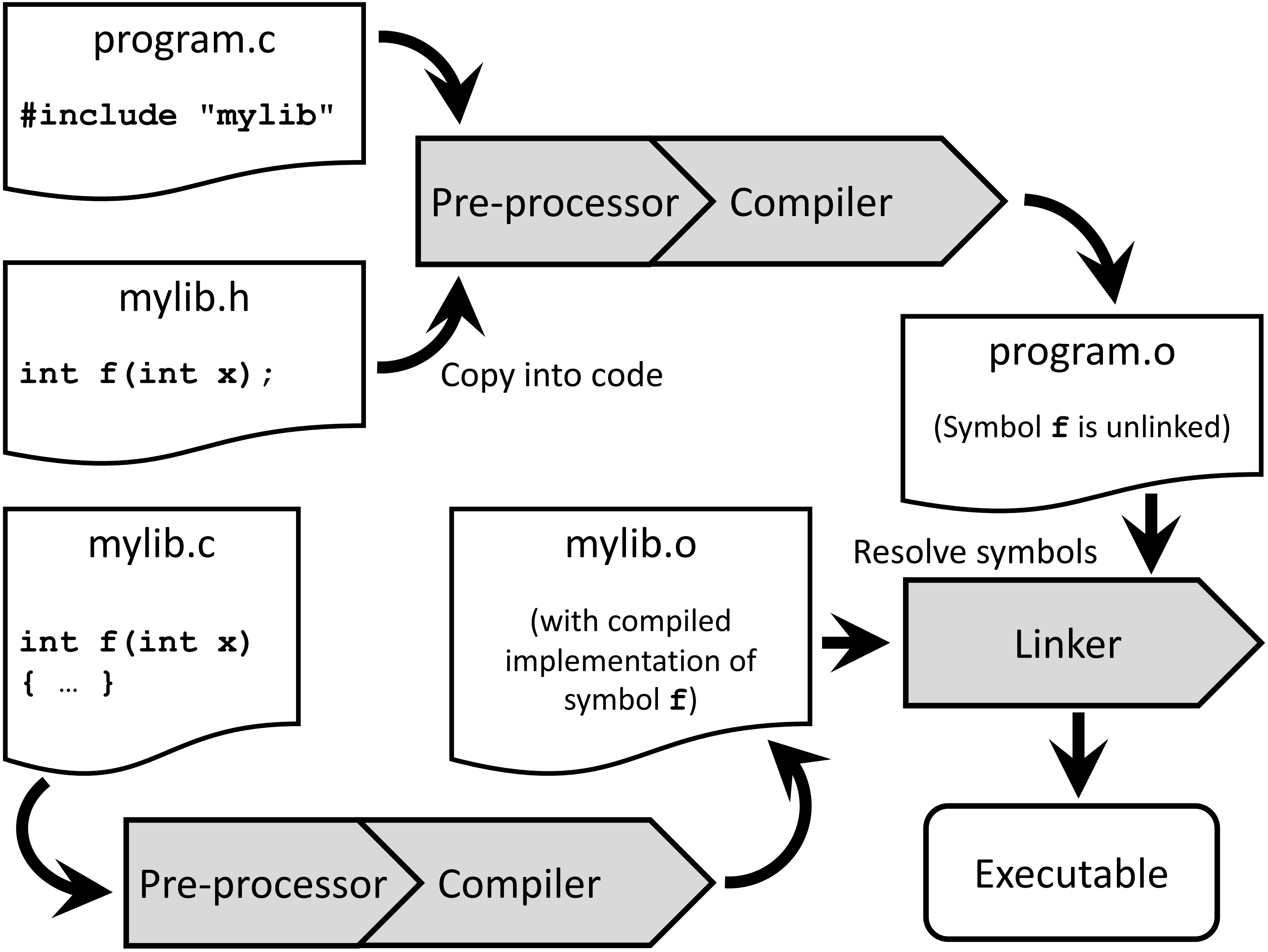
Libraries

Pre-processor

Compiler

Linker

Executable



program.c

```
#include "mylib"
```

mylib.h

```
int f(int x);
```

Copy into code

Pre-processor

Compiler

program.o

(Symbol **f** is unlinked)

mylib.c

```
int f(int x)
{ … }
```

mylib.o

(with compiled implementation of symbol **f**)

Resolve symbols

Linker

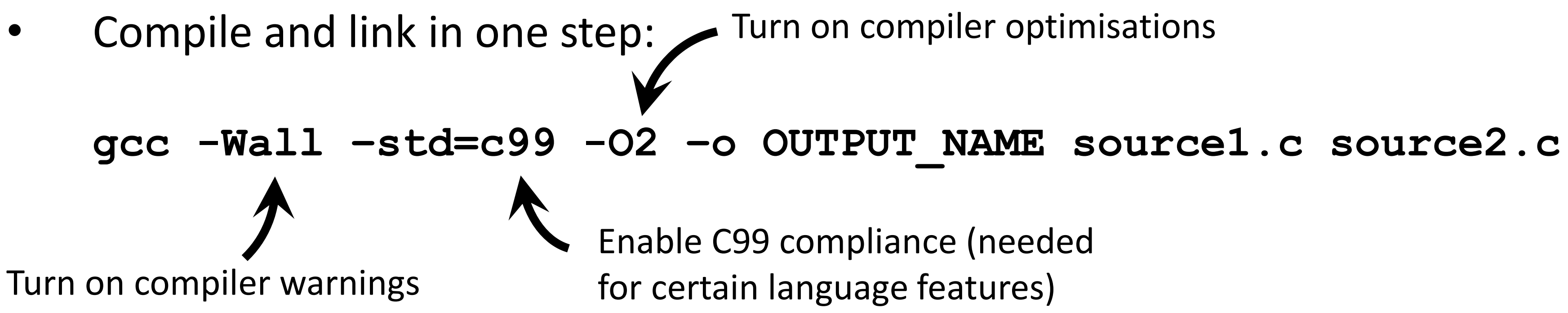
Pre-processor

Compiler

Executable

Useful gcc commands

- Compile and link in one step:



Turn on compiler optimisations

```
gcc -Wall -std=c99 -O2 -o OUTPUT_NAME source1.c source2.c
```

Turn on compiler warnings

Enable C99 compliance (needed for certain language features)

- Compile to object code only (create `.o` files but don’t link):

```
gcc -Wall -O2 -c source1.c source2.c
```

- Link object code:

```
gcc -o OUTPUT_NAME file1.o file2.o file3.o
```

- Link with libraries:

```
gcc -o OUTPUT_NAME -lLIBRARY_NAME file1.o file2.o
```

Summary

- The C build process converts source code to machine code, and comprises three stages: pre-processor, compiler, linker

- The **pre-processor** prepares source code for compilation

    - It processes directives such as **`#include`** and **`#define`**

- The **compiler** converts prepared source code to object code

- The **linker** combines the program’s object code with object code from supporting libraries, producing the final executable

- **`gcc`** can do all this in one command, but it is useful to break it down into stages when working on large projects to avoid the need to rebuild everything following minor changes