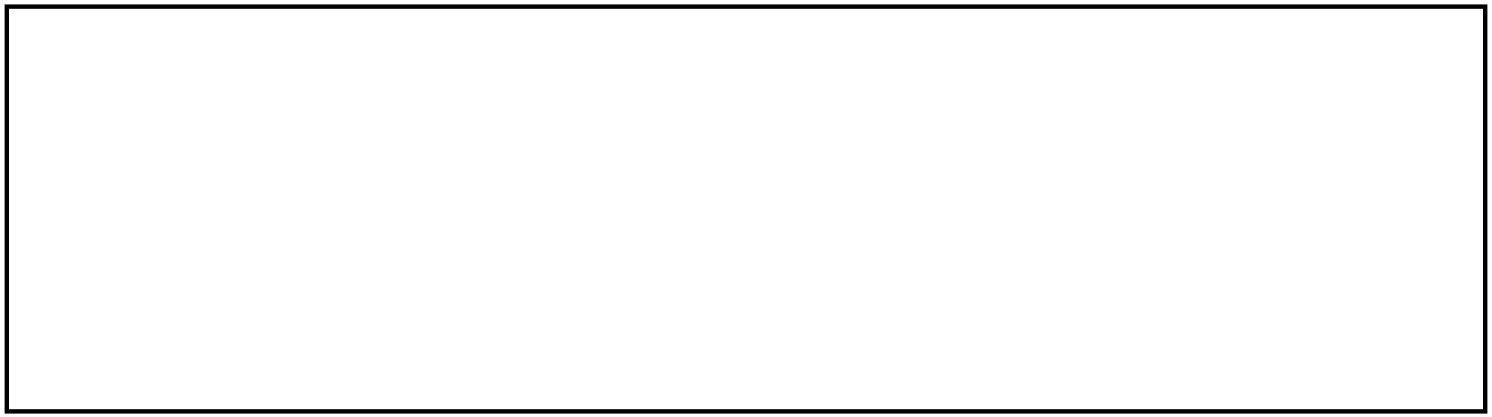
ITC213: STRUCTURED PROGRAMMING



Lecture 09: Functions

Readings: Chapter 7

Introduction

- Divide and Conquer
  - Construct a program from smaller pieces or components
  - These smaller pieces are called modules
  - Each piece is more manageable than the original program
- Functions
  - Modules in C
  - Building blocks of C and the place where all program activity occurs
  - Programs combine user-defined functions with library functions
  - C standard library has a wide variety of functions

Function Defined

- A function is a named, independent section of a program that performs a specific, well-defined task and can optionally return a value to the calling program
  - Each function has a unique name
  - A function can perform its task without interference from or interfering with other parts of the program
  - A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root
  - When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program

Functions (1/2)

- Every C program consists of one or more functions
- One of these functions is the main function
- Execution of a program will always begin by carrying out the instructions in main
- Additional functions will be subordinate to main, and perhaps one to another
- If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another

Functions (2/2)

- A function will carry out its action whenever the function is called from some other portion of the program
- Same function can be called from several different parts of the program
- Generally, a function will process information that is passed to it from the calling portion of the program and return a single value
- Information is passed to the function via special identifiers called arguments and returned via the return statement

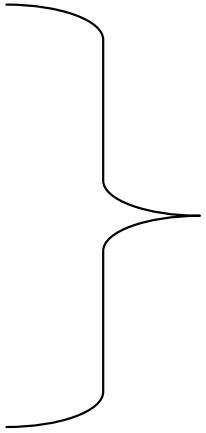
Using Functions

- To use a C function, you must
  - Provide a function definition
  - Provide a function prototype
  - Call the function
- For library functions
  - Already defined and compiled
  - Use header files to provide prototype
  - Only call the function properly
- For user-defined functions you have do all the three things yourself

Example

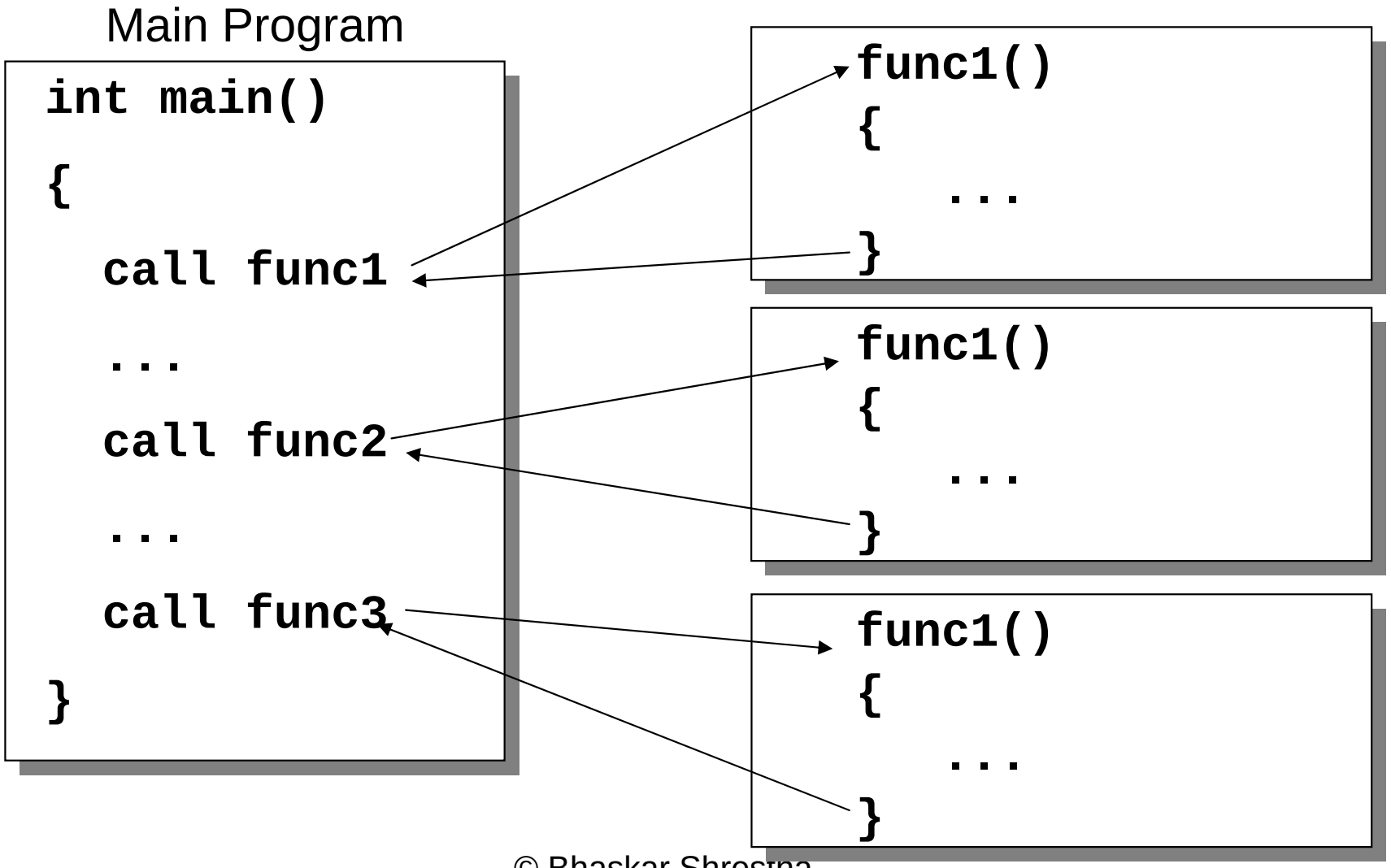
```
long cube(long x); /* function prototype */
int main()
{
  long answer, input;
  printf("Enter an integer value: ");
  scanf("%ld", &input);
  answer = cube(input); /* function call */
  printf("\nThe cube is %ld.\n", answer);
  return 0;
}

long cube(long x)
{
     long x_cubed;
     x_cubed = x * x * x;
     return x_cubed;
}
```



function definition

How Function Works



Advantages of Using Functions

- Manageable program development
  - Allows a program to be broken down into a number of smaller, self-contained components
- Code Reuse
  - Use existing functions as building blocks for new programs
  - Avoids the need for repeated programming of the same instructions
- Abstraction
  - hide internal details (library functions)

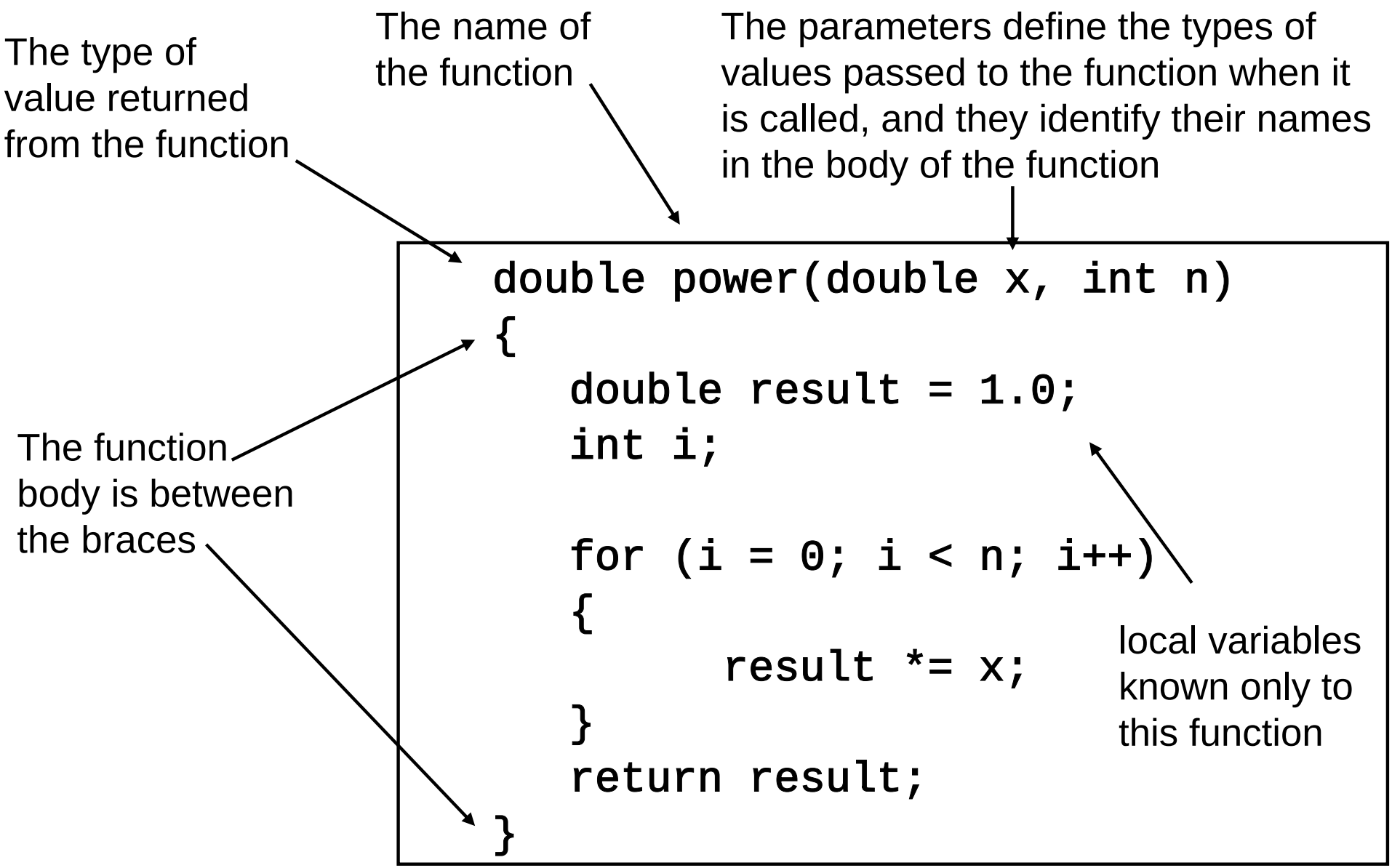
Function Definition (1/2)

- Function definition format
  - *return-value-type function-name*(*parameter-list*)
    ```
    {
       declarations and statements
    }
    ```
- *function-name*: any valid identifier
- *return-value-type*: data type of the result (can be omitted, defaults `int`)
  - `void` – indicates that the function returns nothing
- *parameter-list*: comma separated list, declares parameter variables
  - A type must be listed explicitly for each parameter

Function Definition (2/2)

- Declarations and statements: function body (block)
  - The function body will contain the statements that defines the action to be taken by the function
  - The statements inside the body can be expression statements, compound statements, control statements and so on
  - Variables can be declared inside blocks (can be nested)
    - All variables declared inside functions are local variables (Known only inside the function)
  - Functions can not be defined inside other functions

Function Definition Example



More Examples

```
void sayhello(void)
{
  puts("Hello");
}

double squared(double number)
{
  return (number * number);
}

void printchar(char ch, int times)
{
  int i;
  for (i = 0; i < times; i++)
     putchar(ch);
}
```

The `return` keyword

- The **`return`** keyword is used to return the program control from a function to the calling function
- In general terms, the return statement is written as **`return`** ***`expression`***`;`
  - The value of ***`expression`*** is returned to the calling function of the program
  - No ***`expression`*** is present if a function return type is **`void`**
- A function definition can contain multiple **`return`** statements. However, only one gets executed

Examples

```
void func(int n)
{
  if (n < 1)
  {
    puts("Number is not positive");
    return;
  }
  if (n%2 == 0)
    puts("Number is even");
  else
    puts("Number is odd");
}
```

```
int maximum(int a, int b)
{
     if ( a > b)
          return a;
     else
          return b;
}
```

Calling a function

- When a function is called its code gets executed
  - Call a function by specifying its name, followed by list of arguments enclosed in parentheses and separated by commas
- Arguments
  - the value of the arguments is transferred to the corresponding parameter in the function definition
  - arguments can be constants, single variables, or more complex expressions
  - type of argument must match the parameter type
  - the no of arguments must be equal to no. of parameters in the function definition
  - if no arguments are required, an empty () must be present
- Can be called more than once as desired

Calling Functions Example

```
result = power(10.0, 5);
c = maximum(a, b);
y = squared(x);
```

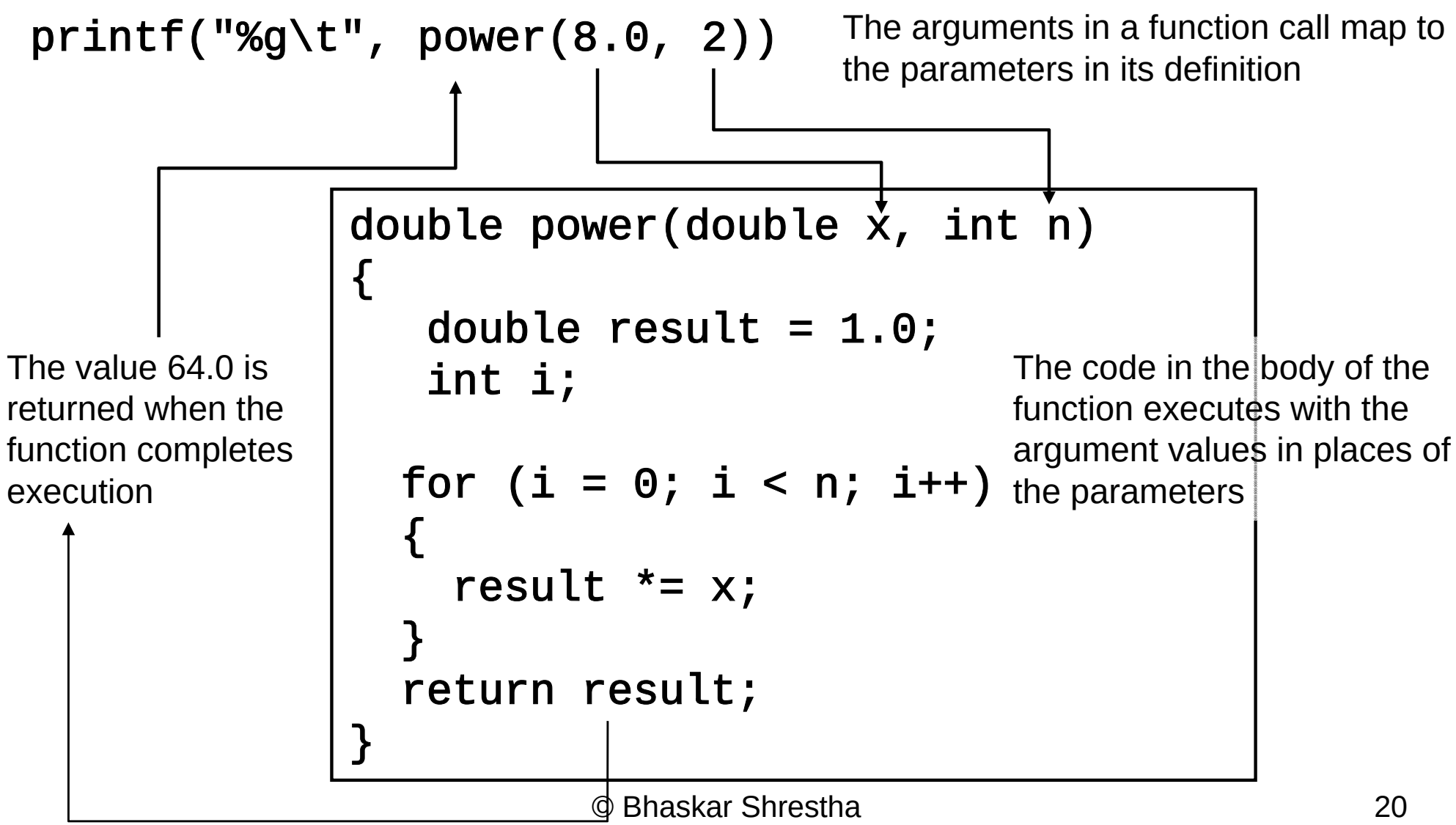
```
print_report(1);
sayhello();
solvequadratic(a, b, c);
```

```
g = power(x, maximum(y*2, n));
if (maximum(x,y) > 10 )
{
...
}
```

Parameters and Arguments

- You pass information to a function by means of the arguments that you specify when you call a function
- The arguments are placed between parentheses following the function name in the call
  - `printf("%g\t", power(8.0, i));`
- When a function is called the value of the arguments is copied to the corresponding parameters in the function definition
- Arguments are also called ***actual arguments*** and parameters are called ***formal parameters***

Parameters and Arguments



Function Prototypes (1/2)

- Function prototype is a declaration statement that
  - specifies the function name, the no and types of its parameters and the return type of the function
- Prototype only needed if function definition comes after use in program
- They are generally written at the beginning of a program, ahead of any user-defined function
- The function with the prototype
  `double power(double x, int n);`
  - Takes in one `double` and one `int`
  - Returns a `double`

Function Prototypes (2/2)

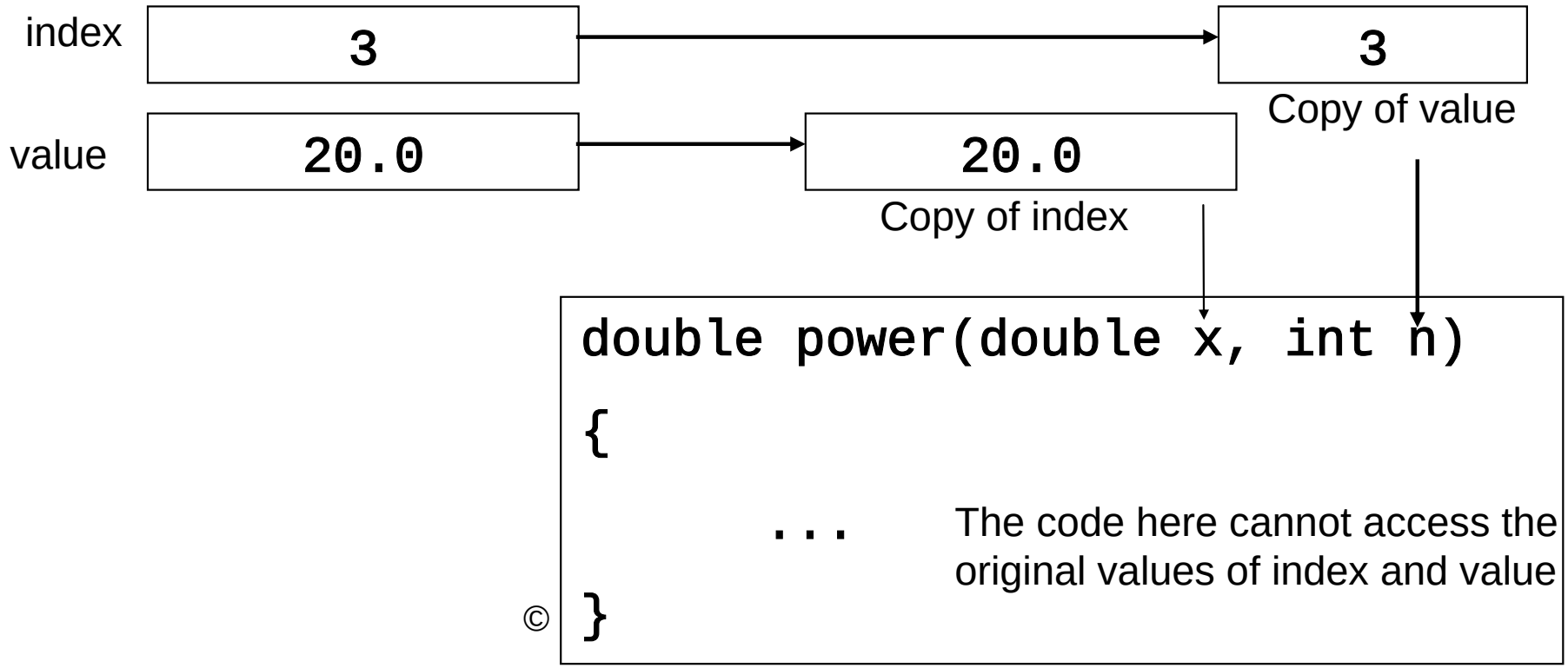
- Function prototype is identical to the function header, with a semicolon appended
- Parameters name can be different than that of the function definition
  - `double power(double val, int exponent);`
- Parameter names can be omitted
  - `double power(double , int );`
- Function prototype facilitate error checking between calls to a function and the corresponding function definition

Passing Arguments to a Function

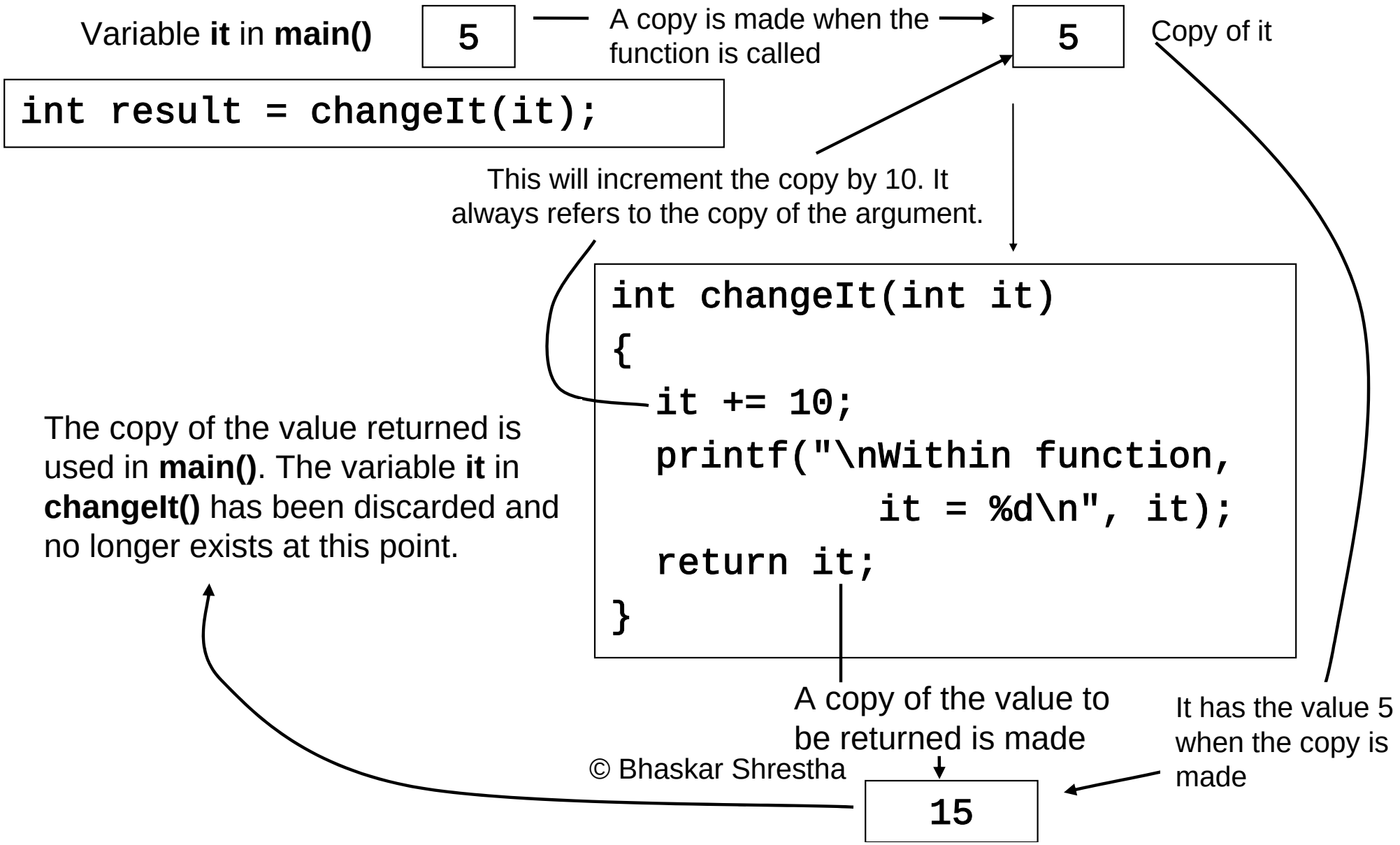
- Two mechanisms used generally in C to pass arguments to functions
- Pass by value
  - Copy of argument passed to function
  - Changes in function do not effect original argument passed
  - Use when function does not need to modify argument
  - Avoids accidental changes
- Pass by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

Pass By Value

```
double value = 20.0;
int index = 3;
double result;
result = power(value, index);
```



Failure to modify arguments value



Declarations Vs. Definitions

- Declaration: introduces a name – an identifier – to the compiler
  - tells the compiler: “This function or this variable exists somewhere, and here is what it should look like
    - Function Prototypes are declarations
- Definition: allocates storage for the name
  - tells the compiler: “Make this variable here” or “Make this function here
  - For a variable, determines how big that variable is and causes space to be generated in memory For a function, generates code, which ends up occupying storage in the memory
- You can declare a variable or a function in many different places, but there must be only one definition in C
- A definition can also be a declaration

Storage Classes

- The ***storage class*** of a variable determines its scope, its storage duration, and its linkage
- ***Storage duration*** **–** how long a variable exists in memory
- ***Scope*** **–** where the variable can be referenced in program
- ***Linkage*** **–** specifies the files in which the variable is known
- The storage class of a variable is determined by the position of its declaration in the source file and by the storage class specifier, if any
  - `auto`, `extern`, `static`, `register`

Automatic Storage Duration (1/2)

- Variables defined inside a block (including formal parameters) have automatic storage
- In automatic storage, the variable is created each time program flow enters the block in which it is defined
- When the block is terminated, the memory occupied by the variable is freed
- The storage class specifier **`auto`** or **`register`** may be used for variables defined inside functions

Automatic Storage Duration (2/2)

- `auto`:
  - default for variables defined inside functions
  - `auto double x, y;`
- `register`:
  - tries to put variable into high-speed registers
  - Can only be used for automatic variables
  - `register int counter = 1;`
- Automatic variables have unknown values if they are not initialized

Examples

```
int x = 10, y;
printf("Outer x: %d\n", x);
printf("y: %d\n", y);
{
        int x = 20;
        printf("Inner x: %d\n", x);
}
printf("Outer x: %d\n", x);
```

```
int i;
for (i = 1; i <= 5; i++)
{
        int x = 10;
        x += 10;
        printf("x = %d\n", x);
}
/* ERROR: Can't access x here */
/* printf("%d\n", x); */
```

Static Storage

- The variable is created and initialized once for the first time its definition statement is encountered
- It exists continuously throughout the execution of the program
- Default value of zero
- Variables defined outside of any function and variables defined with **`static`** storage class specifier inside function have static storage

Static Variables

- Static global variables:
  - Variables defined outside of any function with **`static`** storage class specifier
  - Can be used by any functions in the same file
- Static local variables
  - Variables defined inside a block with **`static`** storage class specifier
  - Can be used only in the function where it is declared

```
int i;
for (i = 1; i < 10; i++)
{
  static int x = 10;
  x += 10;
  printf("x = %d\n", x);
}
/*ERROR: Can't access x here*/
/* printf("%d\n", x); */
```

External Variables

- Extern:
  - Can be used by any function
  - Variables defined outside of any function
  - Default for global variables and functions

```
int x = 10;
void func(void)
{
        x += 10;
        printf("%d\n", x);
}
main()
{
        printf("%d\n", x);
        func();
        x += 10;
        printf("%d\n", x);
}
```

Use of extern keyword

- Use the extern keyword to declare a global variable that has been defined elsewhere in the program

```
extern int;
void func(void)
{
        x += 10;
        printf("%d\n", x);
}
main()
{
        printf("%d\n", x);
        func();
        x += 10;
        printf("%d\n", x);
}
int x = 10;
```

Scope Rules

- File scope
  - Identifier declared outside function, known in all functions
  - Used for global variables, function definitions, function prototypes
- Function scope
  - Can only be referenced inside a function body
  - Used only for labels (start:, case: , etc.)

Scope Rules

- Block scope
  - Identifier declared inside a block
    - Block scope begins at declaration, ends at right brace
  - Used for variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block
- Function prototype scope
  - Used for identifiers in parameter list