CSE 303
Lecture 16

Multi-file (larger) programs

reading: *Programming in C* Ch. 15

slides created by Marty Stepp
http://www.cs.washington.edu/303/

Motivation

- single-file programs do not work well when code gets large
  - compilation can be slow
  - hard to collaborate between multiple programmers
  - more cumbersome to edit

- larger programs are split into multiple files
  - each file represents a partial program or *module*
  - modules can be compiled separately or together
  - a module can be shared between multiple programs

Partial programs

- A `.c` file can contain a partial program:

```
#include <stdio.h>

void f1(void) {                    // part1.c
    printf("this is f1\n");
}
```

- such a file cannot be compiled into an executable by itself:

```
$ gcc part1.c
/usr/lib/gcc/crt1.o: In function `_start':
(.text+0x18): undefined reference to `main'
collect2: ld returned 1 exit status
```

Using a partial program

- We have `part2.c` that wants to use the code from `part1.c`:

```
#include <stdio.h>

void f2(void);                  // part2.c

int main(void) {
    f1();                       // not defined!
    f2();
}

void f2(void) {
    printf("this is f2\n");
}
```

- The program will not compile by itself:

```
$ gcc -o combined part2.c
In function `main':
part2.c:6: undefined reference to `f1'
```

Including .c files (bad)

- One solution (bad style): include `part1.c` in `part2.c`

```
#include <stdio.h>
#include "part1.c"          // note "" not <>

void f2(void);

int main(void) {
    f1();                   // defined in part1.c
    f2();
}

void f2(void) {
    printf("this is f2\n");
}
```

- The program will compile successfully:

```
$ gcc -g -Wall -o combined part2.c
```

Multi-file compilation

```
#include <stdio.h>

void f2(void);                  // part2.c

int main(void) {
    f1();                       // not defined?
    f2();
}

void f2(void) {
    printf("this is f2\n");
}
```

- The gcc compiler can accept multiple source files to combine:

```
$ gcc -g -Wall -o combined part1.c part2.c
$ ./combined
this is f1
this is f2
```

Object (.o) files

- A partial program can be compiled into an *object (.o) file* with **`-c`** :

```
$ gcc -g -Wall -c part1.c
$ ls
part1.c   part1.o   part2.c
```

  - a `.o` file is a binary blob of compiled C code that cannot be directly executed, but can be directly inserted into a larger executable later

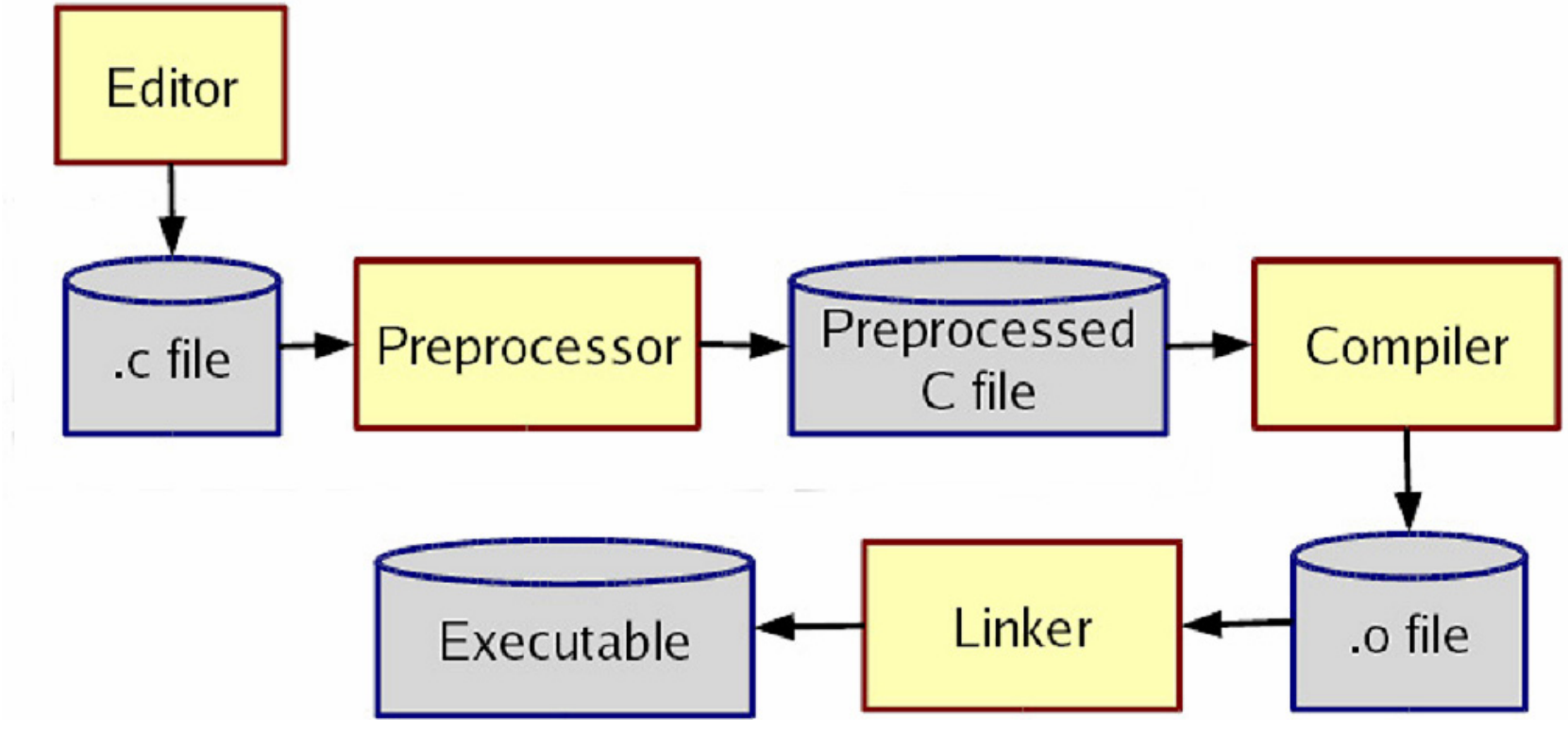
- You can compile a mixture of `.c` and `.o` files:

```
$ gcc -g -Wall -o combined part1.o part2.c
```

  - avoids recompilation of unchanged partial program files

The compilation process

- each step's output can be dumped to a file, depending on arguments passed to gcc



Problem

- with the previous code, we can't safely create `part2.o` :

```
$ gcc -g -Wall -c part2.c
part2.c: In function `main':
part2.c:6: warning: implicit declaration of function `f1'
```

- The compiler is complaining because f1 does not exist.
  - But it *will* exist once `part1.c/o` is added in later

- we'd like a way to be able to declare to the compiler that certain things will be defined later in the compilation process...

Header files

- **header** : A C file whose only purpose is to be included.
  - generally a filename with the `.h` extension
  - holds shared variables, types, and function declarations

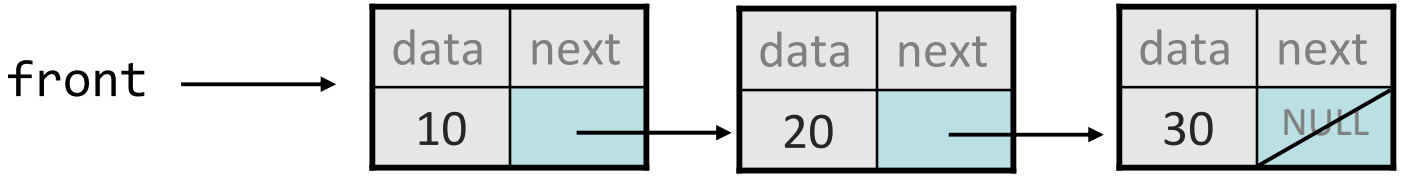
- key ideas:
  - every ***name*`.c`** intended to be a module has a ***name*`.h`**
  - ***name*`.h`** declares all global functions/data of the module
  - other `.c` files that want to use the module will #include ***name*`.h`**

- some conventions:
  - `.c` files never contain global function prototypes
  - `.h` files never contain definitions (only declarations)
  - never `#include` a `.c` file (only `.h` files)
  - any file with a .h file should be able to be built into a `.o` file

Exercise

- Write a program that can maintain a linked list of integers.
  - You should have functions for printing a linked list and summing it.
  - The `main` function should create a list, put some elements in it, and print/sum them.

- Appropriately divide your program into multiple `.c` and `.h` files.



Multiple inclusion

- *problem* : if multiple modules include the same header, the variables/functions in it will be declared twice

- *solution* : use preprocessor to introduce conditional compilation
  - convention: `ifndef`/`define` with a variable named like the .h file
  - first time file is included, the preprocessor won't be defined
  - on inclusions by other modules, will be defined → not included again

```
#ifndef _FOO_H
#define _FOO_H
... // contents of foo.h
#endif
```

Global visibility

```
// example.c
int passcode = 12345;

// example2.c
int main(void) {
    printf("Password is %d\n", passcode);
    return 0;
}
```

- by default, global variables/functions defined in one module can be seen and used by other modules it is compiled with
  - *problem* : `gcc` compiles each file individually before linking them
  - if `example2.c` is compiled separately into a .o file, its reference to passcode will fail as being undeclared

extern

```
// example2.c
extern int passcode;
...
printf("Password is %d\n", passcode);
```

- **extern** (when used on variables/functions) :
  - does not actually define a variable/function or allocate space for it
  - instead, promises the compiler that some *other* module will define it
  - allows your module to compile even with an undeclared variable/function reference, so long as eventually its `.o` object is linked to some other module that declares that variable/function
    - if `example.c` and `example2.c` are linked together, the above will work

static

```
// example.c
int passcode = 12345;                  // public
static int admin_passcode = 67890;     // private
```

- **static** (when used on global variables/functions) :
  - visible only to the current file/module (sort of like Java's `private`)
  - declare things `static` if you do not want them exposed
  - avoids potential conflicts with multiple modules that happen to declare global variables with the same names

  - `passcode` will be visible through the rest of `example.c`, but not to any other modules/files compiled with `example.c`

Function static data

- When used inside a function:

```
static type name = value;
```

  - declares a static local variable that will be remembered across calls

Example:

```
int nextSquare() {
    static int n = 0;
    static int increment = 1;
    n += increment;
    increment += 2;
    return n;
}
```

`nextSquare()` returns 1, then 4, then 9, then 16, ...