

# LAB2

## FUNCTION BASICS

# Learning Objectives

- Standard Library Predefined functions
  - Must **include** appropriate library header file
- Programmer-defined functions
  - declaration, definition, call
  - recursive functions
- Scope rules
  - local names (constants, variables, ...)
  - global names
- Header file

# Some Predefined Math Functions (1/2)

**Display 3.2** Some Predefined Functions

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0, 3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath

# Some Predefined Math Functions (2/2)

ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End pro- gram	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

Check [www.cplusplus.com/reference/](http://www.cplusplus.com/reference/) for more details

# Introduction to Functions

- Building blocks of programs
- Other terminologies equal to functions in other languages:
  - ▣ procedures, subprograms, subroutines, methods, ...
- Input-process-output model
  - ▣ e.g., `double root = sqrt(9.0);`

# Components of Function Use

- 3 steps for using functions
  - ▣ Function **declaration** (or function **prototype**)
    - Information required by compiler to properly interpret calls
  - ▣ Function **definition**
    - **Actual** **implementation/code** for what function does
  - ▣ Function **call** (or function **invocation**)
    - Transfer **control** to the function

# Function Declaration

- Also called function prototype
- Can define **multiple** functions with the same name but different parameters
- An informational declaration for compiler
- Tell compiler how to interpret calls

Syntax: <return\_type> FuncName(<**formal-parameter-list**>);

Example

double totalCost(int **numberParameter**, double **priceParameter**);

or,

double totalCost(int, double);



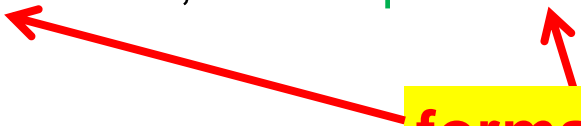
**optional**

- Placed **before** any calls, **declaration-before-use** scenario

# Function Definition

- **Implementation** of function, just like implementing function main()
- Definition: **one and only one**
- Example

```
double totalCost( int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05;
    double subTotal;
    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal * TAXRATE);
}
```



**formal parameter, mandatory**



# Function Call

- Just like calling predefined function

`bill = totalCost(number, price);`

**actual arguments,  
mandatory**

- `totalCost` returns double value, which is assigned to a variable named `bill`
- Arguments here – `number` and `price`
- Arguments can be literals, variables, expressions, or combinations of above
- In function call, arguments often called **actual arguments** because they contain the **actual data** being sent

# Local Names

- Local names
  - ▣ Declared **inside** a function
  - ▣ Scope
    - Available (visible) **from its declaration to the end of the block in which its declaration occurs**
- Hence, different functions can define their own variables/constants even with a same name

# Global Names (1/2)

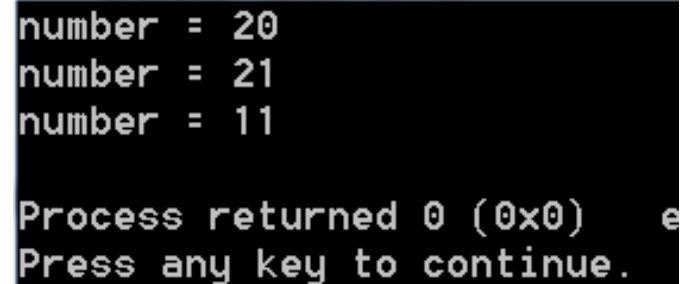
- Global names
  - ▣ Scope
    - Available (visible) from its declaration to the end of the file
- Typically, it is declared at the beginning of the file (before function definitions)

# Global Names (2/2)

- Global names are typical for constants
  - ▣ e.g., `const double TAXRATE = 0.05;`
  - ▣ All functions **in that file** can use it
- Global variables
  - ▣ You can use them, but you'd better avoid using them
  - ▣ Hard to understand and maintain, a disaster for debugging!

# Example (1/2)

```
1  #include <iostream>
2  using namespace std;
3
4  double number = 20 ; //global variable
5
6  void add()
7  {
8      number++ ;
9  }
10
11 int main()
12 {
13     cout<<"number = "<<number<<endl ;
14
15     add() ; //operation 1
16
17     cout<<"number = "<<number<<endl ;
18
19     number -= 10 ; //operation 2
20
21     cout<<"number = "<<number<<endl ;
22
23 }
```



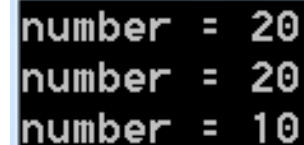
The screenshot shows the output of a C++ program. It displays the value of a global variable 'number' at three different points: initially 20, then 21 after an increment operation, and finally 11 after a decrement operation. The program ends with a message indicating it returned 0 and prompts the user to press any key to continue.

```
number = 20
number = 21
number = 11

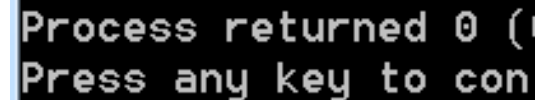
Process returned 0 (0x0)
Press any key to continue.
```

# Example (2/2)

```
1  #include <iostream>
2  using namespace std;
3
4  double number = 20 ; //global variable
5
6  void add()
7  {
8      double number = 10 ; // local variable
9      number++ ;
10 }
11
12 int main()
13 {
14     cout<<"number = "<<number<<endl ;
15
16     add() ; //operation 1
17
18     cout<<"number = "<<number<<endl ;
19
20     number -= 10 ; //operation 2
21
22     cout<<"number = "<<number<<endl ;
23
24 }
```



```
number = 20
number = 20
number = 10
```



```
Process returned 0 (
Press any key to con
```

# Header File (1/2)

- A file with extension “.h”
- **Library** and **user-defined** header files
- Separate function **declarations** and **definitions**
- Often contain functions with high correlation. E.g.,
  - ▣ sqrt, pow in <cmath>
  - ▣ Class member functions
- Included by source files whenever it is used
- Easy to maintain, improve the readability

# Header File (2/2)

- You should avoid defining a function in a header file

func.h:

```
1  #ifndef FUN_H_INCLUDED
2  #define FUN_H_INCLUDED
3  int return_zero()
4  {
5      return 0;
6  }
7  #endif // FUN_H_INCLUDED
```

main.cpp:

```
1  #include <iostream>
2  #include "fun.h"
3  using namespace std;
4  int main()
5  {
6      return 0;
7  }
```

func.cpp:

```
1  #include "fun.h"
2
3
4
```



main.o



func.o

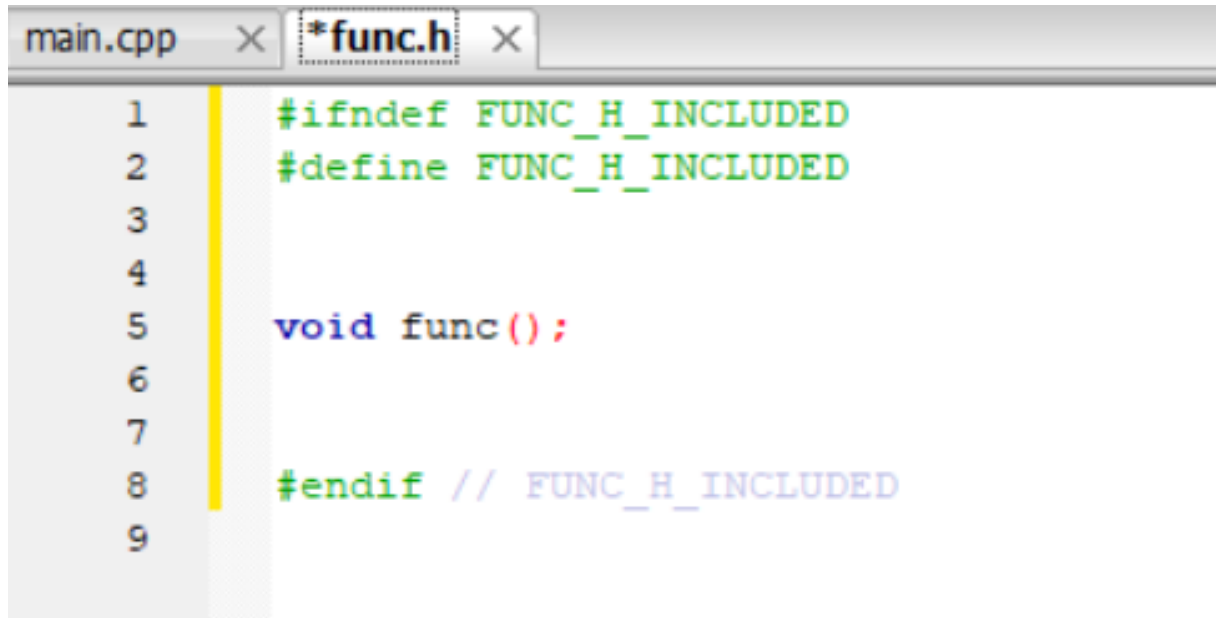


Linking error !!  
(multiple definitions)



# Header Guard

- Prevent you from including one header file multiple times
- Write your code between **#define** and **#endif**

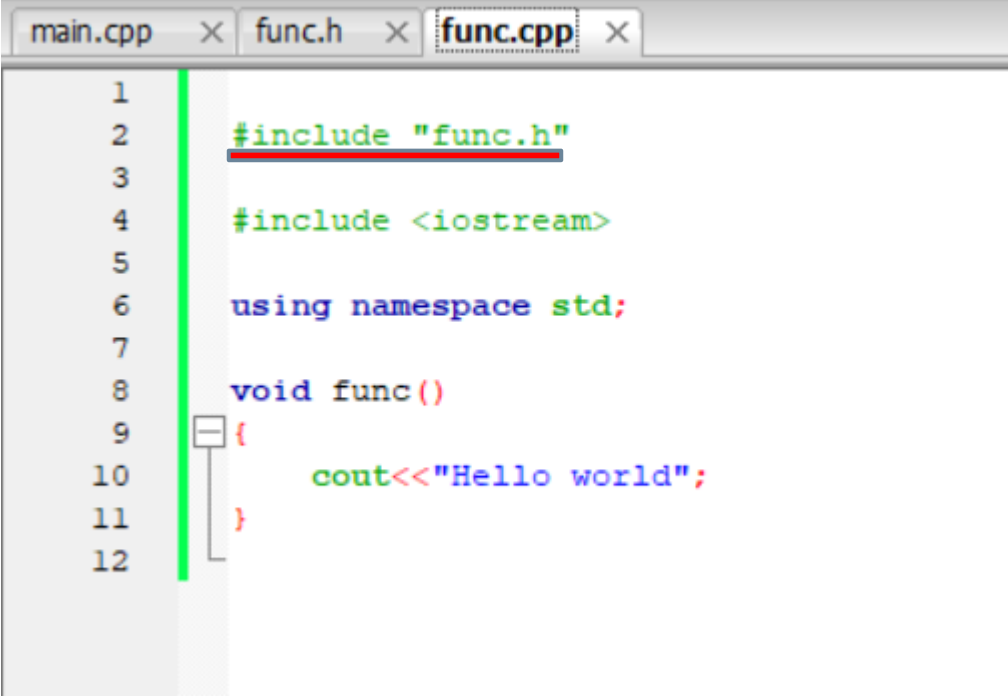


The screenshot shows a code editor with two tabs: 'main.cpp' and '\*func.h'. The '\*func.h' tab is active, displaying the following code:

```
1  #ifndef FUNC_H_INCLUDED
2  #define FUNC_H_INCLUDED
3
4
5  void func();
6
7
8  #endif // FUNC_H_INCLUDED
9
```

# Function Definition

- Write your function definitions here
- Use **quotes** instead of angled brackets for your header file



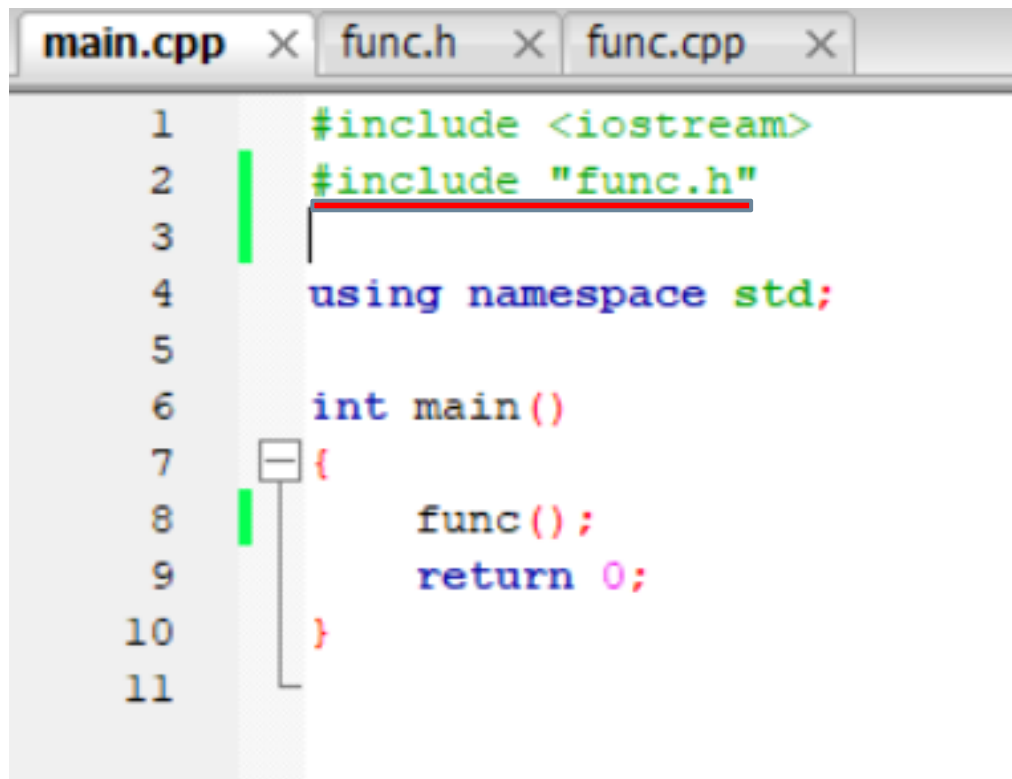
The screenshot shows a code editor with three tabs: main.cpp, func.h, and func.cpp. The func.cpp tab is active, displaying the following code:

```
1
2  #include "func.h"
3
4  #include <iostream>
5
6  using namespace std;
7
8  void func()
9  {
10     cout<<"Hello world";
11 }
12
```

A vertical green line is positioned at the start of line 9. A bracket on the left side of the code block indicates the scope of the function definition from line 9 to line 11.

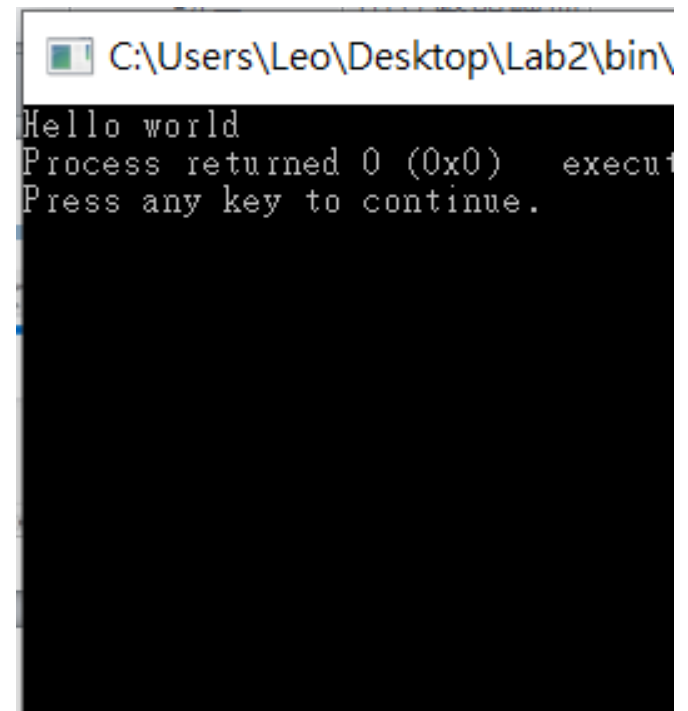
# Function Call

- Call the function in source file



The screenshot shows a code editor with three tabs: main.cpp, func.h, and func.cpp. The main.cpp file is open, showing the following code:

```
1  #include <iostream>
2  #include "func.h"
3
4  using namespace std;
5
6  int main()
7  {
8      func();
9      return 0;
10 }
11
```



The screenshot shows a command prompt window with the following output:

```
C:\Users\Leo\Desktop\Lab2\bin\
Hello world
Process returned 0 (0x0)   execut
Press any key to continue.
```

# Comile multiple files

- Method1:

- ▣ `g++ <file1.cpp> <file2.cpp> ... -o <name>`

- Method2 (prefered):

- ▣ `g++ -c file1.cpp // generate file1.o`

- ▣ `g++ -c file2.cpp // generate file2.o`

- ▣ ...

- ▣ `g++ file1.o file2.o ... -o <name>`

# Lab exercise (1/4)

## Input

Three points' coordinates (type: double)

## Output

1. **Three side lengths** of the triangle
2. **Area of the triangle** using Heron's formula
3. **Absolute value of difference** between longest side and shortest side

## Hint

You can use **pow**, **sqrt**, and **fabs** functions in Library

You must write the following three files in this Lab

**main.cpp**: input, output and function calls

**func.h file** : function declarations

**func.cpp file** : function definitions

# Lab exercise (2/4)

## □ Function 1

- ▣ **Input:** 2 points' coordinates
- ▣ **Output:** one side length

## □ Function 2

- ▣ **Input:** Three side lengths
- ▣ **Output:** area

## □ Function 3

- ▣ **Input:** Three side lengths
- ▣ **Output:** absolute difference between longest side and shortest side

# Lab exercise (3/4)

Heron's formula:

$$s = \frac{a + b + c}{2}$$

$$A = \sqrt{s(s - a)(s - b)(s - c)}$$

# Lab exercise (4/4)

```
Point 1's coordinate:
0 0
Point 2's coordinate:
3 3
Point 3's coordinate:
0 3
====RESULT====
Side Length: 4.24264 , 3 , 3
Area: 4.5
Max Difference: 1.24264

Process returned 0 (0x0)   execution time : 3.454 s
Press any key to continue.
```