CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska

Winter 2007

Lecture 14 – The C Preprocessor

Tools: introduction to the linker

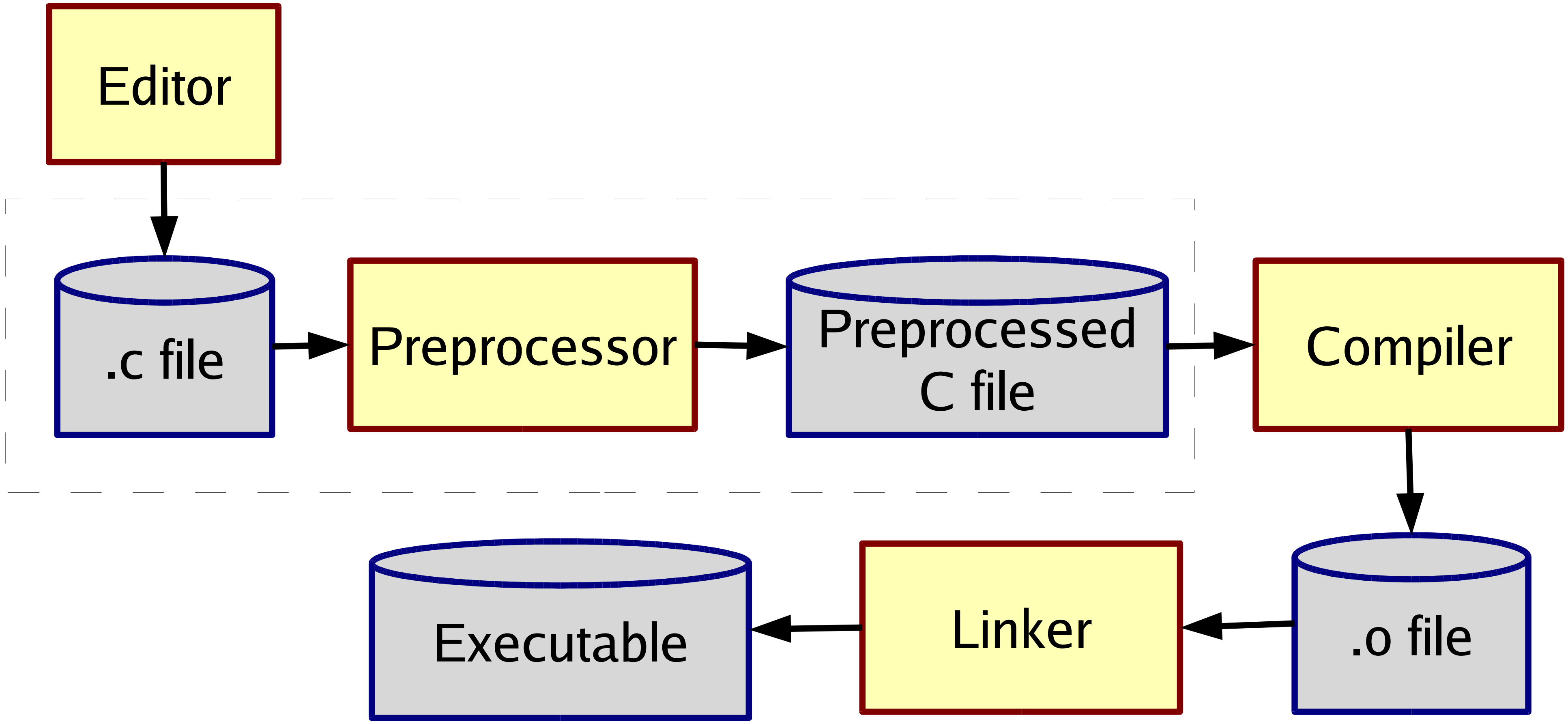
Where We Are

- After today, we will have covered
  - Linux (just an introduction to Linux)
  - Shell scripts and utilities
  - Programming in C

- This week an in future weeks, we will cover
  - Tools
  - C++
  - Introduction to software engineering

Steps Involved in Creating a C Program

- Preprocessing occurs before compilation
- Use `gcc -E` to perform only preprocessing



C Preprocessor

- All preprocessor directives begin with pound sign: #

- Three main uses of C preprocessor
  - Include files
  - Define symbolic constants and macros
  - Compile parts of code conditionally

Preprocessor: Including Files

- The `#include` *directive*
  - Causes a copy of a specified file to be included in place of the directive
  - File is itself preprocessed before being included
- `#include <filename>`
  - Search in pre-defined system include file directories (these directories are implementation dependent)
  - Used for standard libraries
- `#include “filename”`
  - Search in local directory

Compiler -I option

- `gcc -I dir ...`
  - Add the directory `dir` to the list of directories to be searched for header files
  - Directories named by `-I` are searched before the standard system include directories

- Example `include.c, includeA.h, headers/includeB.h`

Preprocessor: Defining Constants

- The `#define` *directive*
  - Creates symbolic constants and macros
- `#define id text`
  - All subsequent occurrences of `id` are replaced with `text` *before program is compiled*
- `#define BUFFER_SIZE 4096`
- `#define DEFAULT_FILE “output.txt”`
- Examples: `constant.c`
  - `stdbool.h` defines `bool`, `true`, and `false`
  - `stddef.h` define `NULL`

Preprocessor: Defining Macros

- A lot like constants, but can take *arguments*
- During preprocessing
  - Step 1: Arguments are substituted
  - Step 2: Macro is *expanded*
- `#define SUM(x,y) ( (x) + (y) )`
- Then
  - `int a = SUM(3,4);`
  - Becomes `int a = ( (3) + (4) );`
- Examples: `macro.c`

More about Macros

- Try to avoid them if you can
  - It is better to use functions!
  - Your goal: clarity and correctness
  - Do not worry about optimization until you know that something is a bottleneck
- Use them only when truly needed

```
#define PRINT(x) \
printf("%s:%d %s\n",__FILE__,__LINE__,x);
```

- (`__FILE__` and `__LINE__` are predefined symbolic constants)

Preprocessor: Conditional Constructs

- Preprocessor supports other useful statements
  - `#if`, `#else`, `#endif`, `#ifdef`, etc.
- These statements enable programmers to control
  - Execution of preprocessor directives
  - Compilation of program code
  - By switching various statements on or off

Typical Usage 1

- Ensure header files are included only once

```
#ifndef INCLUDEA_H
#define INCLUDEA_H
... content of includeA.h ...
#endif
```

- Check if symbolic name is already defined
- If not, then define it

- Example: `include2.c`, `includeA.h`, `includeB.h`, and `includeC.h`

Typical Usage 2

- Conditional compilation

```
#ifdef DEBUG
#define PRINT(x) printf("%s”,x);
#else
#define PRINT(x)
#endif
```

- Example: `conditional.c`
  - `gcc -D DEBUG conditional.c`
  - `gcc conditional.c`
- Other usage: adapt code to architecture, OS

Typical Usage 2 (Example 2)

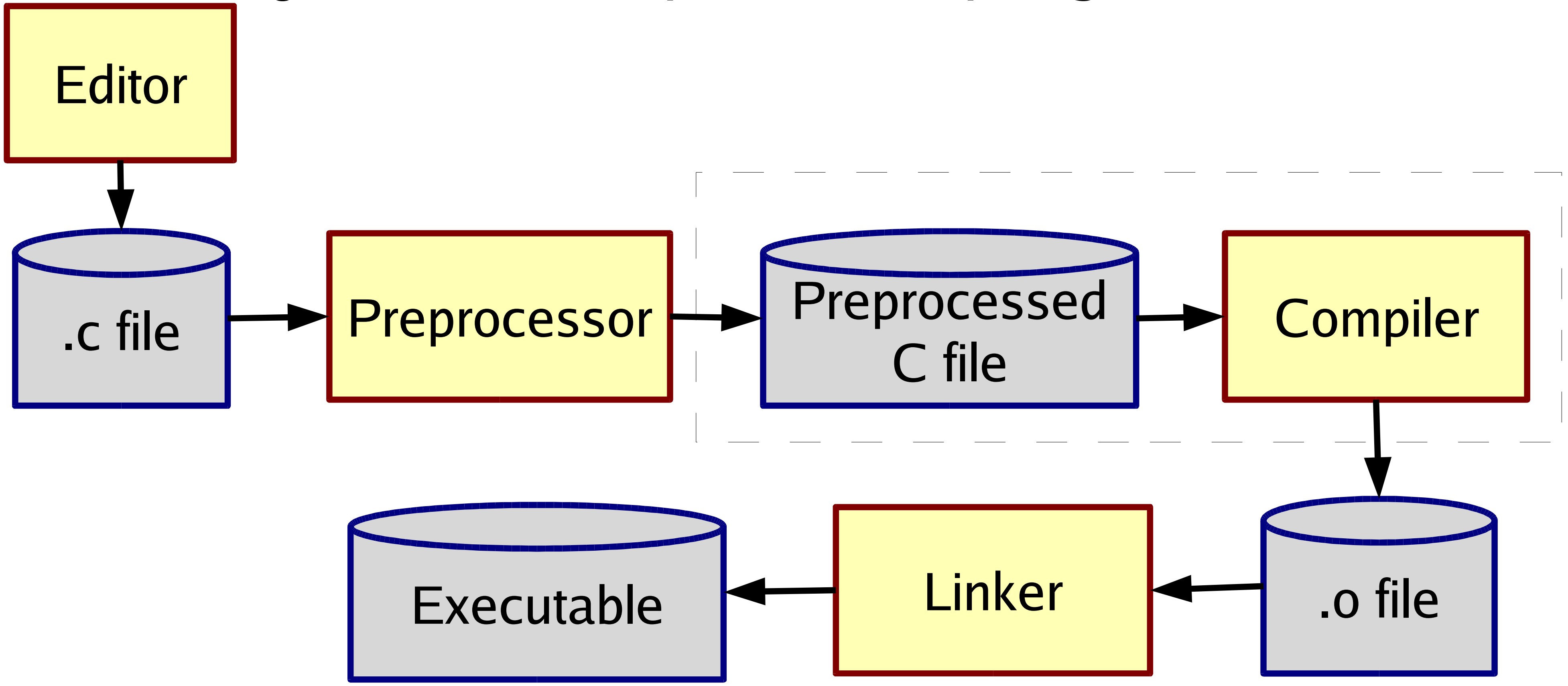
- Example: `fancy-conditional.c`
  - `gcc -D LOG_LEVEL=2 fancy-conditional.c`
  - `gcc -D LOG_LEVEL=1 fancy-conditional.c`
  - `gcc fancy-conditional.c`

Useful macro: assert (in assert.h)

- Usage: `assert(expression)`
  - If value of expression is true, nothing happens
  - If value of expression is false, assert prints an error message and calls abort
- Especially useful for
  - Testing preconditions (example stack not empty)
- Example: `assert.c`
- Disable asserts by defining `NDEBUG`
  - `gcc -D NDEBUG assert.c`

Steps Involved in Creating a C Program

- Compiler transforms source code (.c files) into machine language code, a.k.a. object code (.o files)
- Use `gcc -c` to stop after compiling

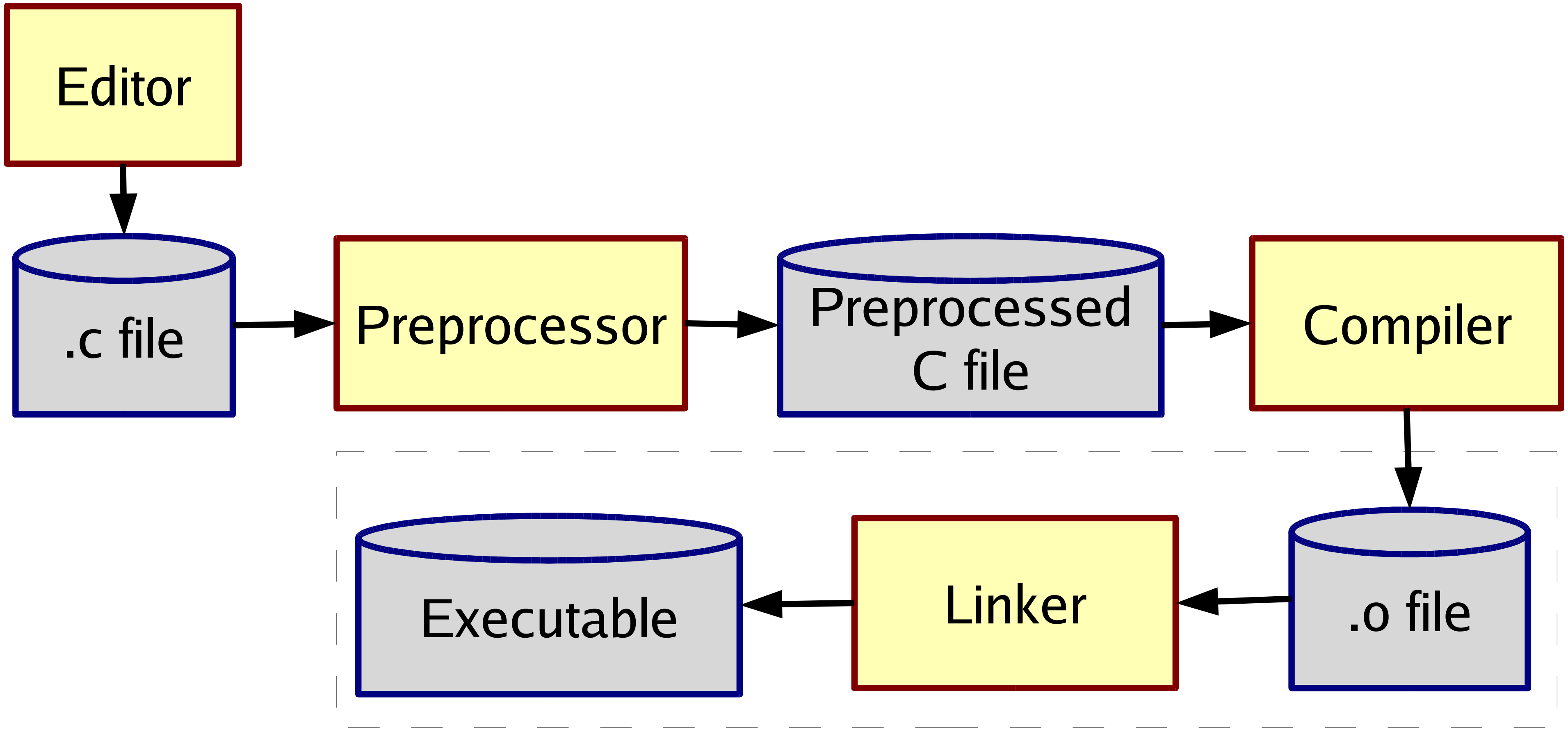


The Goal of the Linker

- Use option `-c` to produce the `.o` file
- Compiled code (`.o` file) is not “runnable”
- We have to link it with other code to make an executable
  - Where is the code for `printf` and `malloc`?
  - We only included the header files…
  - Need to find that code and put it in executable
  - That is what the linker does
- Normally, gcc/g++ hides this from you

Steps Involved in Creating a C Program

- Linker transforms compiled code (.o files) into executable programs



Linking Overview

- If a C/C++ file uses but does not define a function (or global variable), then the `.o` has "undefined references"
  - Note: declarations do not count, only definitions
- Linker takes multiple .o files and "patches them" to include the references
- Executable has no unresolved references
- Linker is called `ld`, but we will not invoke it directly. We will use `gcc`... more next lecture

Readings

- Programming in C
  - Chapter 13
  - Chapter 18, section on “Debugging with the preprocessor”
  - Appendix C “Compiling prorams with gcc”

- Scheme through the man page for gcc
  - `man gcc`