COSC 120: Computer Science I

Module 3

Instructors:

Dr. Xiaohong (Sophie) Wang

(xswang@salisbury.edu)

Department of Computer Science

Salisbury University

Spring 2024

Module 3: Arrays and Functions

1. Arrays
2. Modular Programming
3. Defining and Calling Functions
4. Passing Data into a Function
5. Returning a Value from a Function
6. Using Reference Variables as Parameters
7. Local and Global Variables
8. Default Arguments
9. Overloading Functions
10. Recursion Function

Arrays

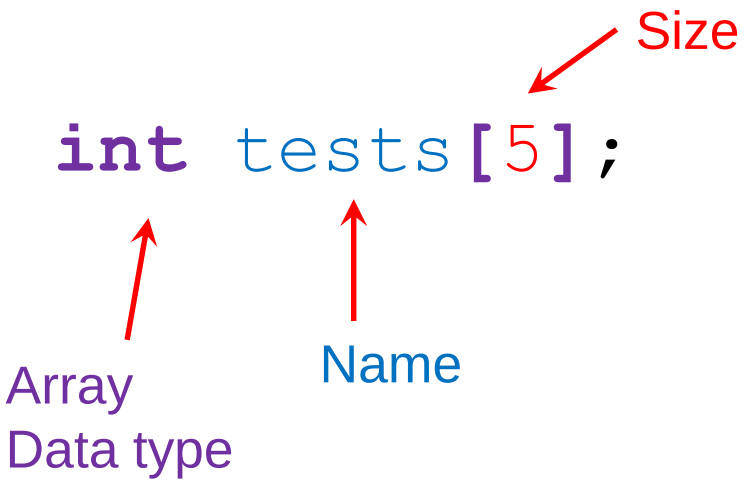
- Array in C++
- Range-Based `for` loop
- Processing Array Contents
- Arrays as Function Arguments
- Two-Dimensional Arrays

Array in C++

- Array:

  *is a data type that allows variables of this type store **multiple** values of the **same type***

- Values are stored in **adjacent memory** locations
- Defined using the date type of each element and a `[]` operator:

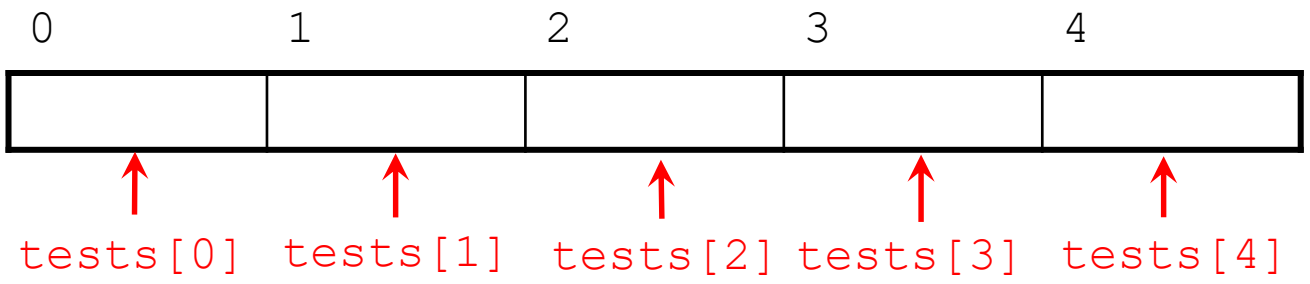


Accessing Array Elements

- Each element in an array is assigned a **unique** *subscript* from `0` to `n-1` while `n` is the size of the array
- To access an element in an array:

```
array_name[subscript]
```

```
int tests[5];
```



Accessing Array Elements (Cont'd)

- Each array element can be used as a regular variable:

```
tests[0] = 79;
cout << tests[0]; //
cin >> tests[1];
tests[4] = tests[0] + tests[1];
```

- Arrays must be accessed via **individual** elements:

```
cout << tests; // not legal
```

Using a Loop to Step Through an Array

- Example – The following code defines an array, `numbers`, and assigns 99 to each element:

```
const int ARRAY_SIZE = 5;
int numbers[ARRAY_SIZE];

for (int count = 0; count < ARRAY_SIZE; count++)
     numbers[count] = 99;
```

The variable `count` starts at `0`, which is the first valid subscript value

The loop ends when the variable `count` reaches 5, which is the first invalid subscript value

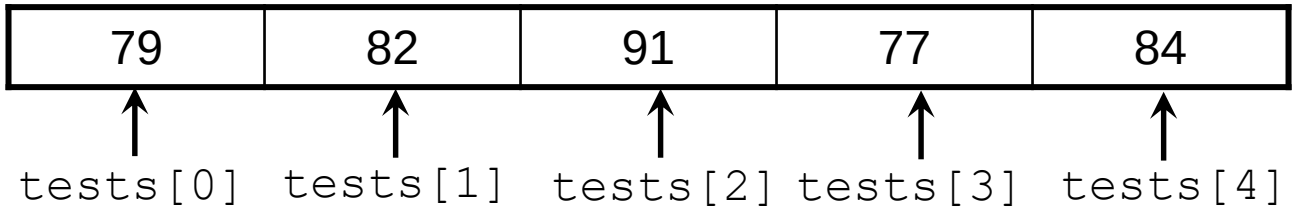
The variable `count` is incremented after each iteration

Array Initialization

- An array can be initialized with an initialization list:

```
const int SIZE = 5;
int tests[SIZE] = {79,82,91,77,84};
```

- The values are stored in the array in the order in which they appear in the list.



- The initialization list cannot exceed the array size.

No Bounds Checking in C++

- When you use an array subscript, C++ does not check whether it is a *valid* subscript or not
  - You can use subscripts that are beyond the bounds of the array

```
int values[3] = {5, 8, 10};

// Syntax correct, but may corrupt other memory
// locations, crash program, or cause elusive bugs
values[3] = 12;
```

- A common mistake: ***off-by-one error***
  - Subscripts are between `0` and `n-1`, not `1` and `n`

```
int numbers[10];
for (int count = 0; count < 10; count++)
   numbers[count] = 0;
```

Processing Array Contents

- Array elements can be treated as ordinary variables of the same type as the array
  - Each element is a variable
  - Processing an element is no different than processing other variables

- When using ++, −− operators, don't confuse the element with the subscript: i=2; tests[i]=10;

```
cout << tests[i]++; //add 1 to tests[i] then display
cout << tests[i++]; // increment i (i=i+1), then display
```

Array Assignment

To copy one array to another,

- Don’t try to assign one array to the other:

```
newTests = tests;  // Won't work
```

- Instead, assign element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
   newTests[i] = tests[i];
```

Note: Anytime the name of an array is used without brackets and a subscript, it is seen as the array’s **beginning memory address** (not a variable).

In-class practice

- Take 5 integers from user and store these numbers in an array
- Use a `for` loop to find the largest element of this array
- Display this element
- Test you code

Question: How to implement this practice using range-based `for` loop?

Reference code: LargestInteger.cpp

Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules

This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.

```
int main()
{
   statement;
   statement;
   statement;
}
```

main function

```
void function2()
{
   statement;
   statement;
   statement;
}
```

function 2

```
void function3()
{
   statement;
   statement;
   statement;
}
```

function 3

Advantages of using functions

- Code reuse:
  - Golden rule of function: if you need to write one sentence more than once, write it as a function
- Decompose complex problem into simpler ones
- Reliability: avoid to copy bugs
- Maintainability: modify one place, affect all calls to this function
- Improve the clarity of code

Question: Is there any disadvantage of using functions?

Defining and Calling Functions

- Function definition: contains the statements that make up a function

- Function call: a statement that causes a function to execute

Note:

- You MUST place either the **function definition** or the **function prototype** ahead of all calls to the function. Otherwise, the program will not compile.
- We will introduce **function prototype** later.

Function Definition

```
return_type function_name(parameter list)
{
    body of the function
}
```

- return type: data type of the value that function returns (*the output of the function*)
- name: name of the function. Function names follow same rules as variables
- parameter list: variables containing values passed to the function (*the input of the function*)
- body: statements that perform the function's task, enclosed in { }

Take a function as a black box. The input and output are its interfaces.

https://www.tutorialspoint.com/cplusplus/cpp_functions.htm

Example

```
// function returning the max between two numbers

int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Questions:

- What is the return type?
- How many input parameters? What are their types?

https://tutorialink.com/cpp/cpp-functions.cpp

Calling a Function

- Function `main` is called automatically when a program starts; all other functions are called by `function call` statements

- How to call a function:

```
function_name(arguments);
```

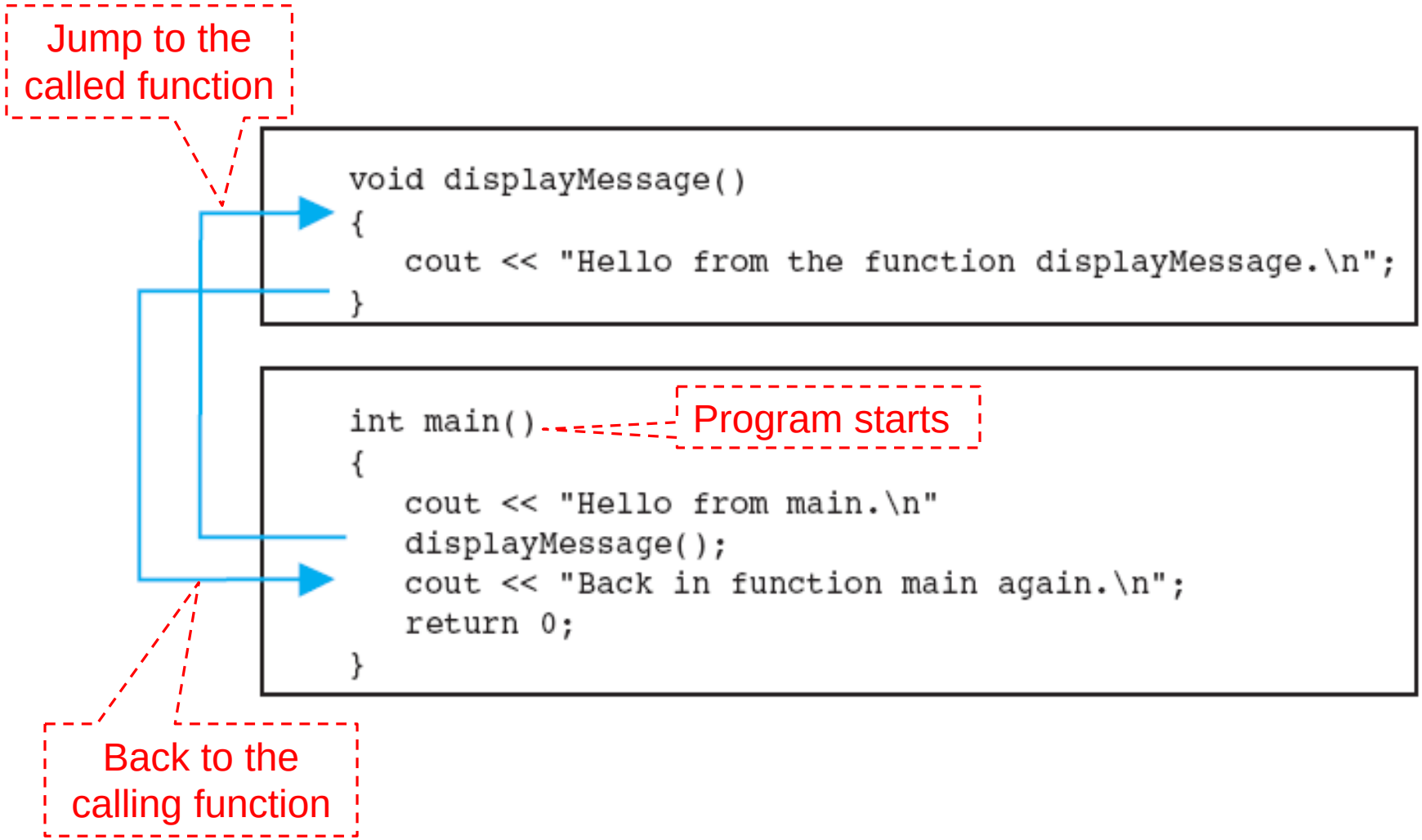
  - `E.g.: max(a, 200);`

Values/variables corresponding to the parameter list

- When called, program executes the body of the called function

- After the function terminates, execution resumes in the calling function at point of call

Flow of Function Call



In-class practice

- Define a function named `rectDraw` to draw the following rectangle with asterisks
- Call the `rectDraw` function in the `main` function
- Test your code

```
*
        ********
        ********
        ********
        ********
```


Reference code: RectDraw.cpp

Function Prototypes

- Compiler must know the following about a function before it is called:
  - name
  - return type
  - number of parameters
  - data type of each parameter

- Two solutions:
  - Define a function before call it
  - Declare a function with a `function prototype` (i.e. `function declaration`) before call it

Function Prototypes

- Syntax of function prototype:

```
return_type function_name(parameter_list);
```

  - Must end with a semicolon
  - Usually placed near the top of a program
  - When using prototypes, can place function definitions in any order in source file
  - In the parameter list, the parameter names are optional. For example, the following two are both correct.

```
int protofunction (int first, int second); or
int protofunction (int, int);
```

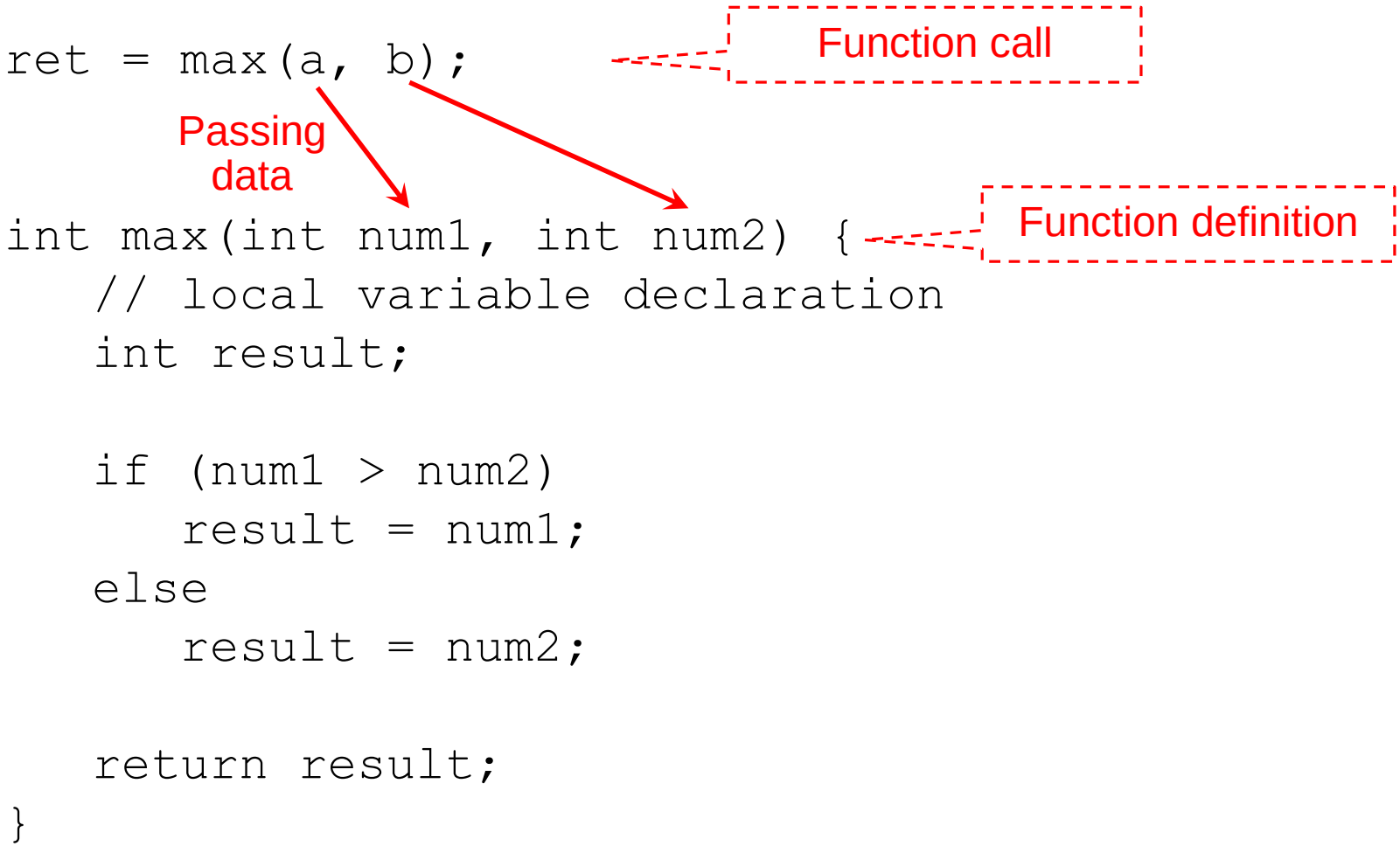
Example

```
...
void deep();
void deeper();

int main() {
    cout << "I am starting in function main. \n"
    deep();
    cout << "Back in function main again. \n"
    return 0;
}

void deep() {
    cout << "I am now in the function deep. \n"
    deeper();
    cout << "Now I am back in deep. \n"
}

void deeper() {
    cout << "I am now in the function deeper. \n"
}
```

Function prototypes

Passing Data into a Function

- Can pass values into a function at time of call:



Parameter and Argument

- Values passed to function are arguments

```
ret = max(a, b);
```

Arguments

- Variables in a function that hold the values passed as arguments are parameters

```
int max(int num1, int num2)
```

Parameters

In-class practice

- Define a function named `rectDraw` to draw a rectangle of size `width * height` with asterisks.
  - `width` and `height` are two parameters of the `rectDraw` function
- In the `main` function, ask the user to input the values of these two arguments; call the `rectDraw` function to draw the rectangle
- Test your code

Reference code: RectDraw2.cpp

Arrays as Function Arguments

- To pass an array to a function, use the array name:

```
int tests[5] = {79,82,91,77,84};
showScores(tests);
```

- To define a function that takes an array parameter, use empty `[]` for array argument:

```
// function prototype
void showScores(int []);

// function header
void showScores(int scores[])
```

No size declarator inside the brackets

Note: When an entire array is passed to a function, it is not passed by value, but passed by reference (only the starting memory address is passed).

Arrays as Function Arguments

- When passing an array to a function, it is common to pass **array size** so that function knows how many elements to process:

```
showScores(tests, ARRAY_SIZE);
```

# of elements

- Array size must also be reflected in prototype, header:

```
// function prototype
void showScores(int [], int);

// function header
void showScores(int scores[], int size)
```

Example

```
#include <iostream>
using namespace std;

void showValues(int [], int);  // Function prototype

int main() {
    const int ARRAY_SIZE = 8;
    int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};

    showValues(numbers, ARRAY_SIZE);
    return 0;
}

void showValues(int nums[], int size) {
    for (int index = 0; index < size; index++)
        cout << nums[index] << " ";
    cout << endl;
}
```

Output:
5 10 15 20 25 30 35 40

In-class practice: Array Rotation

- Write a function `Rotate` that rotates an array of size `n` by `d` elements to the left
- Use array as argument
- In the `main` function, call the function `Rotate` and show the rotated array
- Test your code

```
For example:
Input: [1 2 3 4 5 6 7], n = 7, d = 2
Output: [3 4 5 6 7 1 2]
```


Reference code: ArrayRotation.cpp https://www.geeksforgeeks.org/array-rotation/

Returning a Value from a Function

- The `return` Statement
  - Used to return a value from a function or end execution of a function
  - Can be placed anywhere in a function
    - Statements that follow the `return` statement will not be executed
  - Can be used to prevent abnormal termination of program
  - In a `void` function without a `return` statement, the function ends at its last `}`

End Function Execution Using `return`

```
//*************************************************************
// Definition of function divide.                             *
// Uses two parameters: arg1 and arg2. The function divides arg1*
// by arg2 and shows the result. If arg2 is zero, however, the  *
// function returns.                                          *
//*************************************************************

void divide(double arg1, double arg2)
{
   if (arg2 == 0.0)
   {
      cout << "Sorry, I cannot divide by zero.\n";
      return;
   }
   cout << "The quotient is " << (arg1 / arg2) << endl;
}
```

Returning a Value From a Function

- In a value-returning function, the `return` statement can be used to return a value from function to the point of call. Example:

Return Type →

```
int sum(int num1, int num2)
{
    double result;
    result = num1 + num2;
    return result;
}
```

Value Being Returned ←

In-class practice

- Define a function that takes an integer `n` as input and return whether it is a power of 4 or not
- In the `main` function, ask the user to input an integer; call this function to show the result
- Example :

```
Input: 16
Output: 16 is a power of 4

Input: 20
Output: 20 is not a power of 4
```

Reference code: PowerOf4.cpp

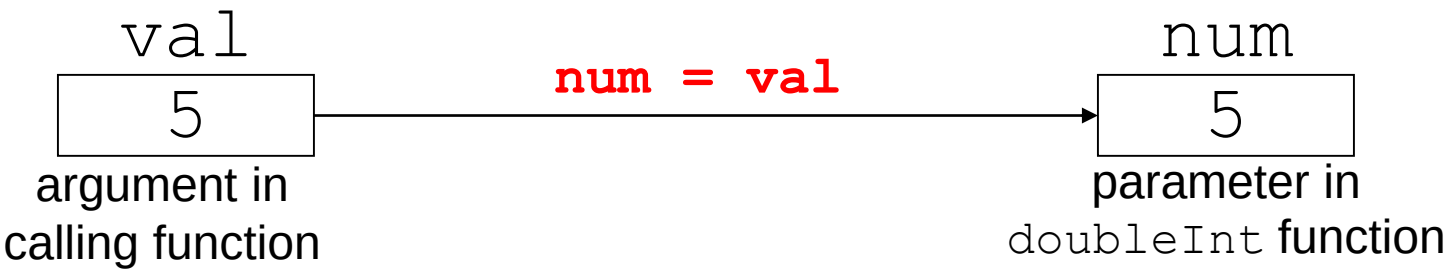
https://www.geeksforgeeks.org/find-whether-a-given-number-is-a-power-of-4-or-not/

Using Reference Variables as Parameters

- Passing data into a function (**passing by value**)
  - The argument value is **copied** into the parameter (`num = val`)
  - Value change of the parameter (`num`) does **not** affect the argument (`val`) because they are two **independent** variables

```
int val=5;
doubleInt(val);
```

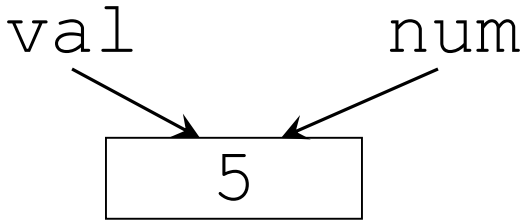
```
void doubleInt(int num){
    num *= 2;//num = num*2
}
```



Q: How to change the argument value synchronously when we change the parameter value?

Passing by Reference

- A reference variable is an **alias** for another variable
  - An alias is another name for an existing variable



If we change `num`, what will happen to `val`? Why?

- Use a reference variable as a function parameter
  - Any changes made to the reference variable are actually performed on the argument variable

Passing by Reference

- How to define?
  - Place an ampersand (&) in front of the parameter name in the function header

```
void doubleInt(int & num){
    num *= 2
}
```

  - In the function prototype, also include the ampersand after the data type

```
void doubleInt(int & num);
```

or

```
void doubleInt(int&);
```

Why Passing by Reference?

- Allow a function to work with the original argument from the function call, not a copy of the argument
- Allow the function to modify values stored in the calling environment
- Provide a way for the function to ‘return’ more than one value

Example

```
#include <iostream>
using namespace std;

void swap (int &first, int &second)
{
    int temp = first;
    first = second;
    second = temp;
}

int main()
{
    int a = 2, b = 3;
    swap(a, b);
    cout << a << " " << b;
    return 0;
}
```

Output:
3 2

https://www.geeksforgeeks.org/references-in-c/

In-class practice

- Ask the user to input three integers
- Define a function to sort these integers in ascending order, using reference variables as parameters
- Test your code

```
Input three integers: 1 3 2
Sorted List: 1 2 3
```

Reference code: Sort3Integers.cpp

This practice refers to Don's Handouts_COSC120_Spring2012.pdf

Local and Global Variables

- Local variables
  - Defined within a block
  - Do not exist outside the function in which they are declared, i.e. they can not be accessed or used outside that function
  - Other functions may have separate, distinct variables with the same name

```
void func() {
    // age is a local variable and cannot be
    // accessed outside this function
    int age = 18;
    cout << age;
}
```


https://www.geeksforgeeks.org/scope-of-variables-in-c/

Global Variables

- Are usually declared **outside** of all of the functions, at the **top** of the program
- Can be accessed from **any part** of the program
- Are available **throughout the lifetime** of a program

```
int g = 10; // Global variable

void func1() {
    g = 20;
    cout << g << endl;
}

int main() {
    func1();
    g = 30;
    cout << g << endl;
    return 0;
}
```

Output:
20
30

https://www.geeksforgeeks.org/scope-of-variables-in-c/

Local Variables VS Global Variables

Local Variables	Global Variables
• Declares within a function	• Declared outside any function
• Accessible only within the function it is declared	• Accessible by all functions in the program
• Created when the function starts executing and is destroyed when control leave the function	• Are destroyed when the program is terminated
• More reliable and secure (the value can't be changed by other functions)	• Are used when values are to be shared among different functions

https://www.slideshare.net/03062679929/functions-in-c-ppt
https://pediaa.com/difference-between-local-and-global-variable/

Lifetime of Variables

```
#include <iostream>
using namespace std;

int num = 0;

void func() {
    int num = 2;
    cout << "num defined in func: " << num << "\n";
}

int main() {
    cout << "Global variable num: " << num << "\n";
    int num = 1;
    cout << "num defined in main: " << num << "\n";
    func();
    return 0;
}
```

Global variable

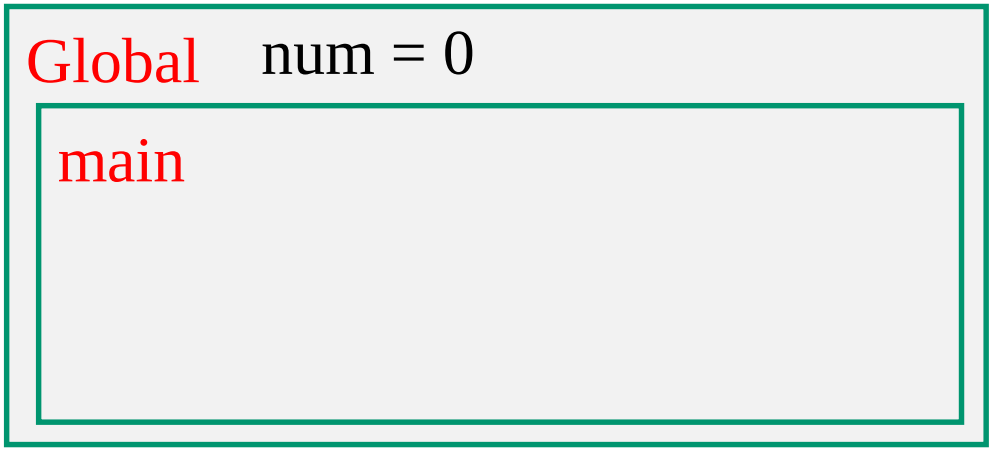
Local variable

Local variable

Output:
Global variable num: 0
num defined in main: 1
num defined in func: 2

Note: Local variables → smaller scope → larger priority
Global variables → larger scope → smaller priority

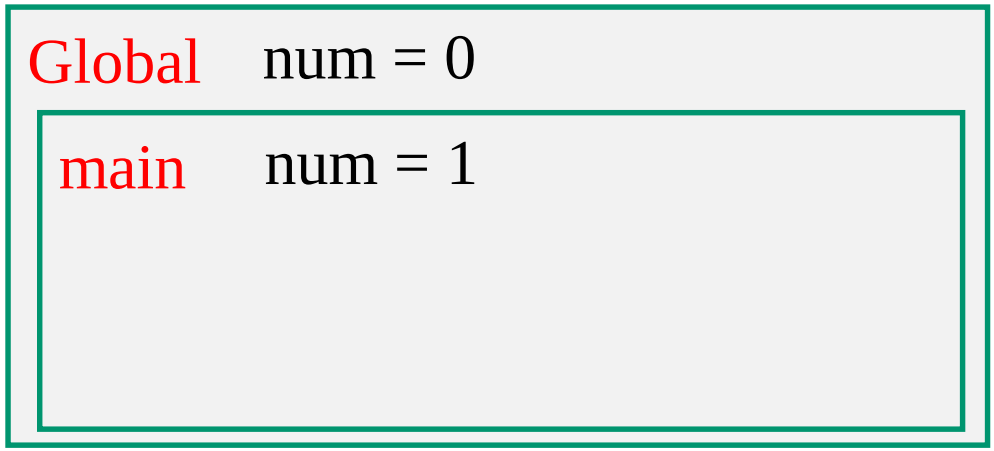
Lifetime of Variables




```
#include <iostream>
using namespace std;

int num = 0;

void func() {
     int num = 2;
     cout << "num defined in func: " << num << "\n";
}

int main() {
     cout << "Global variable num: " << num << "\n";
     int num = 1;
     cout << "num defined in main: " << num << "\n";
     func();
     return 0;
}
```

Here →

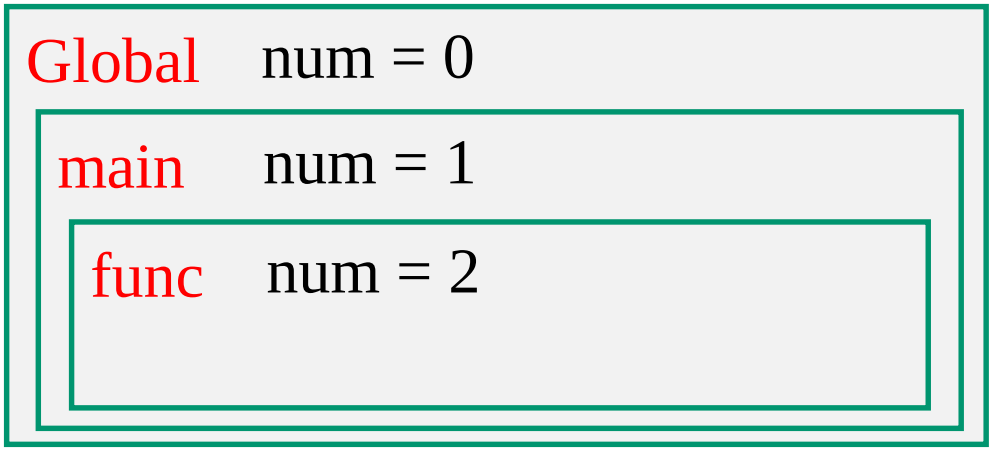
Lifetime of Variables




```
#include <iostream>
using namespace std;

int num = 0;

void func() {
     int num = 2;
     cout << "num defined in func: " << num << "\n";
}

int main() {
     cout << "Global variable num: " << num << "\n";
     int num = 1;
     cout << "num defined in main: " << num << "\n";
     func();
     return 0;
}
```

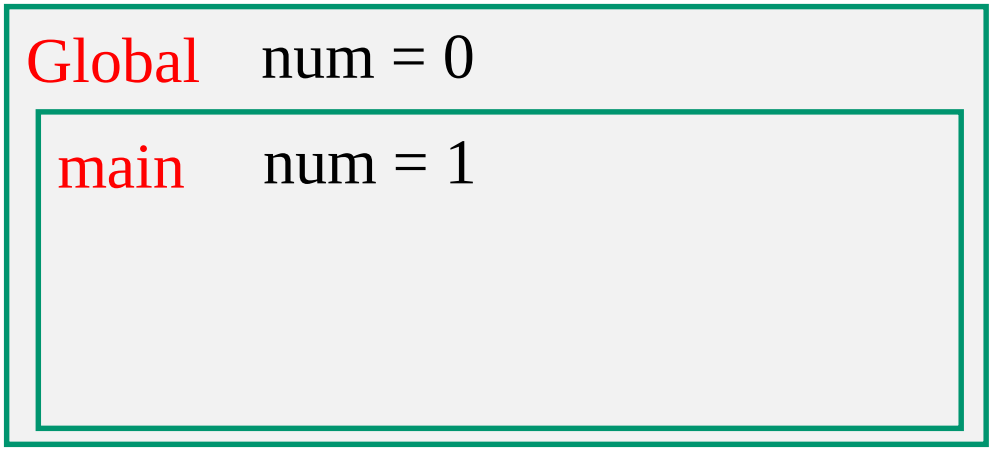
Here →

Lifetime of Variables




```
#include <iostream>
using namespace std;

int num = 0;

void func() {
Here ⟶  int num = 2;
     cout << "num defined in func: " << num << "\n";
}

int main() {
     cout << "Global variable num: " << num << "\n";
     int num = 1;
     cout << "num defined in main: " << num << "\n";
     func();
     return 0;
}
```

Lifetime of Variables




```
#include <iostream>
using namespace std;

int num = 0;

void func() {
    int num = 2;
    cout << "num defined in func: " << num << "\n";
}

int main() {
    cout << "Global variable num: " << num << "\n";
    int num = 1;
    cout << "num defined in main: " << num << "\n";
    func();
    return 0;
}
```

Here →

Lifetime of Variables

```
#include <iostream>
using namespace std;

int num = 0;

void func() {
    int num = 2;
    cout << "num defined in func: " << num << "\n";
}

int main() {
    cout << "Global variable num: " << num << "\n";
    int num = 1;
    cout << "num defined in main: " << num << "\n";
    func();
    return 0;
}
```

Here ⟶

Question: How can C++ distinguish these variables with same names?

Static Local Variables

- How can a program “remember” what value is stored in a local variable?
  - `static` local variables retain their contents between function calls.
  - `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.

Example

```
#include <iostream>
using namespace std;

void showLocal();  // Function prototype

int main() {
    showLocal();
    showLocal();
    return 0;
}

void showLocal() {
    static int localNum = 0; // Static local variable
    cout << localNum << endl;
    localNum += 5;
}
```

The default initialization value is 0, which is equivalent to:

```
static int localNum = 0;
```

Output:

0

5

Thinking: Is static local variable a good design?

In-class practice

- What is the output of the following program?

```
int globvar;

void fct1() {
    int i = 5;
    cout << globvar << " " << i << endl;
}

void fct2() {
    int i;
    i = 17;
    globvar = 3;
}

int main() {
    globvar = 25;
    fct1();
    fct2();
    fct1();
    return 0;
}
```

Output:
25 5
3 5

This practice refers to Don's Handouts_COSC120_Spring2012.pdf

Default Arguments

- A default argument is a constant that is passed **automatically** to a parameter if the argument is missing on the function call.
- Usually listed in the function prototype. E.g.:

```
void showArea(double = 20.0, double = 10.0);
```

or

```
void showArea(double length = 20.0, double width = 10.0);
```

```
void showArea(double length, double width) {
    double area = length * width;
    cout << "The area is " << area << endl;
}
```

```
showArea();              The area is 200
showArea(12.0);          The area is 120
showArea(12.0, 5.5);     The area is 66
```

Default Arguments

- Default arguments can be declared in function header if no prototype

```
void showArea(double length = 20.0, double width = 10.0)
{
    double area = length * width;
    cout << "The area is " << area << endl;
}
```

- Multi-parameter functions may have default arguments for some or all of them:
  - The defaultless ones are declared first in the parameter list

```
int getSum(int, int=0, int=0);
```

Two-Dimensional Arrays

- A 2-D array is an array of 1-D arrays
- Use two size declarators in definition:
  - First declarator is number of rows; second is number of columns

```
const int ROWS = 4, COLS = 3;
int exams[ROWS][COLS];
```

columns

rows

exams[0][0]	exams[0][1]	exams[0][2]
exams[1][0]	exams[1][1]	exams[1][2]
exams[2][0]	exams[2][1]	exams[2][2]
exams[3][0]	exams[3][1]	exams[3][2]

- Use two subscripts to access element:

```
exams[2][2] = 86;
```

2D Array Initialization

- Two-dimensional arrays are initialized row-by-row:

```
const int ROWS = 2, COLS = 2;
int exams[ROWS][COLS] = {{84, 78}, {92, 97}};
```

84	78
92	97

- Some array elements without initial values will be set to `0` or `NULL`

```
int exams[ROWS][COLS] = {{84}, {92, 97}};
```

exams[0][1] is automatically set to 0

Passing Two-Dimensional Array to Function

- When a 2-D array is passed to a function, the parameter type must contain a size declarator for the columns
  - The size declarator for rows is optional (use empty [ ])

```
const int COLS = 2;
// Prototype
void getExams(int [][COLS], int);

// Header
void getExams(int exams[][COLS], int rows)
```

Here COLS is a global constant

- Use array name as argument in function call:

```
getExams(exams, 2);
```

Use Nested Loop to Step through 2D Array

- What is the output of the following program?

```
#include <iostream>
using namespace std;

void readArray(int n[][2], int row) {
}
int sumOfArray(int n[][2], int row) {
    int total = 0;
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < 2; j++) {
            total += n[i][j];
        }
    }
    return total;
}

int main() {
    int num[3][2] = {{3, 4}, {9, 5}, {7, 1}};
    readArray(num,3);
    cout << "The sum is: " << sumOfArray(num, 3);
    return 0;
```

Output:
The sum is: 29

Overloading Functions

- Sometimes we want to create functions that perform the same task but take different parameter types or different number of parameters

```
float absolute(float var){ //float type parameter
    if (var < 0.0)
        var = -var;
    return var;
}

int absolute(int var) {  //int type parameter
     if (var < 0)
         var = -var;
    return var;
}
```


https://www.programiz.com/cpp-programming/function-overloading

Overloading Functions

- Overloaded functions have the same name but different parameter lists
  - Different number of parameters

```
int sum(int num1, int num2);
int sum(int num1, int num2, int num3);
int sum(int num1, int num2, int num3, int num4);
```

  - Different data types of parameters

```
float absolute(float var);
int absolute(int var);
```

  - Do **NOT** consider different return types
- Compiler will determine which version of function to call by argument and parameter lists

Example

```
#include <iostream>
using namespace std;

int plusFunc(int x, int y) {
  return x + y;
}

double plusFunc(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

Output:
Int: 13
Double: 10.56

https://www.w3schools.com/cpp/cpp_function_overloading.asp

In-class practice

- Define 3 overloading functions
  - have the same name `Average`
  - have 2, 3, and 4 integer parameters, respectively
  - return the average of input arguments
  - the return type is double
- In the main function, call `Average` functions to calculate and show the average of
  - 2 and 3
  - 2, 3, and 5
  - 2, 3, 5, and 8

Reference code: OverloadAverage.cpp https://www.bestprog.net/en/2018/08/07/...

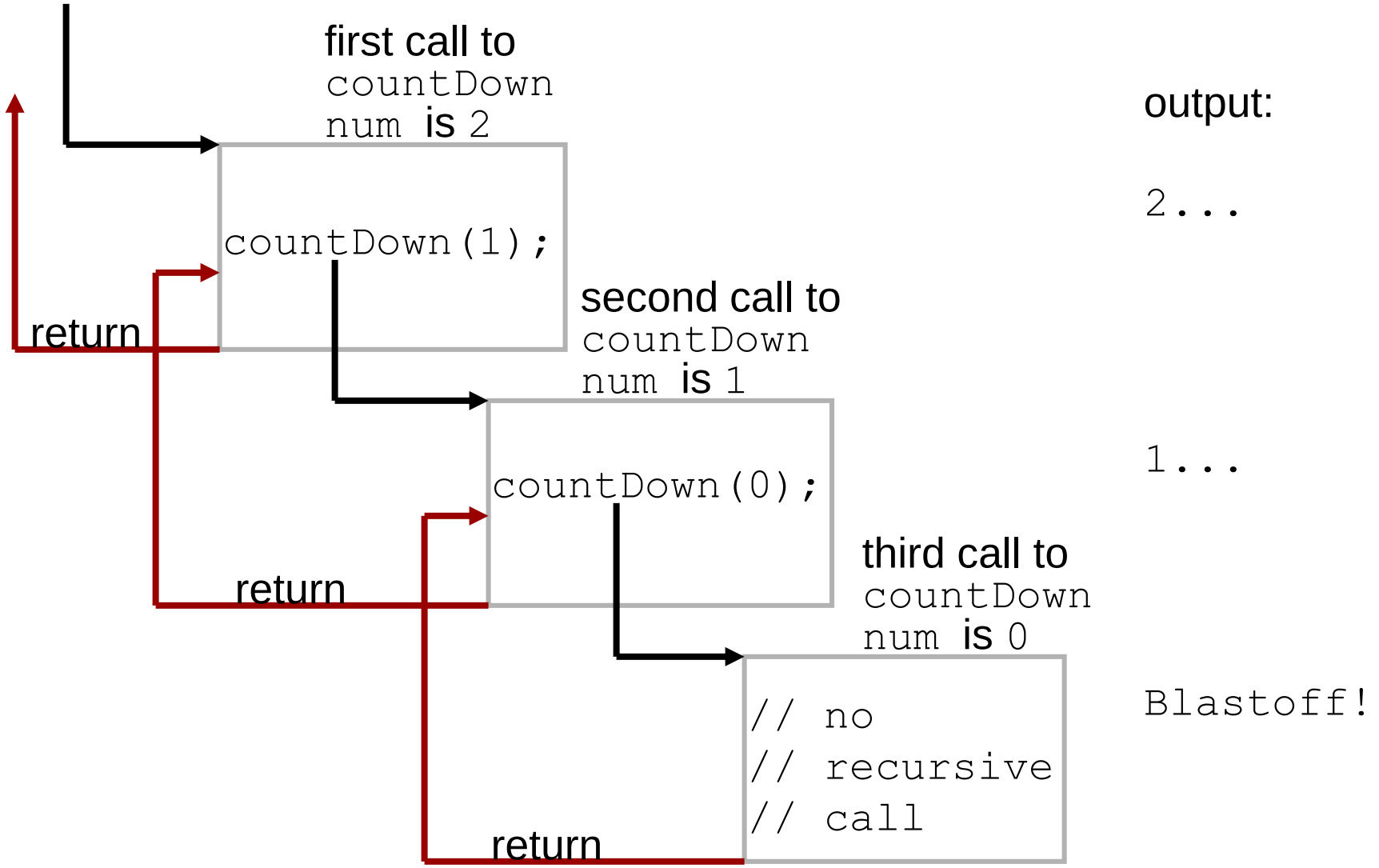
9. Recursion Function

- A recursive function is one that **calls itself**

```
void countDown(int num){
    if (num == 0)                    // stop condition
        cout << "Blastoff!";
    else{
        cout << num << "...\n";
        countDown(num-1);            // recursive call
    }
}
```

- Assume the input argument is 2:
  - `countDown(2)` outputs `2...`, then it calls `countDown(1)`
  - `countDown(1)` outputs `1...`, then it calls `countDown(0)`
  - `countDown(0)` outputs `Blastoff!`, then returns to `countDown(1)`
  - `countDown(1)` returns to `countDown(2)`
  - `countDown(2)` returns to the calling function

What Happens When Called?



Solving Problems with Recursion

- Two important steps:
  - Define the recursive function
  - Define the stop condition

- Example: factorial calculation

  `n! = n*(n-1)*(n-2)*...*3*2*1` if `n > 0`

  `n! = 1` if `n = 0`

  - Define the recursive function:

    `n! = n * (n-1)!`

  - Define the stop condition:

    `0! = 1` (base case)

Recursive Factorial Function

```
#include <iostream>
using namespace std;

int factorial(int);    // Function prototype

int main(){
    int number;
    cout << "Enter an integer value to display its factorial: ";
    cin >> number;

    cout << "The factorial of " << number << " is " << factorial(number);
    return 0;
}

int factorial(int n){
    if (n == 0)
        return 1;                       // Base case
    else
        return n * factorial(n - 1); // Recursive case
}
```

```
Enter an integer value to display its factorial: 5
The factorial of 5 is 120
```

Example code: Factorial.cpp

In-class practice

- The Fibonacci numbers are the numbers in the following integer sequence

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …`

- In mathematical terms, the sequence `Fn` of Fibonacci numbers is defined as

  `Fn = Fn-1 + Fn-2`

  where:

  `F0 = 0` and `F1 = 1`

- Use recursive function to calculate and display the first `10` Fibonacci numbers
- Test your code

 Reference code: Pr20-6.cpp

Thanks

Reading textbook

- Chapter 6, 7, 8, 20

Reference

- The teaching materials of this course refer to:
  - Professor Xiaohong (Sophie) Wang. COSC 120 teaching materials
    - Salisbury University
  - Textbook:
    - Starting Out with C++: From Control Structures through Objects, by Tony Gaddis, Pearson (9th Edition)
    - Instructor materials of the above textbook (All rights reserved)