

3

Functions

1

Review

- What have you learnt so far:
– Development of a C Program
– Data Types, Constants, Variables and Operators
– Simple Input/Output (printf, scanf, putchar, getchar)
– Relational and Logical Operators **(1) Sequential structure**
– Branching (if....else if....else if....else, switch, conditional operators) **(2) Branching structure**
– Looping (for, while, do...while) **(3) Looping structure**
– break and continue
- With Sequential, Branching and Looping, you will be able to build programs for simple applications. However, for more complex applications, your programs may be long and certain code may be repeated in the programnot so good!

2

Review

- Today: **... Functions ...** to group specific tasks into functions, so that code will not be repeated. It also helps to improve your program readability and efficiency.

3

This Lecture

- At the end of this lecture, you will be able to understand the following:
 - Function Definition
 - Function Prototypes
 - Function Flow
 - Parameter Passing: Call by Value
 - Storage Scope of Variables
 - Functional Decomposition

4

Functions

To solve real-world problems, we need larger programs than the ones presented in the previous lectures. When developing large and complex programs, the programming task will be much simplified if we use functions. Every C program is made up of one or more functions. A function is a self-contained unit of code to carry out a specific task. Any C functions can call any of the other functions in a program. One of the functions must be named as **main()**. Program execution begins with **main()** and terminates when the last statement of the **main()** function has been executed. In this lecture, we discuss the concepts of functions.

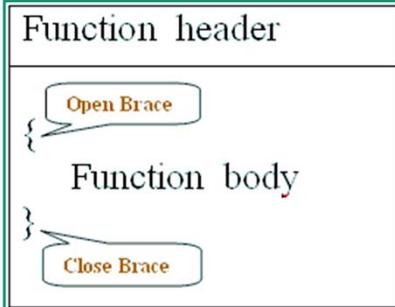
Functions

- **Function Definition**
- Function Prototypes
- Function Flow
- Parameter Passing: Call by Value
- Storage Scope of Variables
- Functional Decomposition

5

Function Definition

- A **function** is a self-contained unit of code to carry out a specific task, e.g. `printf()`, `sqrt()`.
- A **function** consists of
 - a header
 - an opening curly brace
 - a function body
 - a closing curly brace



Example:

```
float findMax(float x, float y) // header
{
    // function body
    float maxnum;

    if (x >= y)
        maxnum = x;
    else
        maxnum = y;

    return maxnum;
}
```

6

Function Definition

A function definition is a piece of code, which specifies the actions of the function, and the local data used by the function. A function definition has the following structure:

```
function_header
{
    function_body
}
```

Each function definition consists of a function header and a function body.

An example is illustrated in the `findMax()` function, which has the function header and function body.

Function Header

Return_type Function_name (Parameter_list)

- **Function_name**

- specifies the name given to the function. Try to give a meaningful name to the function.

- **Parameter_list**

- specifies a list of parameters which contain the data that are passed in by the calling function.

- **Return_type**

- specifies the **type** of the data to be returned to the calling function.

7

Function Header

The *function header* has the following format:

Return_type Function_name(Parameter_list)

where **Function_name** is the name given to the function. **Parameter_list** specifies a list of parameters which contain the data that are passed in by the calling function. **Return_type** specifies the type of the data to be returned to the calling function.

Function Header: Parameter List

- Parameters define the **data passed into** the function.
- A function can have **no** parameter, **one** parameter or **many** parameters.

type parameterName[, type parameterName]

Example: float **findMaximum**(float x, float y)

- Each parameter has:
 - parameter name**
 - data type** (e.g. int, char, etc.) of the parameter
- The function assumes that these inputs will be supplied to the function when they are being called.

8

Function Header: Parameter List

The **Parameter_list** can be **void** or a list of declarations for variables called *parameters*:

type parameterName[,type parameterName]

where [] can be repeated zero or more times. The parameters are known only inside the function body. They receive values from the calling function when the function is called. This provides a mechanism for passing data between functions.

Function Header: Return_type

- **Return Type** is the data type returned from the function, it can be **int**, **float**, **char**, **void**, or nothing.
 - **int** -- the function will return a value of the type **int**.
 - **float** -- the function will return a value of the type **float**
 - **void** -- the function will not return any value.

```
void hello_n_times(int n)
{
    int count;
    for (count = 0; count < n; count++)
        printf("hello\n");
    /* no return statement */
}
```

- **nothing** – if defined with no type, the **default type** is **int**.

```
successor(int num) /* i.e. int successor(int num) */
{
    return num + 1; /* has a return statement */
}
```

Function Header: Return_type

Return_type is the data type of the value returned by the function, it can be **int**, **float**, **char**, **void**, etc.

The **return** statement is used for functions that return a value. The syntax for the **return** statement is

return (expression);

For example,

```
int successor(int num) /* function header */
{
    /* function body begins here */
    return num + 1;
} /* function body ends here */
```

the function **successor()** will return a value of type **int**.

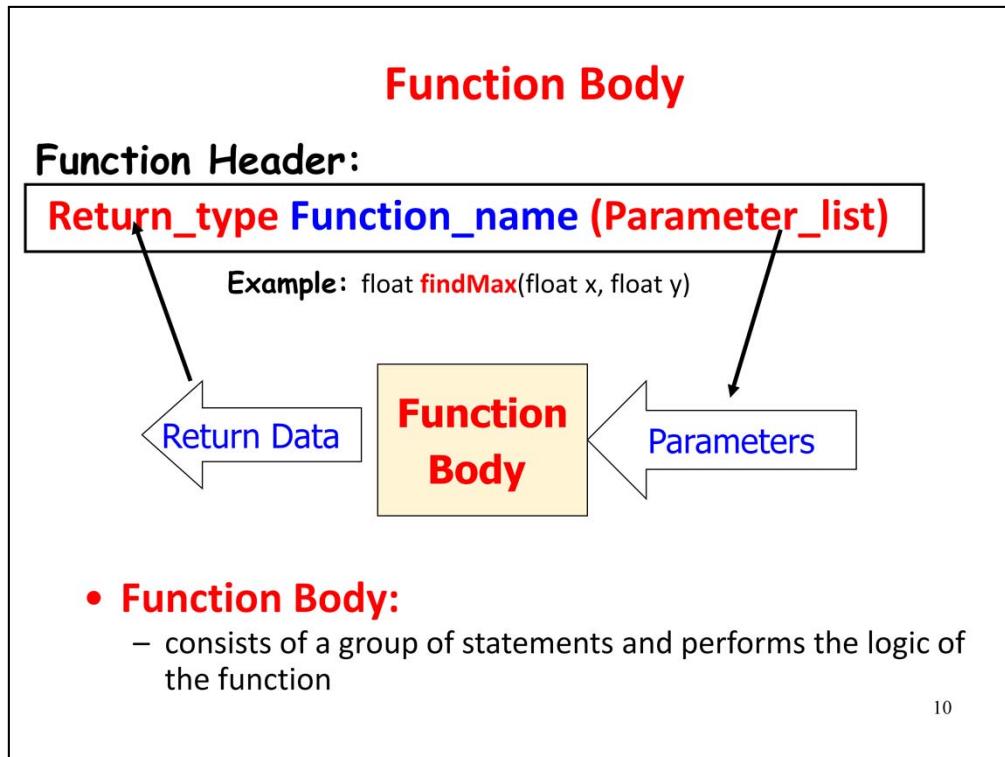
For the return type **void**, the function will not return any value. The function **hello_n_times()**

```
void hello_n_times(int n)
{
    int count;
    for (count = 0; count < n; count++)
        printf("Hello\n");
    /* no return statement here */
}
```

```
}
```

prints a string "**Hello**" to the screen the number of times specified by the parameter **n**, which is defined to be of type **int**.

If nothing is specified for **Return_type** of a function header, i.e. when a function is defined with no type, e.g. **main()**, then the default type **int** is used.



Function Body

The function body consists of a group of *statements*, it can be declaration statements, simple statements or compound statements. The statements are executed when the function is called. The variables declared inside the function body are called *local* variables and are only known within the function. Its main purpose is to perform the logic of the function.

For example, in the following **distance()** function:

```
double distance(double x, double y)
{
    return sqrt(x * x + y * y);
}
```

the function is defined with two parameters, **x** and **y**, of data type **double**. It returns a value of type **double**. The function body consists of a statement to compute the distance of the two values.

In the following **hello()** function:

```
void hello(void)
{
    printf("hello\n");
}
```

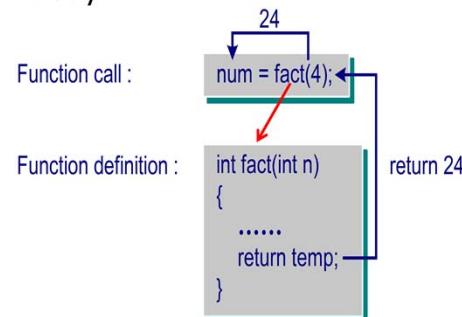
the **parameter_list** is **void**, which means that no data will be passed into this function. In addition, the function will not return any value to the calling function. The function body

just prints out the message **hello** on the screen.

Multiple Return Statements

- The return statement may appear in any place and in more than one place inside the function body.

```
int fact(int n)
{
    int temp = 1; // local variable
    if (n < 0) {
        printf("error: must be +ve\n");
        return 0;
    }
    else if (n == 0)
        return 1;
    else
        for ( ; n > 0; n--)
            temp *= n;
    return temp;
}
```



Output

Enter a positive number: 4
The factorial of 4 is 24

Multiple Return Statements

The **return** statement may appear in any place and in more than one place inside the function body. Also, the **return** statement terminates the execution of the function and passes the control back to the calling function. In the program, the function **fact()** has **return** statements in various locations in the function body. If **n** is less than 0, then an error message is displayed, and the control is returned to the calling function. If **n** equals to 0, then the function returns a value 1. If **n** is greater than 0, then the factorial of **n** is evaluated using a **for** loop, and the result is returned.

The value returned by the function can be assigned to a variable or used in other functions. In the program, the **main()** function calls the function **fact()**. The value 24 returned by the function **fact()** is then assigned to the variable **num**.

A type **void** function may also have a **return** statement to terminate the function. However, it must not have a **return** expression. An example is given below:

```
void hello_n_times(int n)
{
    int count;
    if (n <= 0)
        return; /* a return statement to return to the calling function */
    else
        for (count = 0; count < n; count++)
```

```
    printf("Hello!\n");
}
```

If the function does not have a **return** statement, then the control will be passed back to the calling function when the closing brace of the function body is encountered.

Sometimes, it is not necessary to use the value returned by a function. This is illustrated in the use of a number of C library functions such as **printf()** and **scanf()**. The **printf()** statement returns a value of type **int** that counts the number of characters printed, whereas the **scanf()** statement returns the number of items that are successfully read. However, if we do not require this information, we do not need to use the return value returned by these functions.

Function: Examples

Compute Grade:

```
char findGrade(float marks) {  
    char grade; // variable  
  
    /* function body */  
    if (marks >= 50)  
        grade = 'P';  
    else  
        grade = 'F';  
    return grade;  
}
```

12

Function: findGrade()

The function **findGrade()** expects a parameter of type **float** and returns a value of type **char**. There is one variable defined in the function. The variable is a local variable and can only be accessed within the function. The variable is created when the function is called, and destroyed when the function ends.

Function: Examples

Compute Grade:

```
char findGrade(float marks) {
    char grade; // variable
    /* function body */
    if (marks >= 50)
        grade = 'P';
    else
        grade = 'F';
    return grade;
}
```

Compute Circle Area:

```
float areaOfCircle(float radius) {
    const float pi = 3.14;
    float area;
    /* function body */
    area = pi*radius*radius;
    return area;
}
```

13

Function: areaOfCircle()

The function **areaOfCircle()** expects one parameter of type **float**. It returns a value of type **float**. A local variable **area** is also declared in the function. This variable is only accessible within the function **areaOfCircle()**. It is also created when the function is called, and will be destroyed when the function exits.

Functions

- Function Definition
- **Function Prototypes**
- Function Flow
- Parameter Passing: Call by Value
- Storage Scope of Variables
- Functional Decomposition

14

Function Prototype: Format

- **function prototype** - used to declare a function. It provides the information about
 1. the **return type** of the function
 2. the **name** of the function
 3. the **number** and **types** of the arguments
- The declaration may be the same as the function header but terminated by a **semicolon**.
- Two ways to declare parameters in the parameter list of function prototype:
 - (1) **void hello_n_times(int n); // with parameter name n**
 - Or to be declared without giving the parameter names:
 - (2) **double distance(double, double); // no parameter names**

Example:

```
float findMax(float x, float y);
```

Function Prototype

We need to declare a function before using it in the **main()** function or other functions. A function declaration is called a *function prototype*. It provides the information about the type of the function, the name of the function, and the number and types of the arguments. The declaration is the same as the function header but terminated by a semicolon:

```
void hello_n_times(int n);
```

This declaration specifies that the function **hello_n_times()** expects one argument of type **int** and does not return any value. The function prototype can also be declared without giving the argument names:

```
double distance(double, double);
```

Function prototypes enable the compiler to ensure that functions are being called properly. The compiler will check whether the number of arguments and the type of the arguments of the function call match with the parameters used in the function definition. Warning messages will be given if the number of arguments is different. Type casting will also be done if the type of the argument is mismatch.

Function Prototype: Where to declare it

- The declaration has to be done **before** the function is called:
 - (1) before the `main()` header
 - (2) inside the `main()` body or
 - (3) inside any function which uses it

16

Function Prototype: Where to declare it

A function must be declared before it is actually called. It can be declared either before the `main()` header, inside the `main()` body or inside any function which uses it.

Function Prototype: Before the main()

- The declaration has to be done **before** the function is called:
 - (1) before the main() header
 - (2) inside the main() body or
 - (3) inside any function which uses it

Before the main():

```
#include <stdio.h>
int factorial(int n); // function prototype

int main( )
{
    int x;
    x = factorial(5); // use factorial() here
}

int factorial(int n) /* function definition*/
{
    ...
}
```

17

Function Prototype: Before the main()

If the function prototype is placed before the **main()** function and at the beginning of the program, it makes the function available to all the functions in the program. The function **factorial()** is declared outside the **main()** and can be used by all the functions in the program.

Function Prototype: Inside the main()

- The declaration has to be done **before** the function is called:
 - before the main() header
 - inside the main() body or
 - inside any function which uses it

Before the main():

```
#include <stdio.h>
int factorial(int n); // function prototype

int main( )
{
  int x;
  x = factorial(5); // use factorial() here
}

int factorial(int n) /* function definition*/
{
  ...
}
```

18

Inside the main():

```
#include <stdio.h>

int main()
{
  int x;
  int fact(int); // function prototype
  x = fact(5); // use fact() here
  ...
}

int fact(int n) /* function definition*/
{
  ...
}
```

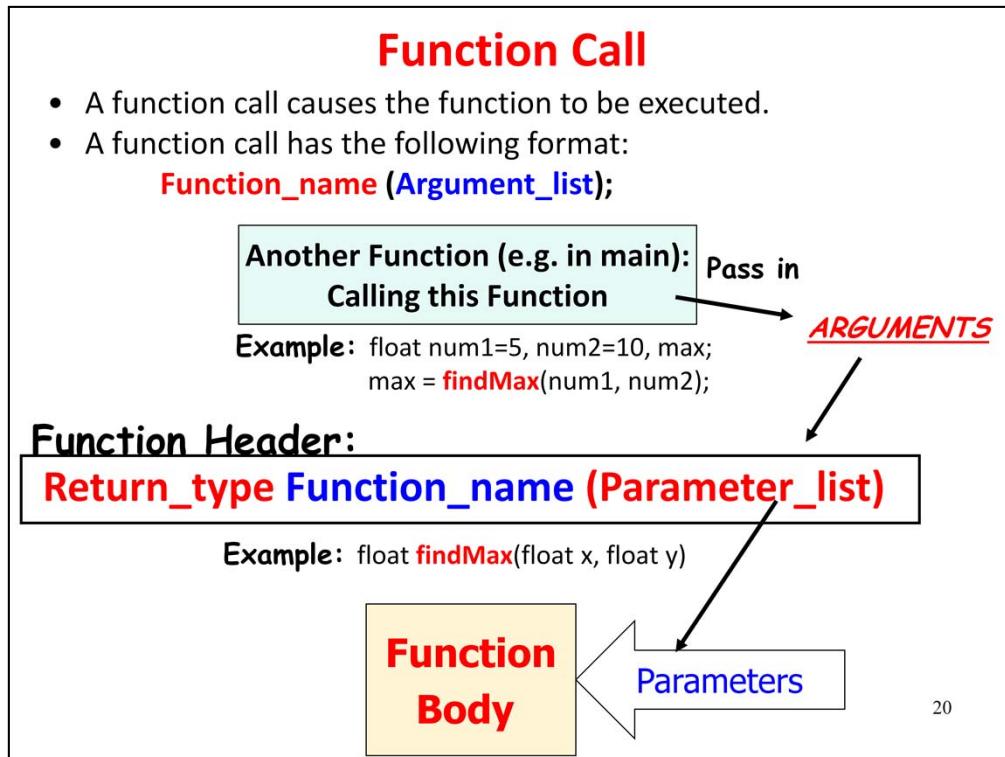
Function Prototype: Inside the main()

The function **fact()** is declared inside the **main()** function. This makes the function callable only within the **main()** function.

Functions

- Function Definition
- Function Prototypes
- Function Flow**
- Parameter Passing: Call by Value
- Storage Scope of Variables
- Functional Decomposition

19



Function Call

A function is executed when it is called. A function call has the following format:

function_name(argument_list);

The function can be called by using the function name followed by a list of *arguments*. The statement

hello_n_times(4);

calls the function **hello_n_times()**. One argument is passed to the function. The function does not return a value.

Function arguments can be constants, variables or expressions. In the function calls:

hello_n_times(nooftimes);

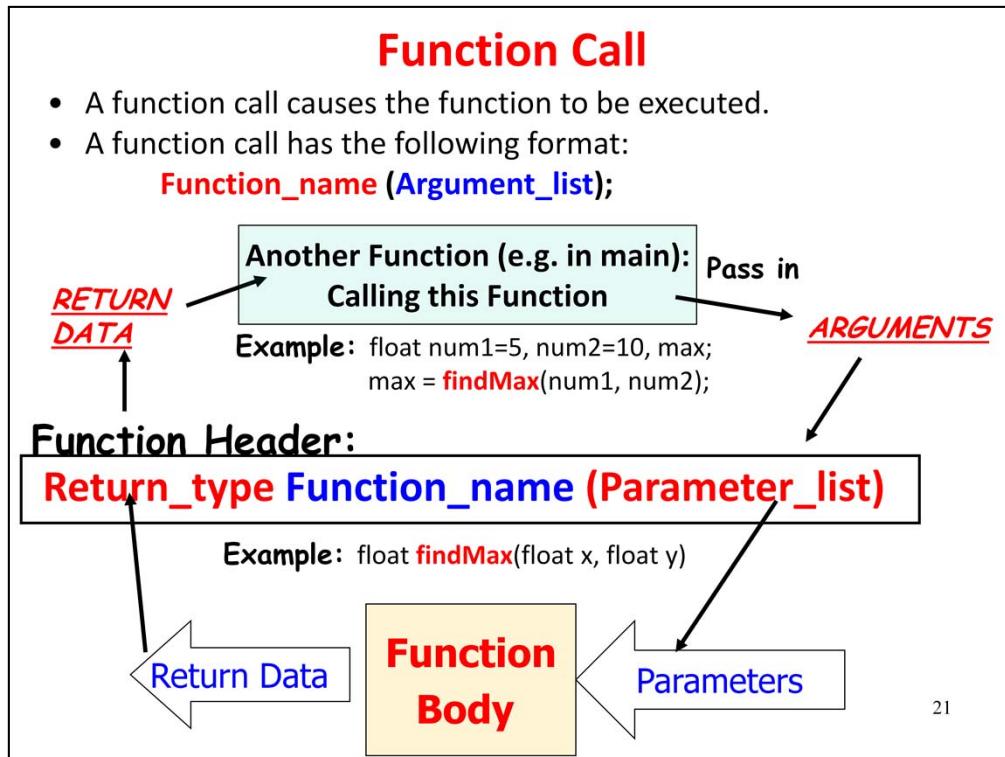
hello_n_times(nooftimes * 4);

where the argument **nooftimes** is a variable, and **nooftimes*4** is an expression.

We can also call a function that returns a value:

dist = distance(2.0, 4.5);

The function call has two arguments, separated by a comma. The function also returns a value. The returned value is assigned to the variable **dist**.



Function Call

A function is executed when it is called. A function call has the following format:

function_name(argument_list);

The function can be called by using the function name followed by a list of *arguments*. The statement

hello_n_times(4);

calls the function **hello_n_times()**. One argument is passed to the function. The function does not return a value.

Function arguments can be constants, variables or expressions. In the function calls:

hello_n_times(nooftimes);

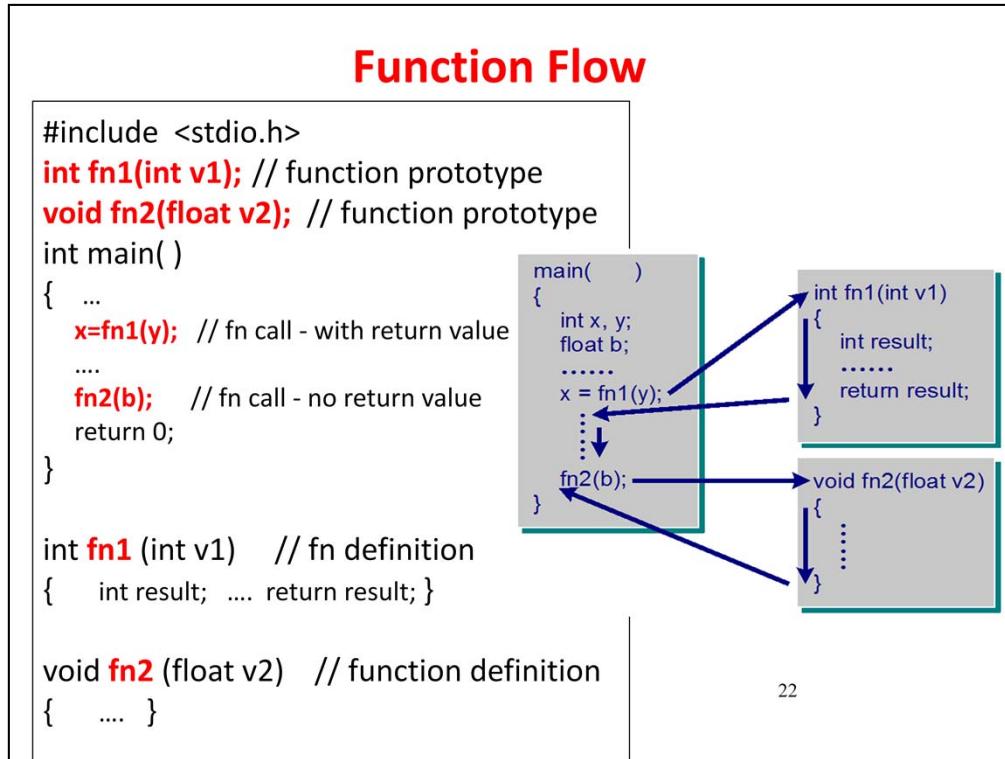
hello_n_times(nooftimes * 4);

where the argument **nooftimes** is a variable, and **nooftimes*4** is an expression.

We can also call a function that returns a value:

dist = distance(2.0, 4.5);

The function call has two arguments, separated by a comma. The function also returns a value. The returned value is assigned to the variable **dist**.



Function Flow

In the program, the **main()** function will start the execution. When the function **fn1()** is called, the program transfers the control to the **fn1()** function which then starts execution. As **fn1()** will return a value back to the calling function, the statements in the function body of **fn1()** will be executed until a **return** statement is encountered. Control is then transferred back to the **main()** function. The value of the variable **result** will be assigned to the variable **x** in **main()**. The next statement after the function call then starts execution. When the second function **fn2()** is called. The control is then transferred to **fn2()**. The function will execute until the end of the function body. Control will then be transferred back to the **main()** function.

Function Flow: Examples

Compute Grade:

```
#include <stdio.h>
char findGrade(float marks);
int main()
{
    char answer;
    answer = findGrade(68.5); →
    printf("Grade is %c", answer);
    return 0;
}
char findGrade(float marks){ ←
    char grade; // variable
    if (marks >= 50)
        grade = 'P';
    else
        grade = 'F';
    return grade;
}
```

Function Flow: **findGrade()**

In the program, the **main()** function calls the function **findGrade()**. When the statement
answer = findGrade(68.5);

is executed, it calls the function **findGrade()**. Control is then transferred to the function **findGrade()**. Information is passed between the calling function and the called function through the argument. In this case, the function receives one argument with the value of **68.5**. It is assigned to the corresponding parameter in the function definition to compute the grade. When the execution of statements in the function body encounters the **return** statement, the control is then transferred back to the **main()** function, and the statement just after the function call in **main()** will continue to execute.

The name for parameter needs not be the same as function argument. However, the number of arguments and the data type of the arguments must be the same as parameters defined in function definition. In the program, the argument **68.5** must correspond to the parameter **marks** in the function call. Notice that the function prototype is declared as:

float findGrade(float marks);

and is placed at the beginning of the program before the **main()** function.

Function Flow: Examples

Compute Grade:

```
#include <stdio.h>
char findGrade(float marks);
int main( )
{
    char answer;
    answer = findGrade(68.5);
    printf("Grade is %c", answer);
    return 0;
}
char findGrade(float marks)
{
    char grade; // variable
    if (marks >= 50)
        grade = 'P';
    else
        grade = 'F';
    return grade;
}
```

Compute Circle Area:

```
#include <stdio.h>
float areaOfCircle(float);
int main( )
{
    float answer;
    answer = areaOfCircle(2.5);
    printf("Area is %.1f", answer);
    return 0;
}
float areaOfCircle(float radius)
{
    const float pi = 3.14;
    float area;
    /* function body */
    area = pi*radius*radius;
    return area;
}
```

Function Flow: findGrade()

In the program, the **main()** function calls the function **findGrade()**. When the statement
answer = findGrade(68.5);

is executed, it calls the function **findGrade()**. Control is then transferred to the function **findGrade()**. Information is passed between the calling function and the called function through the argument. In this case, the function receives one argument with the value of **68.5**. It is assigned to the corresponding parameter in the function definition to compute the grade. When the execution of statements in the function body encounters the **return** statement, the control is then transferred back to the **main()** function, and the statement just after the function call in **main()** will continue to execute.

The name for parameter needs not be the same as function argument. However, the number of arguments and the data type of the arguments must be the same as parameters defined in function definition. In the program, the argument **68.5** must correspond to the parameter **marks** in the function call. Notice that the function prototype is declared as:

float findGrade(float marks);

and is placed at the beginning of the program before the **main()** function.

Functions

- Function Definition
- Function Prototypes
- Function Flow
- **Parameter Passing: Call by Value**
- Storage Scope of Variables
- Functional Decomposition

25

Parameter Passing: Call by Value

- **Call by Value** - Communications between a function and the calling body is done through arguments and the return value of a function.

```
#include <stdio.h>
int add1(int);
```

Output

The value of num is: 6

```
int main()
{
    int num = 5;
    num = add1(num);    // num – called argument
    printf("The value of num is: %d", num);
    return 0;
}

int add1(int value)    // value – called parameter
{
    value++;
    return value;
}
```

num 5 → 6

value 5 → 6

- **Call by Reference – using pointers (see next chapter)**

26

Parameter Passing

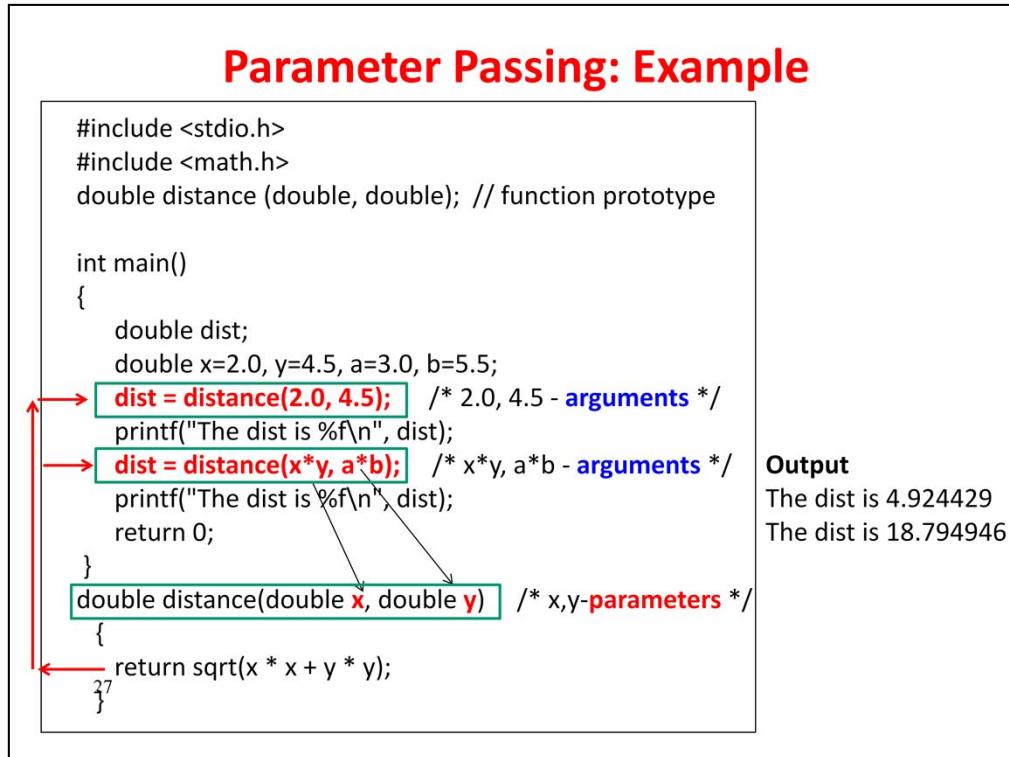
Communications between a called function and the calling function is through *arguments*. The called function then performs the task based on the received argument values. The called function can also return a value back to the calling function. For example, in the function call,

`x = sqrt(y);`

`y` is the argument that is passed into the function `sqrt()`. The types and the number of arguments must match to those of the parameters in the function definition.

Parameter passing between functions may be performed in two ways: *call by value* and *call by reference*. In call by value, the parameters must be declared in the function definition as regular variables. The arguments in function calls can be constants, variables or expressions. When the function is called, the parameters hold a *copy* of the arguments locally. Therefore, any changes to the parameters in a function are done on the copy of the arguments.

In any programs, there are two ways for a called function to return values back to the calling function. The first way is to use the `return` statement as shown in the function `add1()`. However, this can only be used when only a **single value** needs to be returned back from a function. If **two or more values** need to be passed back from a called function, we need to use another approach called call by reference via pointers.



Parameter Passing: Example

In the program, it calls the function **distance()**. When the statement

dist = distance(2.0, 4.5);

is executed, it calls the function **distance()** in **main()**. Control is then transferred to the function **distance()**. Information is passed between the calling function and the called function through arguments. In this case, the function receives two arguments with values of 2.0 and 4.5. They are assigned to the corresponding parameters in the function definition. In addition, we can also use expression as an argument in the function as shown in the following statement:

dist = distance(x*y, a*b);

When the execution of statements in the function body encounters the **return** statement, the control is then transferred back to the **main()** function, and the statement just after the function call in **main()** will continue to execute.

The names for parameters need not be the same as function arguments. However, the number of arguments and the data type of the arguments must be the same as parameters defined in function definition. In the program, the arguments **2.0** and **4.5** correspond respectively to the parameters **x** and **y** in the first function call. Similarly, the arguments **x*y** and **a*b** in the **main()** function also correspond respectively to the parameters **x** and **y** in the second function call.

Function Calling Another Function

```

#include <stdio.h>
int max3(int, int, int);      /* function prototypes */
int max2(int, int);
int main()
{
    int x, y, z;
    printf("input three integers => ");
    scanf("%d %d %d", &x, &y, &z);
    printf("Maximum of the 3 is %d\n", max3(x, y, z));
    return 0;
}
int max3(int i, int j, int k)
{
    printf("Find the max in %d, %d and %d\n", i, j, k);
    return max2(max2(i, j), max2(j, k));
}
int max2(int h, int k)
{
    printf("Find the max of %d and %d\n", h, k);
    return h > k ? h : k;
}

```

Output

input three integers => 7 4 9
 Find the max in 7, 4 and 9
 Find the max of 7 and 4
 Find the max of 4 and 9
 Find the max of 7 and 9
 Maximum of the 3 is 9

28

Function Calling Another Function

A function may be called by **main()** or another function through call by value. In the program, the function **max2()** specifies two parameters, **h** and **k**, of type **int**, and receives two function arguments from the calling function. The values of the arguments are then stored in the memory locations of the two parameters, **h** and **k**. The function then compares their values, and returns the larger value back to the calling function.

The function **max3()** specifies three parameters, **i**, **j** and **k**, and receives the function arguments from the calling function, and compares their values to determine the largest value. The **max3()** function calls the **max2()** function to compare two values at a time and returns the maximum value:

```
return max2(max2(i,j), max2(j,k));
```

Here, the function **max2()** is specified in the function **max2()** itself. The returned value from the called function **max2()** will be used again as arguments in the same function **max2()**. The maximum value is then returned back to the calling function.

Functions

- Function Definition
- Function Prototypes
- Function Flow
- Parameter Passing: Call by Value
- **Storage Scope of Variables**
- Functional Decomposition

29

Scope of Variables in a Function

- Scope of a variable
 - the sections of code that can use the variable. In other words, the variable is visible in that section.
- Variables declared in a function is ONLY visible within that function. We call it **block scope**.
- Example below: variables **radius**, **pi** and **area** are **NOT** visible **outside** this function.

```
float areaOfCircle(float radius) {      // parameter
    const float pi = 3.14;              // const variable
    float area;                      // local variable
    area = pi*radius*radius;
    return area;
}
```

30

Storage Class

The storage class of a variable is a set of properties about the variable. It is determined by where it is defined and which keyword (i.e. **auto**, **extern**, **static** or **register**) it is used with. The storage class of a variable determines the scope, linkage and storage duration of the variable. The *scope* determines the region of the code that can use the variable. In other words, the variable is visible in that section of code. The *linkage* determines how a variable can be used in a multiple-source file program. It identifies whether the variable can only be used in the current source file or it can be used in other source files with proper declarations. The *storage duration* determines how long a variable can exist in memory.

•Scope

A C variable has one of the following three scopes: file scope, block scope or function prototype scope. A variable has the *file scope* if the variable is visible until the end of the file containing the definition. A variable has the *block scope* if the variable is visible until the end of the block containing the definition. A variable has the *function prototype scope* if the variable is visible to the end of the prototype declaration.

•Linkage

A C variable has one of the following three linkages: external linkage, internal linkage and no linkage. A variable has *external linkage* if the variable can be used anywhere in a multiple-source file program. A variable has *internal linkage* if the variable can be used anywhere in a file. If a variable can be used only within a block, the variable has *no linkage*. Variables with block scope or function prototype scope have no linkage.

•Duration

The storage duration of a C variable can be static or automatic. A variable has *static storage*

duration if the variable exists throughout the program execution. A variable has *automatic* storage duration if the variable only exists within a block where the variable is defined.

Static Variables

- Using the **static** keyword

- The duration of a static variable is fixed.
- Static variables are created at the **start** of the program and are destroyed only at the **end** of program execution. That is, they exist throughout the program execution once they are created.

```
#include <stdio.h>
void function();
int main() {
    int i;
    for (i=0; i<3; i++) // calling the function three times
        function();
    return 0;
}
void function()
{
    static int static_var = 0; /* static variable */
    int auto_var = 0; /* automatic variable */
    ++static_var;
    ++auto_var;

    printf("Static variable: %d\n", static_var);
    printf("Automatic variable: %d\n", auto_var);
}
```

Note:

Automatic variable (local) – the variable disappears after each function execution.

Static variable (like global) – the variable stays until end of program execution.

Output

```
Static variable: 1
Automatic variable: 1
Static variable: 2
Automatic variable: 1
Static variable: 3
Automatic variable: 1
```

Static Variables

A **static** variable may be defined inside or outside a function's body. The duration of a static variable is fixed. Static variables are created at the start of the program and are destroyed only at the end of the program execution. We can define static variables *inside* a function's body by changing an automatic variable using the keyword **static**. If a static variable is defined and initialized, it is then initialized once when the storage is allocated. If a static variable is defined, but not initialized, it will be initialized to zero by the compiler. The initialization is done when the storage is allocated. If the static variable is defined inside a function's body, then the variable is only visible by the block containing the variable. Static variables are very useful when we need to write functions that retain values between functions. We may use global variables to achieve the same purpose. However, static variables are preferable as they are local variables to the functions, and the shortcomings of global variables can be avoided.

Automatic Variables

Automatic (or **auto**) variables are declared with the storage class keyword **auto** inside the body of a function or block. A block contains a complex statement that is enclosed by braces **{}**. We can define an automatic variable by using the storage class keyword **auto** as:

```
auto int i, j, k;
```

or we can omit the keyword as

```
int i, j, k;
```

An automatic variable has automatic storage. The lifetime of an automatic variable is only

within the function or block where it is defined. It has no meaning outside the function or block. It is also called the *local variable* of the corresponding function or block. Automatic variables are destroyed after the execution of the corresponding function or block where they are defined. It is because they are no longer needed and the memory occupied by the variables can be reused by other functions. Therefore, the value stored in an automatic variable is lost after exiting from the function or block. An automatic variable has the block scope. Only the function in which the variable is defined can access that variable by name. Further, an automatic variable has no linkage. The variable cannot be declared twice in the same block.

External Variables

An external (or **extern**) variable is defined outside a function's body. However, it can also be declared inside a function that uses it by using the **extern** keyword. External variables have static storage duration and external linkage. The storage is allocated to the variable when it is declared and it lasts until the end of program execution. If the variable is defined in another file, declaring the variable with the keyword **extern** is mandatory. External variables are initialized once when the storage is allocated. If the external variable is not explicitly initialized in the program, it will be automatically initialized to zero by the compiler. External variables allow us to break down a large program into smaller files and compile each file separately. This helps to reduce program development time, as only those files that have been changed are required to be compiled again.

Register Variables

An automatic variable can be defined using the keyword **register**. It informs the compiler that the variables will be made reference to on numerous occasions and, where possible, to use the CPU registers. Instructions using register variables execute faster than instructions that use no register variables. However, only a limited number of registers are available. The use of register variables is applicable to automatic variables and function arguments only and is restricted to certain data types (which is machine dependent) such as **int** and **char**. Pointers are not allowed to take the address of register variables, i.e. **&** operator will not work with register variables.

Local and Global Variables

- **Local variables:**

- are variables defined inside a function.
They have **block scope**.

- **Global variables:**

- are variables defined outside the functions. They have **file (or program) scope**.

32

Local and Global Variables

Local variables are variables defined inside a function. They have function or block scope. They can be accessed only within the function. They cannot be accessed by other functions. Local variables are created when the function is invoked, and destroyed after the complete execution of the function.

Global variables are variables defined outside the functions. They have file (or program) scope. Thus, global variables are visible to all the functions that are defined following its declaration.

The advantages of global variables in programs are that global variables are the simplest way of communication between functions and they are efficient. The disadvantages of programs using global variables are that they are less readable and more difficult to debug and modify as any functions in the program can change the value of the global variable. Therefore, it is a good programming practice to use local variables, and use parameter passing between functions for communication between functions. In this way, the value of each variable in the function is protected.

Local and Global Variables: Example

```

#include <stdio.h>
int g_var = 5;           // global variable – has file scope
int fn1(int, int);
float expn(float);

int main( ) {
    char reply;          // local - these two variables are only
    int num;              // known inside main() function - block scope
    ...
}

int fn1(int x, int y) {
    float fnum;          // local x,y - formal parameters are only
    int temp;              // known inside this function – block scope
    g_var += 10;           // local - these two variables are known
    ...
}

float expn(float n) {
    float temp;          // local - this variable is known in expn()
    ...
}

```

33

Local and Global Variables: Example

The program illustrates the use of global variables and local variables. The global variable **g_var** is declared outside the **main()** function. Global variables will have the file scope. The variables **fnum** and **temp** are local variables which will have block scope.

Local and Global Variables

- Should **global variables** be used in your programs?
 - **Advantages** of using global variables:
 - simplest way of communication between functions
 - efficiency
 - **Disadvantages** of using global variables:
 - less readable program
 - more difficult to debug and modify
- Strongly discouraged to use global variables – instead you should use parameter passing between functions to achieve the same effect. So that errors will be localized within each function for easy debugging.

34

Local and Global Variables

Local variables are variables defined inside a function. They have function or block scope. They can be accessed only within the function. They cannot be accessed by other functions. Local variables are created when the function is invoked, and destroyed after the complete execution of the function.

Global variables are variables defined outside the functions. They have file (or program) scope. Thus, global variables are visible to all the functions that are defined following its declaration.

The advantages of global variables in programs are that global variables are the simplest way of communication between functions and they are efficient. The disadvantages of programs using global variables are that they are less readable and more difficult to debug and modify as any functions in the program can change the value of the global variable. Therefore, it is a good programming practice to use local variables, and use parameter passing between functions for communication between functions. In this way, the value of each variable in the function is protected.

Functions

- Function Definition
- Function Prototypes
- Function Flow
- Parameter Passing: Call by Value
- Storage Scope of Variables
- **Functional Decomposition**

35

Functional Decomposition: Example

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

int main()
{
.....
} /* line 20 */

float f1(float h)
{
.....
} /* line 55 */

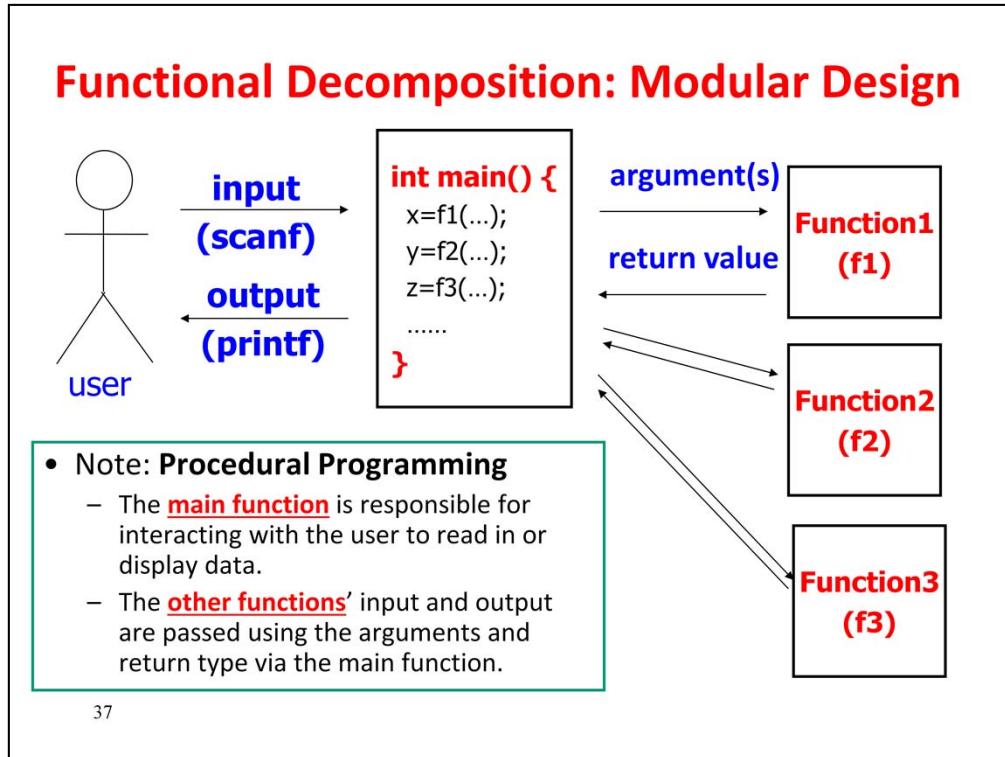
.....
void f18()
{
.....
} /* line 1560 */
```

36

Functional Decomposition

Functional decomposition basically means the top-down stepwise refinement technique that uses the divide-and-conquer strategy. It starts with the high level description of the program and decomposes the program into successively smaller components until we arrive at a set of suitably sized functions (or algorithms). Then, we design the code for the individual functions using stepwise refinement. At each level of refinement, we are only concerned with what the lower level functions will do. Functional decomposition produces smaller functions that are easier to understand. Smaller functions promote software reusability. In general, functions should be small, so that they can be developed and tested separately. They should also be independent of each other.

In the program, it is decomposed into a number of smaller functions. The **main()** function will start the execution of the program and call other functions to perform different required operations.



Functional Decomposition: Modular Design

Using the functional decomposition and top-down stepwise refinement technique, a problem is broken up into a number of smaller subproblems or functions. We then develop the algorithms for the functions. These functions can then be implemented using a programming language such as C. These functions are also called *modules*. This approach of designing programs as functional modules is called *modular design*. The functions or modules should be small and self-contained, so that they can be developed and tested separately. They should also be independent of each other. There are a number of advantages for modular design. Modular programs are easier to write and debug, since they can be developed and tested separately. Another advantage is that modular programs can be developed by different programmers as each programmer can work on a single module of the program independently. Moreover, a library of modules can be developed which can then be reused in other programs that require the same implementation. This can reduce program development time and enhances program reliability. Therefore, modular design can simplify program development significantly.

In the figure, it shows a typical structure of a program consisting of the main function and other functions for solving a problem. Usually, the functions could be quite complex as well, and they can be divided further into smaller functions.

Additional Note:

Placing Functions in Different Files

A library contains some general-purpose functions that we may use in the future. In the previous lectures, we have discussed the use of standard libraries (such as `stdio`) to help

develop structured programs. Therefore, if we can develop functions in libraries, we may use them in many other programs, thereby improving software reusability. To achieve this, we need to organize our programs. Each program should consist of one or more header files, one or more implementation files, and an application file.

Header Files

Header files are C source files. A header file will contain only declarations of functions, global variables, constants and typedefs (for data type declarations). It also contains the function prototypes of the implemented functions. The extension for header file is **.h**. An example header file **def.h** is given as follows:

```
#include <math.h>
#define CONST1 80
#define CONST2 100
int function1(int, int);
void function2(int, float);
```

Implementation Files

Implementation files are also C source files. An implementation file contains the implementation of the functions declared in the header file. An example implementation file **support.c** that implements the two supporting functions defined in **def.h** is given as follows:

```
#include <stdio.h>
#include "def.h" /* located in the current directory */
int function1(int f, int g)
{
    ....
}
void function2(int p, float q)
{
    ....
}
```

Application File

Application file contains the **main()** function and any other related functions that are not implemented in the implementation files. An example application file **mainF.c** is given as follows:

```
#include <stdio.h>
#include "def.h" /* located in the current directory */
int main()
{
    ....
    count = function1(i, j);
    function2(h, k);
    ....
```

```
}
```

In **mainF.c**, the **main()** body calls **function1()** and **function2()**.

Program Compilation Process

When we have a program comprising several files, we need to compile the multiple-source file program. Most Integrated Development Environment can support the creation of project for you to place all the files. The compilation process can also be supported by the IDE. For example, the compiler will compile **mainF.c** and produces **mainF.o**. Then, it compiles **support.c** and produces **support.o**. The linker will produce the executable file **mainF** after linking the two object files and the object files from the library functions. After successful compilation, if any changes are made to **mainF.c** but not **support.c**, then we only need to recompile the **mainF.c** file. Similarly, If any changes are made to **support.c** but not **mainF.c**, then we only need to recompile the **support.c** file. The advantage of placing parts of a program into different files is that the functions in different files can be used by more than one program, and only the files, which have been changed, need to be re-compiled.

Programming Problem

38

Problem: Writing a C Program with a Function

```
/* Purpose: Write a function to find the maximum of two numbers.
The main() function calls the findMax() function using call by value
*/
#include <stdio.h>
float findMax(float x, float y);
int main() {
    float x,y, max;
    printf("Enter 2 numbers: ");
    scanf("%f %f", &x, &y);
    max = findMax(x, y);
    printf("The max is %.2f.\n", max);
    return 0;
}
float findMax(float x, float y)
{
    float maxnum;
    /* write your code here */
}
```

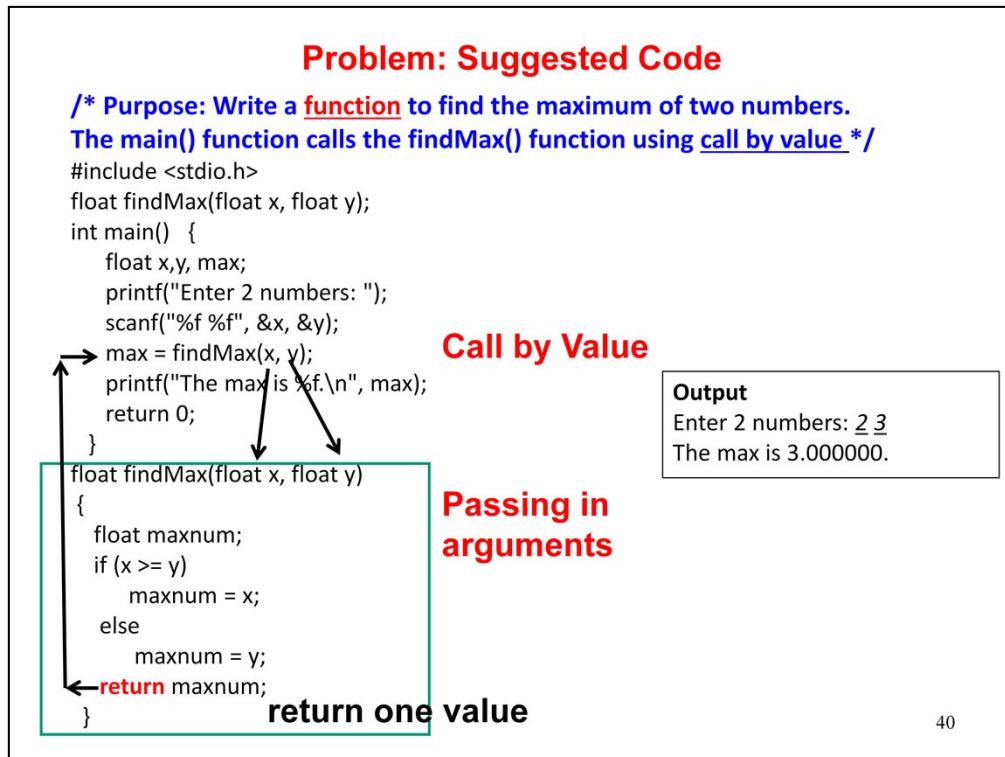
Output

Enter 2 numbers: 2 3
The max is 3.000000.

39

Programming Problem

The purpose of the program is to compute the maximum of two floating point numbers. Write a function called **findMax()** to achieve this purpose. The function will take in two arguments in floating point type, compute the maximum of the two values and return the maximum number to the calling function.



Programming Problem: Suggested Code

The suggested code for the function **findMax()** is given. It will take in two arguments **x** and **y** from the calling function, compute the maximum number and returns the maximum number to the calling function. The calling function **main()** will then stores the returned result in the variable **max** which is then printed on the screen.