CS1010 http://www.comp.nus.edu.sg/~cs1010/

Programming Methodology

UNIT 13

Separate Compilation

Unit 13: Separate Compilation

Objective:

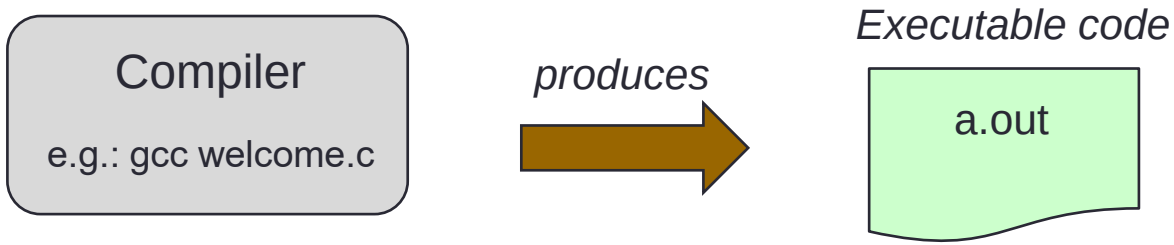
- Learn how to use separate compilation for program development

Unit 13: Separate Compilation

1. Introduction
2. Separate Compilation
3. Notes

1. Introduction (1/4)

- So far we have compiled our programs directly from the source into an executable:



- For the development of large programs with teams of programmers the following is practised
  - "Break" the program into multiple modules (files)
  - Compile the modules separately into object files (in C)
  - Link all object files into an executable file

1. Introduction (2/4)

- Header Files and Separate Compilation
  - Problem is broken into sub-problems and each sub-problem is tackled separately – divide-and-conquer
  - Such a process is called modularization
  - The modules are possibly implemented by different programmers, hence the need for well-defined interfaces
  - The function prototype constitutes the interface (header file). The function body (implementation) is hidden – abstraction
  - Good documentation (example: comment to describe what the method does) aids in understanding

1. Introduction (3/4)

- Example of documentation
  - The function header is given
  - A description of what the function does is given
  - How the function is implemented is not shown

```
double pow(double x, double y);
// Returns the result of raising
// x to the power of y.
```

C library function - pow()

Advertisements

Previous Page Next Page

Description

The C library function **double pow(double x, double y)** returns **x** raised to the power of **y** i.e. x^y.

Declaration

Following is the declaration for pow() function.

```
double pow(double x, double y)
```

Parameters

- **x** -- This is the floating point base value.
- **y** -- This is the floating point power value.

Return Value

This function returns the result of raising x to the power y.

Example

The following example shows the usage of pow() function.

Try it

```
#include <stdio.h>
#include <math.h>

int main ()
{
   printf("Value 8.0 ^ 3 = %lf\n", pow(8.0, 3));

   printf("Value 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));

   return(0);
}
```

1. Introduction (4/4)

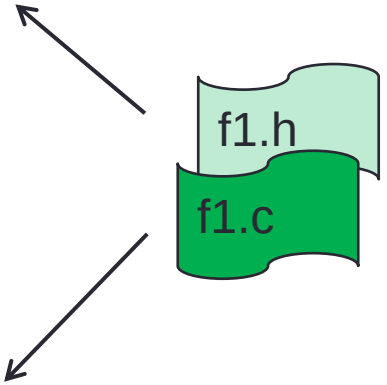
- Reason for Modular Programming
  - Divide problems into manageable parts
  - Reduce compilation time
    - Unchanges modules do not eed to be re-compiled
  - Facilitate debugging
    - The modules can be debugged separately
    - Small test programs can be written to test the functions in a module
  - Build libraries of useful functions
    - Faster development
    - Do not need to know how some functionality is implemented, e.g., image processing routines
    - Example: OpenCV – a computer vision library.

2. Separate Compilation (1/2)

- From http://encyclopedia2.thefreedictionary.com/
- Separate Compilation:
  - A feature of most modern programming languages that allows each program module to be compiled on its own to produce an object file which the linker can later combine with other object files and libraries to produce the final executable file.
- Advantages
  - Separate compilation avoids processing all the source code every time the program is built, thus saving development time. The object files are designed to require minimal processing at link time. The can also be collected together into libraries and distributed commercially without giving away source code (through they can be disassembled).
- Examples of output of separate compilation:
  - C object files (.o files) and Java .class files.

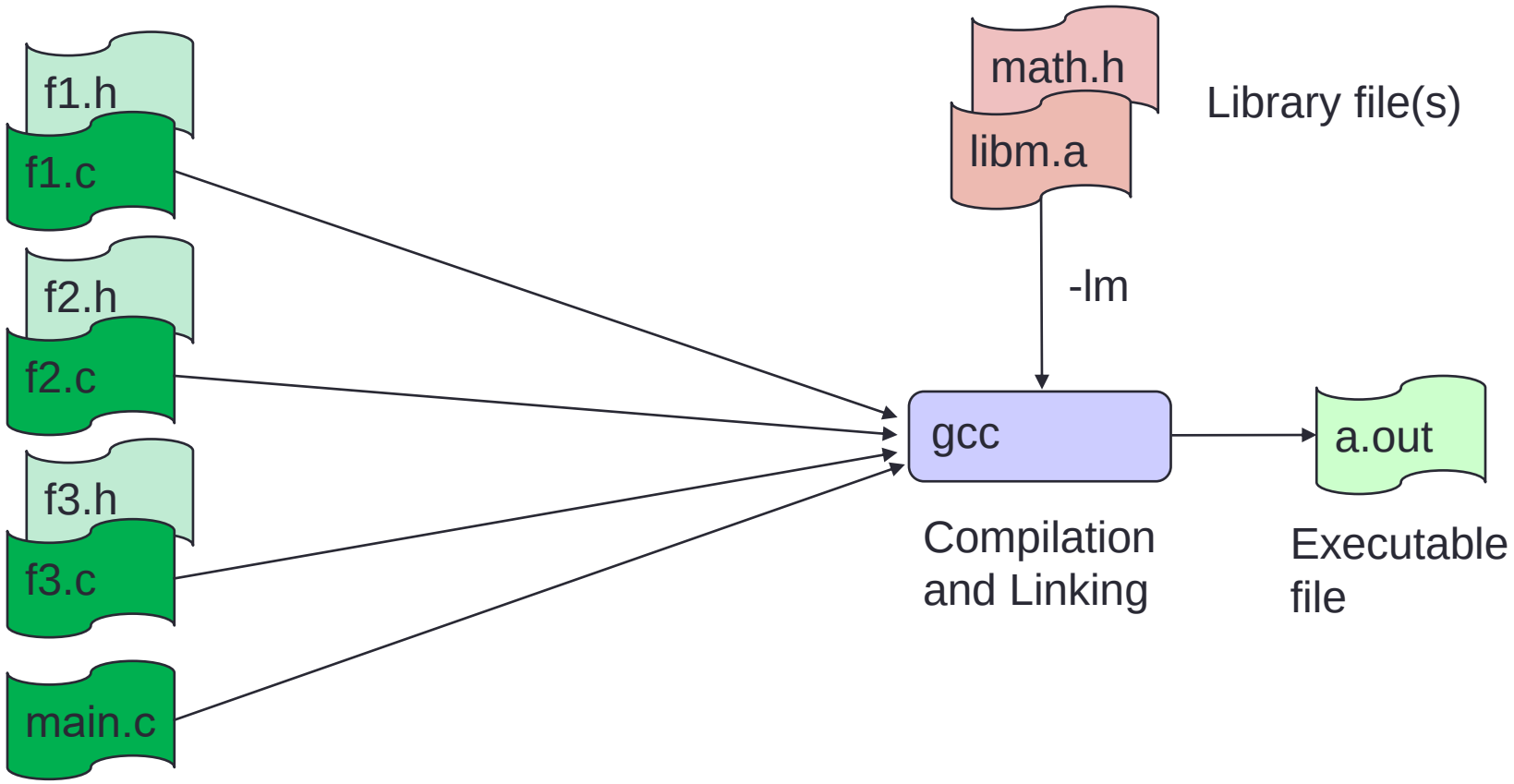
2. Separate Compilation (2/2)

- In most cases, a module contains functions that are related, e.g., math functions.
- A module consists of
  - A header file (e.g. f1.h) which contains:
    - Constant definitions, e.g.:
      - `#define MAX 100`
    - Function prototypes, e.g.:
      - `double mean(double, double);`
  - A source file (e.g. f1.c) which contains:
    - The functions that implement the function prototypes in the header file (e.g., the code for the function `mean(…)`).
    - Other functions, variables, and constants that are only used within the module (i.e., they are module-local).



2.1 Separate Compilation: Case 1

Case 1:
All the source files are compiled and linked in one step.



2.1 Case 1 Demo

- Let's re-visit the Freezer version 2 program in Unit 4 Exercise 6. We will create a module that contains a function to calculate the freezer temperature:
  - Module header file:

Unit13_FreezerTemp.h

```
// Compute new temperature in freezer
float calc_temperature(float);
```

  - Module source file:

Unit13_FreezerTemp.c

```
#include <math.h>

// Compute new temperature in freezer
float calc_temperature(float hr) {
    return ((4.0 * pow(hr, 10.0))/(pow(hr, 9.0) + 2.0)) - 20.0;
}
```

2.1 Case 1 Demo: Main Module

Unit13_FreezerMain.c

```c
#include <stdio.h>
#include "Unit13_FreezerTemp.h"

int main(void) {
    int hours, minutes;
    float hours_float;  // Convert hours and minutes into hours_float
    float temperature;  // Temperature in freezer

    // Get the hours and minutes
    printf("Enter hours and minutes since power failure: ");
    scanf("%d %d", &hours, &minutes);

    // Convert hours and minutes into hours_float
    hours_float = hours + minutes/60.0;

    // Compute new temperature in freezer
    temperature = calc_temperature(hours_float);

    // Print new temperature
    printf("Temperature in freezer = %.2f\n", temperature);

    return 0;
}
```

Include the header file (Note "..." instead of <…>).
Header file should be in the same directory as this program.

Now we can write a program which uses the new external function

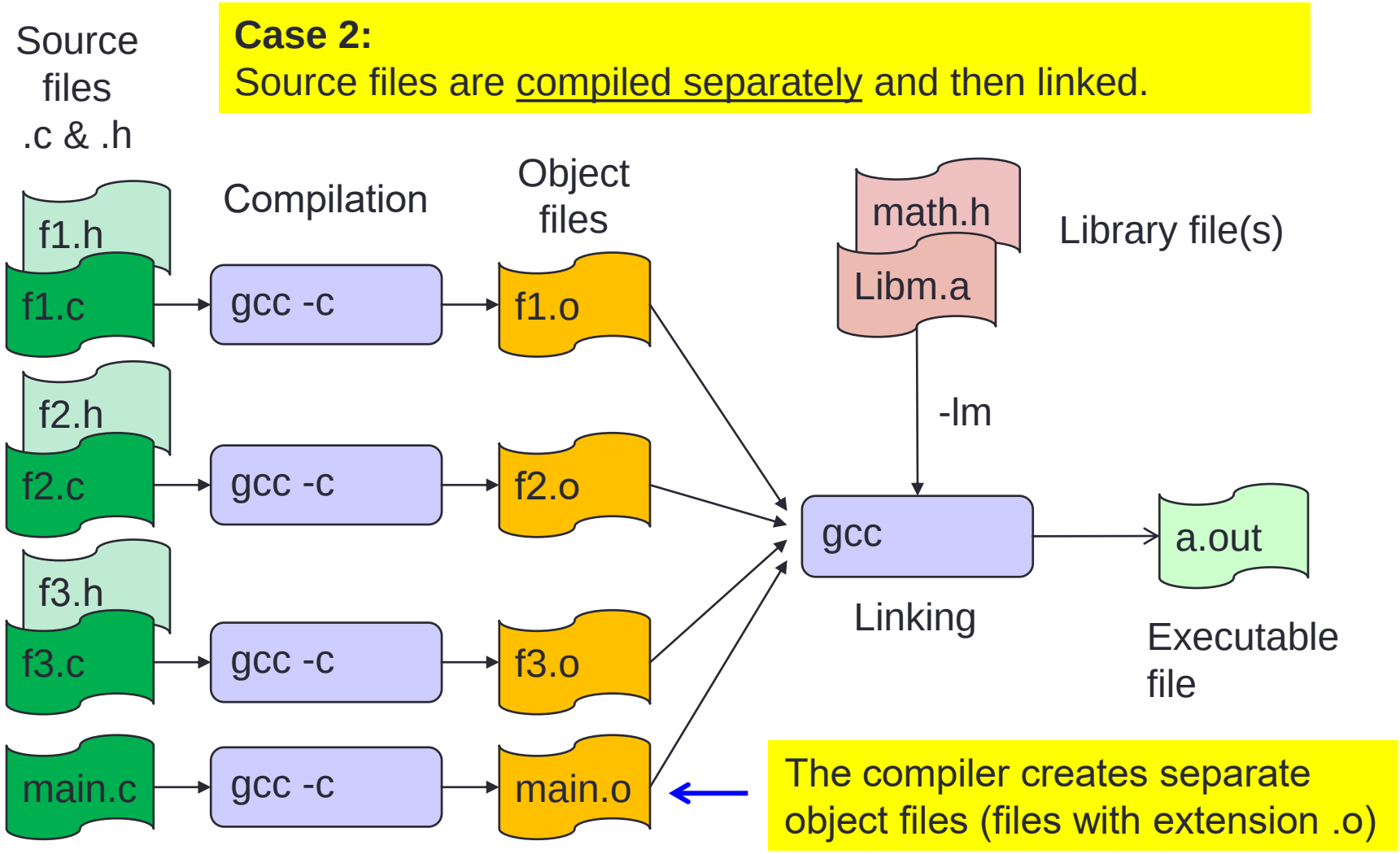
2.1 Case 1 Demo: Compile and Link

- How do we run Unit13_FreezerMain.c, since it doesn't contain the function definition of calc_temperature()?
- Need to compile and link the programs

```
$ gcc Unit13_FreezerMain.c Unit13_FreezerTemp.c -lm
```

- Here, the compiler creates temporary object files (which are immediately removed after linking) and directly creates a.out
- Hence, you don't get the chance to see the object files (files with extension .o)
- (Note: The option –Wall is omitted above due to space constraint. Please add the option yourself.)

2.2 Separate Compilation: Case 2



2.2 Case 2 Demo: Compile and Link

- For our Freezer program:

```
$ gcc -c Unit13_FreezerMain.c
$ gcc -c Unit13_FreezerTemp.c
$ gcc Unit13_FreezerMain.o Unit13_FreezerTemp.o -lm
```

- Here, we first create the Unit13_FreezerMain.o and Unit13_FreezerTemp.o object files, using the –c option in gcc.
- Then, we link both object files into the a.out executable
- (Note: The option –Wall is omitted above due to space constraint. Please add the option yourself.)

3. Notes (1/2)

- Difference between
  - #include < … > and #include " … "
  - Use " … " to include your own header files and < … > to include system header files. The compiler uses different directory paths to find < … > files.
- Inclusion of header files
  - Include *.h files only in *.c files, otherwise duplicate inclusions may happen and later may create problems:
    - Example: Unit13_FreezerTemp.h includes <math.h>
      Unit13_FreezerMain.c includes <math.h> and
      "Unit13_FreezerTemp.h"
      Therefore, Unit13_FreezerMain.c includes <math.h> twice.

3. Notes (2/2)

- ‘Undefined symbol’ error
  - ld: fatal: Symbol referencing errors.
  - The linker was not able to find a certain function, etc., and could not create a complete executable file.
    - Note: A library can have missing functions → it is not a complete executable.
  - Usually this means you forgot to link with a certain library or object file. This also happens if you mistyped a function name.

Summary

- In this unit, you have learned about
  - How to split a program into separate modules, each module containing some functions
  - How to separately compile these modules
  - How to link the object files of the modules to obtain the single executable file

End of File