

# Introduction to C, C++, and Unix/Linux

CS 60

Functions

Today

→ C functions, program structure

- Reading for Monday: K&R ch. 1-4 & 7.1-7.4

# Functions

- C is a procedural programming language, and all the action takes place in functions

<code>main</code>	<code>fabs</code>	<code>strcmp</code>
<code>printf</code>	<code>putchar</code>	<code>strcpy</code>
<code>scanf</code>	<code>getchar</code>	<code>sqrt</code>

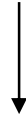
- `main` is expected and must be included
- Other functions are supplied by C libraries or by the programmer

# Function header

**return\_type**    **func\_name** ( **param\_list** )



Type returned by the function (int, double, int \*, char\*, void, ...)



Unique function name



List of arguments – values passed to the function:

**type param\_name**

“void” means the function has no return value

# Functions (cont.)

- Format of a function:

Function  
header

```
return_type func_name ( param_list )
```

Function  
body

```
{  
    declarations  
    statements  
}
```

Actually, **gcc** won't complain if you define variables at other places in the function, but it's good practice to define all your variables at the beginning of a block

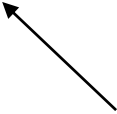
```
int main(void)
int main(int argc, char **argv)
int fact(int num)
double sqrt(double x)
char *error_message(int errnum)
int printf(char *str, ...) ← Special case
double *MakeArray(int x, int y)
int *Resize(int x, int y, int *data)
void beep(void)
```


## Note

```
int *Resize(int x, int y, int *data)
```

could be written as

```
int* Resize(int x, int y, int* data)
```

return value is a pointer

parameter is a pointer

# Call by value

- Arguments to C functions are evaluated, and copies of the values are passed into the function
  - This is called “call by value”
- The same is true for the return value of a function
  - It is copied into memory reserved for the function’s return value
  - How much space is allocated?

# Call by value (cont.)

- Benefit
  - This protect the variable values so the function doesn't “accidentally” modify them
  - Pointers reduce this benefit, though
- Drawback
  - It is inefficient (what about a large array?)
  - Pointers will help here



```
void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

```
int main(void)
{
    int a=5, b=8;
    swap(a, b);
    printf("( %d, %d)", a, b);
    return(0);
}
```

Prints

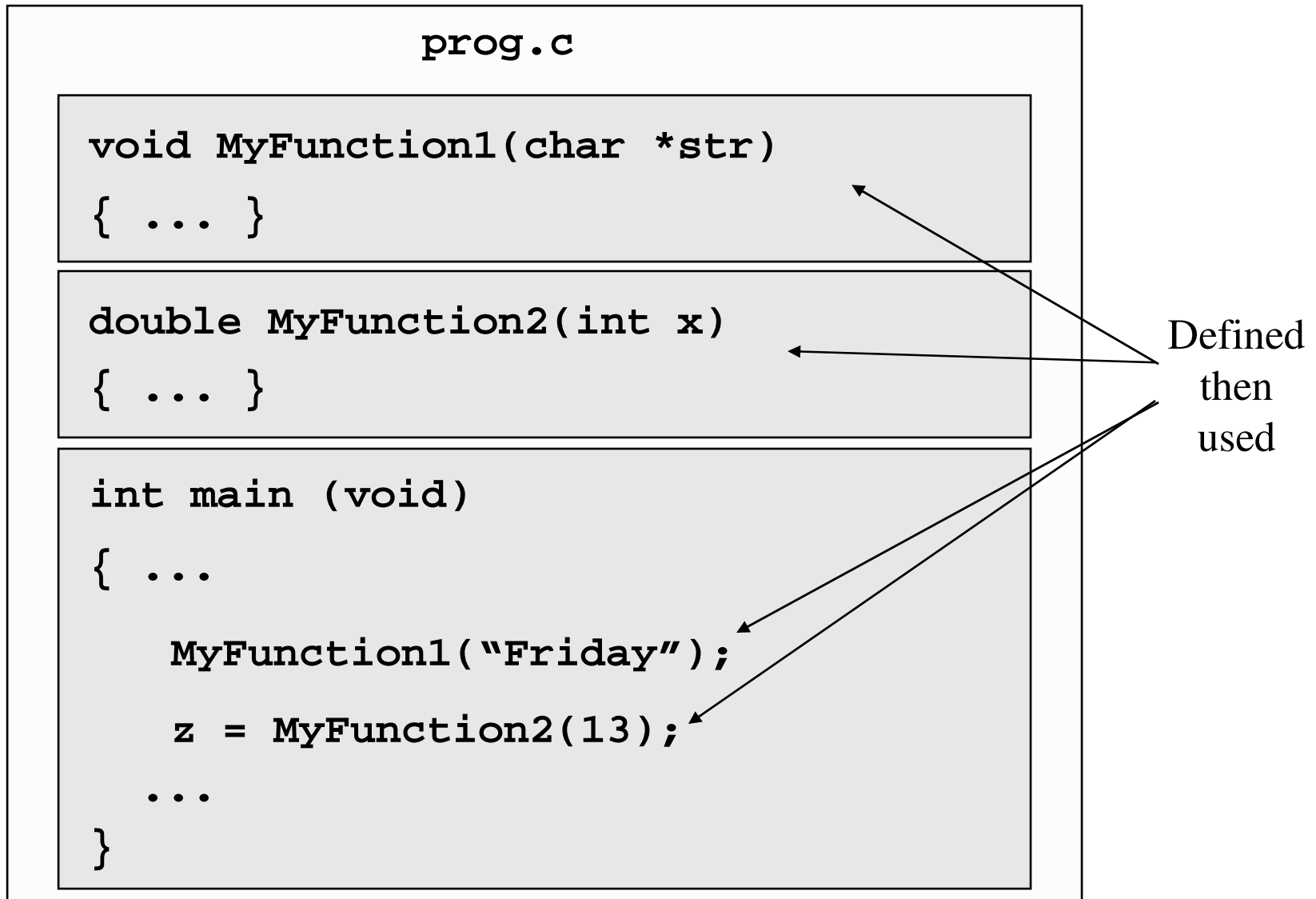
(5, 8) ←

(Doesn't swap)

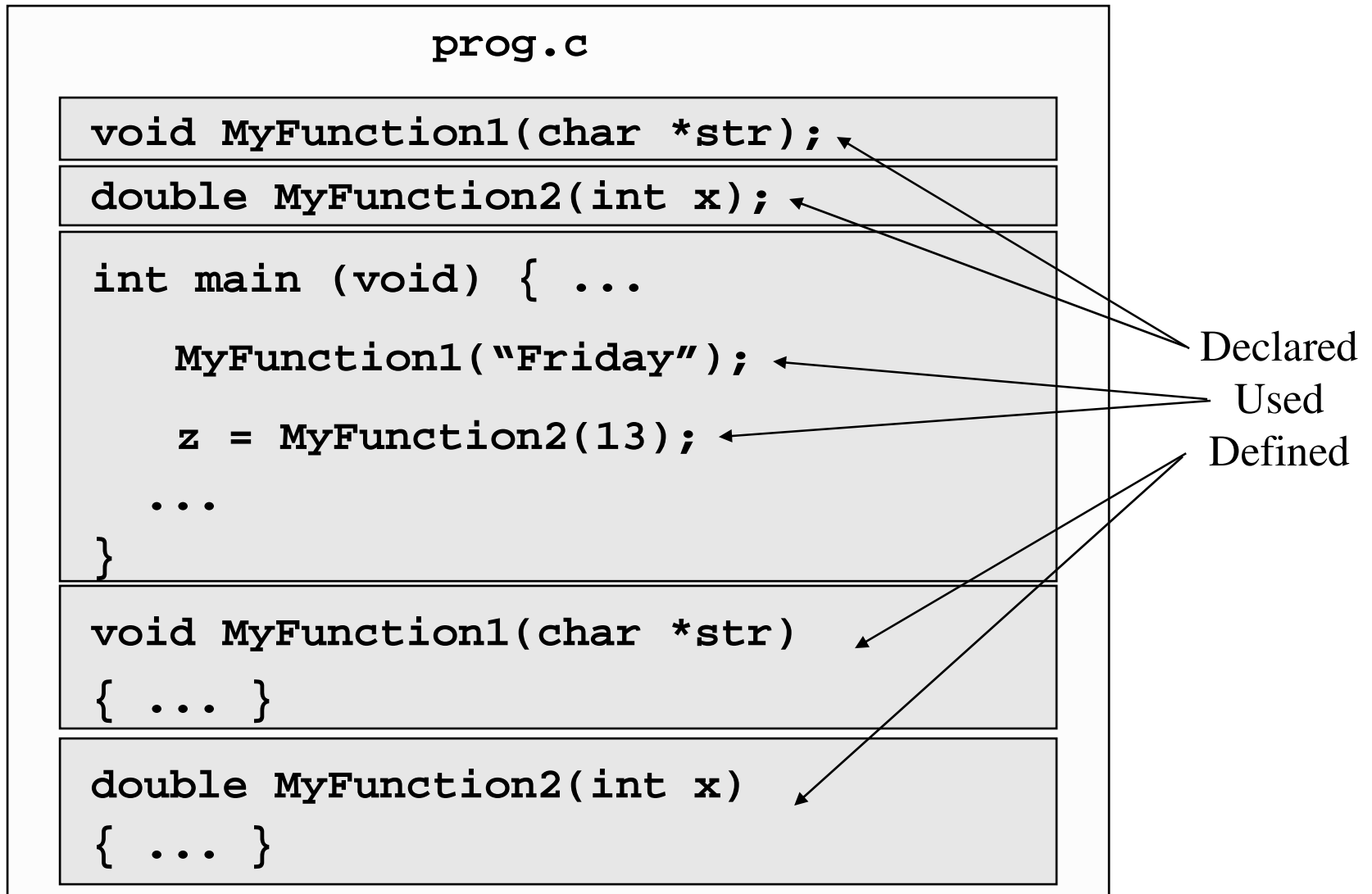
# Program structure

- A C program can have only a **main** function, but non-trivial programs should be modular, with a collection of small, meaningful functions
- These make the program
  - logically clear and well structured
  - easier to read, debug, and modify
  - reusable in other programs

# Single file structure (1)



# Single file structure (2) – Function prototypes



# Single file structure (2a) – Function prototypes

```
prog.c  
  
void MyFunction1(char *);  
double MyFunction2(int);  
  
int main (void) { ...  
    MyFunction1("Friday");  
    z = MyFunction2(13);  
    ...  
}  
  
void MyFunction1(char *str)  
{ ... }  
  
double MyFunction2(int x)  
{ ... }
```

Variable names in function declaration are optional

But of course they're required in the function definition

# Multiple file structure (1)

```
prog.c
```

```
void MyFunction1(char *);
```

```
double MyFunction2(int);
```

```
int main (void)
```

```
{ ...
```

```
    MyFunction1("Friday");
```

```
    z = MyFunction2(13);
```

```
    ...
```

```
}
```

```
functions.c
```

```
void MyFunction1(char *str)
```

```
{ ... }
```

```
double MyFunction2(int x)
```

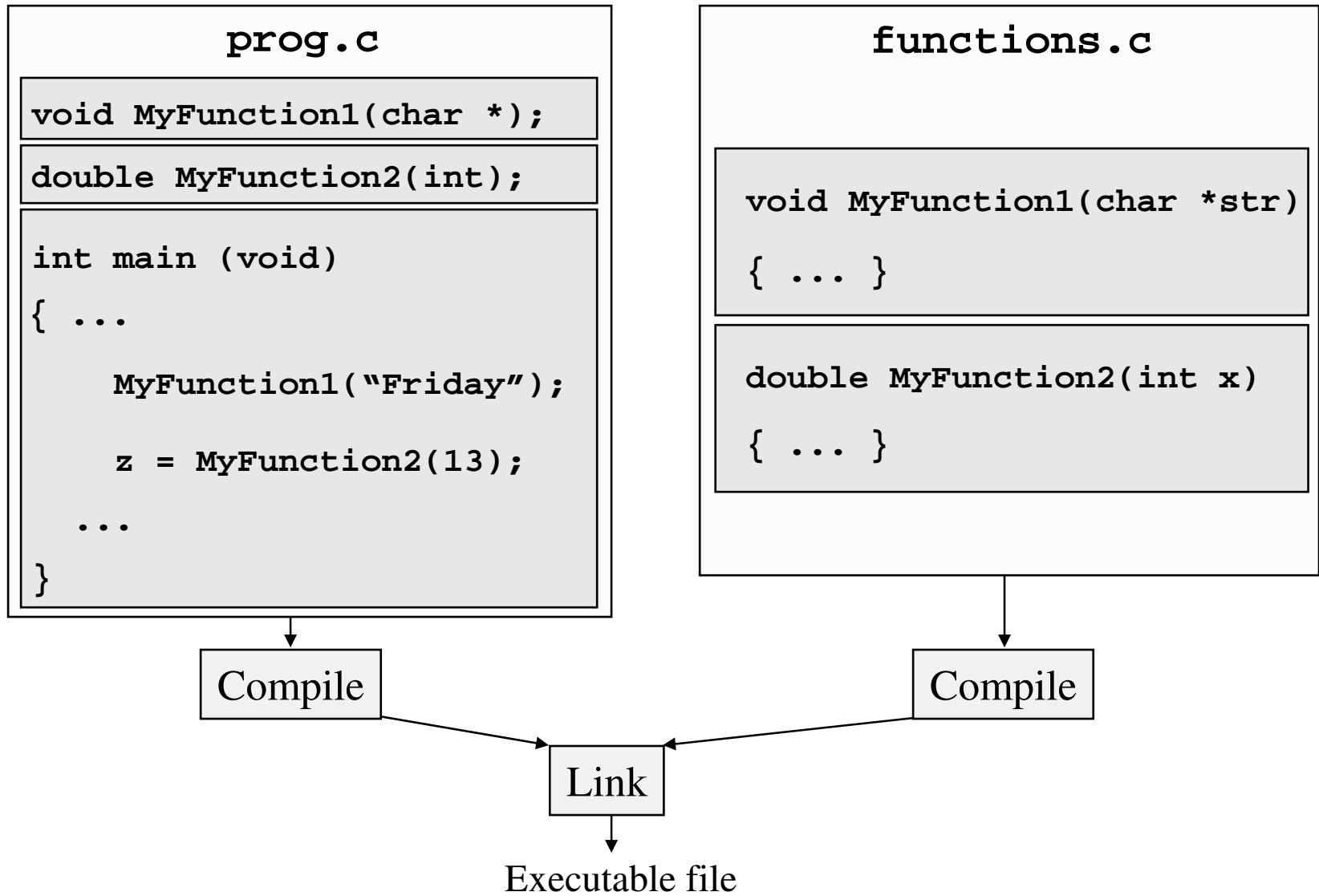
```
{ ... }
```

```
gcc -c prog.c           → prog.o
```

```
gcc -c functions.c     → functions.o
```

```
gcc -o prog prog.o functions.o
```

# Multiple file structure (2)



# Multiple file structure (3)

```
prog.c
#include "functions.h"

int main (void)
{ ...
    MyFunction1("Friday");
    z = MyFunction2(13);
    ...
}
```

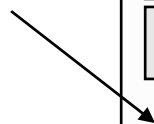
```
functions.c
#include "functions.h"

void MyFunction1(char *str)
{ ... }

double MyFunction2(int x)
{ ... }
```

```
functions.h
void MyFunction1(char *);
double MyFunction2(int);
```

Also: global variables, constants, macros, typedefs, structs, ....





Source → executable:

```
gcc -o prog prog.c functions.c
```

Source → object → executable

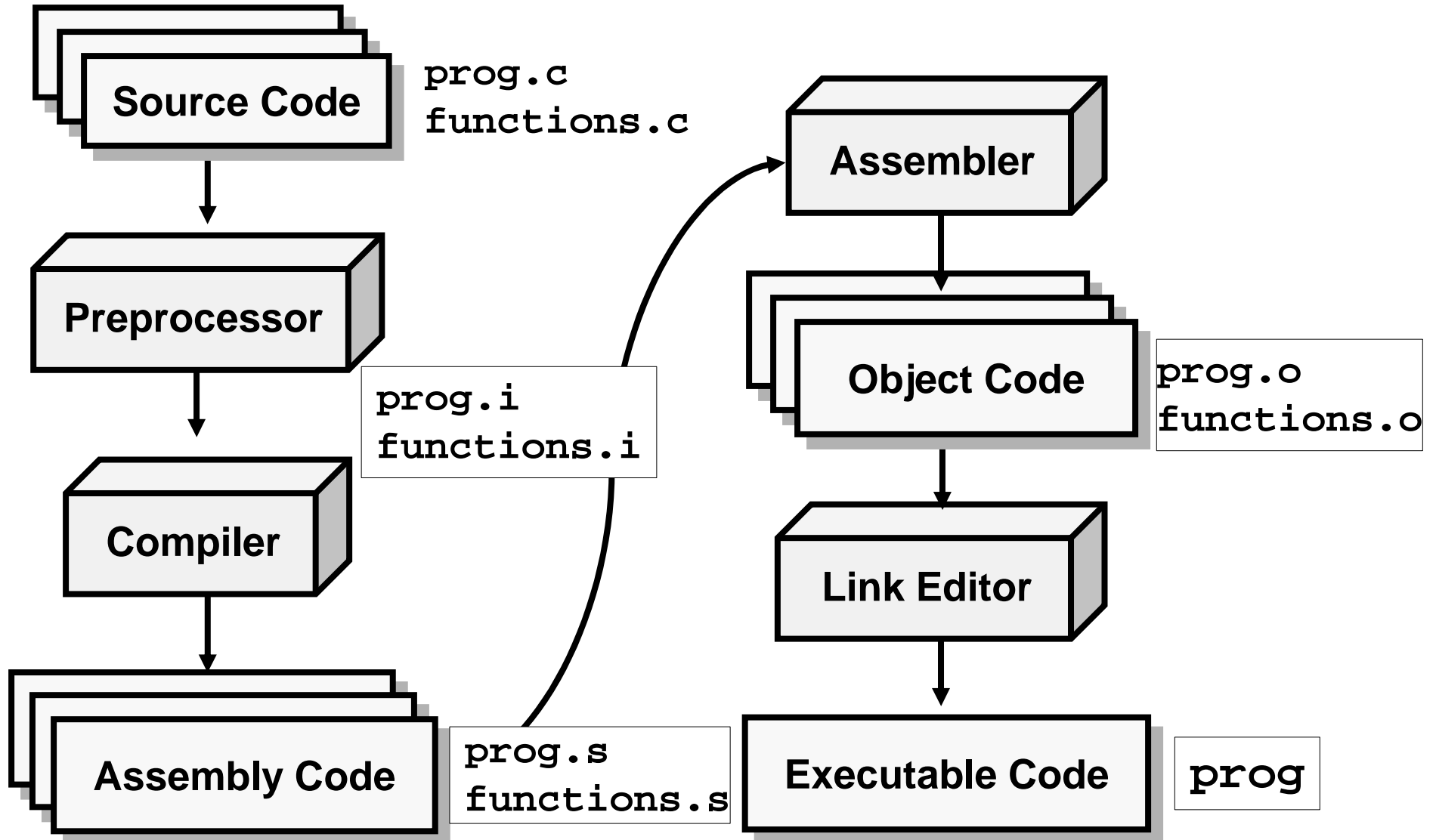
```
gcc -c prog.c
gcc -c functions.c

gcc -o prog prog.o functions.o
```

Source → preprocessed source → assembly → object → executable

```
gcc -E prog.c > prog.i
gcc -E functions.c > functions.i
gcc -S prog.i
gcc -S functions.i
gcc -c prog.s
gcc -c functions.s
gcc -o prog prog.o functions.o
```

# Compiling in C (via `gcc`)



## By the way...

- In big projects, it's hard to keep track of which object files are up to date and which need to be compiled
  - Linking with an out-of-date object file can cause major problems
  - It can take hours to compile the whole project so efficiency is important
- So how can we easily keep track of all this?
  - **Makefiles** (this week's discussion session)

```
gcc -o prog prog.c iofuncts.c ui.c calc.c misc.c
```

### prog.c

```
#include "iofuncts.h"  
#include "ui.h"  
#include "calc.h"  
#include "misc.h"
```

```
int main (void)  
{  
    ...  
    ...  
    ...  
    return(0);  
}
```

### iofuncts.c

```
#include "iofuncts.h"  
...
```

### ui.c

```
#include "ui.h"  
...
```

### calc.c

```
#include "calc.h"  
...
```

### misc.c

```
#include "misc.h"  
...
```

# Multiple file structure (4) – Library

```
prog.c

#include <mt.h>

int main (void)
{
    ...
    ...
    ...
    return(0);
}
```

```
mt.h

#include "iofuncts.h"
#include "ui.h"
#include "calc.h"
#include "misc.h"
```

iofuncts.o  
ui.o  
calc.o  
misc.o

} libmt.a

```
gcc -o prog prog.c -lmt
```

## How to create a library – **ar**

```
ar rs libmt.a iofuncts.o ui.o calc.o misc.o
```

- Compile with **-lmt**

```
gcc -o prog prog.c -lmt -L <dir>
```

Use the “libmt.a” library file

Tells the compiler where to look for the “mt” library.

Not needed if **libmt.a** is in a directory that’s in the default library path

```
setenv LIBRARY_PATH <dirs>
```

## Some libraries in /usr/lib, /usr/include

<u>Lib file</u>	<u>Header file</u>	<u>gcc flag</u>
libcrypt.a	crypt.h	-lcrypt
libjpeg.a	jpeglib.h	-ljpeg
libm.a	math.h	-lm
libogg.a	ogg.h	-logg
libgdbm.a	gdbm.h	-lgdbm
libss.a	ss.h	-lss

# The C standard library

- **printf**, **scanf**, **getchar**, etc. aren't part of the basic C language, yet we can use them without explicitly linking to a library. How???
- The Standard C Library
  - Provides a set of common support functions
  - Appendix B of K&R
- Some of the standard library functions are automatically linked (e.g., **printf( )**); others are not (e.g., **sin( )**)  
Requires the math library, **libm.a (-lm)**