C Revision

(this is a fast overview of the contents of the course, you should read the actual lectures for the detail: this is just to remind you of the topics)

Types and Variables

	int	**long**	**float**	**double**	**char**	**(void)**	**(pointer)**
holds	integers	integers	floating point	floating point	single ASCII character	(no type/ no value)	memory location
unsigned version?	unsigned int	unsigned long	N/A	N/A	unsigned char	N/A	N/A
size (bytes, x86_64)	4	8	4	8	1	0	8
format string	%d	%ld	%f	%lf	%c	N/A	%p

Variable declarations

- A variable declaration does two things:
  - Ask for memory to be (automatically) allocated to store values of a given type.
  - Assigns a name to that memory so you can refer to it ("variable name").

"Put aside some memory on the stack to store a long int, and let me refer to it as big_number"

```
long big_number;
```

"Put aside some memory on the stack to store a double, let me refer to it as precise_float", "Then set the contents to 5.6778"

```
double precise_float = 5.6778;
```

Operations

`+ - * / %` **Fundamental operations**

`=` **ASSIGNMENT**

`==` **EQUALITY TEST**

`<= >= > <` **Comparisons**

`(int)` **cast (value to the right) to int**

`( )` **parentheses change priority (as in standard maths)**

```
x = x+1;
x += 1;
x++;
```

Shortcut notations.

```
3/2 == 1
3/2.0 == 1.5
```

Operations happen in the context of the "most precise" type involved

Blocks and statements

- A statement in C is a complete "instruction".
- They end with a ;
- A block is a set of statements surrounded by { }
- A block groups together the statements inside it, so they can be treated collectively.
- Most C structures (loops etc) use blocks to group the code they apply to.

lecture 2.3

Scope and Allocation

Scope is the range of the text over which a name has meaning.

Scope reads top to bottom.

Scope is delineated by blocks - a name defined in a block loses scope when that block ends.

A name outside all blocks is defined over all the file (after its definition)

```
int g = 5;

void f(int d,int r){
 //some stuff
}

int main(void){
   int a;
   {
      int b;
   }
   f(a,24);
}
```

Lecture 2.3

Scope and Allocation

Allocation range is the period (when the code is executing) during which memory is actually assigned to store a thing.

Unlike Scope, it doesn't "end" if (for example) another function is called.

Automatically allocated variables are only deallocated when their containing function returns.

```
int g = 5;


void f(int d,int r){
 //some stuff
}


int main(void){
    int a;
    {
       int b;
    }
    f(a,24);
}
```

Loops

Repeat stuff in this colour

Check this at end of each loop (continue if true)

Do this before starting loop for first time

Do this at end of each loop

```
for(int i=0;i<5;i++) {
//stuff
}
```

Check condition at start

```
while(i<5) {
i++;
}
```

```
do {
i++;
}while(i<5);
```

Check condition at end

Conditionals (If, switch)

```
if(i<2){
   //do this
} else if (i==3){
   //or this
} else {
   //otherwise, this
}
```

```
switch (i) {
 case 0:
 case 1:
  //do this
  break;
 case 3:
  //or this
  break;
 default:
  //otherwise, this
}
```

lecture 3

Arrays and Strings

An array is a variable which is allocated space to store multiple values of a type.
Array indices start at 0.

Strings are stored in *arrays of char* - they have special functions to help you use them.

Allocate me space for 5 integers, and let me refer to this as `a`

```
int a[5];


double c[2] = {6.1,2.3};
```

Allocate me space for 2 doubles,
let me refer to this as `c`**, and assign values 6.1 and 2.3 to the slots in it.**

A single character

```
char c = 'c';
```

Two ways to initialise a string

```
char s[7] = "string";
char s[] = "string";
```

Arrays and Strings

```
char c[7] = "string";
```

'\0' character used to mark end of string

is the same as

```
char c[7] = {'s','t','r','i','n','g','\0'};
```

Functions to use strings are in `string.h`

```
strcat(deststring,srcstring)
```

```
strlen(stringtomeasure)
```

The length of a string is not necessarily the size of the containing array

```
strcmp(string1, string2)
```

```
strcpy(deststring,srcstring)
```

Pointers

A pointer is a variable which can hold the address of memory locations in which other variables are stored.

Make me a thing capable of pointing at integers, and let me refer to this as a

```
int *a;

int b;
```

Assign to a **the address of (memory location) of** b

```
a = &b;
```

Get the value in the memory location a **points at (**b**)**

```
printf("The value in b is %d\n", *a);
```

Pointers and arrays

Pointers and arrays are closely related.

You can use "array" indexes on pointers, and a "bare" name of an array gives a pointer to the start of its memory.

```
char s[] = "A string";
```

**"Allocate enough space to store the string literal "A string",
copy the characters into the resulting space.
Let me refer to this variable as `s`"**

The difference is that making an array also allocates memory needed for it!

```
int a[5];
int *p;
p = a;
```

```
p[1] == a[1]
*a = *p = a[0]
```

```
char *s = "A string";
```

**"Create a pointer-to-char.
Assign the pointer the address in memory in which the (constant) literal value "A string" is stored.
Let me refer to this pointer as `s`**

lecture 3.4

User input and output

Functions to use Input and Output are in `stdio.h`

```
printf(formatstring,variables…)

sscanf(string, formatstring, &variables…)
```

Format Strings
normal text - matches identical text in string
`%d` - matches a decimal representation of integer
`%f` - matches a decimal representation of a float
`%lf` - matches a decimal representation of a double
`%c` - matches a single ASCII character
`%s` - matches a sequence of ASCII characters (up to a space)

Structured Data Types

```
struct mytype {
 int r;
 double q[56];
};

struct mytype example;

example.r = 57;
example.q[6] = 66.0;
```

Define a new kind of value, called "struct mytype", which contains an int called r an array of 56 doubles, called q

Allocate memory for a struct mytype, and call it example

Assign values to components of example.

Functions

Functions are for packaging up an algorithm so you can reuse it.

When you call a function, values are *copied* into variables used inside function.

Type of value function will return (`void` for "doesn't return anything")

Types of values function needs to be given (`void` for "doesn't need anything")

Name (inside f) to use for this value

```
double f(double t){
  int i;
  double acc;
  for(i=0;i<4;i++){
    acc = t*t;
  }
  return acc;
}
```

Variables local to function

Return the value of acc and exit function

Command Line Arguments

```
int main f(int argc, char *argv[]){


}
```

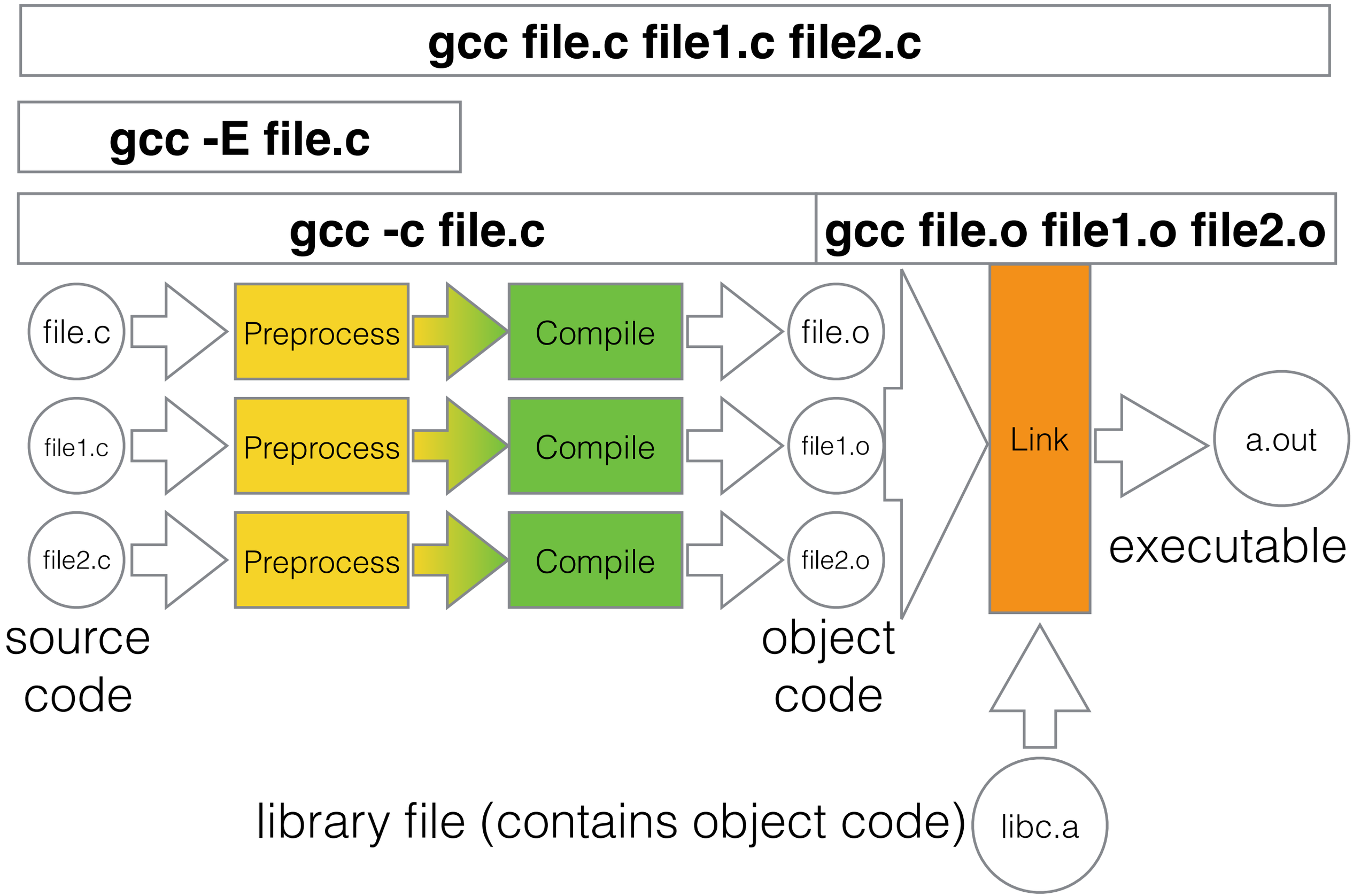
We can get command line arguments into our code with this form of main().

`argc` is "number of arguments" (including the name of program)
`argv[i]` is the **i**th argument (as a string).
`argv[0]` is the name of the program

The Preprocessor

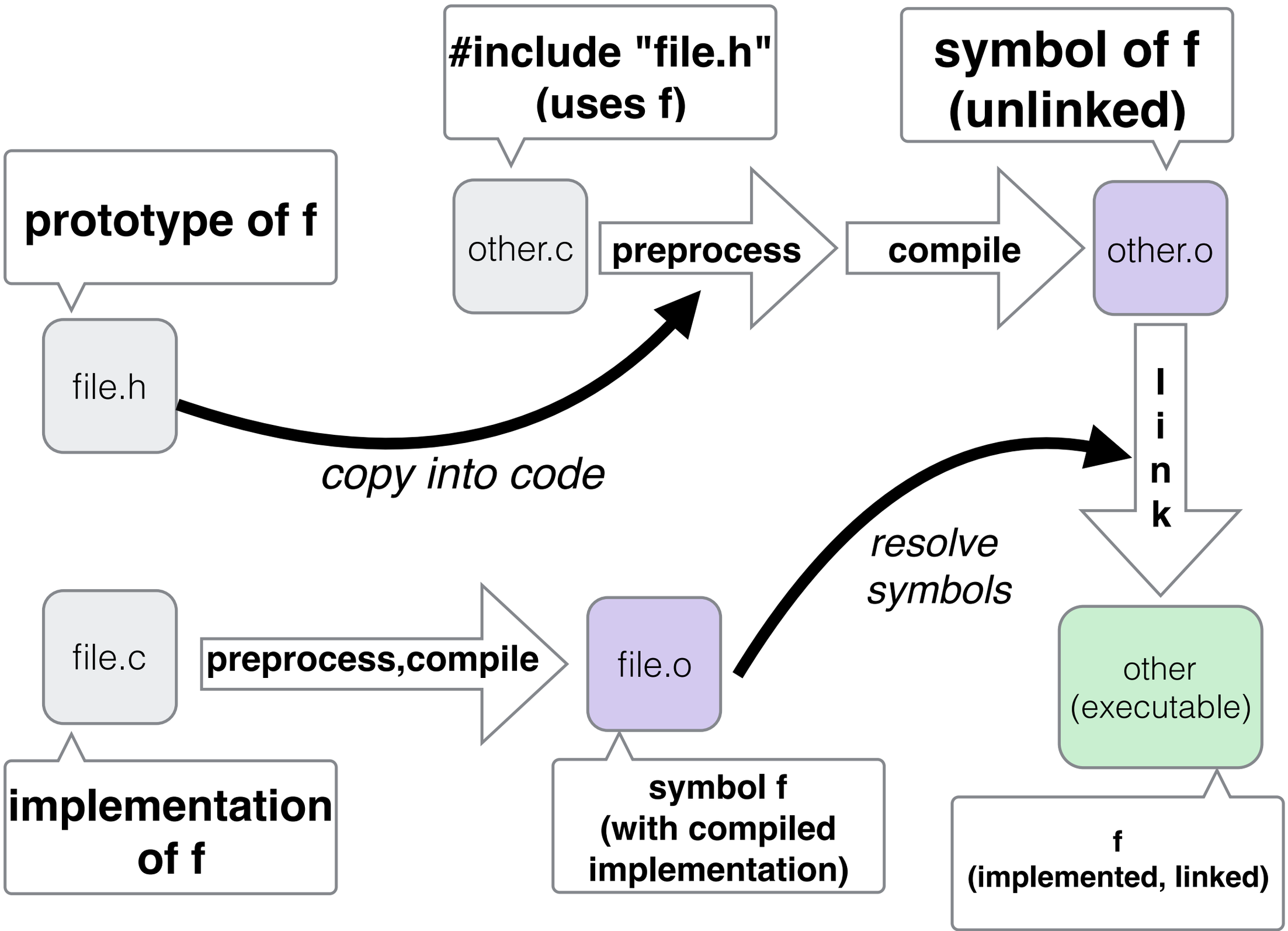
```
#define NAME things to replace with

#include <name of system header>
#include "name of local header"

#ifdef SOMEDEF
 code which depends on SOMEDEF
#else
 code if SOMEDEF not defined
#endif
```

lecture 6


gcc file.c file1.c file2.c
gcc -E file.c
gcc -c file.c
gcc file.o file1.o file2.o
file.c
Preprocess
Compile
file.o
file1.c
Preprocess
Compile
file1.o
file2.c
Preprocess
Compile
file2.o
Link
a.out
executable
source code
object code
library file (contains object code)
libc.a

lecture 6

Compiling and Linking



Text File I/O

Functions to use Input and Output are in `stdio.h`

```
int i = 6;
FILE * f = fopen("Myfile.txt", "r+");

fprintf(f, "%d is a number\n", i);

fscanf(f, "%d is a number, &i);

fflush(f);
fclose(f);
```

Binary File I/O

Functions to use Input and Output are in `stdio.h`

```
int i = 6;
FILE * f = fopen("Myfile.bin", "wb");

fwrite(&i,sizeof(int),1,f);

fflush(f);
fclose(f);
```

lecture 8

Pseudo Random Number Generation

Functions to use PRNGs are in `stdlib.h`

```
srand(78);
int a = rand();
```

set PRNG internal seed to 78

generate next int in pseudo-random sequence, and assign it to a

Seeding the PRNG with srand requires an integer value.
One way to get "different seeds" each time you run a program is to use the current time as a seed.

(needs `time.h` , **time()** function)

```
srand(time(NULL));
```

the current time, expressed as an int, seconds since midnight, 1 Jan 1970

lecture 8

Pseudo Random Number Generation

```
rand();
```

a pseudo-random int between 0 and RAND_MAX

```
rand()/(double) RAND_MAX;
```

a pseudo-random double between 0 and 1, by division

We can generate other distributions from rand() via appropriate transformations.

e.g. approx gaussians via sum of uniform deviates (with rescaling), or by other methods.