FUNCTIONS AND AN INTRODUCTION TO RECURSION

Chapter 6, Part 2

TOPICS

More on Scope

- Global Variables
- Unary Scope Resolution Operator

Math Library Functions

Case Study: Random Number Generation

Introduction to Recursion

More on Scope

TYPES OF SCOPE

Block

Function (chapter 23)

Global namespace

Function-prototype (chapter 23)

Class (chapter 9)

Namespace (chapter 23)

Block Scope

Begins at declaration and ends at closing curly brace

- Local variables, function parameters

Block

- E.g., a loop within a function, or a nested loop

Hiding / Shadowing

- A variable used in an inner scope with the same name as a variable in outer scope
- Not a syntax error, but usually a logic error
- The inner variable hides the outer variable

Once a variable goes out of scope, its value is lost

Local Variables May Be Declared Static

Static variables retain their values even when function ends

Numeric types initialized to 0 by default

Scope <> duration

Examples:

```
static unsigned int count{1};        //count initialized to 1

static unsigned int count;           //count initialized to 0
```

GLOBAL NAMESPACE SCOPE

Identifier declared **outside of a function or class**

Known in all functions from the point of declaration to the end of file

Examples:

- Function definitions
- Function prototypes outside a function
- Class definitions
- Global variables

GLOBAL VARIABLES

Declared outside of a class or function definition

Should be avoided

Violates principle of least privilege

SCOPE DEMONSTRATION

Fig06_11.cpp

Line	Variable	Scope	Result
10	declares x	global	global x is created
15	declares x	local	local x created, hides global x declared on line 10
20	declares x	local	local x created, hides local x declared on line 16
39	declares x	local	Local x created & reinitialized every time useLocal is called
50	declares x	static local	Local x reused & NOT reinitialized every time useStaticLocal is called
61-63			Refers to global x

Unary Scope Resolution Operator

::

Use to access a global variable when a local variable of the same name is in scope

Cannot be used to access a local variable of the same name in an outer block

Fig06_19.cpp

Good practice to always use :: to access a global variable, even if it has a unique name

MATH LIBRARY FUNCTIONS

MATH LIBRARY FUNCTIONS IN CMATH HEADER

<cmath> header contains helpful math-related functions

These are **global functions**

- Not members of a class
- Do not need an object to be created first

Example: **sqrt()** function returns square root of a number

See other <cmath> functions p. 214-15

mathLibDemo.cpp

CASE STUDY: RANDOM NUMBER GENERATION

STANDARD LIBRARY FUNCTION RAND()

Generates **random unsigned int** between 0 and **RAND_MAX**

RAND_MAX defined in **<cstdlib>**

- Value may differ on different systems

printRandMax.cpp

ROLLING A SIX-SIDED DIE – SCALING

fig06_06.cpp simulates rolling a die 20 times

Scaling

- Limiting the result of random number generation to a certain number of consecutive possibilities, starting at 0
- Use the **mod operator %**

Line 12 – the 6 is **scaling factor**

- Produces 6 possible numbers, 0 through 5, randomly

Rolling a Six-Sided Die – Shifting

But a die uses the numbers 1 through 6

Shifting

- Adding to the result of the random number generation to start the possibilities at a certain value

Line 12 – the 1 is **shifting factor**

- Produces 6 possible numbers, 1 through 6, randomly

ROLLING A SIX-SIDED DIE

fig06_07.cpp simulates rolling a die 60,000,000 times

Line 18 – note the C++14 **digit separators**

- Makes longer literals more readable

In Dev-cpp:

- Add compiler option **-std=c++14**

RANDOMIZING THE RANDOM NUMBER GENERATOR

If we run fig06_06.cpp or fig06_07.cpp again and again…

rand() produces **pseudorandom** numbers

- Sequence of numbers that will repeat

srand()

- **Seeds** the rand() function to produce a different sequence each time
- Must be called before rand()

SEEDING WITH SRAND() EXAMPLES

fig06_08.cpp

Demonstrates different values for the seed:

- an unsigned int entered by user
- current time

Function **time(0)**

- Produces # of seconds since 1/1/70 at midnight GMT
- Data type **time_t**
- Cast to unsigned int for use with srand()

GENERALIZED SCALING & SHIFTING OF RANDOM NUMBERS

General format:

random number = shiftingValue + rand() % scalingFactor;

Shifting value

- First number in desired set of consecutive numbers

Scaling factor

- Total number of values in set of consecutive numbers

RANDOM NUMBERS WITH <RANDOM>

Function rand() may not be as statistically unpredictable as we'd like

<random> header

- Provides more secure, randomer random numbers

Engine

- Random number generation algorithm
- Default engine is **default_random_engine**

Distribution

- Controls the range, type and statistical properties of values produced
- **uniform_int_distribution** evenly distributes pseudorandom numbers over a specified range of values

RANDOM NUMBERS WITH <RANDOM> EXAMPLE

fig06_10.cpp rolls 6-sided die again

Creates a **default_random_engine** object

- Line 13, variable name is engine
- Constructor argument is the seed – if no seed passed, engine produces same sequence of random ints

Creates **uniform_int_distribution** object

- Line 14, variable name is randomInt
- Constructor arguments indicate range of random numbers to create
- Also an example of a class template

Returns a random unsigned int between 1 and 6 (line 19)

INTRODUCTION TO RECURSION

REVIEW – CALL STACK

What happens on the call stack when these statements execute?

1. main calls function 1

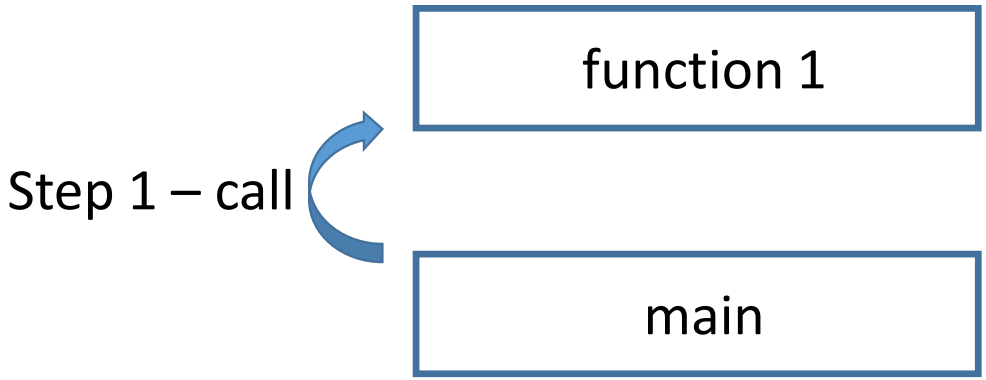
2. function 1 calls function 2

3. function 2 ends

4. function 1 ends

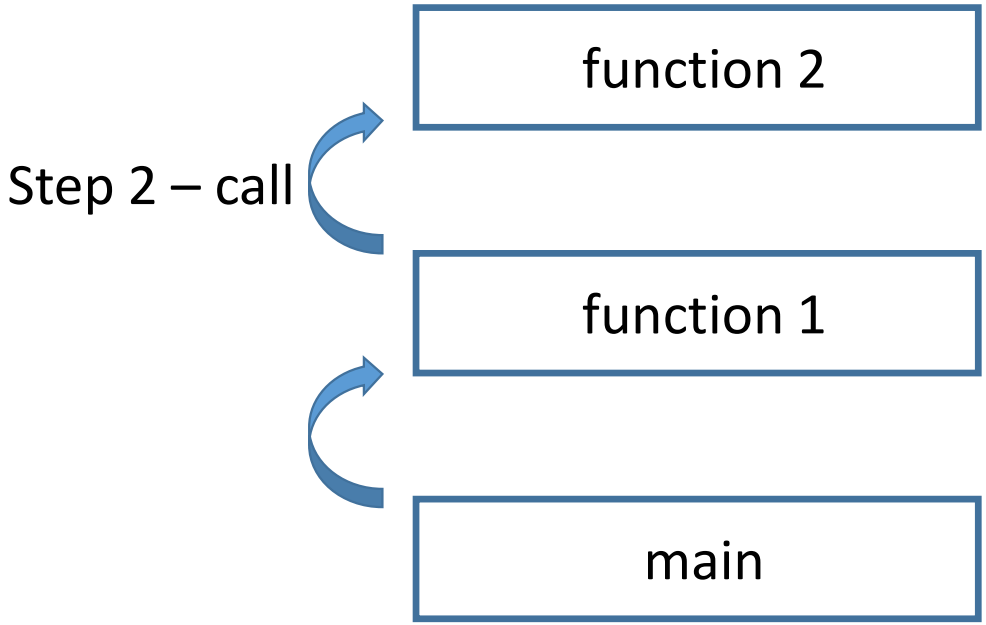
REVIEW – CALL STACK – STEP 1

Example – main calls function 1:



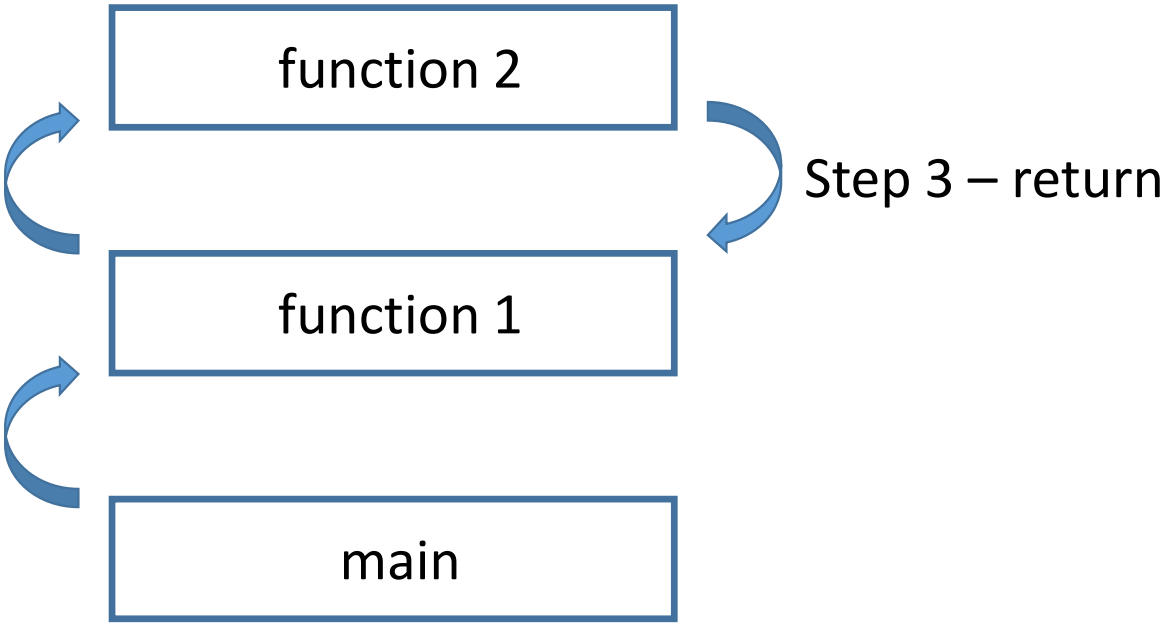
REVIEW – CALL STACK – STEP 2

Then function 1 calls function 2:



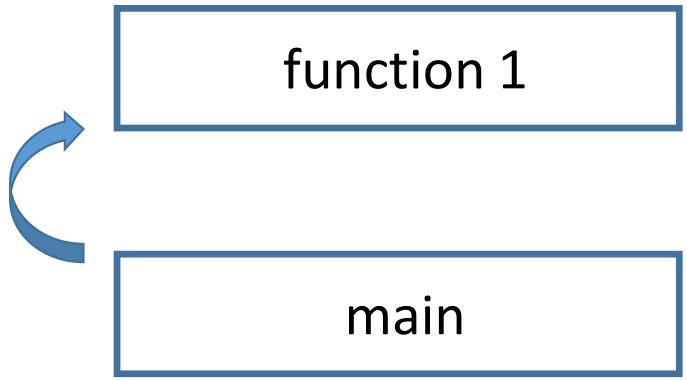
REVIEW – CALL STACK – STEP 3

function 2 ends and returns control to function 1:



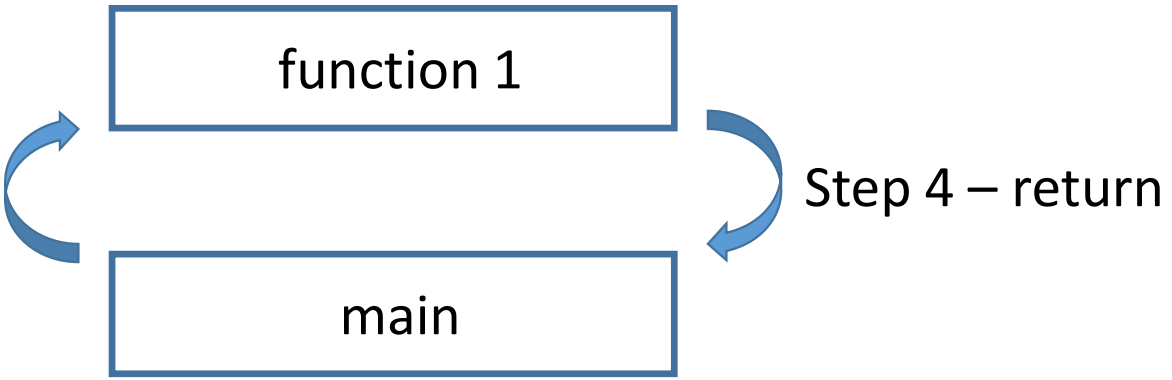
REVIEW – CALL STACK – FUNCTION 2 IS GONE

All memory space associated with function 2 is gone:



REVIEW – CALL STACK – STEP 4

Then function 1 ends and returns control to main:

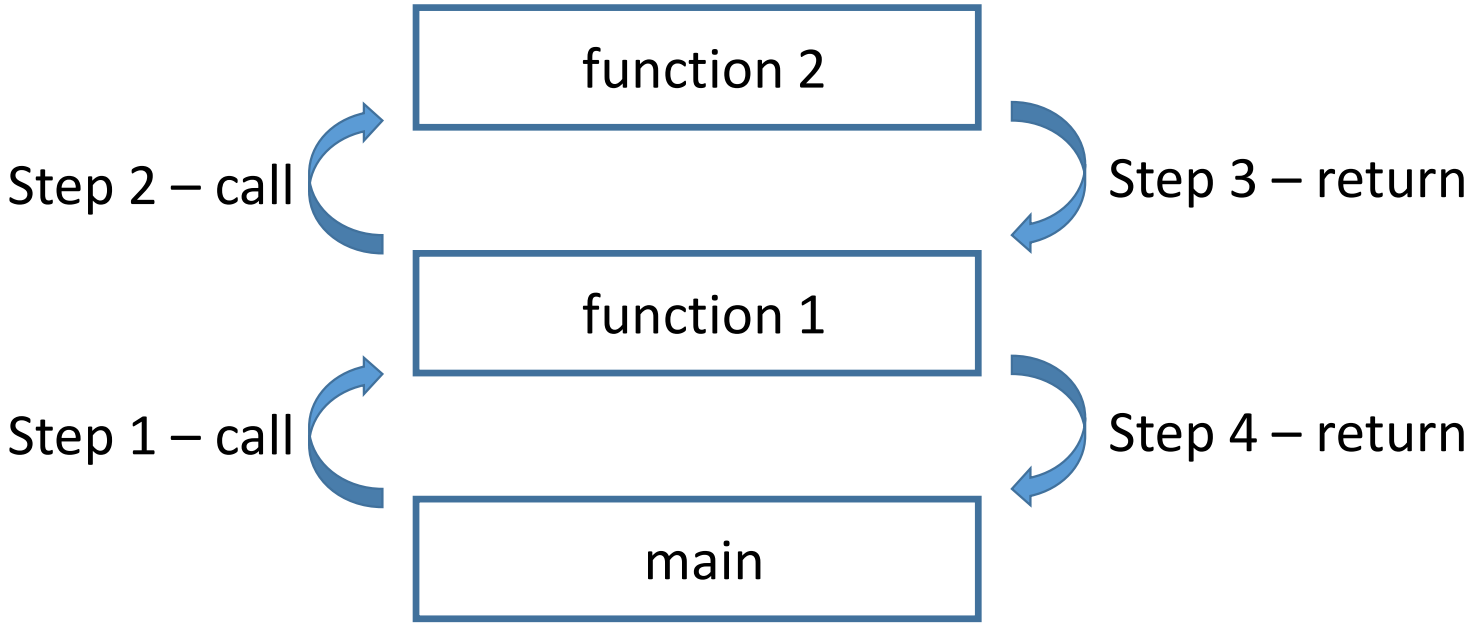


REVIEW – CALL STACK – FUNCTION 1 IS GONE

All memory space associated with function 1 is gone:

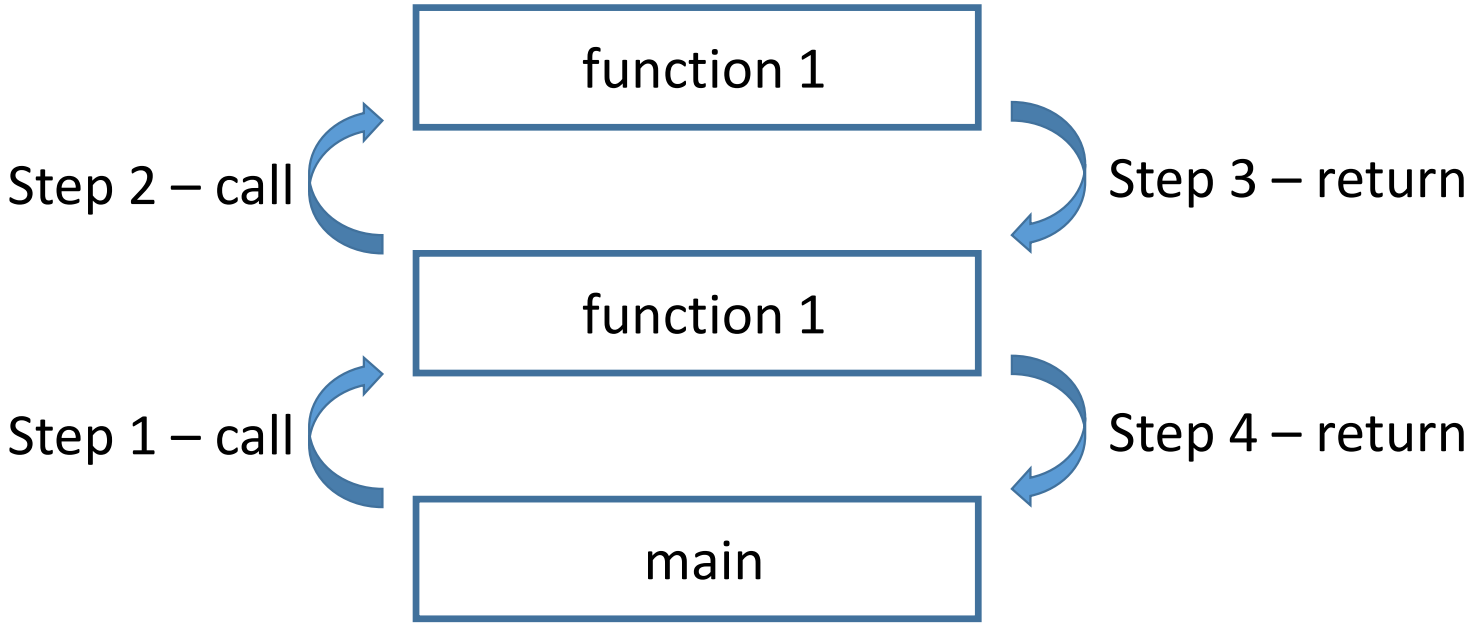
REVIEW – CALL STACK – SUMMARY

Summary of steps:



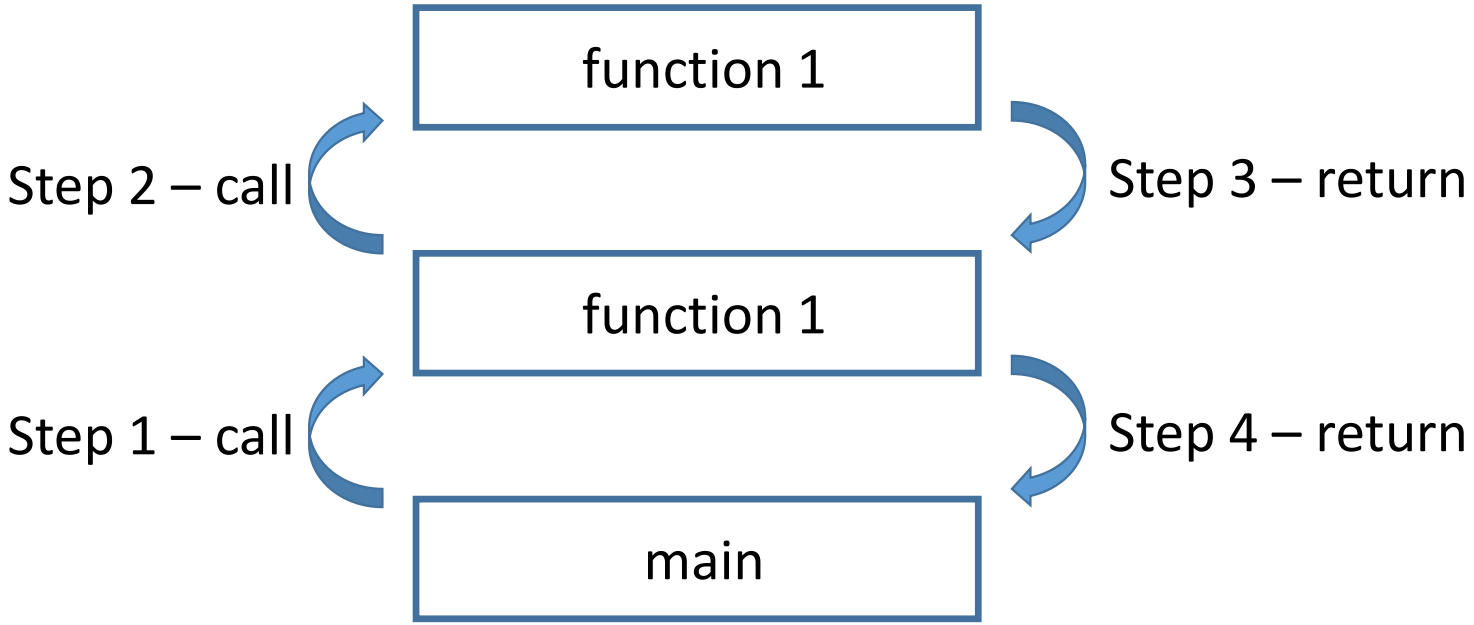
REVIEW – CALL STACK – USE THE SAME FUNCTION

This example used different functions, but we could just as easily have used the same function:



Review – Call Stack – Separate Activation Records

This is possible because **each activation record is separate**, and has no relation to any other AR, except via the call and return.



THE NATURE OF RECURSION

A recursive function calls a function too

The function it calls is a **copy of itself**

A recursive function may also return a value to the code that called it.

- Which was a copy of itself…

Recursion can be used as an alternative to looping

EXAMPLE 1 – FACTORIAL

The factorial of an integer is the product of the integer and all the integers below it (up to and including 1, but not 0)

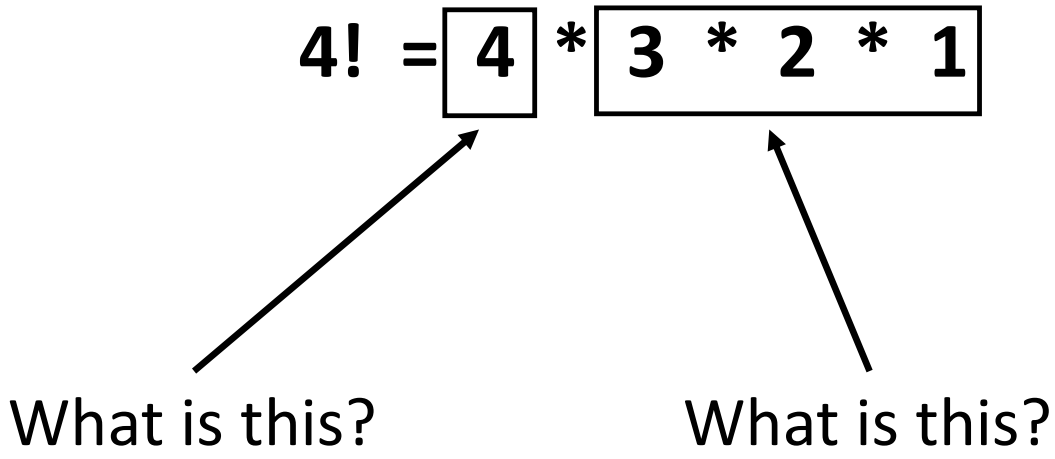
Why do factorials stop at 1, and not 0?

Example: factorial of four is equal to 24:

4! = 4 * 3 * 2 * 1

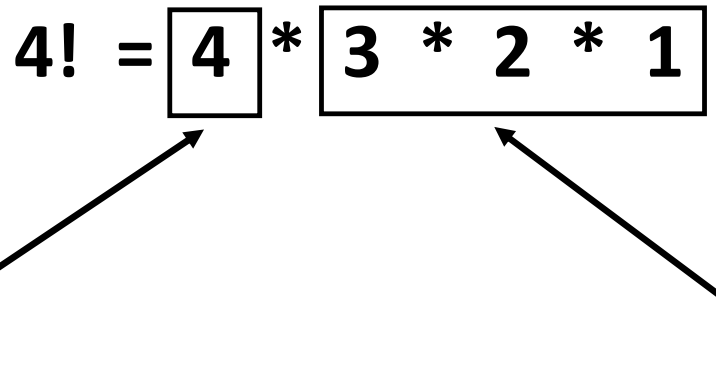
EXAMPLE 1 – FACTORIAL OF 4

There are two parts this problem:



EXAMPLE – FACTORIAL OF 4

There are two parts this problem:

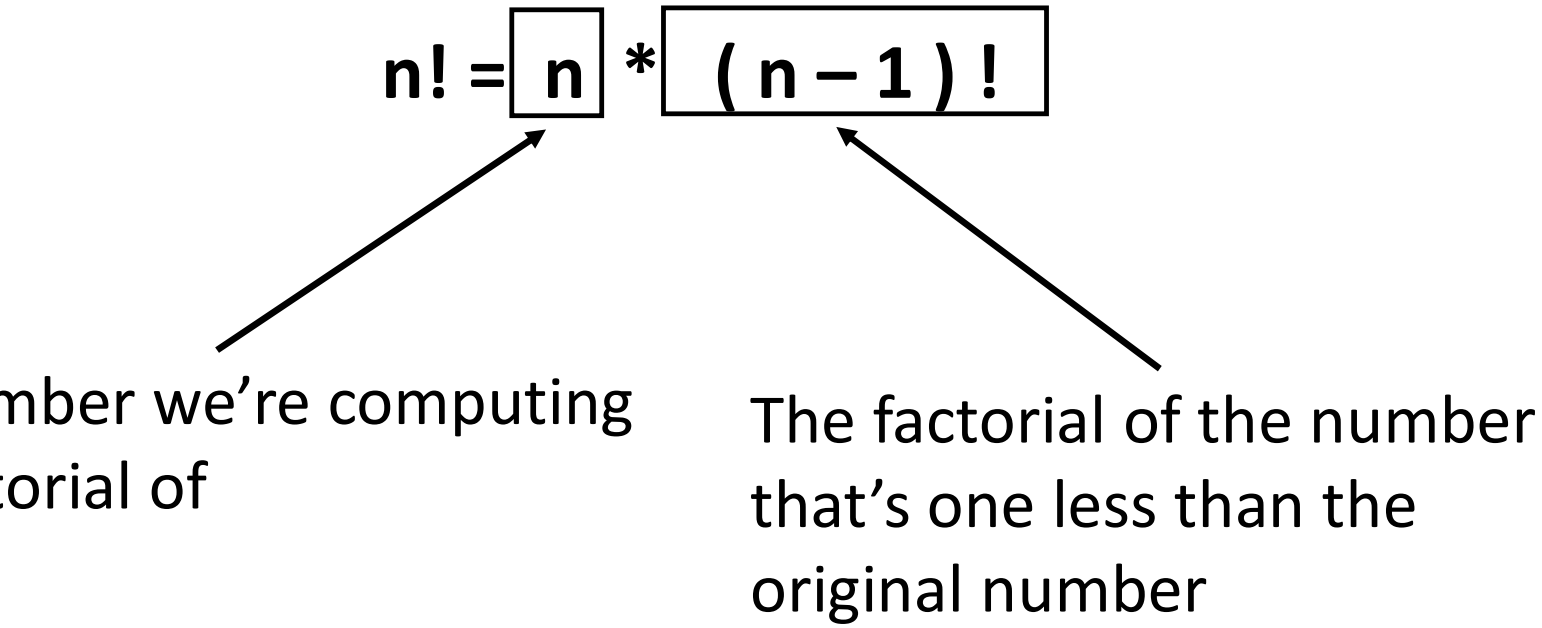


The number we're computing the factorial of

The factorial of the number that's one less than the original number

EXAMPLE – FACTORIAL OF N

If we replace 4 with n:



EXAMPLE – FACTORIAL OF AN INTEGER N

Factorial of any number n is:

factorial (n) = n * factorial (n – 1)

So, factorial (4) is:

factorial (4) = 4 * factorial (3)

In order to solve this, we need to compute the factorial of (3)

EXAMPLE – FACTORIAL OF 3

factorial (3) is:

factorial (3) = 3 * factorial (2)

In order to solve this, we need to know the factorial of (2)…

EXAMPLE– FACTORIAL OF 2

factorial (2) is:

factorial (2) = 2 * factorial (1)

In order to solve this, we need to know the factorial of (1)…

EXAMPLE – FACTORIAL OF 1

factorial (1) has been defined:

factorial (1) = 1

Now we have one of the answers we need to solve the whole problem.

So we work backwards…

EXAMPLE – FACTORIAL OF 2

factorial (2) is:

factorial (2) = 2 * factorial (1)

factorial (2) = 2 * 1

factorial (2) = 2

EXAMPLE – FACTORIAL OF 3 RESULT

factorial (3) is:

factorial (3) = 3 * factorial (2)

factorial (3) = 3 * 2

factorial (3) = 6

EXAMPLE – FACTORIAL OF 4 RESULT

factorial (4) is:

factorial (4) = 4 * factorial (3)

factorial (4) = 4 * 6

factorial (4) = 24

Factorial Example Code

Fig06_25.cpp – recursive factorial

Recursion is very memory-intensive due to all the function calls

Fig06_28.cpp – iterative factorial

Recursion is Repetition

Remember that a loop must have:

1. A variable that controls the repetition
2. A condition that checks the LCV
3. A change in the LCV's value

A recursive function has these parts too

Parts of a Recursive Function

Variable that controls the recursion

- No fancy name for this variable

Condition that checks the value of this variable

- For the existence of the base or recursive case

Base case

- This is the stopping point

Recursive case

- Call to the recursive function with an **updated value** for the variable controlling the recursion, if it's not the base case

Locate These Parts in the Factorial Example Code

Fig06_25.cpp – recursive factorial

Variable controlling the recursion – line 18

- Parameter **number**

Condition that checks variable – line 20

- If true, we have the base case, and we stop
- If false, we have the recursive case, and need to continue

Call to recursive function – line 24

- Uses an updated value of **number**

Class Demos – Printing Numbers

Print the numbers 1 through 10, ascending

- Step 1 – Set up a regular loop, and identify its components
- Step 2 – Write a recursive version using the same components

Print the numbers 10 through 1, descending

- Step 1 – Set up a regular loop, and identify its components
- Step 2 – Write a recursive version using the same components

Print the even numbers from 1 to 50

- Step 1 – Set up a regular loop, and identify its components
- Step 2 – Write a recursive version using the same components