COMsW 1003-1

Introduction to Computer Programming in C

Lecture 18

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Modular Programming

Review - Header files

- Header files are fundamentally libraries
- Their extension is .h
- They contain function definitions, variables declarations, macros
- In order to use them, the preprocessor uses the following code

```
#include <nameOfHeader.h>
#include "nameOfHeader.h"
```

→ For standard C libraries

→ For user defined headers

- So far, we have used predefined C header files, but we can create our own! (more on this next week)

Modular Programming

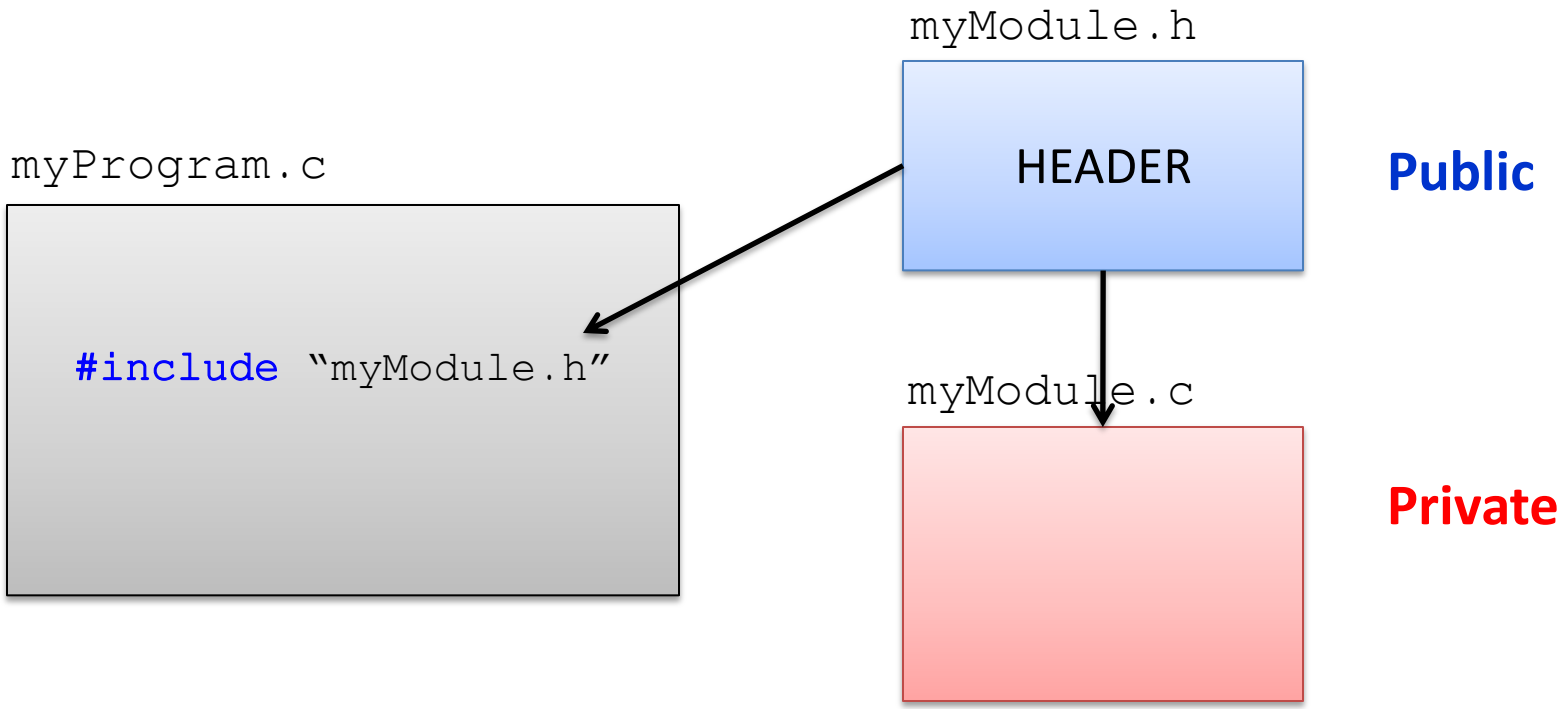
- So far we have seen only small programs, in one single file
- What about bigger programs? Need to keep them organized, especially if multiple people work on the same project
- They are organized in multiple, organized parts : MODULES

Modules

- A module is “a collection of functions that perform related tasks” [PCP Ch18]
- A module is basically a **user defined library**
- Two parts:
  - Public : tells the user how to use the functions in the module. Contains declaration of data structures and functions
  - Private : implements the functions in the module

Modules

- Two parts:
  - **Public** : tells the user how to use the functions in the module. Contains definition of data structures and functions
  - **Private** : implements the functions in the module



Header

- A header should contain:
  - A section describing what the module does
  - Common constants
  - Common structures
  - Public functions declarations
  - **Extern** declarations for public variables

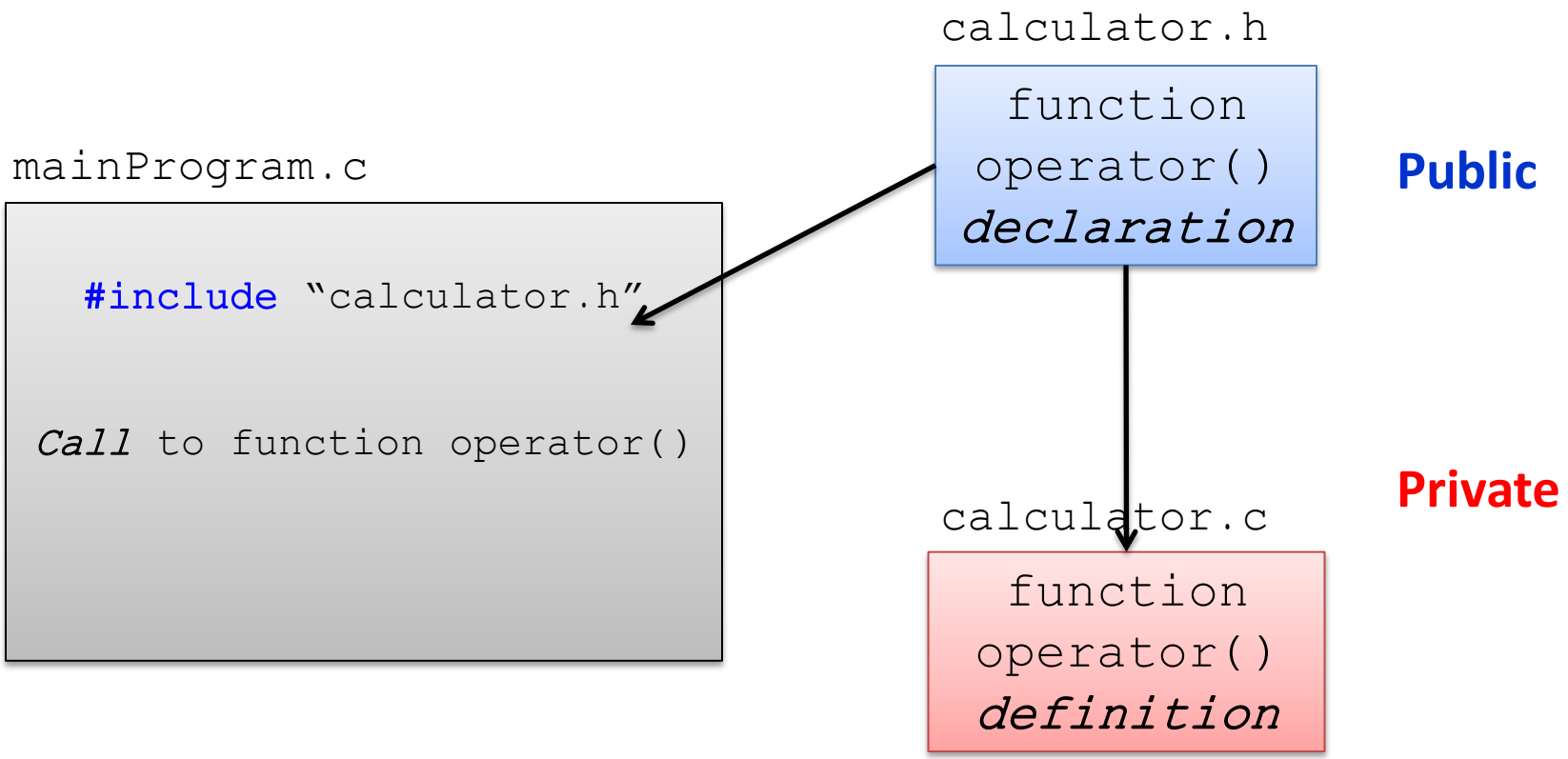
Function Declaration vs. Definition

- All identifiers in C need to be declared before they are used, including functions
- Function declaration needs to be done before the first call of the function
- The **declaration** (or **prototype**) includes
  - return type
  - number and type of the arguments

- The function **definition** is the actual implementation of the function
- Function definition can be used as implicit declaration

Modules

mainProgram.c
calculator.h
calculator.c



Compile modules together

- We need a way to “glue” the modules together
- We need to compile not only the main program file, but also the user defined modules that the program uses
- Solution : makefile

Makefile

- `make` routine offered in UNIX (but also in other environments)

- `make` looks at the file named *Makefile* in the same folder and invokes the compiler according to the **rules** in *Makefile*

Makefile – Single file

```
#-----------------------------------------------------#
#     Makefile for UNIX system                        #
#     using a GNU C compiler (gcc)                    #
#-----------------------------------------------------#

# this is a comment

oldCalculator: oldCalculator.c
	gcc –Wall -o oldCalculator oldCalculator.c
```

Makefile – Single file

```
#-----------------------------------------------------#
#      Makefile for UNIX system                       #
#      using a GNU C compiler (gcc)                   #
#-----------------------------------------------------#
```

```
# this is a comment
```

→ Comments start with a # sign

```
oldCalculator: oldCalculator.c
	gcc –Wall -o oldCalculator oldCalculator.c
```

Rule: gcc command we are used to

The second statement MUST start with a TAB!

Makefile – Single file

```
#-----------------------------------------------------#
#     Makefile for UNIX system                        #
#     using a GNU C compiler (gcc)                    #
#-----------------------------------------------------#

CC=gcc
CFLAGS=-Wall

oldCalculator: oldCalculator.c
	$(CC) $(CFLAGS) -o oldCalculator oldCalculator.c
```

Makefile – Single file

```
#-----------------------------------------------------#
#     Makefile for UNIX system                        #
#     using a GNU C compiler (gcc)                    #
#-----------------------------------------------------#
```

```
CC=gcc
CFLAGS=-Wall
```

macros

```
oldCalculator: oldCalculator.c
	$(CC) $(CFLAGS) -o oldCalculator oldCalculator.c
```

Rule: gcc command we are used to

The second statement MUST start with a TAB!

Makefile

- Macros

```
name=data
$(name) ⟶ data
```

Whenever $(name) is found, it gets substituted with data
Same as object-type macros for Preprocessor

- Rules

```
target: source [source2] [source3] …
        command
        command2
        command3
            ⋮
```

UNIX compiles `target` from `source` using `command`

Default command is `$(CC) $(CFLAGS) -c source`

Predefined by make

Makefile – Single file

```
#-----------------------------------------------------#
#     Makefile for UNIX system                        #
#     using a GNU C compiler (gcc)                    #
#-----------------------------------------------------#

CC=gcc
CFLAGS=-Wall

oldCalculator: oldCalculator.c
	$(CC) $(CFLAGS) -o oldCalculator oldCalculator.c

clean:
	rm -f oldCalculator
```

Makefile – Single file

```
#-------------------------------------------------------#
#      Makefile for UNIX system                         #
#      using a GNU C compiler (gcc)                     #
#-------------------------------------------------------#
```

```
CC=gcc
CFLAGS=-Wall
```

macros

```
oldCalculator: oldCalculator.c
	$(CC) $(CFLAGS) -o oldCalculator oldCalculator.c
```

Rule: gcc command we are used to

The second statement MUST start with a TAB!

```
clean:
	rm -f oldCalculator
```

Rule: Clean up files

Makefile

- If I have multiple rules, I can use the name of the target to execute only the rule I want
- By default, make executes only the first rule

Example

```
$make clean
```

Makefile – Multiple Modules

```
#-------------------------------------------------------#
#     Makefile for UNIX system                          #
#     using a GNU C compiler (gcc)                      #
#-------------------------------------------------------#

CC=gcc
CFLAGS=-Wall

mainCalc : mainProgram.c  calculator.o
	$(CC) $(CFLAGS) -o mainCalc mainProgram.c calculator.o

calculator.o : calculator.c calculator.h
	$(CC) $(CFLAGS) -c calculator.c

clean:
	rm -f calculator.o mainProgram
```

Makefile – Multiple Modules

```
#-------------------------------------------------------#
#     Makefile for UNIX system                          #
#     using a GNU C compiler (gcc)                      #
#-------------------------------------------------------#

CC=gcc
CFLAGS=-Wall

mainCalc : mainProgram.c  calculator.o
	$(CC) $(CFLAGS) -o mainCalc mainProgram.c calculator.o

calculator.o : calculator.c calculator.h
	$(CC) $(CFLAGS) -c calculator.c

clean:
	rm -f calculator.o mainCalc
```

We must use the –c option to compile a module instead of an executable!

Makefile

- Rules

```
target: source [source2] [source3] …
        command
        command2
        command3
           ⋮
```

UNIX compiles `target` from `source` using `command`

Default command is `$(CC) $(CFLAGS) -c source`

`make` is smart: it compiles only modules that need it

If `target` has already been compiled and `source` did not change, make will skip this rule

```
target:
        command
```

This rule instead is ALWAYS executed by the compiler, because `source` is not specified in the first line

calculator.h
calculator.c

Extern/Static Variables

- `Extern` is used to specify that a variable or function is **defined outside** the current file

  When same variable is used by different modules, `extern` is a way to declare a global variable which can be used in all modules

- `Static` is used to specify that a variable is local to the current file (for global variables)

  Remember the use for local variables (Lec7): local static means permanent