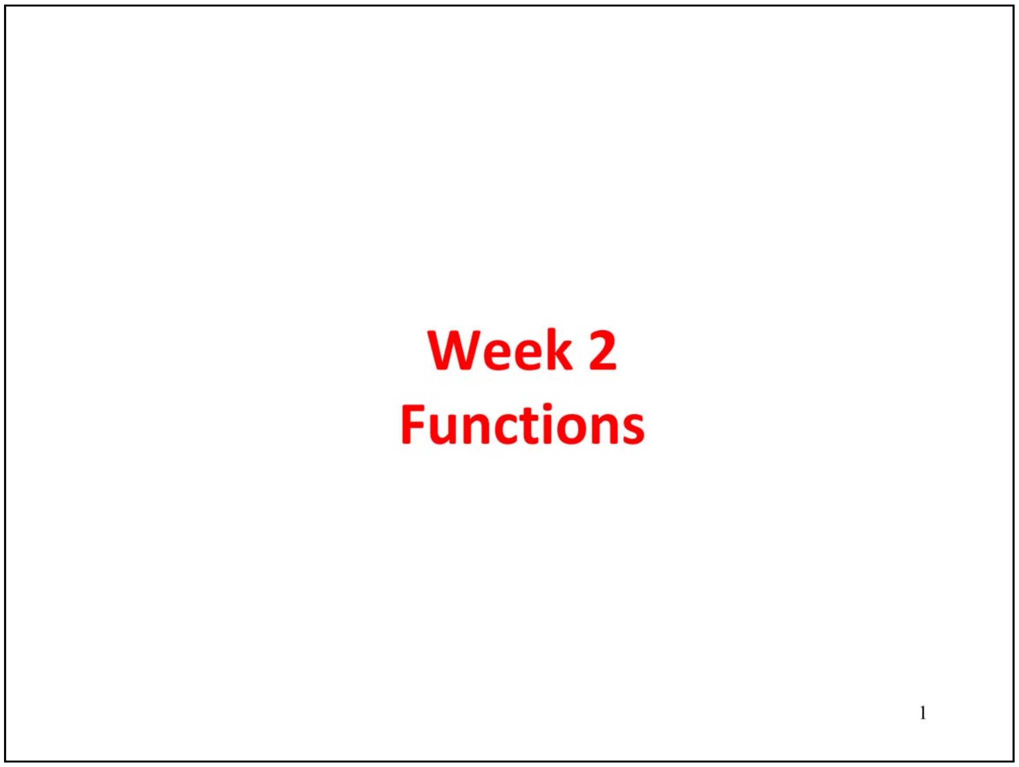


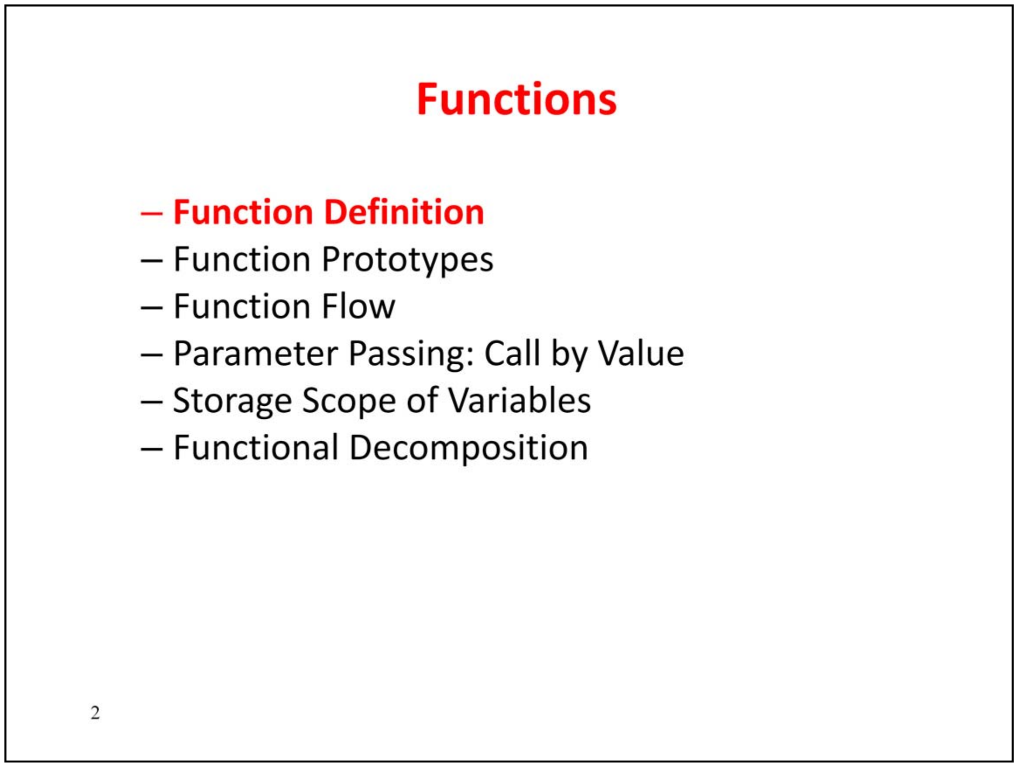
Week 2
Functions

1

Why Learning Functions

1. 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 program.
2. Functions aim to group specific tasks, so that code will not be repeated. It also helps to improve your program readability and efficiency.
3. In this lecture, we discuss the concepts on functions.

To solve real-world problems, we need larger programs than the ones presented in the previous lecture. 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.



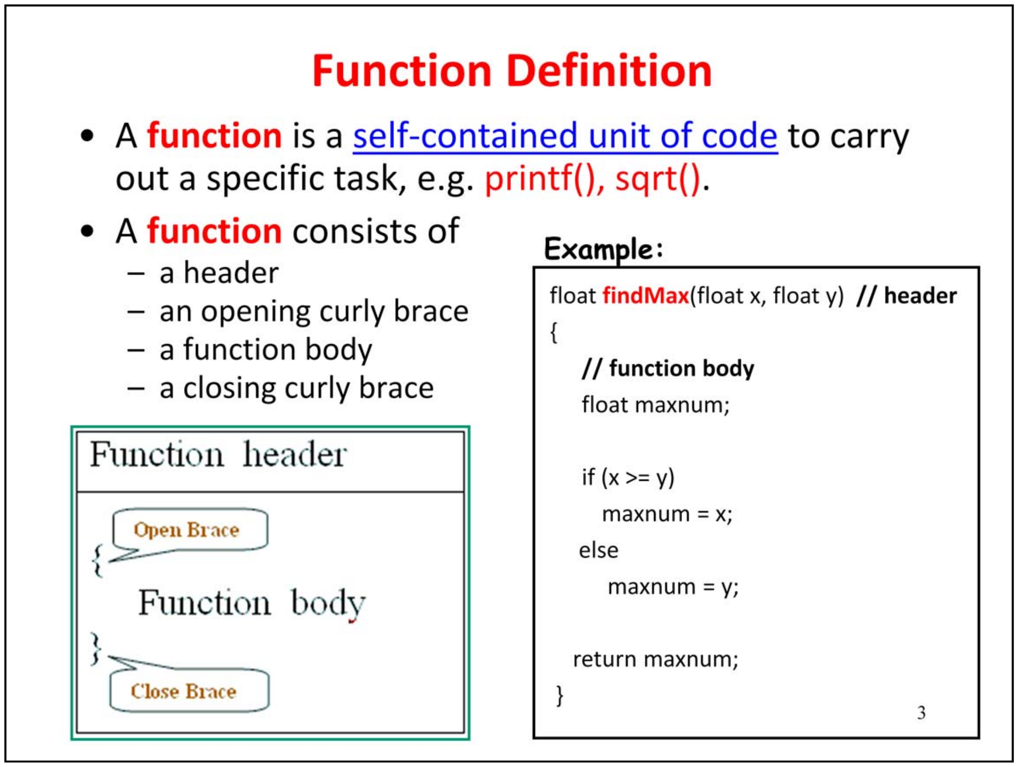
Functions

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

2

Functions

1. Here, we start by discussing function definition.



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

Function header
Open Brace
{
Function body
}
Close Brace

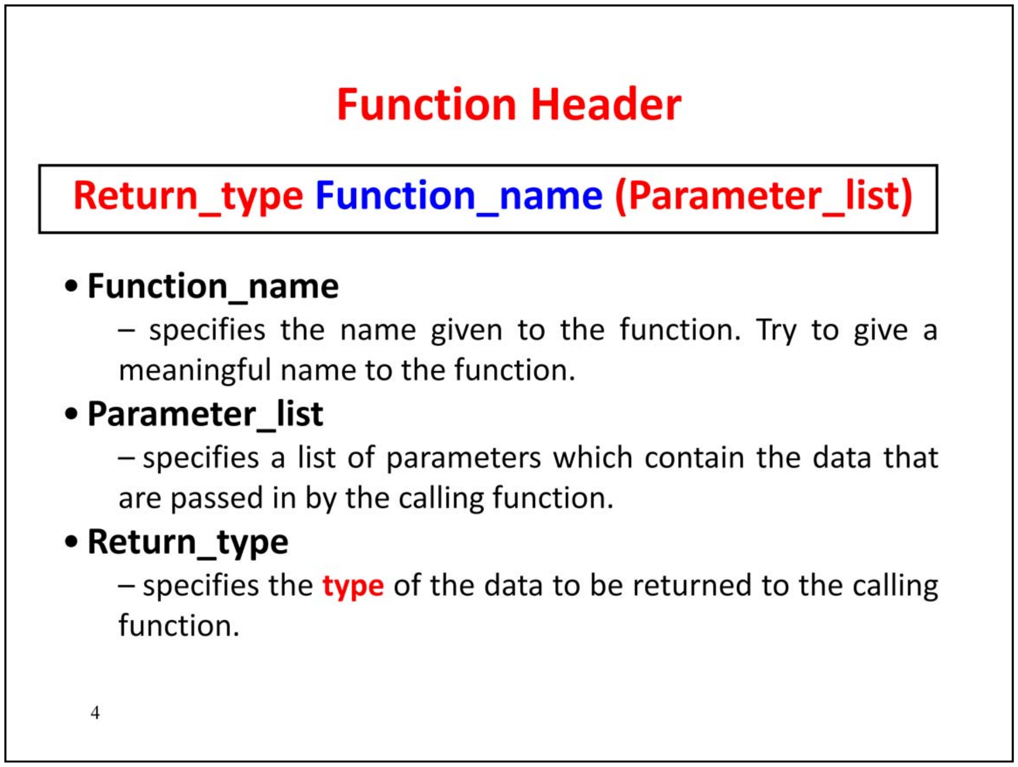
Example:

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

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

   return maxnum;
 }
```

3

Function Definition

1. A function is a self-contained unit of code to carry out a specific task, e.g. **printf()**, **sqrt()**.
2. Each function definition consists of a function header and a function body.
3. The function body contains the code, which specifies the actions of the function, and the local data used by the function.
4. 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.

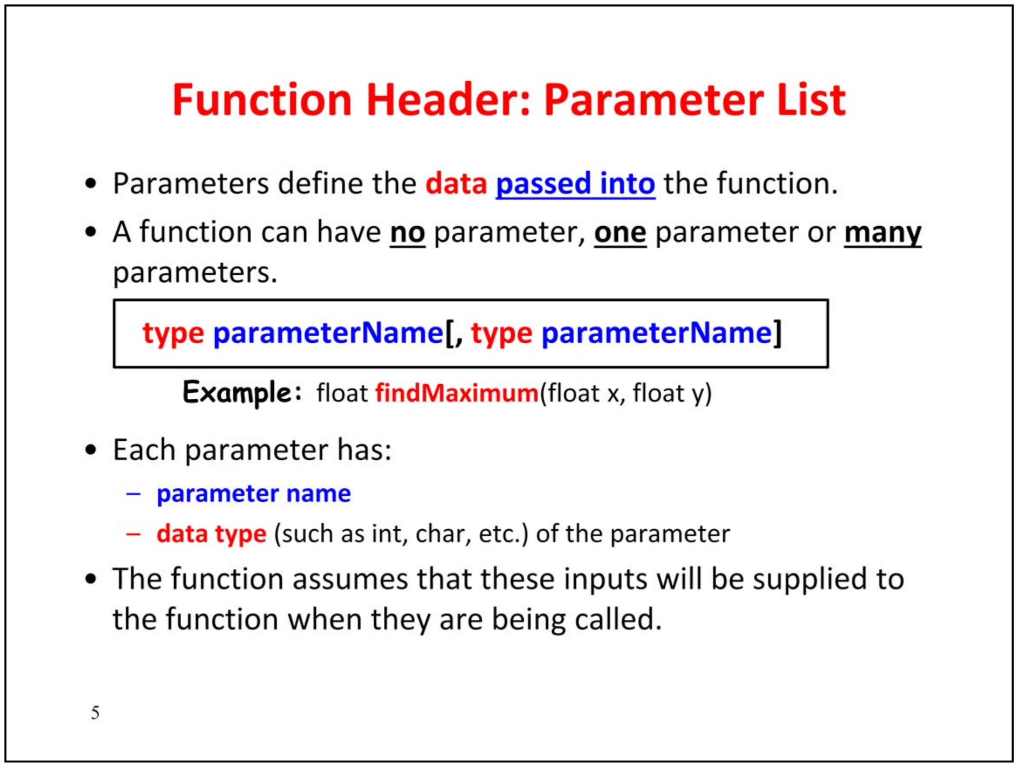
4

<u>Function Header</u>

1. The function header has the following format:

   Return_type Function_name(Parameter_list)

2. Function_name specifies the name given to the function.
3. Parameter_list specifies a list of parameters which contain the data that are passed in by the calling function.
4. 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** (such as int, char, etc.) of the parameter
- The function assumes that these inputs will be supplied to the function when they are being called.

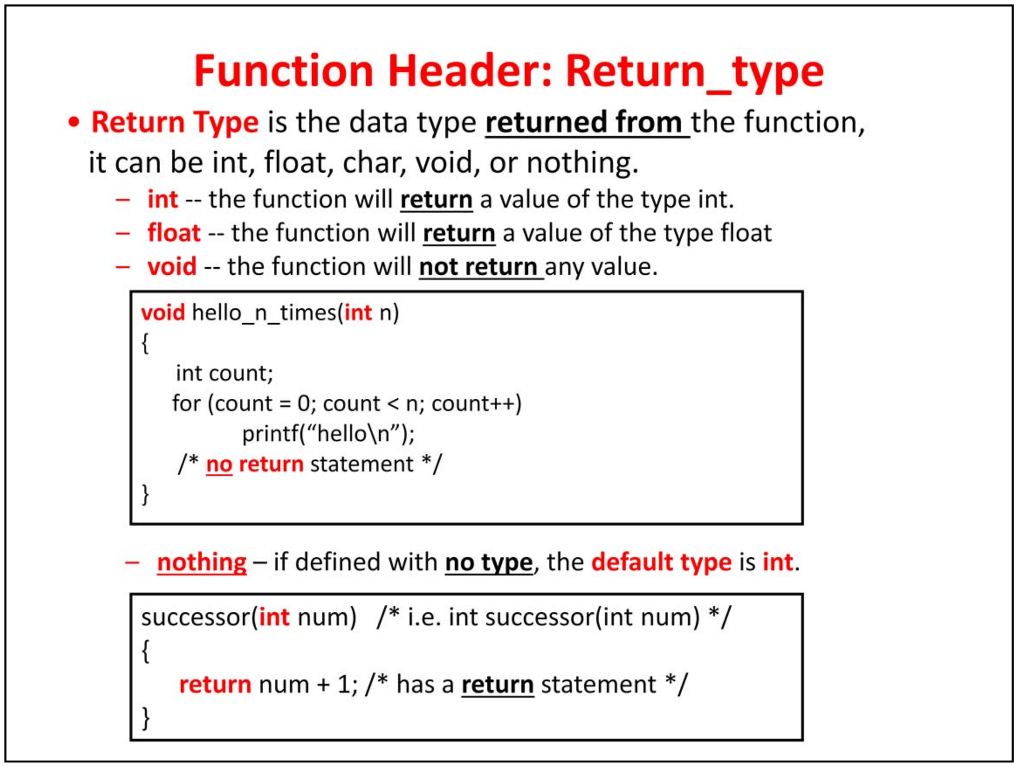
5

Function Header: Parameter List

1. Parameters define the data passed into the function.
2. A function can have no parameter, one parameter or many parameters.
3. The **Parameter_list** can be **void** or a list of declarations for variables called *parameters*:

   type parameterName[, type parameterName]

4. Each parameter has a parameter name and data type of the parameter (such as **int**, **char**, etc.) .
5. The function assumes that these inputs will be supplied to the function when they are being called.



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

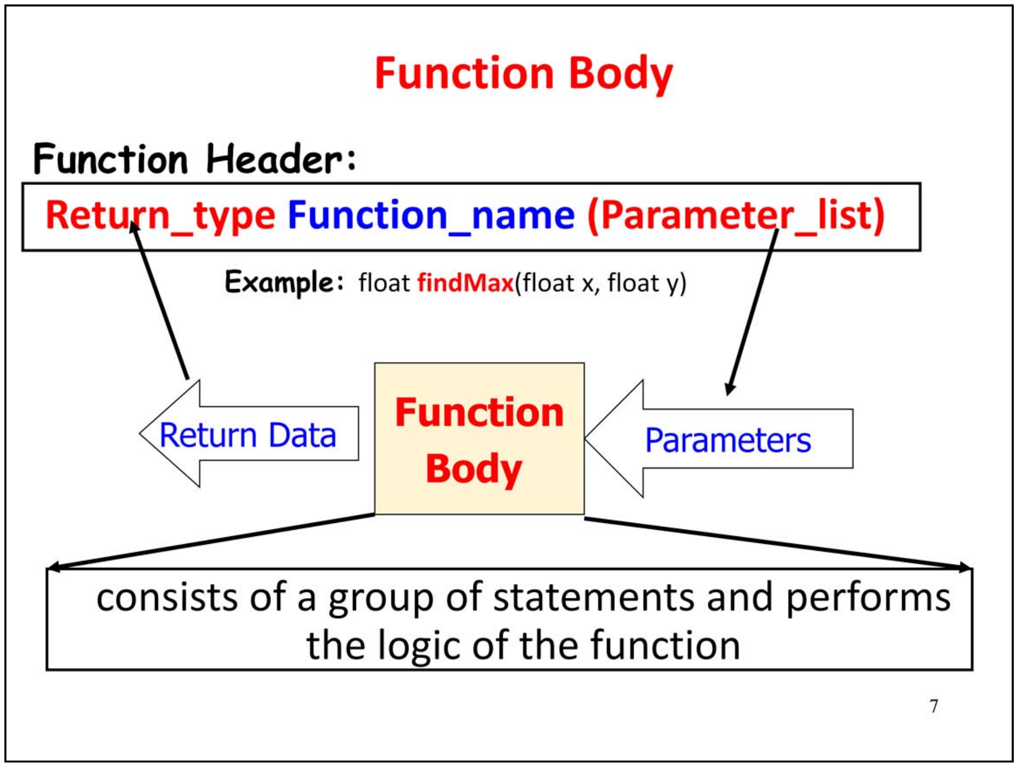
1. **Return_type** is the data type of the value returned by the function, it can be **int**, **float**, **char**, **void**, etc.
2. The **return** statement is used for functions that return a value.
3. The syntax for the **return** statement is **return (expression);**
4. In the example function **hello_n_times()**, when the return type is **void**, the function will not return any value.
5. If nothing is specified for **Return_type** of a function header, i.e. when a function is defined with no type, for example, in the **successor**() function, then the default type **int** is used for that function. It means the function will return an integer value.

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

Function Header:

Return_type Function_name (Parameter_list)

Example: float findMax(float x, float y)

consists of a group of statements and performs the logic of the function

7

Function Body

1. The function body consists of a group of statements.
2. The statements are executed when the function is called.
3. The variables declared inside the function body are called *local* variables and are only known within the function.
4. The main purpose of function body is to perform the logic of the function.

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

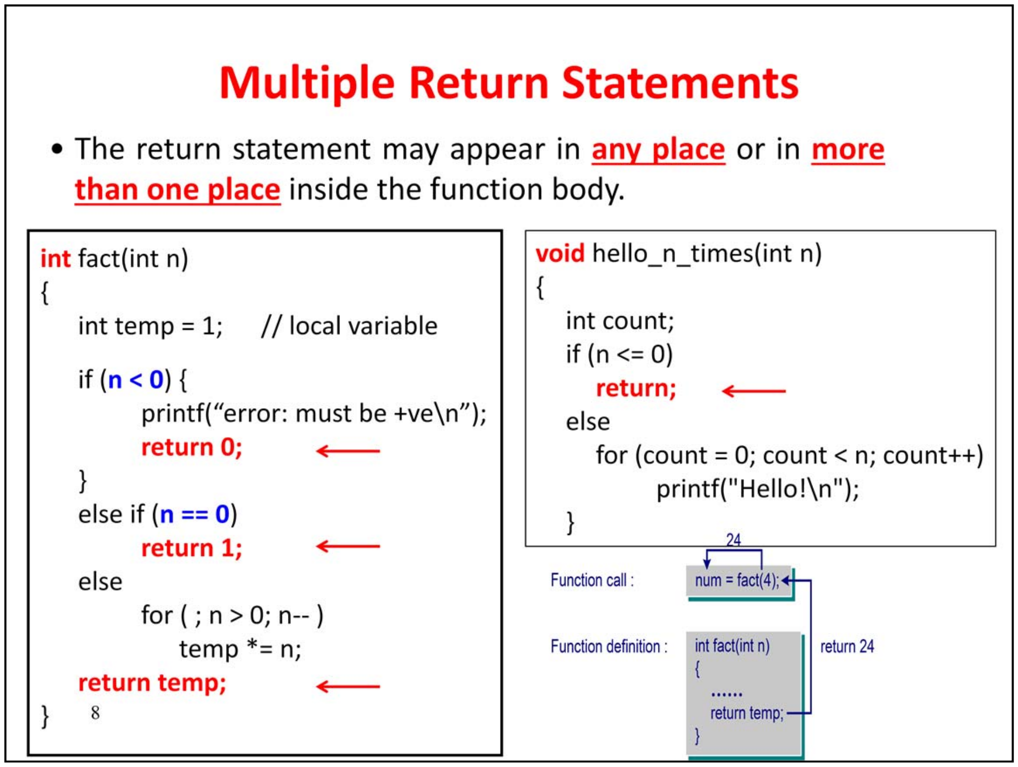
```
double distance(double x, double y)
{
    return sqrt(x * x + y * y);
}
```

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.

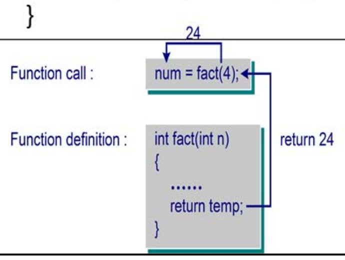


Multiple Return Statements

- The return statement may appear in <u>any place</u> or in <u>more than one place</u> 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;               ⟵
}    8
```

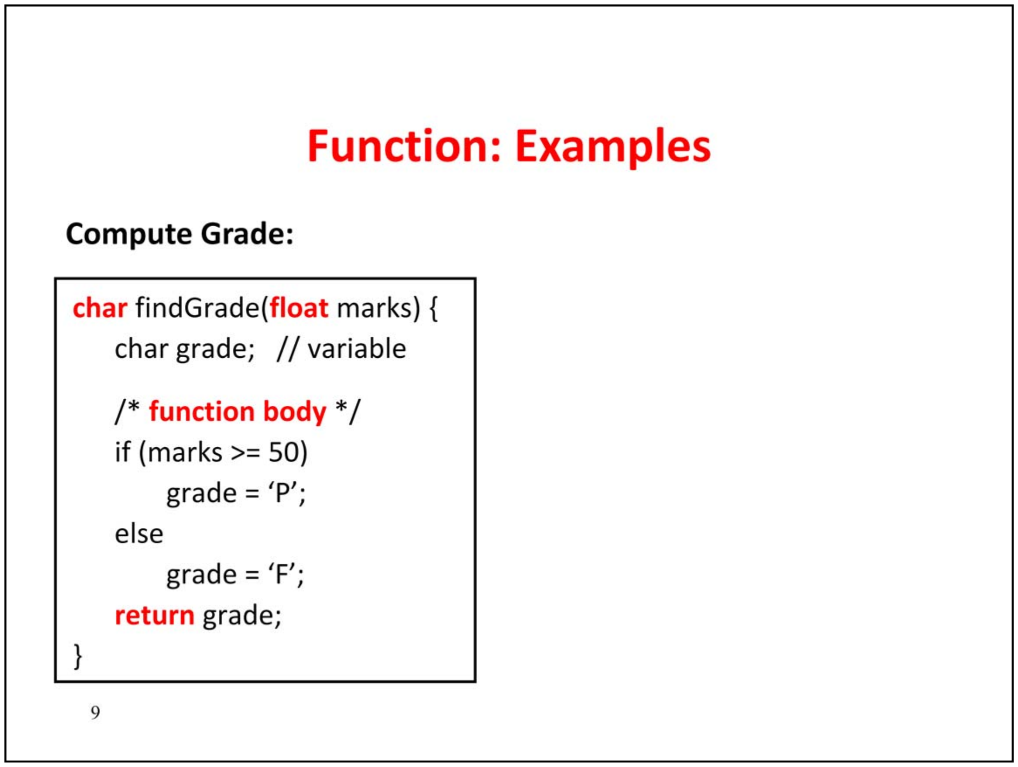
```
void hello_n_times(int n)
{
    int count;
    if (n <= 0)
        return;        ⟵
    else
        for (count = 0; count < n; count++)
              printf("Hello!\n");
    }
```



<u>Multiple Return Statements</u>

1. The **return** statement terminates the execution of the function and passes the control to the calling function.
2. The **return** statement may appear in any place or in more than one place inside the function body.
3. 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.
4. A type **void** function may also have a **return** statement to terminate the function. However, it must not have a **return** expression. 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

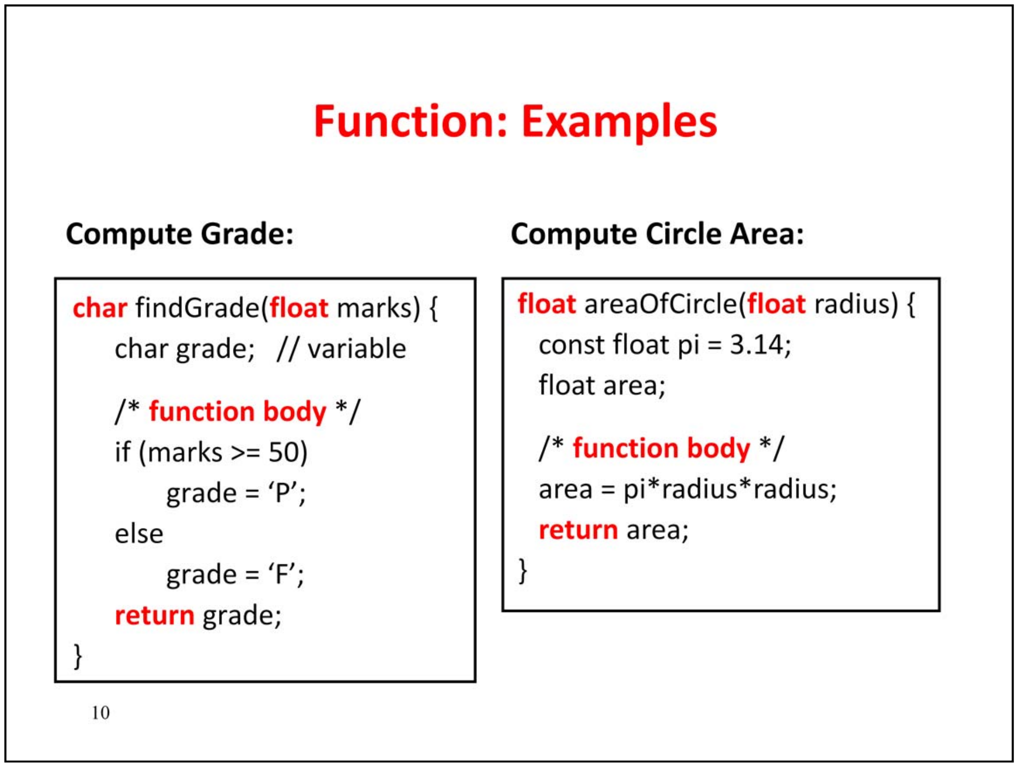
Compute Grade:

```
char findGrade(float marks) {
    char grade;   // variable

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

9

Function: findGrade()

1. The function **findGrade()** expects a parameter of type **float** and returns a value of type **char**.
2. The parameter **marks** is only accessible within the function **findGrade()**.
3. There is one variable **grade** defined in the function **findGrade()**. The variable is a local variable and can only be accessed within the function.
4. 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;
}
```

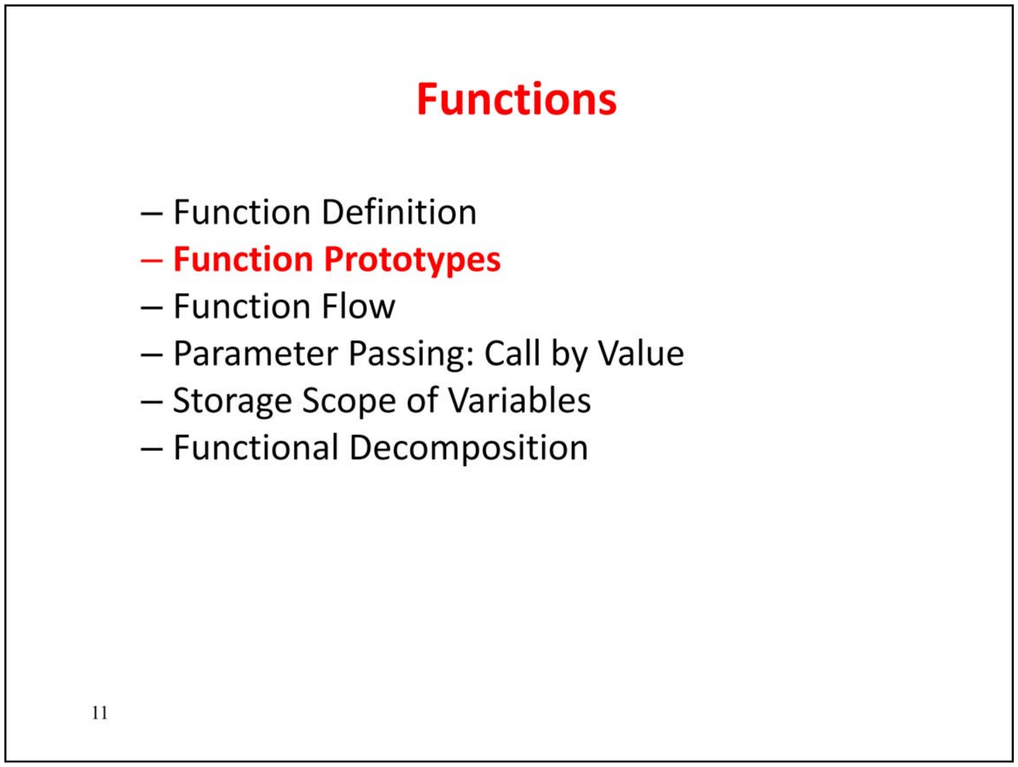
Compute Circle Area:

```
float areaOfCircle(float radius) {
  const float pi = 3.14;
  float area;

  /* function body */
  area = pi*radius*radius;
  return area;
}
```

10

<u>Function: areaOfCircle()</u>

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



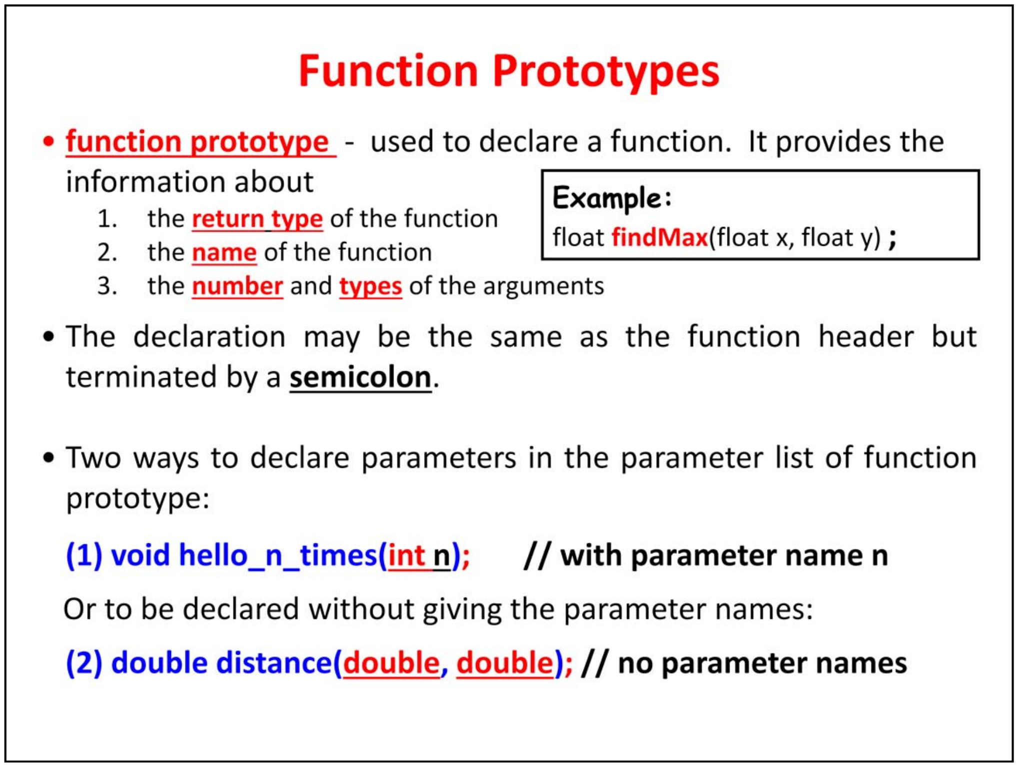
Functions

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

11

Functions

1. Here, we discuss function prototypes.



Function Prototypes

- **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

Example:
float **findMax**(float x, float y) **;**

- 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:

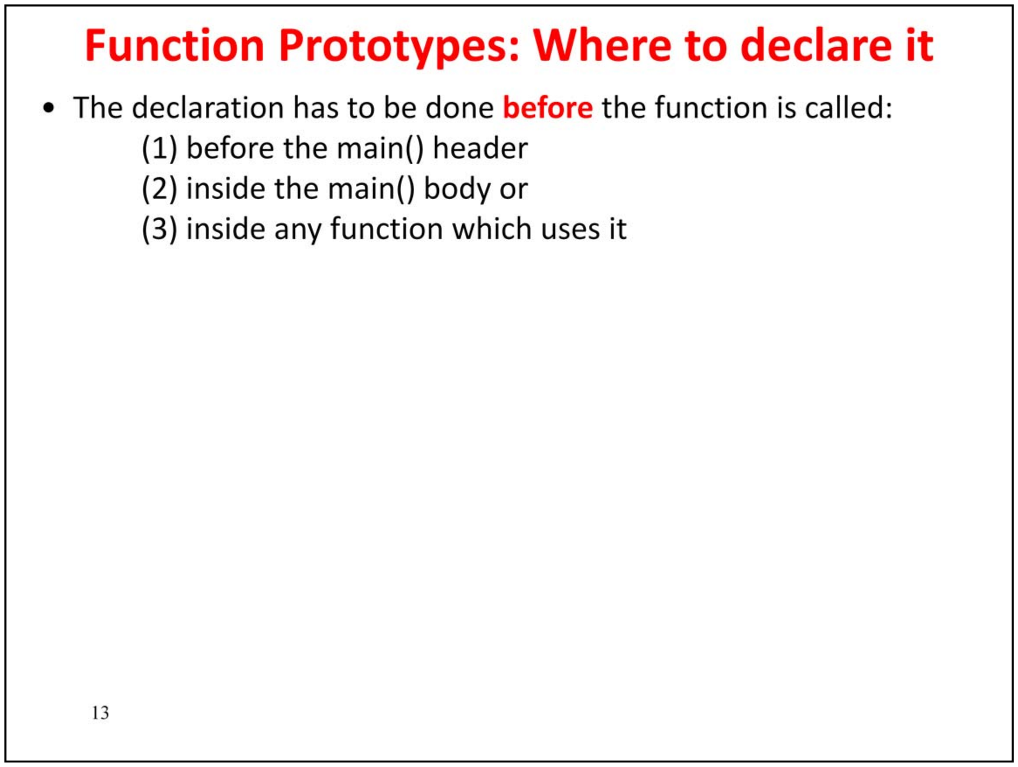
(1) void hello_n_times(int n);   // with parameter name n

Or to be declared without giving the parameter names:

(2) double distance(double, double); // no parameter names

Function Prototypes

1. 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.
2. The declaration is the same as the function header but terminated by a semicolon. For example, **void hello_n_times(int n);**
3. The function prototype can also be declared without giving the parameter names. For example, **double distance(double, double);**
4. 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.



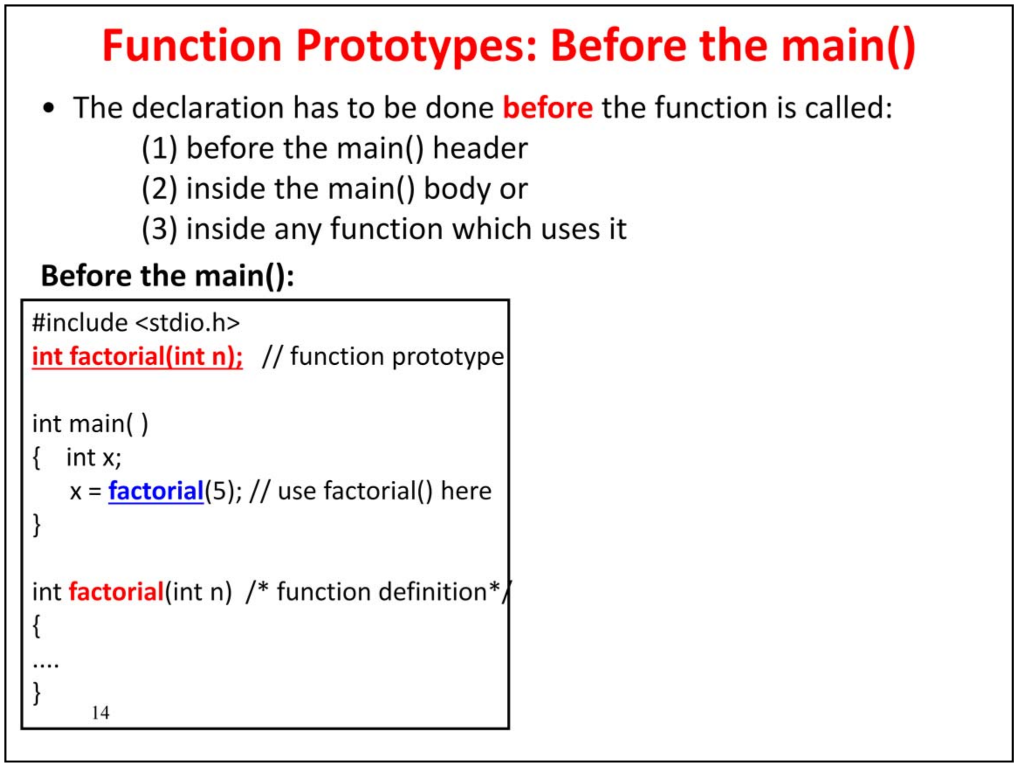
Function Prototypes: Where to declare it

- 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

13

Function Prototypes: Where to declare it

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



Function Prototypes: 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

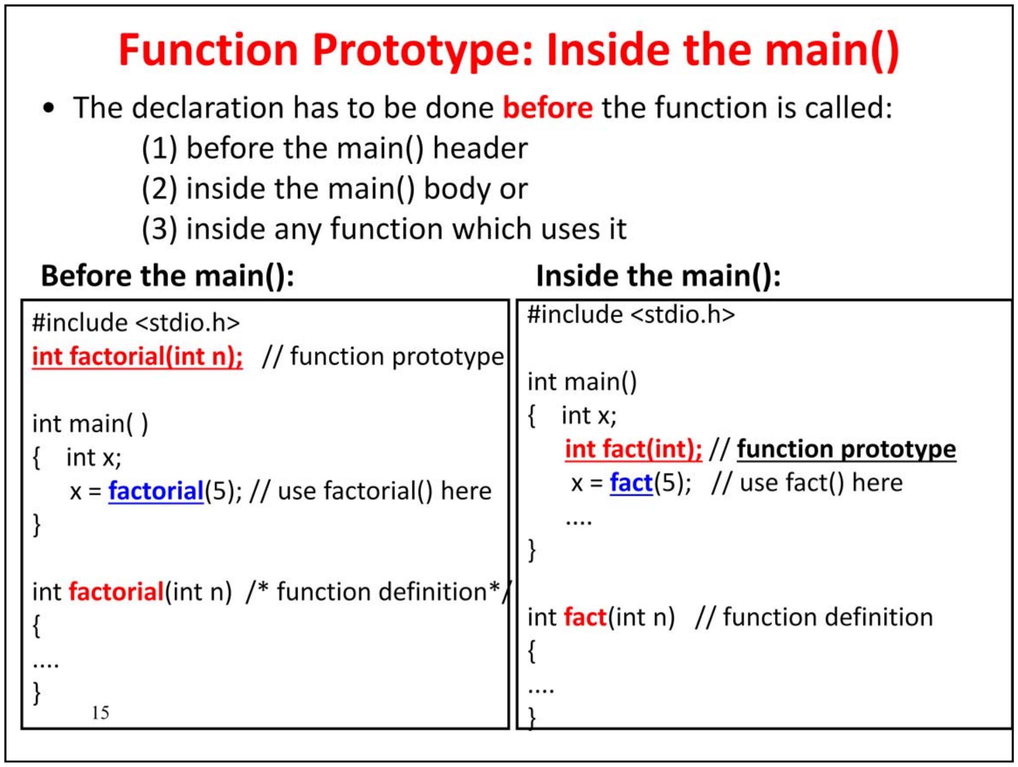
Before the main():

```
#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*/
{
....
}
```

14

Function Prototype: Before the main()

1. 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.
2. In this example program, the function **factorial()** is declared outside the **main()**. Therefore, it 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:
  - (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*/
{
....
}
```

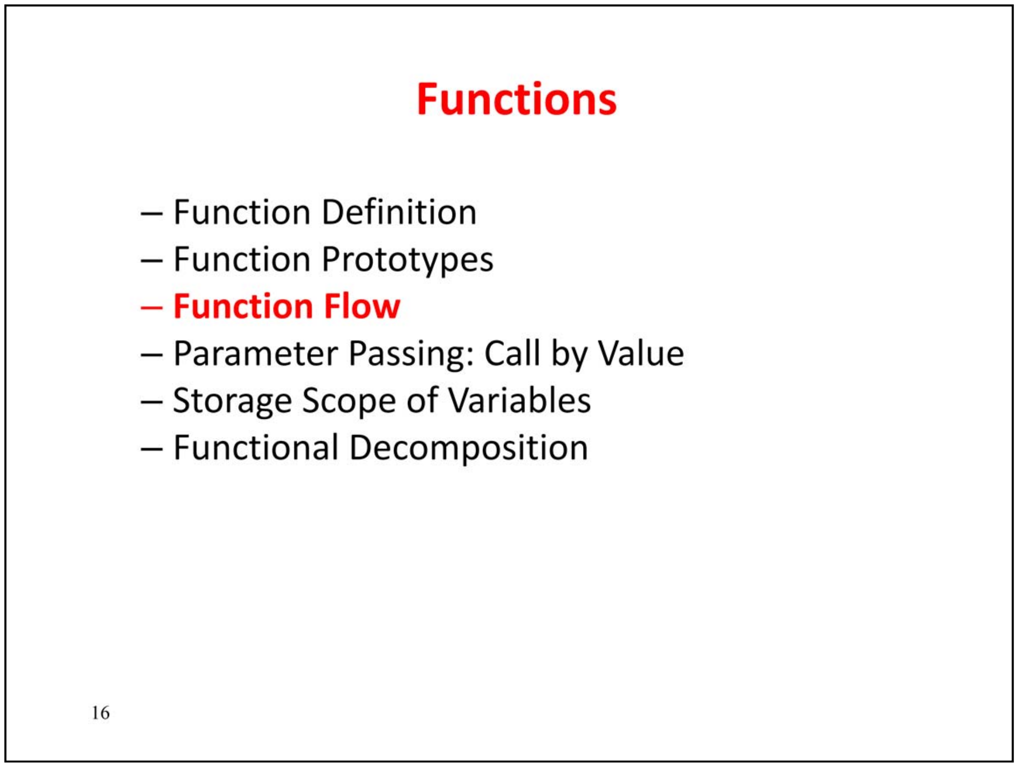
Inside the main():

```
#include <stdio.h>

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

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

15

<u>Function Prototype: Inside the main()</u>

1. In the second example program, the function **fact()** is declared inside the **main()** function.
2. This makes the function callable only within the **main()** function.



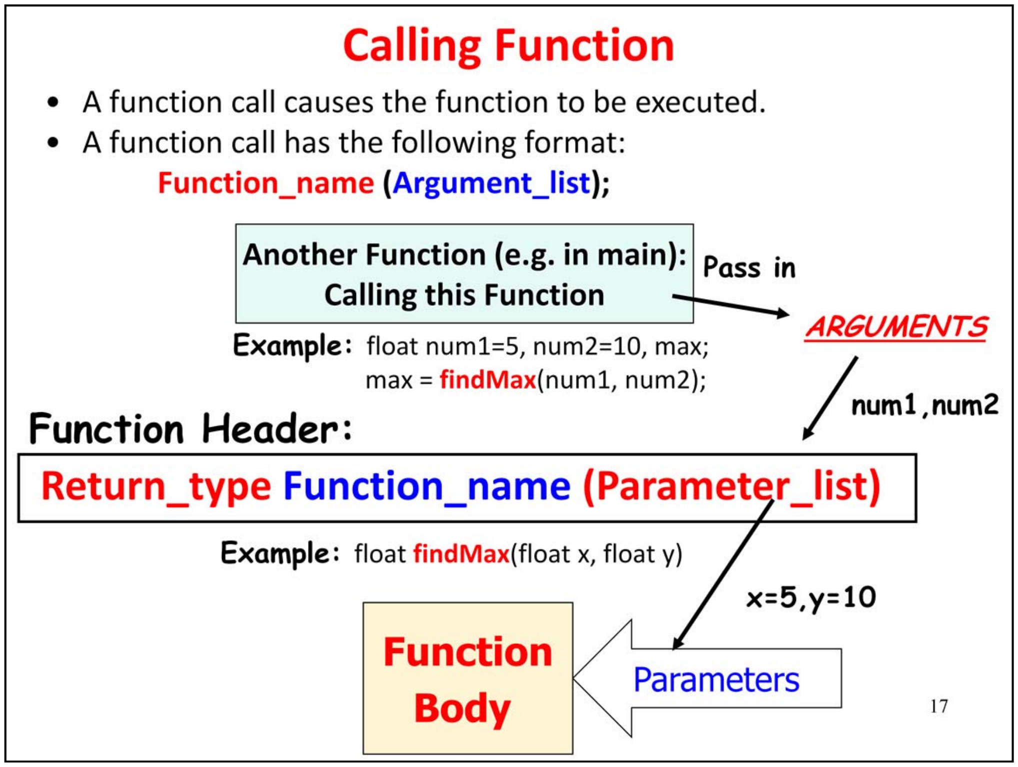
Functions

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

16

Functions

1. Here, we discuss function flow.



Calling Function

- A function call causes the function to be executed.
- A function call has the following format:

  Function_name (Argument_list);

Another Function (e.g. in main): Calling this Function
Pass in
ARGUMENTS
Example: float num1=5, num2=10, max;
max = findMax(num1, num2);
num1, num2
Function Header:
Return_type Function_name (Parameter_list)
Example: float findMax(float x, float y)
x=5, y=10
Function Body
Parameters
17

Calling Function

1. A function is executed when it is called.
2. A function call has the following format: **function_name(argument_list);**
3. A function can be called by using the function name followed by a list of *arguments*. For example, **num1** and **num2** in the **findMax()** function.
4. Function arguments can be constants, variables or expressions.
5. As **num1**=5, **num2**=10, the values 5 and 10 will be passed to the parameters **x** and **y** respectively.

The statement

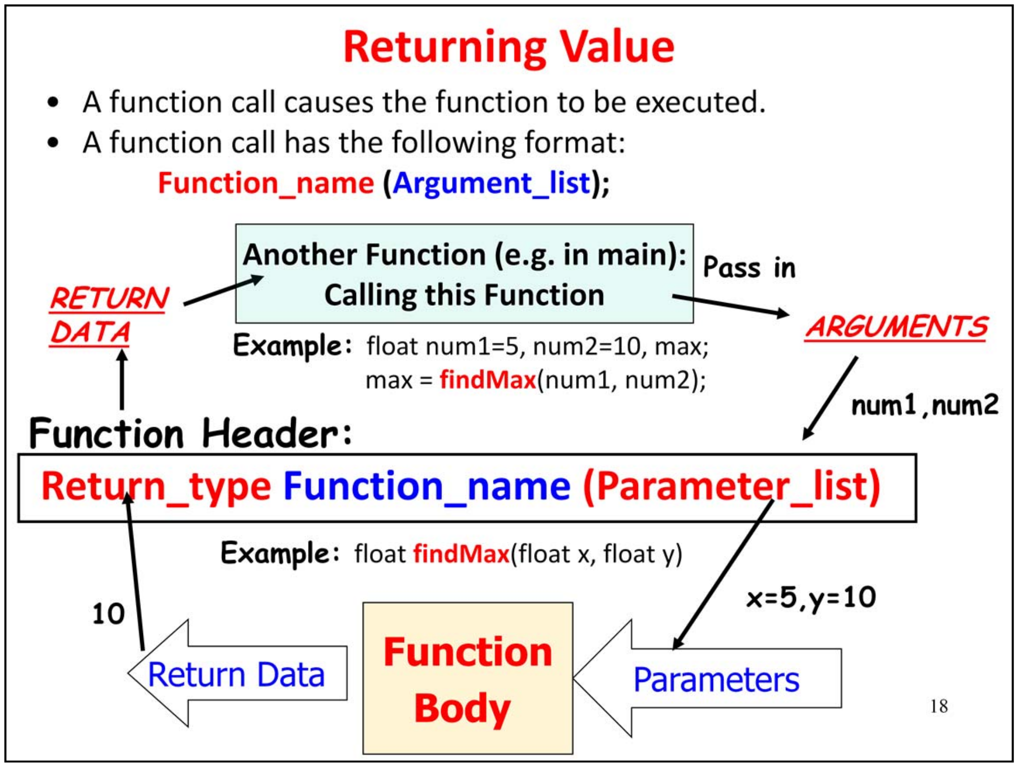
```
hello_n_times(4);
```

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

In the function calls:

```
hello_n_times(nooftimes);
hello_n_times(nooftimes * 4);
```

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



Returning Value

- A function call causes the function to be executed.
- A function call has the following format:

  Function_name (Argument_list);

Example: float num1=5, num2=10, max;
max = **findMax**(num1, num2);

Function Header:

Return_type Function_name (Parameter_list)

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

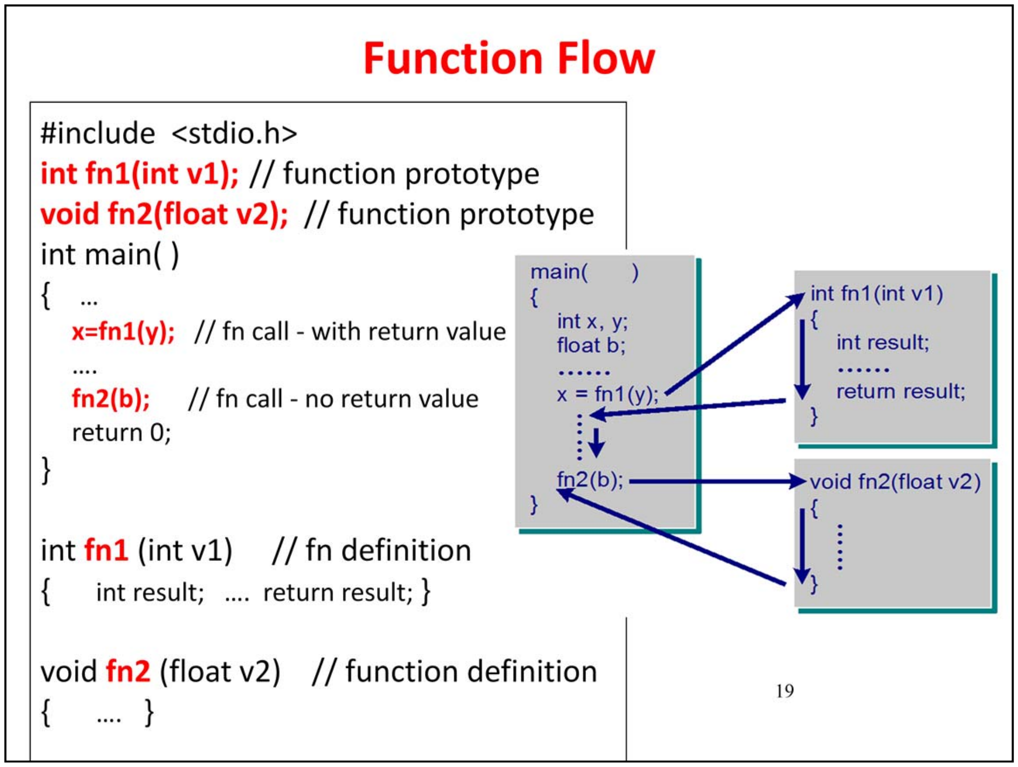
Returning Value

1. A function does not need to return a value such as **void hello_n_times(int n);**
2. However, it can also return a value as shown in the **findMax()** function. The **findMax()** function computes the maximum value of **x** and **y** (i.e. 5 and 10 respectively), and returns the value of 10 to the calling function.
3. The returned value is then assigned to the variable **max**.

A function can return a value such as **double distance(double, double);** We can call the function to return a value as follows:

```
dist = distance(2.0, 4.5);
```

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



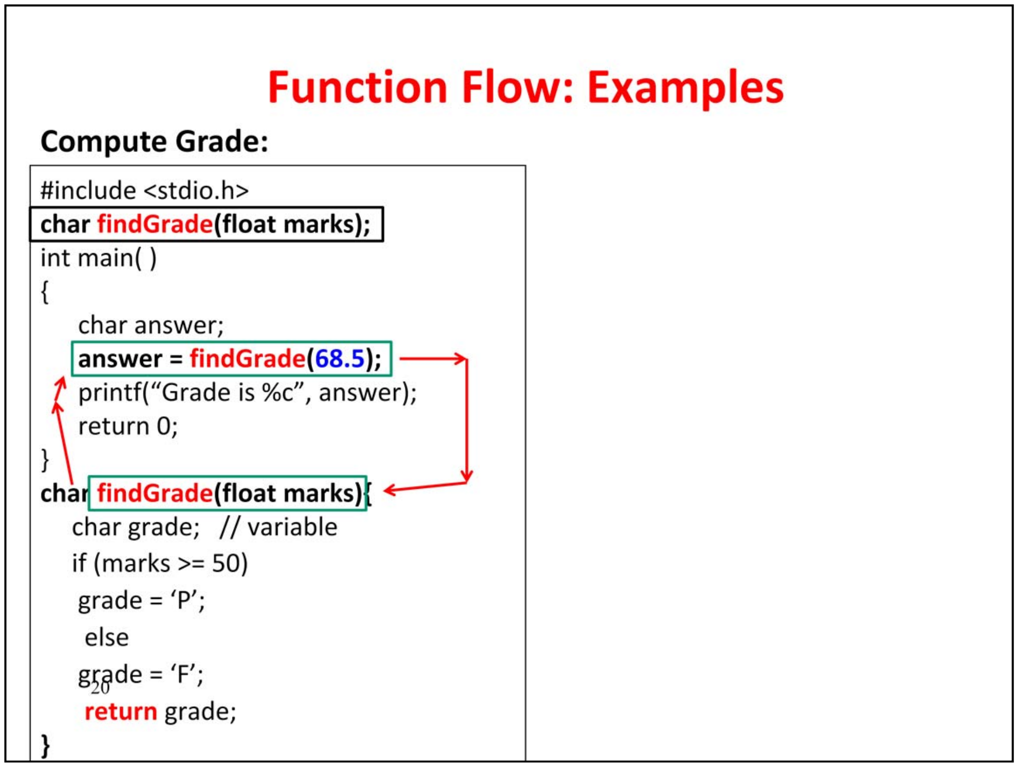
Function Flow

```
#include  <stdio.h>
int fn1(int v1); // function prototype
void fn2(float v2);  // function prototype
int main( )
{   …
    x=fn1(y);   // fn call - with return value
    ….
    fn2(b);     // fn call - no return value
    return 0;
}

int fn1 (int v1)     // fn definition
{     int result;   ….  return result; }

void fn2 (float v2)    // function definition
{     ….   }
```

19

<u>Function Flow</u>

1. In the program, the **main()** function will start the execution.
2. 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.
3. 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.
4. 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

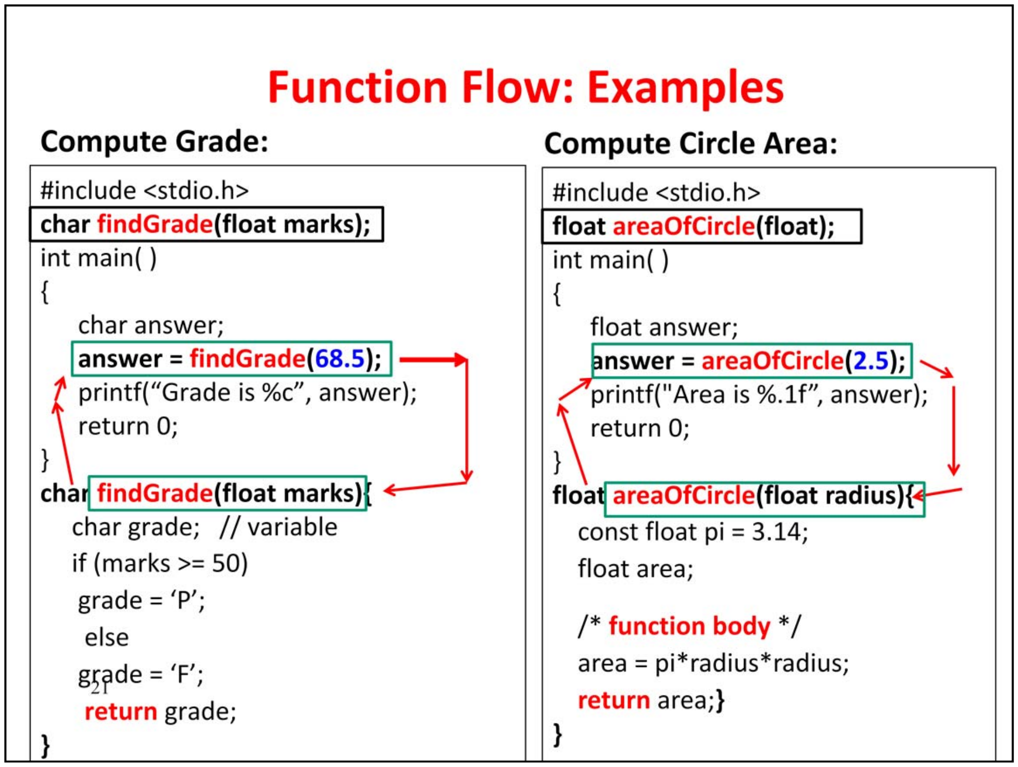
Compute Grade:

```
#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;
}
```

20

Function Flow: findGrade()

1. In the program, the **main()** function calls the function **findGrade()**. When the statement: **answer = findGrade(68.5);** is executed, it calls the function **findGrade()**.
2. 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.
3. 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.
4. 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.
5. Note 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:

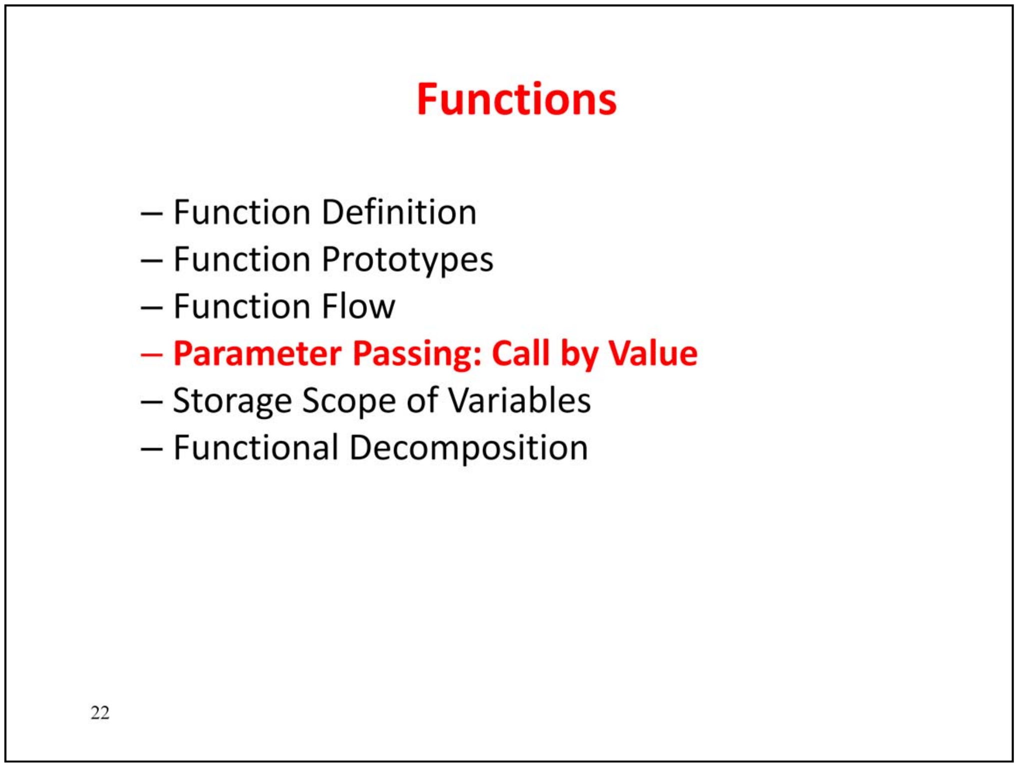
```
#include <stdio.h>
char findGrade(float marks);
int main( )
{
    char answer;
    answer = findGrade(68.5);
    printf("Grade is %c", answer);
    return 0;
}
char findGrade(float marks){
   char grade;   // variable
   if (marks >= 50)
    grade = 'P';
     else
    grade = 'F';
     return grade;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

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: areaOfCircle()

1. In the program, the **main()** function calls the function **areaOfCircle()**. When the statement: **answer = areaOfCircle(2.5);** is executed, it calls the function **areaOfCircle()**.
2. Control is then transferred to the function **areaOfCircle()**. 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 **2.5**. It is assigned to the corresponding parameter in the function definition to compute the area of the circle.
3. 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.
4. Note that the function prototype is declared as: **float areaOfCircle(float);** and is placed at the beginning of the program before the **main()** function.



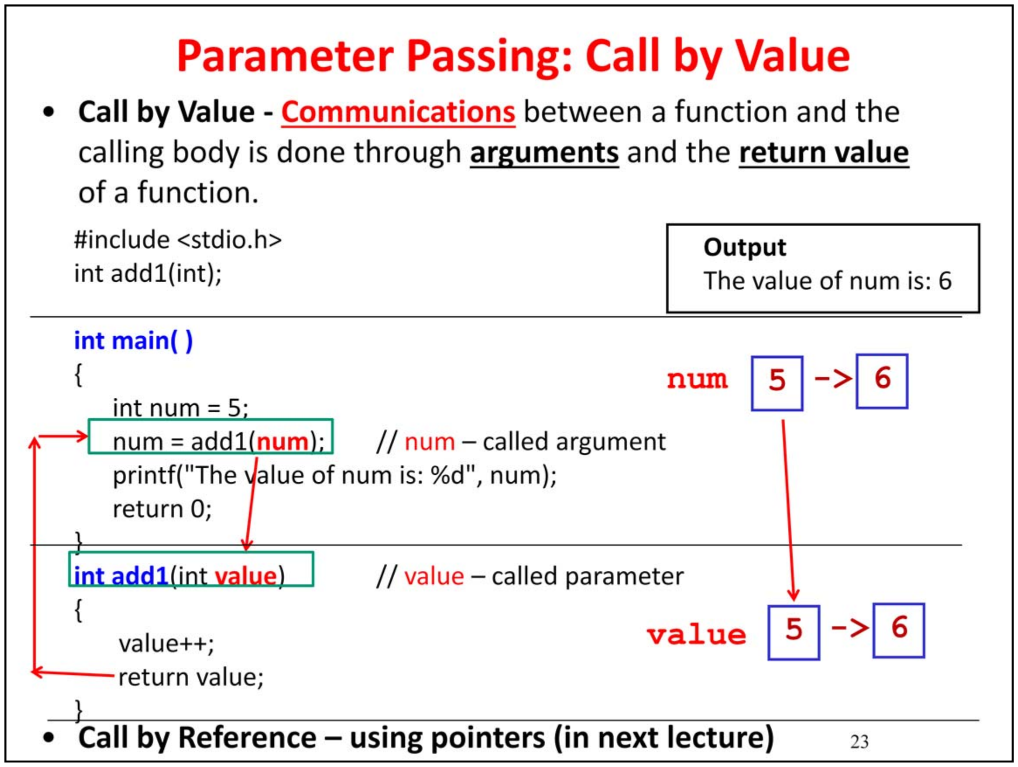
Functions

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

22

Functions

1. Here, we discuss parameter passing using call by value.



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;
}
```

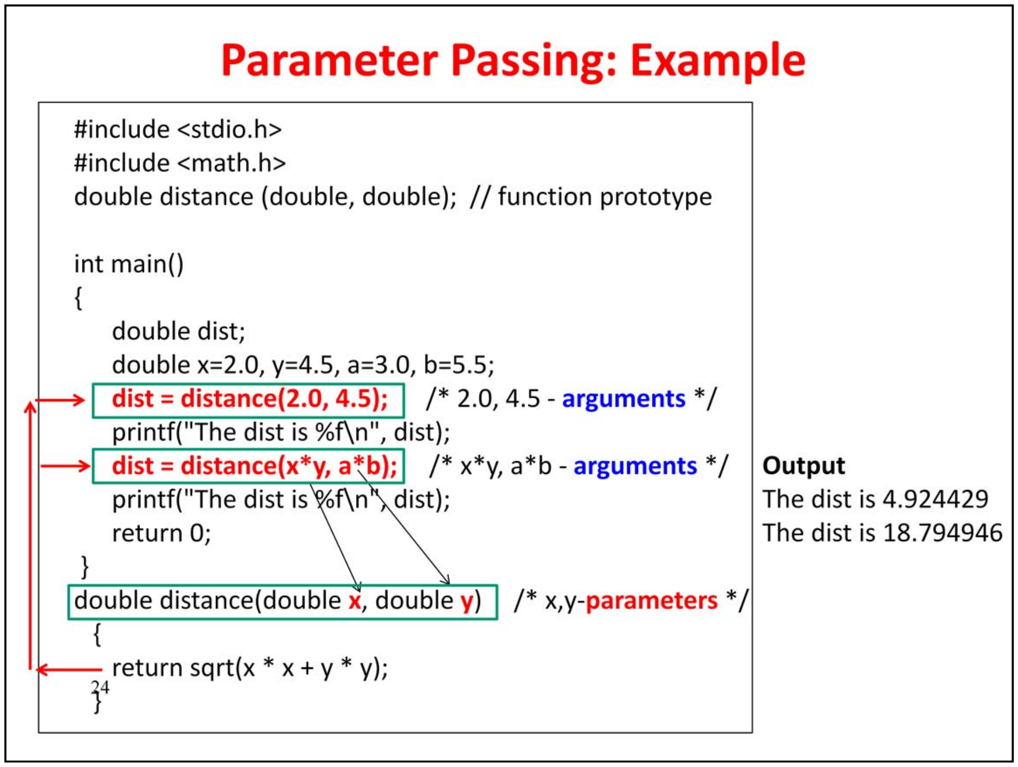
num 5 -> 6
value 5 -> 6

- **Call by Reference – using pointers (in next lecture)**

23

Parameter Passing

1. 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.
2. 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.
3. 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.
4. 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.
5. 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

```
#include <stdio.h>
#include <math.h>
double distance (double, double);  // function prototype

int main()
{
    double dist;
    double x=2.0, y=4.5, a=3.0, b=5.5;
    dist = distance(2.0, 4.5);    /* 2.0, 4.5 - arguments */
    printf("The dist is %f\n", dist);
    dist = distance(x*y, a*b);    /* x*y, a*b - arguments */
    printf("The dist is %f\n", dist);
    return 0;
}
double distance(double x, double y)    /* x,y-parameters */
  {
    return sqrt(x * x + y * y);
  }
```

Output

```
The dist is 4.924429
The dist is 18.794946
```

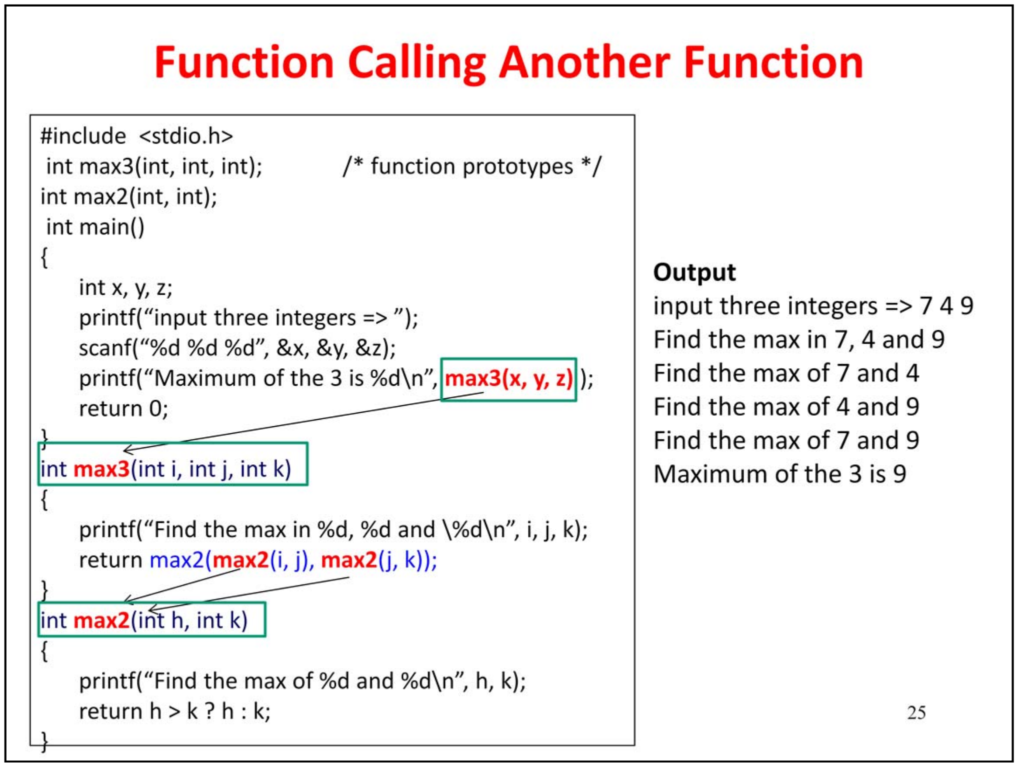
24

<u>Parameter Passing: Example</u>

1. 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()**.
2. 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.
3. 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);
4. 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.
5. Note that 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

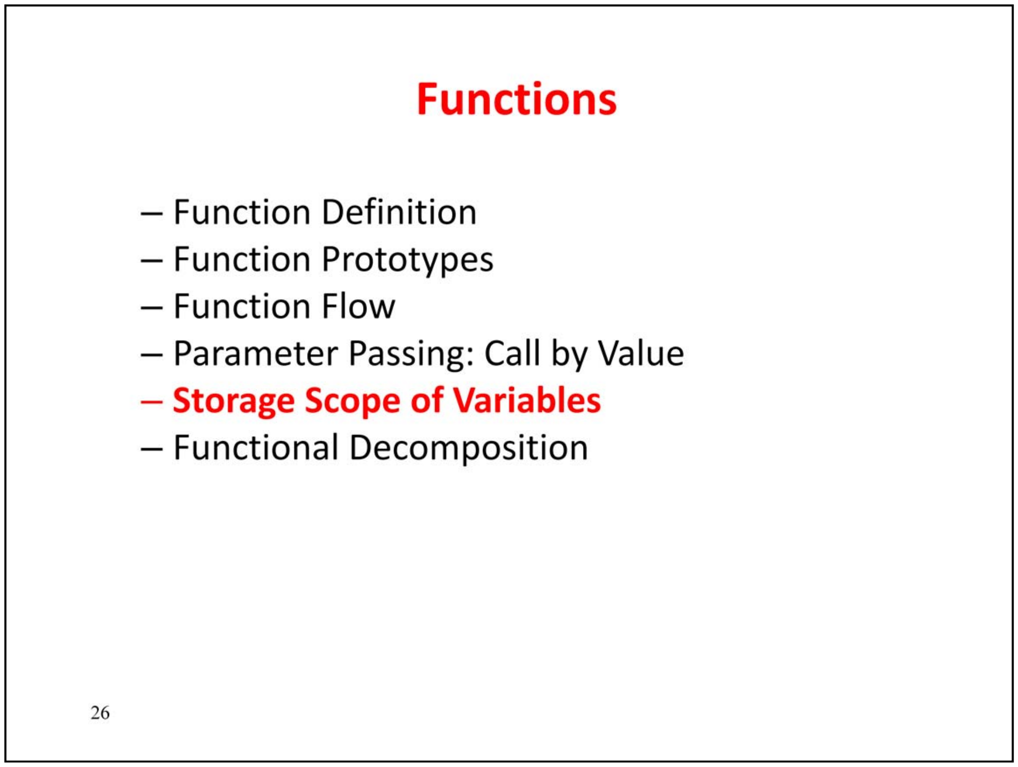
```
input three integers => 7 4 9
Find the max in 7, 4 and 9
Find the max of 7 and 4
Find the max of 4 and 9
Find the max of 7 and 9
Maximum of the 3 is 9
```

25

Function Calling Another Function

1. 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.
2. 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.
3. 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));
4. 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.



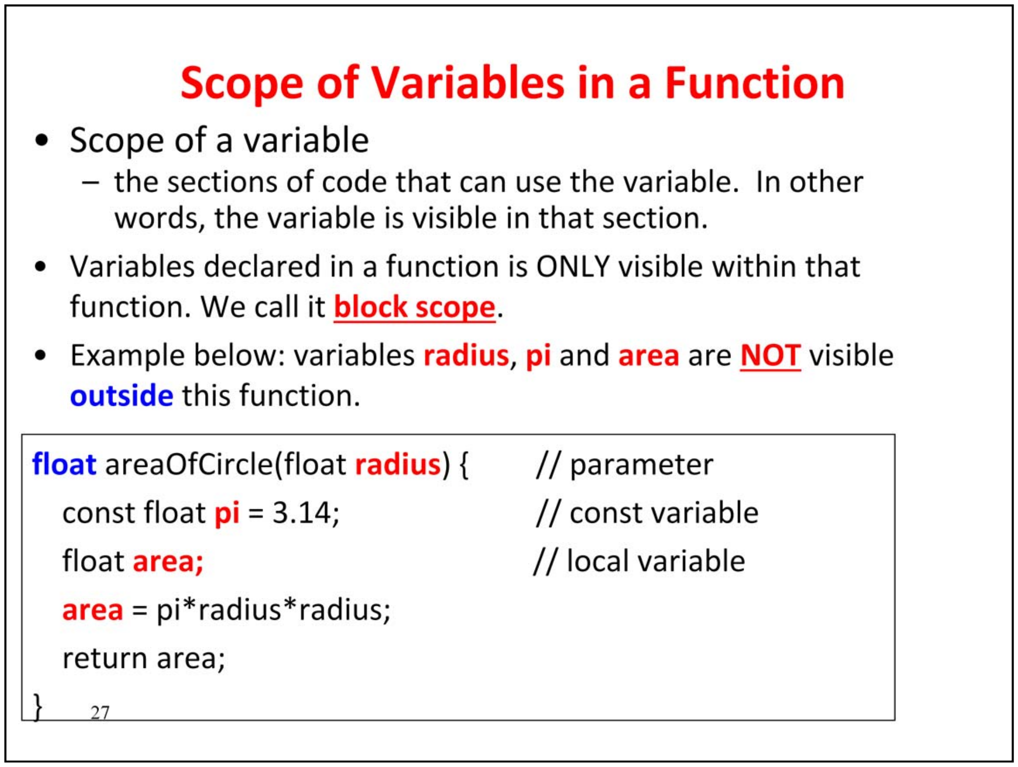
Functions

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

26

Functions

1. Here, we discuss the storage scope of variables.



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;
}
```

27

Scope of Variables in a Functions

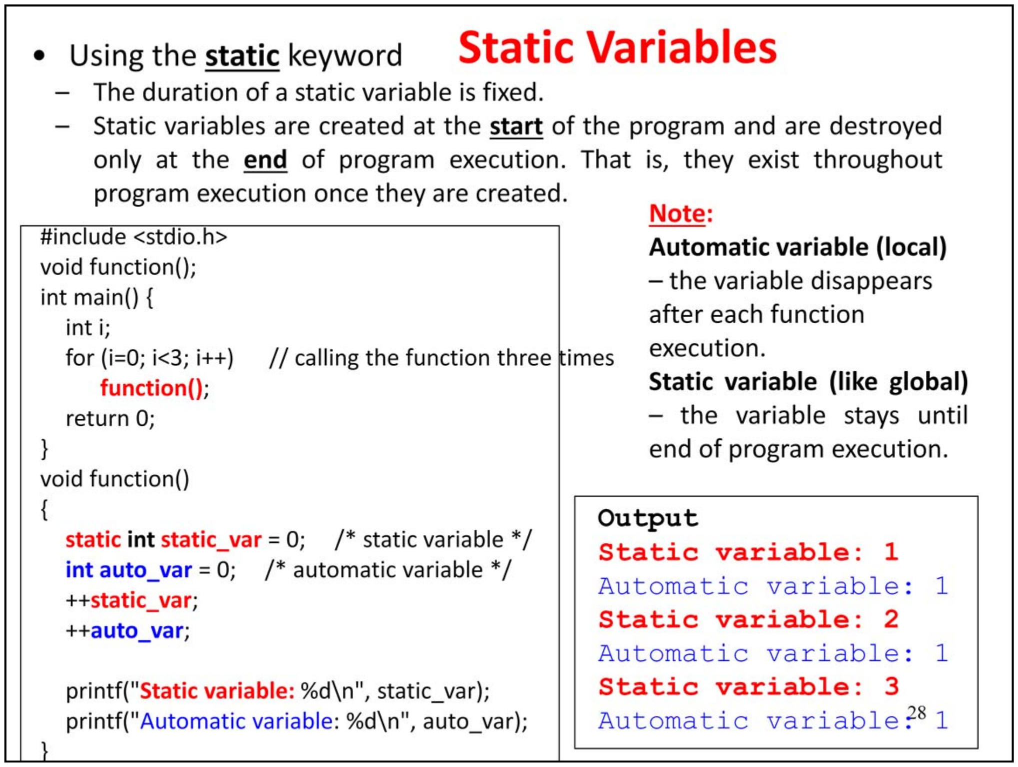
1. The scope of a variable determines the sections of the code that can use the variable. In other words, the variable is visible in that section of code.
2. Variables declared in a function is ONLY visible within that function. We call it block scope.
3. In the example function, the variables **radius**, **pi** and **area** are not visible outside the function **areaOfCircle()**.

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 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
```

28

Static Variables

1. A *static* variable may be defined inside or outside a function's body. The duration of a static variable is fixed.
2. Static variables are created at the start of the program and are destroyed only at the end of program execution.
3. We can define static variables *inside* a function's body by changing an automatic variable using the keyword **static**.
4. 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.
5. Static variables are very useful when we need to write functions that retain values between functions.
6. 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.
7. In the example program, the static variable **static_var** is declared and initialized only once when storage is allocated at the start of the program. The value of the static variable is retained for different calls to **function()**. The value stored in the static variable will remain until the end of the program execution. This is

different from automatic variable as it is created and initialized every time when **function()** is called. However, since the static variable **static_var** is declared inside **function()**, it is only visible inside **function()**.

Other types of variables in C are discussed below for further reading:

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;
```

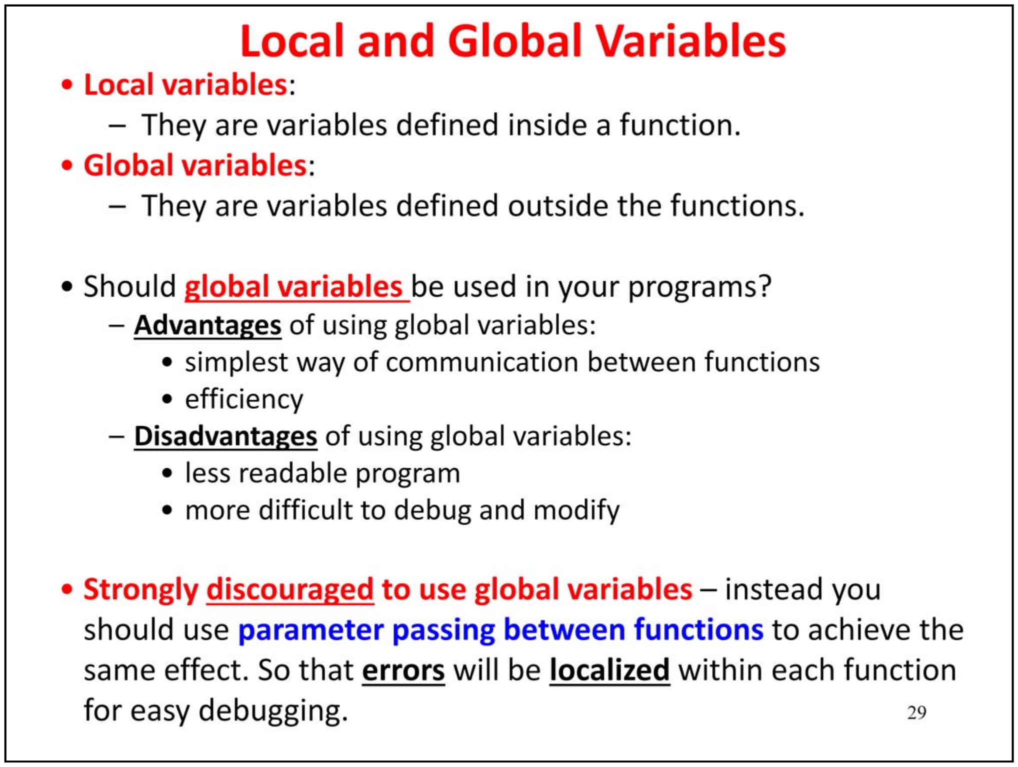
An automatic variable has automatic storage. The lifetime of an automatic variable is only within the function or block where it is defined. It has no meaning outside the function or block. It is also called the *local variable* of the corresponding function or block. Automatic variables are destroyed 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**:
  - They are variables defined inside a function.
- **Global variables**:
  - They are variables defined outside the functions.

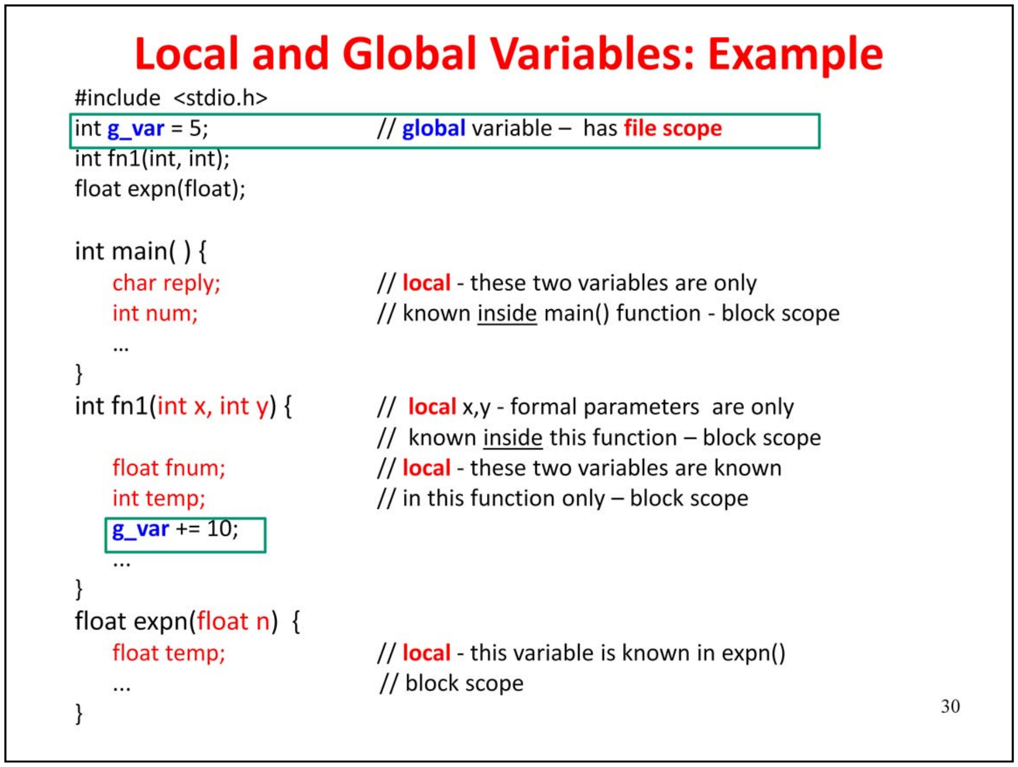
- Should global variables be used in your programs?
  - **Advantages** of using global variables:
    - simplest way of communication between functions
    - efficiency
  - **Disadvantages** of using global variables:
    - less readable program
    - more difficult to debug and modify

- **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.

29

Local and Global Variables

1. Local variables are variables defined inside a function. They have 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.
2. 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.
3. 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.
4. It is strongly discouraged to use global variables. Instead you should use parameter passing between functions to achieve the same effect. So that errors can be localized within each function for easy debugging.



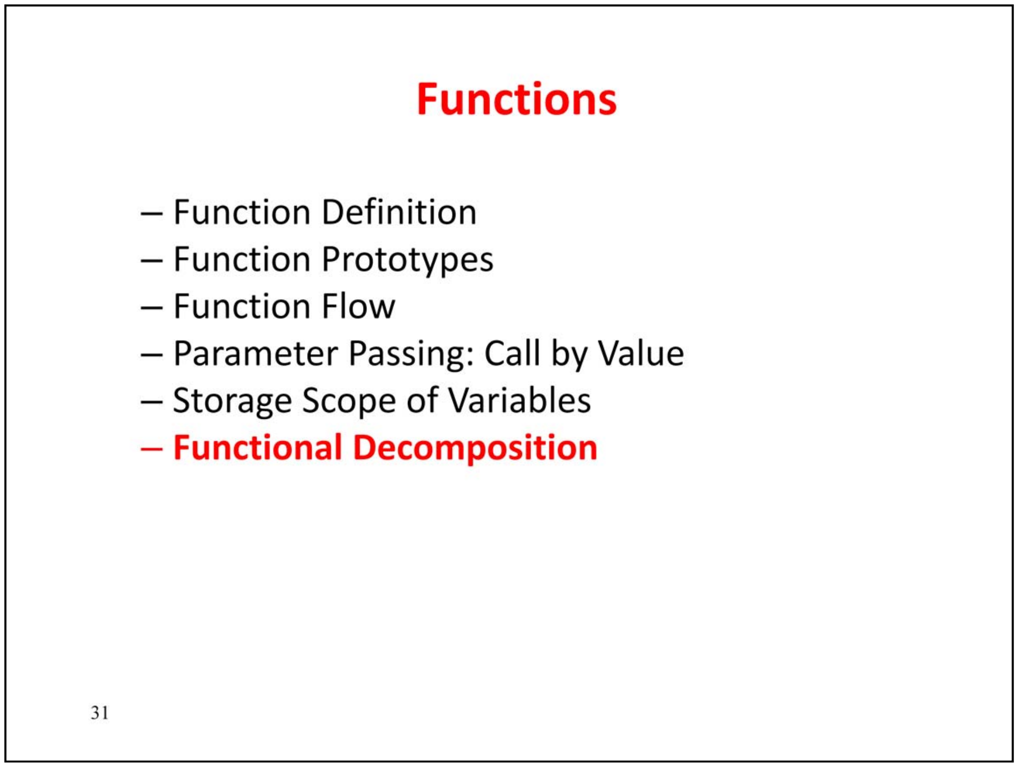
Local and Global Variables: Example

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

30

Local and Global Variables: Example

1. In the example program, the global variable **g_var** is declared outside the **main()** function. Global variables will have the file scope.
2. The variables **fnum** and **temp** are local variables which will have block scope and are only visible inside the function.



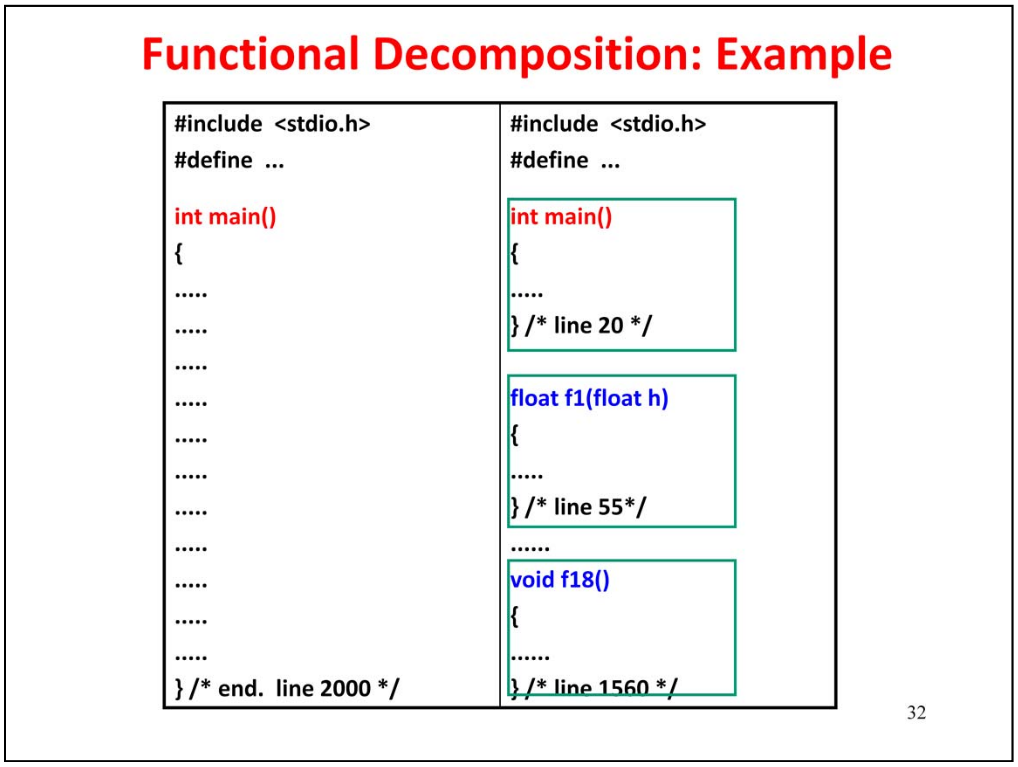
Functions

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

31

Functions

1. Here, we discuss functional decomposition.



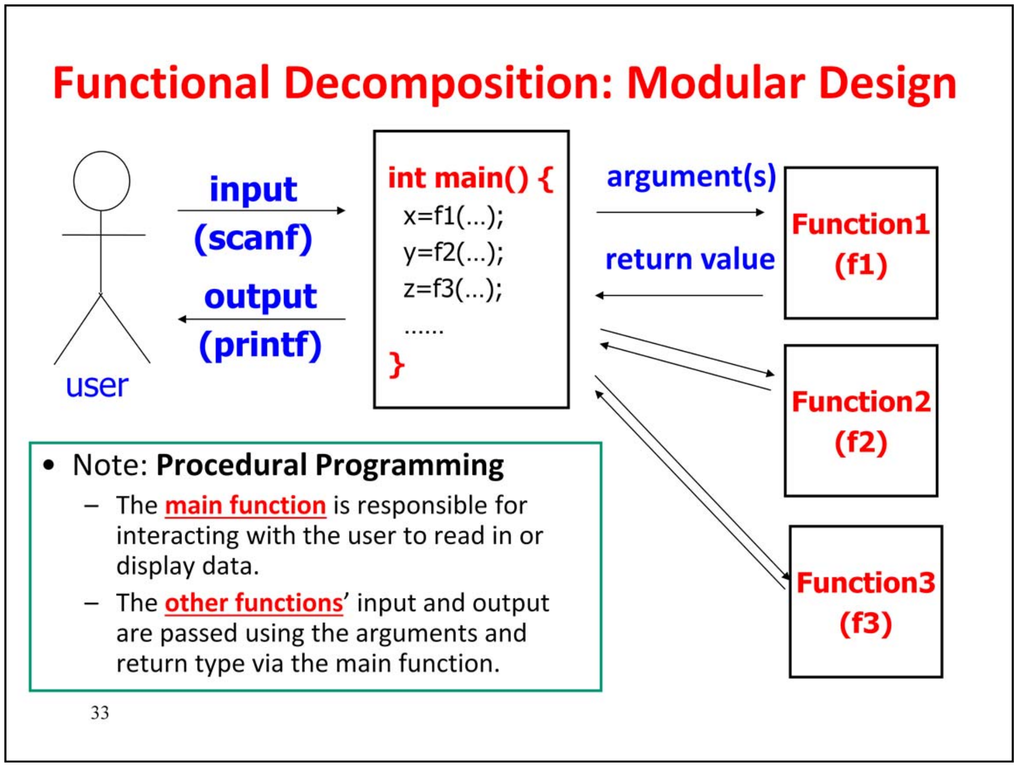
Functional Decomposition: Example

#include <stdio.h> #define ... int main() { .... } /* end. line 2000 */	#include <stdio.h> #define ... int main() { ..... } /* line 20 */ float f1(float h) { ..... } /* line 55*/ ...... void f18() { ...... } /* line 1560 */

32

Functional Decomposition

1. In the original program, the **main()** function contains about 2000 lines of code which is difficult to read and debug.
2. 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). 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.
3. 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.
4. In the example program, it is decomposed into a number of smaller functions. The **main()** function will start program execution and call other functions to perform different required operations.



Functional Decomposition: Modular Design

- Note: **Procedural Programming**
  - The main function is responsible for interacting with the user to read in or display data.
  - The other functions' input and output are passed using the arguments and return type via the main function.

33

Functional Decomposition: Modular Design

1. 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.
2. 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.
3. When writing C programs, we use procedural programming technique, which is different from object-oriented programming paradigm used in Python, Java and C++.
4. In procedural programming, a typical structure of a program consists of the

main function and other functions for solving a problem. Generally, if the functions are still quite complex, then they can be divided further into smaller functions. Generally, each function should not be longer than a page.

Additional note for further reading:

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.

Thank you !!!

34