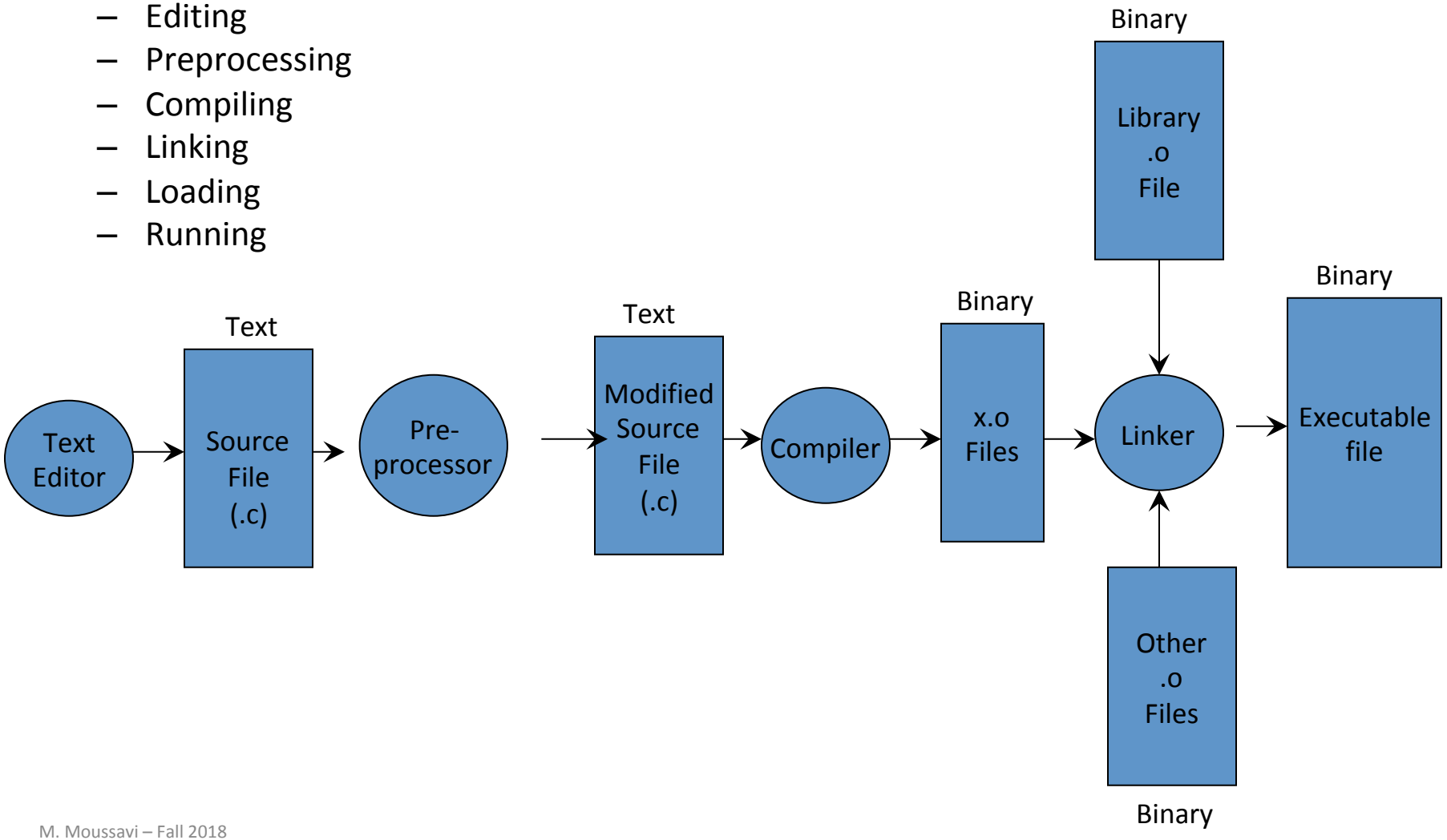
A Brief Introduction to Program Development and Compilation Process

Program Development and Compilation Process

- The process of developing a program and running an executable file consists of several operations:
  - Editing
  - Preprocessing
  - Compiling
  - Linking
  - Loading
  - Running



Preprocessing Directives

Readings from your textbook (C in a Nutshell)

Pages: 209 – 214 and 217 - 220

Processing Directives

- A simplified version of program compilation process which is a multi-step process was briefly explained in the first set of slides.
- One of the steps in this multi-step logical process was outlined as **preprocessor.** Some of the possible actions in this step include:
  - Defining constants:

    #define PI 3.14159

    No semicolon
  - Inserting the content of other source files

    #include <stdio.h>

    No semicolon
  - Identifying sections of code to be compiled under certain conditions
  - Defining function-like macros
- Preprocessing directives always start with # and doesn't need a semicolon.

Processing Directives

- The **preprocessor** uses text substitution to revise the source code.
  - You can use the `-E` option with the gcc to see the output of the preprocessor.
- Once the preprocessor completes text substitution, the revised text is processed by the compiler.
- In addition to defining constants (using #define) and including another source file (using #include) that we have used frequently in our C++ or C programs, there are several other application of preprocessor directives that we will discuss in this set of slides.

What is a C Macro?

C Macro

- In general a simple form of C Macro is in fact a symbolic name for a text
  - You can define macros in C using directive: #define
  - When the name of the macro appears in you code the preprocessor replaces it with the text
  - Examples:

```
#define MESSAGE "Hello C Macros"
#define SIZE 5

int main(void) {
   printf(”Today’s message is: %s", MESSAGE);
   double arr[SIZE];
   for(int i=0; i < SIZE; i++)
      arr[i] = 0.0;
   return 0;
}
```

- The rules for names is similar to naming rules for variables, etc. in C.

Macros with Parameters

- Form:
  - ***#define macro_name(parameter list) macro body***
  - Notes:
    - there must be no space between the name and the left parenthesis.
    - C99 allows for empty brackets, ().
- Example:

```
#include <stdio.h>
#define LABEL_PRINT_INT(label, num) printf("%s = %d", (label), (num))

int main(void)
{
    int i = 5;
    LABEL_PRINT_INT("rabbit", i);
    return 0;
}
```

- The preprocessor replaces the above with:

```
printf("%s = %d", ("rabbit"), (i));
```

Macros with Parameters (continued)

- Multiple substitution is also possible. For example:

```
#define CONTROL          "%d\n"
#define PRINT_INT(i)     printf(CONTROL, (i))
#define PRINT_POS_INT(i)  if ((i) > 0) PRINT_INT(i)

void main(void)
{
    PRINT_POS_INT(10);
}
```

The resulting expansion is:

```
    if ((10) > 0) printf("%d\n", (10))
```

- Lets take a look at the following simple program that uses a function called square to obtain the square of its argument x:

```
#include <stdio.h>
int square(int x)
{
   return (x * x);
}

int main(void)
{
   int a = 3, b;
   b = square(a++);
   printf("a = %d and b = %d .", a, b);
   return 0;
}
```

- What is the program output and why?
- Now lets convert the square function to a C macro.

Developing a C Macro

By convention we use capital letters. I could use square instead of SQUARE

- A possible definition:

```
#define SQUARE(x)  x * x
int main(void)
{
   int a = 3, b;
   b =  SQUARE(a);
   printf("a = %d and b = %d .", a, b);
   return 0;
}
```

- **WARNNING! THIS MACRO IS DEFECTIVE! WHY?**
- The output must be the same as previous version, using function square.
- What is the value of b after the following call? Are expected value is 25. Is it really 25?

```
b =  SQUARE(a + 2);
```

Developing a C Macro

- Slightly better solution.

The red brackets were added

```
#define SQUARE(x)   (x) * (x)
int main(void)
{
   int a = 3, b;
  b =  SQUARE(a);
   printf("a = %d and b = %d .", a, b);
   return 0;
}
```

- Now the value of b is 25
- **BUT STILL DEFECTIVE! WHY?**
- What is the value of b after the following call?

```
b =  100 / SQUARE(a + 2);
```

Developing a C Macro

- This one works definitely better.

The outer (green) brackets were added

```
#define SQUARE(x)   ((x) * (x))
int main(void)
{
   int a = 3, b;
   b =  SQUARE(a);
   printf("a = %d and b = %d .", a, b);
   return 0;
}
```

- **More Side effects? Maybe**

Macros with Parameters (continued)

- Example of unintended side effects:

```
#define SQUARE(x)   ((x) * (x))
…
void main(void)
{
      int a = 3, b;
      b = SQUARE(a++);
}
```

The following expansion takes place:

```
b = ((a++) * (a++));
```

- On my Mac the result is: **a == 5 and b == 12**
- Even, other implementations may result in different results

Macros with Parameters (continued)

Macro

- Macro code is repeated through the program code by text substitution.
- May result in larger program size.
- Faster execution since code is executed "in line."

Function

- Function code is stored only once in a program; execution will "go to" a function, then return to the calling code.
- May result in smaller program size.
- May result in slower execution.

- With the improvements in compilers the execution speed differences are becoming less and less significant.
- Macros may have unintended side effects, resulting in bugs which are difficult to debug.
- In general for most of the cases functions are a better choice.
- Considering that older code may use parameterized macros, so it is still important to understand how macros work.

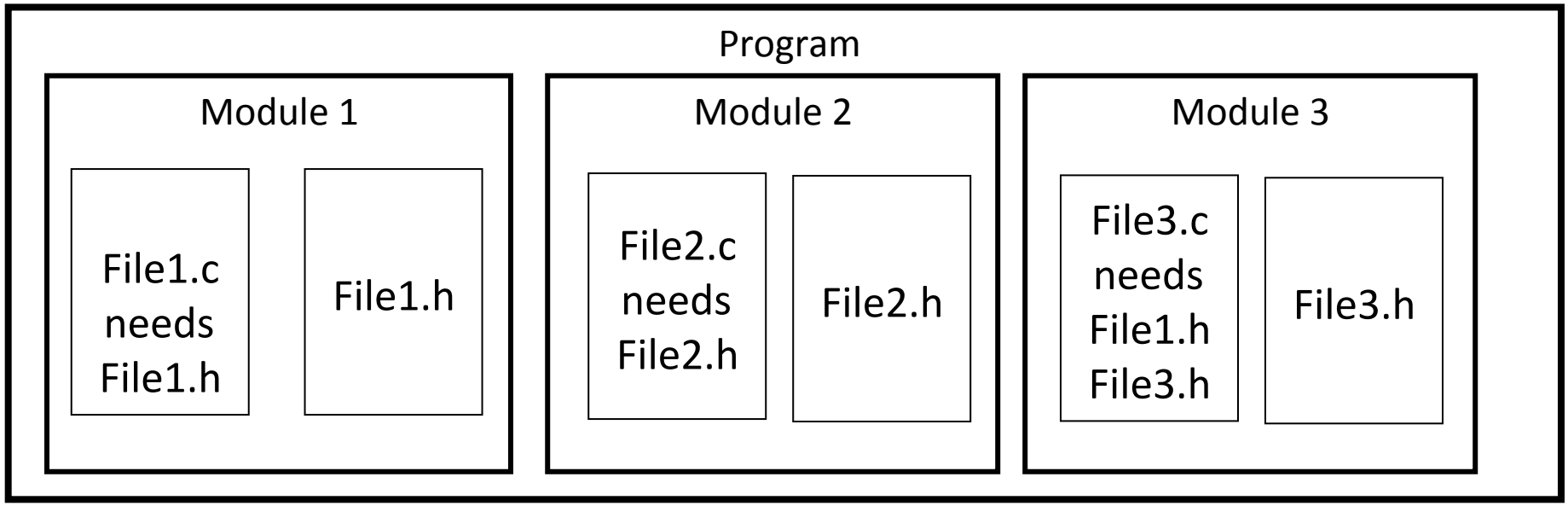
Programming in Large and Division of the Program

Divisions of Program

- A program may include one or more modules, and each module may contain one or more of the following type of files:
  - Implementation files
    - Example: myfile.c
  - Interface files
    - Example: myfile.h
- Interface part of a program normally contains:
    - Function prototypes
    - Constants
    - Definition of abstract data types, such as structures, and classes
    - Other Preprocessor directives: such as #include statements
    - Macros
    - Etc.
- Implementation part of a program normally contains:
  - Definition (implementation) of functions

Divisions of Programs

- Large programs are normally divided into several modules:
  - To divide the program into more manageable unit
  - To help the development process –different developers work on each module
  - To divide the program into more cohesive units



Divisions of Programs - Example

- C-Preprocessor can be used to prevent multiple inclusion of the header files.
- Example

```
// File1.h


#ifndef XYZ
#define XYZ
   // function prototypes
   // constants
   //  user defined data types such as structs, enum, etc.
#endif
```

Conditional Compilation

Conditional Compilation

- It's the technique of using the preprocessor to include or omit code, depending on defined conditions.
  - This is useful for cases such as:
    - A program runs on different computer systems, or is distributed in more than one version.
    - A program contains "debugging code" that needs to be turned on or off during development.
- General form:

```
#if defined(constant1)
// code to be compiled
#elif defined(constant2)
// other code to be compiled
#else
// default code to be compile
#endif
```

Conditional Compilation (continued)

- **Example:**

```
#define LINUX

#if defined(LINUX)
     system("clear");      // uses Linux system function
#elif defined(DOS)
     system("cls");        // uses DOS system function
#endif
```

- Note that:
  - `#ifdef` is the same as `#if defined()`
  - `#ifndef` is the same as `#if !defined()`

Conditional Compilation (continued)

- Another common use of Conditional Compilation is to eliminate the the debug code form regular code:

```
#define DEBUG
…
…
void main(void)
{
     ...
#if defined(DEBUG)
…              // this code will be compiled
…              //  only for debugging
#endif
     ...
     ...
}
```

- Use can also use -D option when compiling if you want to turn on particular code.
- For example:

```
gcc –Wall -DDEBUG myprog.c
```

Conditional Compilation (continued)

- You can temporarily comment out large sections of code by using `#if 0` and `#endif`.
  - This is useful, since comments in C don't nest.
  - You can uncomment this code by changing the 0 to a 1.
  - Example:

```
...
#if 0
   ...      // Code here is temporarily commented
   ...      // out, regardless of comments that
   ...      // already exist
#endif
...
```

- Preprocessor directives can also nest:

```
#if defined(DEBUG)
...
#if defined(ALPHA)
...
...

#endif
...
#endif
```