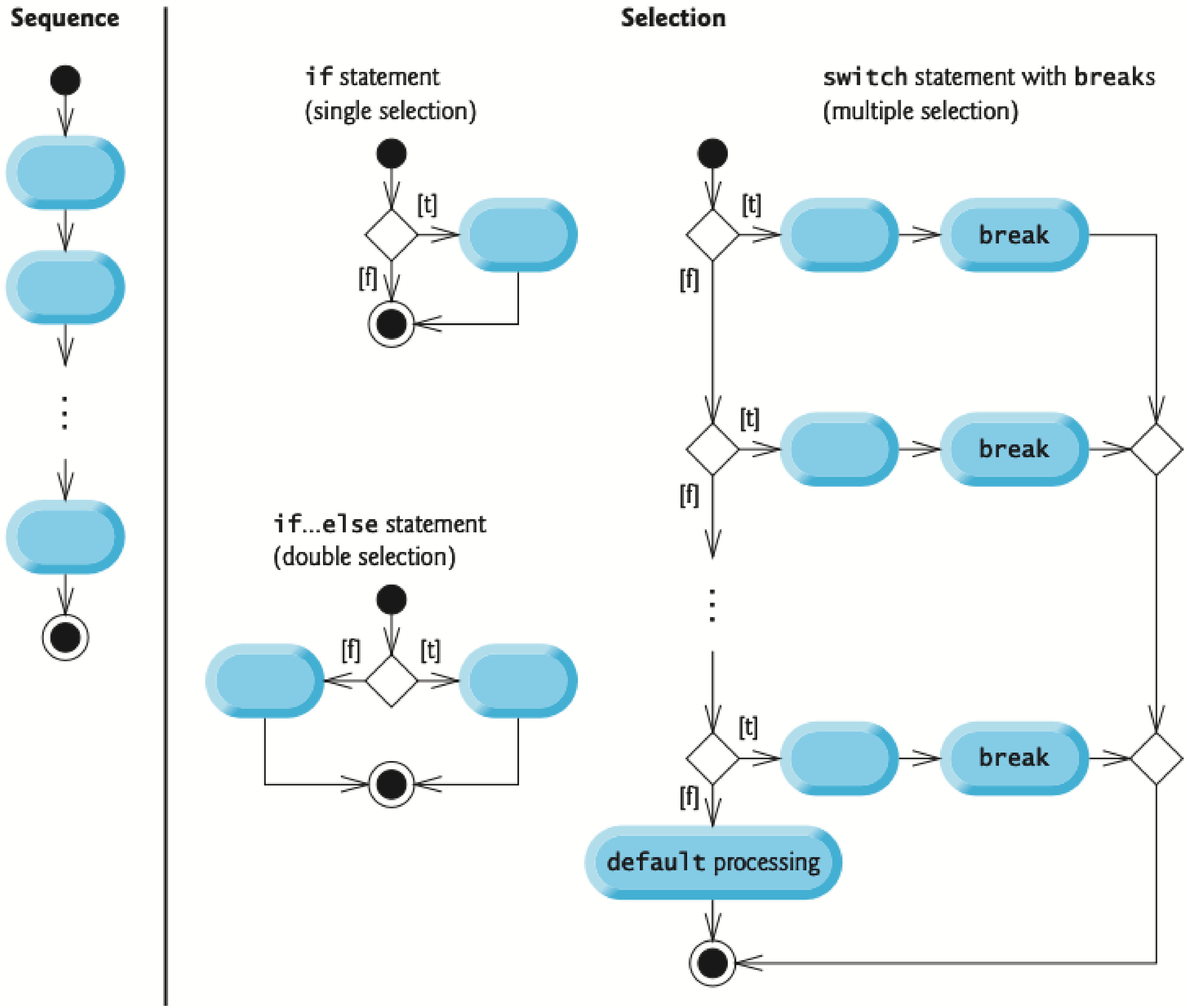
Object-Oriented Programming

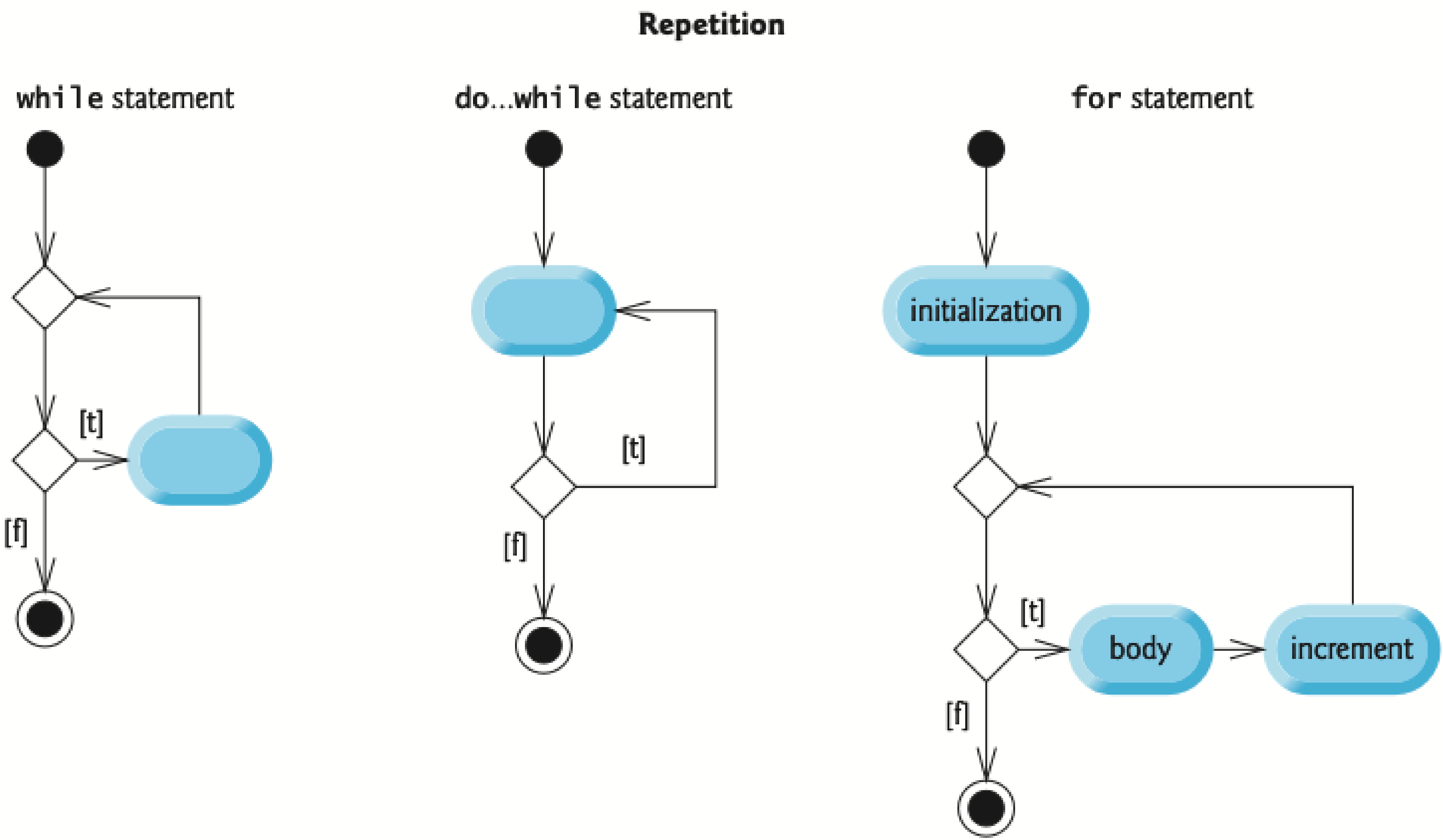
Lecture 5: Functions and Recursion in C++

Objectives

- Summaries of C++ Statement
- Understand modular programming
- Learn function prototypes, math library, and scope rules
- Explore recursion and compare it to iteration
- Lambda Function
- Apply knowledge to real-world scenarios like games and simulations



Sequence
Selection
if statement
(single selection)
[t]
[f]
if...else statement
(double selection)
[f]
[t]
switch statement with breaks
(multiple selection)
[t]
break
[f]
[t]
break
[f]
[t]
break
[f]
default processing


Repetition
while statement
do...while statement
for statement
[t]
[f]
[t]
[f]
initialization
[t]
body
increment
[f]

Switch-Case Example

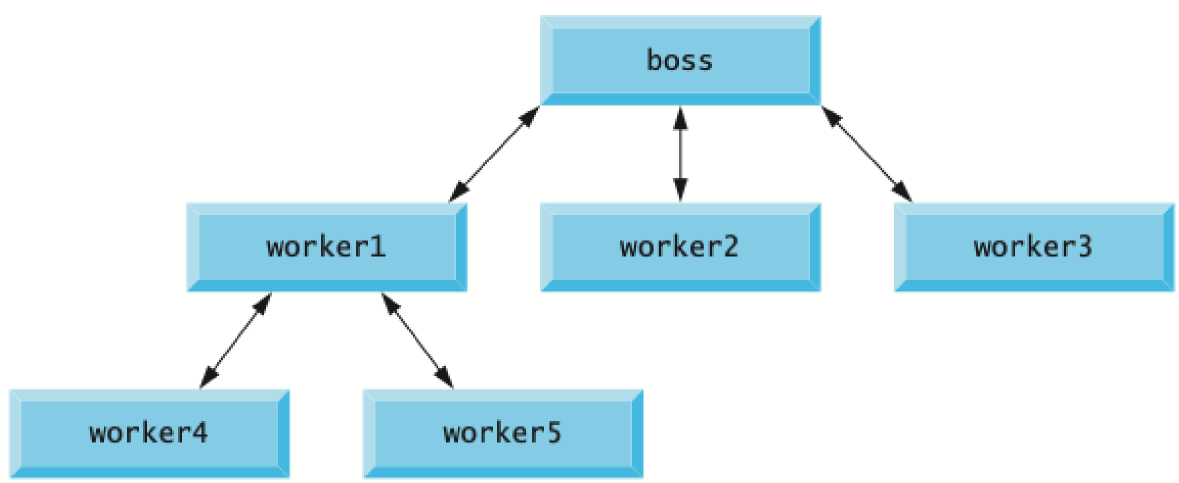
```
#include <iostream>
using namespace std;

int main() {
   int choice;

   cout << "Menu:" << endl;
   cout << "1. Add" << endl;
   cout << "2. Subtract" << endl;
   cout << "3. Multiply" << endl;
   cout << "4. Divide" << endl;
   cout << "Enter your choice: ";
   cin >> choice;

   switch (choice) {
      case 1:
         cout << "You chose to Add." << endl;
         break;
      case 2:
         cout << "You chose to Subtract." << endl;
         break;
      case 3:
         cout << "You chose to Multiply." << endl;
         break;
      case 4:
         cout << "You chose to Divide." << endl;
         break;
      default:
         cout << "Invalid choice. Please select a number between 1 and 4." << endl;
         break;
   }

   return 0;
}
```

What are Functions?

- Definition: A block of code designed to perform a specific task.
- Key Components:
  - Function declaration
  - Function definition
  - Function call

Why Use Functions?

- Benefits:
  - Modularity: Break problems into smaller pieces.
  - Reusability: Use functions across programs.
  - Maintainability: Easier debugging and updates.



Function Prototypes

- A function prototype informs the compiler about the function's name, return type, and parameters before it is used in the code.
- Compiler's Role: The compiler processes code line by line.
  - When a function call is encountered, the compiler must know the function's details (return type, parameter types) to validate the call.
- Function Prototype:
  - Acts as a declaration of the function.
  - If the function is defined after main without a prototype, the compiler does not know about the function at the time it encounters the call. This leads to a compilation error.

lect05_01.cpp

Why Use Prototypes?

- Code Organization: You can place the main logic (main function) at the top and detailed implementations afterward, improving code readability.
- Error Checking: Prototypes ensure that function calls are valid (e.g., correct argument types and order).
- Modularity: Separate prototypes in headers (.h files) allow you to reuse function declarations across multiple files.

Why Use Prototypes?

math_function.h

```cpp
// Function prototype (declaration)
int add(int a, int b);
```

math_functions.cpp

```cpp
#include "math_function.h"

// Function definition (body)
int add(int a, int b) {
    return a + b;
}
```

main.cpp

```cpp
#include <iostream>
#include "math_function.h"

int main() {
    std::cout << "2 + 3 = " << add(2, 3)
<< std::endl;
    return 0;
}
```

Function Prototypes

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

int maximum(int x, int y, int z); // function prototype

int main() {
   cout << "Enter three integer values: ";
   int int1, int2, int3;
   cin >> int1 >> int2 >> int3;

   // invoke maximum
   cout << "The maximum integer value is: "
      << maximum(int1, int2, int3) << endl;
}

// returns the largest of three integers
int maximum(int x, int y, int z) {
   int maximumValue{x}; // assume x is the largest to start

   // determine whether y is greater than maximumValue
   if (y > maximumValue) {
      maximumValue = y; // make y the new maximumValue
   }

   // determine whether z is greater than maximumValue
   if (z > maximumValue) {
      maximumValue = z; // make z the new maximumValue
   }

   return maximumValue;
}
```

function signature

Function overload

int maximum(int, int, int);

int maximum(int, int, int, int);

Argument Coercion

- Compiler automatically converts an argument in a function call to match the type of the corresponding parameter in the function prototype or definition. For example, if a function is defined to take a double, but an integer is passed, the compiler will automatically convert (coerce) the integer to a double.
- **Argument-Promotion Rules:**
  - These rules describe how smaller integer types (like char and short) are automatically promoted to larger integer types (like int) when passed as arguments to a function.
- **Implicit Conversions:**
  - The compiler can perform implicit type conversions to ensure compatibility.

Data Types Promotion Hierarchy

Data types	
`long double`	
`double`	
`float`	
`unsigned long long int`	(synonymous with `unsigned long long`)
`long long int`	(synonymous with `long long`)
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`char` and `signed char`	
`bool`	

Argument Coercion

- Best practices:
  - Use Explicit Casting:
    `printInt(static_cast<int>(num));`
- Prefer Matching Types:
  - Ensure that function arguments and parameters have matching types to avoid unintended type coercion.

Math Library Functions

Function	Description	Example
`ceil(x)`	rounds x to the smallest integer not less than x	`ceil(9.2)` is `10.0` `ceil(-9.8)` is `-9.0`
`cos(x)`	trigonometric cosine of x (x in radians)	`cos(0.0)` is `1.0`
`exp(x)`	exponential function e^x	`exp(1.0)` is `2.718282` `exp(2.0)` is `7.389056`

`fabs(x)`	absolute value of x	`fabs(5.1)` is `5.1`
		`fabs(0.0)` is `0.0`
		`fabs(-8.76)` is `8.76`

`floor(x)`	rounds x to the largest integer not greater than x	`floor(9.2)` is `9.0` `floor(-9.8)` is `-10.0`
`fmod(x, y)`	remainder of x/y as a floating-point number	`fmod(2.6, 1.2)` is `0.2`
`log(x)`	natural logarithm of x (base e)	`log(2.718282)` is `1.0` `log(7.389056)` is `2.0`
`log10(x)`	logarithm of x (base 10)	`log10(10.0)` is `1.0` `log10(100.0)` is `2.0`
`pow(x, y)`	x raised to power y (x^y)	`pow(2, 7)` is `128` `pow(9, .5)` is `3`
`sin(x)`	trigonometric sine of x (x in radians)	`sin(0.0)` is `0`
`sqrt(x)`	square root of x (where x is a nonnegative value)	`sqrt(9.0)` is `3.0`
`tan(x)`	trigonometric tangent of x (x in radians)	`tan(0.0)` is `0`

Random Number Generation

- In C++, random number generation involves producing pseudorandom numbers (not truly random but generated using deterministic algorithms).
- Legacy Approach:
  - Using rand() and srand() from <cstdlib>.
- Modern Approach:
  - Using C++11 <random> header, which offers better randomness and flexibility.

lect05_02.cpp

Random-Number Generation

Quiz 2 – 15 minutes

Write a C++ program to simulate rolling a six-sided die 60,000,000 times and calculate the frequency of each face appearing. The output should display the frequency of each face in a tabular format.

```
Face    Frequency
1       10001623
2        9996515
3        9997615
4       10002603
5       10002201
6        9999443
endl%
phairojjatanachai@P
Face    Frequency
1       10001623
2        9996515
3        9997615
4       10002603
5       10002201
6        9999443
endl%
phairojjatanachai@P
Face    Frequency
1       10001623
2        9996515
3        9997615
4       10002603
5       10002201
6        9999443
```

```
#include <iostream>
#include <random> // For random number generation

int main() {
// Random number generator seeded with current time
std::default_random_engine generator(static_cast<unsigned
int>(time(0)));

// Uniform distribution in the range [1, 10]
std::uniform_int_distribution<int> distribution10(1, 10);
std::cout << "Random number (1-10): " << distribution10(generator)
<< std::endl;


return 0;
}
```

Random-Number Generation

```
#include <iostream>
#include <iomanip>
#include <cstdlib>

using namespace std;

int main() {
   for (unsigned int counter{1}; counter <= 20; counter++) {
      cout << setw(10) << (1+rand() % 6);

      if (counter % 5 == 0 ) {
         cout << endl;
      }
   }
   cout << rand() << " , " << RAND_MAX << endl;
}
```

shiftValue

scalingFactor

```
         2         2         6         3         5
         3         1         3         6         2
         1         6         1         3         4
         6         2         2         5         5
896544303 , 2147483647
phairojjatanachai@Phairojs-MacBook-Air Lec_Code % ./
         2         2         6         3         5
         3         1         3         6         2
         1         6         1         3         4
         6         2         2         5         5
896544303 , 2147483647
phairojjatanachai@Phairojs-MacBook-Air Lec_Code % ./
         2         2         6         3         5
         3         1         3         6         2
         1         6         1         3         4
         6         2         2         5         5
896544303 , 2147483647
```

Enumeration

- An enumeration is a user-defined type consisting of a set of named constants, known as enumerators.
- Useful for representing a group of related values in a readable and organized way.
  Ex. represent the days of the week, the months of the year, or the states of a game.

Traditional Enums

- Implicit Integral Values:
  - By default, enumerators are assigned integer values starting from 0.
  - You can explicitly specify values:

```
enum Color {
    RED = 1,
    GREEN = 5,
    BLUE  // BLUE will be 6
};

switch (myColor) {
    case RED:
        std::cout << "The color is RED." << std::endl;
        break;
    case GREEN:
        std::cout << "The color is GREEN." << std::endl;
        break;
    case BLUE:
        std::cout << "The color is BLUE." << std::endl;
        break;
    default:
        std::cout << "Unknown color." << std::endl;
}
```

Limitation of Traditional Enums

- Global Namespace Pollution

```
enum Color {
   RED,    // RED added to the global namespace
   GREEN,  // GREEN added to the global namespace
   BLUE    // BLUE added to the global namespace
};

enum TrafficLight {
   RED,    // Conflict! RED is already defined in Color
   YELLOW,
   GREEN   // Conflict! GREEN is already defined in Color
};

int main() {
   // Compiler Error: RED and GREEN are ambiguous
   std::cout << RED << std::endl;
   return 0;
}
```

- Implicit Integer Conversion
- Lack of Type Safety

Scoped Enumerations

- Scoped enumerations, introduced in C++11, provide a safer and more modern alternative to traditional enumerations.
- Key features:
  - Explicit scope

    ```
    enum class Color { RED, GREEN, BLUE };
    Color myColor = Color::RED; // Access using the scope resolution operator
    ```
- Type safty:
  - Scoped enums do not implicitly convert to integers, unlike traditional enums

    ```
    enum class Status { WON, LOST, CONTINUE };

    Status gameStatus = Status::WON;

    // ERROR: Cannot compare with an integer
    // if (gameStatus == 0) { ... }
    ```

Enumeration : Example

```cpp
#include <iostream>
#include <string>
enum class TrafficLightState {Red, Yellow, Green };

std::string getTrafficLightStateName(TrafficLightState state) {
   switch (state) {
      case TrafficLightState::Red:
         return "Red";
      case TrafficLightState::Yellow:
         return "Yellow";
      case TrafficLightState::Green:
         return "Green";
      default:
         return "Unknown State";
   }
}

TrafficLightState changeTrafficLightState(TrafficLightState currentState) {
   switch (currentState) {
      case TrafficLightState::Red:
         return TrafficLightState::Green;
      case TrafficLightState::Yellow:
         return TrafficLightState::Red;
      case TrafficLightState::Green:
         return TrafficLightState::Yellow;
      default:
         return TrafficLightState::Red;
   }
}
```

```cpp
int main() {
   TrafficLightState state = TrafficLightState::Red;

   std::cout << "Current Traffic Light State: " << getTrafficLightStateName(state) <<
std::endl;

   // Change the state and print it
   state = changeTrafficLightState(state);
   std::cout << "New Traffic Light State: " << getTrafficLightStateName(state) << std::endl;

   return 0;
}
```

```
Current Traffic Light State: Red
New Traffic Light State: Green
```

Scope Rules

The scope of a variable or function determines the part of the program where that name is accessible. There are several types of scope:

1. Block Scope (Local Scope): Variables declared within a block {} are only accessible within that block and are destroyed once the block is exited. Functions do not have block scope.

```cpp
void myFunction() {
    // localVar has block scope within myFunction
    int localVar = 5;
    // ...
}
// localVar is not accessible here, outside myFunction
```

2. Function Scope: Labels used in goto statements have function scope. They are only valid within the function where they are defined.

```cpp
#include <iostream>
int main() {
    int n = 10;
jump_here:  // This label has function scope
    std::cout << n << " ";
    n--;
    if (n > 0) {
        goto jump_here;  // Jumps back to the label
    }
    std::cout << "\nCountdown complete.";
    return 0;
}
```

3. File Scope (Global Scope): Variables declared outside all functions or blocks have file or global scope. They are accessible from the point of declaration to the end of the file.

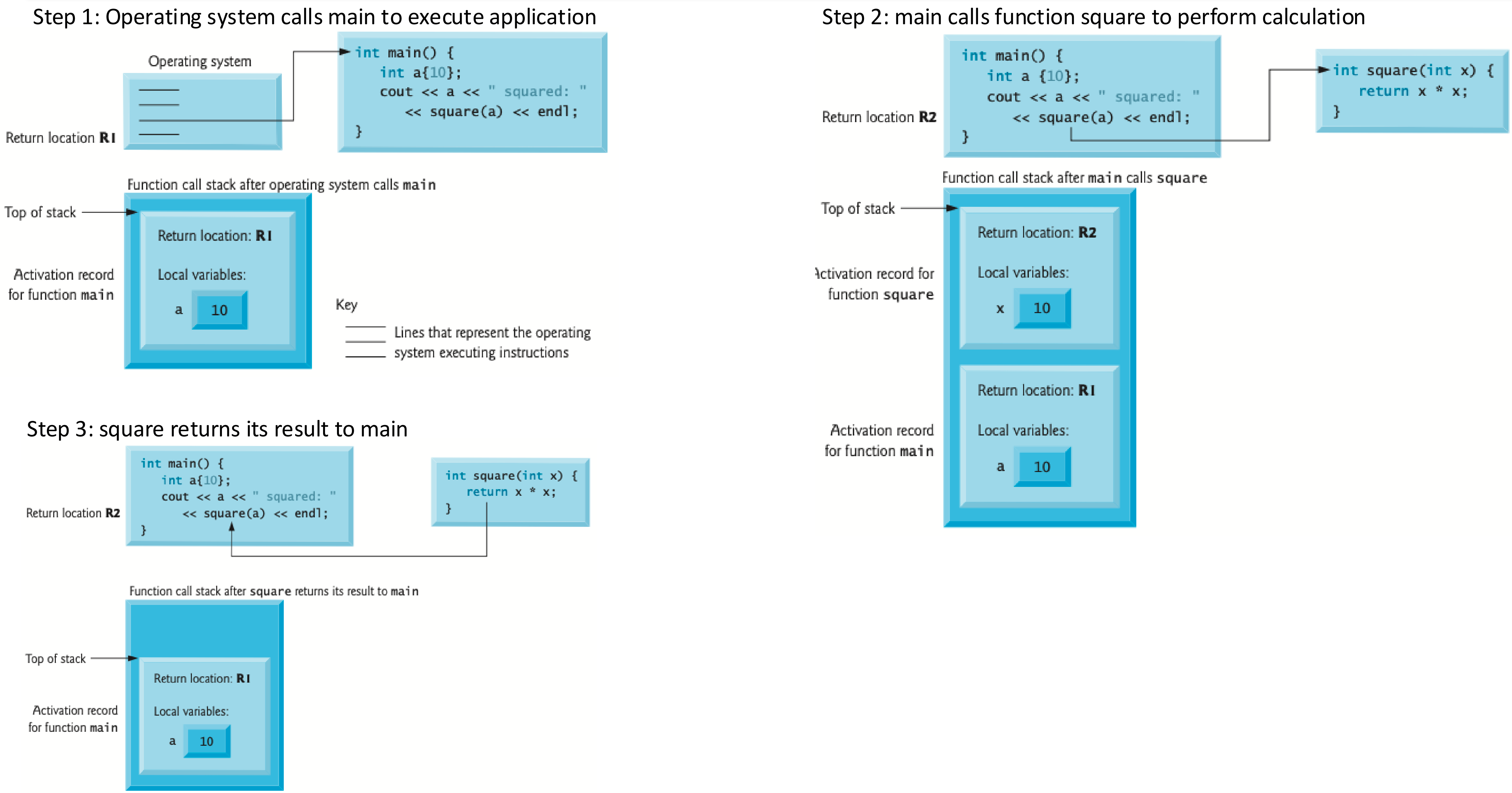
4. Namespace Scope: Namespaces are declarative regions that provide a way to avoid name collisions without the need for overly long variable names.

5. Class Scope: Names declared within a class or struct are accessible only within that class or struct, or through its instances.

Function-Call Stack and Activation Records

```cpp
#include <iostream>

using namespace std;

// Prototype for function square
int square(int x);

int main() {
    int a{10}; // Value to square (local variable in main)
    cout << a << " squared: " << square(a) << endl; // Display a squared
    return 0;
}

// Returns the square of an integer
int square(int x) { // x is a local variable
    return x * x; // Calculate square and return result
}
```

Function-Call Stack and Activation Records



Inline Function

C++ provides inline functions to help reduce function-call overhead. Placing the qualifier inline before a function's return type in the function definition <u>advises the compiler to generate a copy of the function's body code in every place where the function is called</u> (when appropriate) to avoid a function call. This often makes the program larger.

```cpp
#include <iostream>

using namespace std;

// Inline function to calculate the volume of a cube
inline double cube(const double side) {
    return side * side * side; // calculate cube
}

int main() {
    double sideValue; // stores value entered by user

    cout << "Enter the side length of your cube: ";
    cin >> sideValue; // read value from user

    // calculate cube of sideValue and display result
    cout << "Volume of cube with side " << sideValue << " is " << cube(sideValue) << endl;

    return 0;
}
```

References and Reference Parameters

Two ways to pass arguments to functions in many programming languages are pass-by- value and pass-by-reference.

- **passed-by-value**, a copy of the argument’s value is made and passed (on the function-call stack) to the called function.
  Changes to the copy do not affect the original variable’s value in the caller.
- ***pass-by-reference***, the caller gives the called function the ability to *access the caller’s data directly*, and to *modify* that data.

Const References

A const reference is a reference that does not allow modification of the object it refers to.

```
void print(const int &x) {
    std::cout << x << std::endl;
    // x cannot be modified here
}
```

Default Arguments

Default arguments are a feature that allows to specify default values for parameters in a function declaration.
When the function is called, these default values are used if no corresponding arguments are provided in the call.
Benefit:

- make function calls more concise
- maintaining backward compatibility with existing code when new parameters are added to a function.

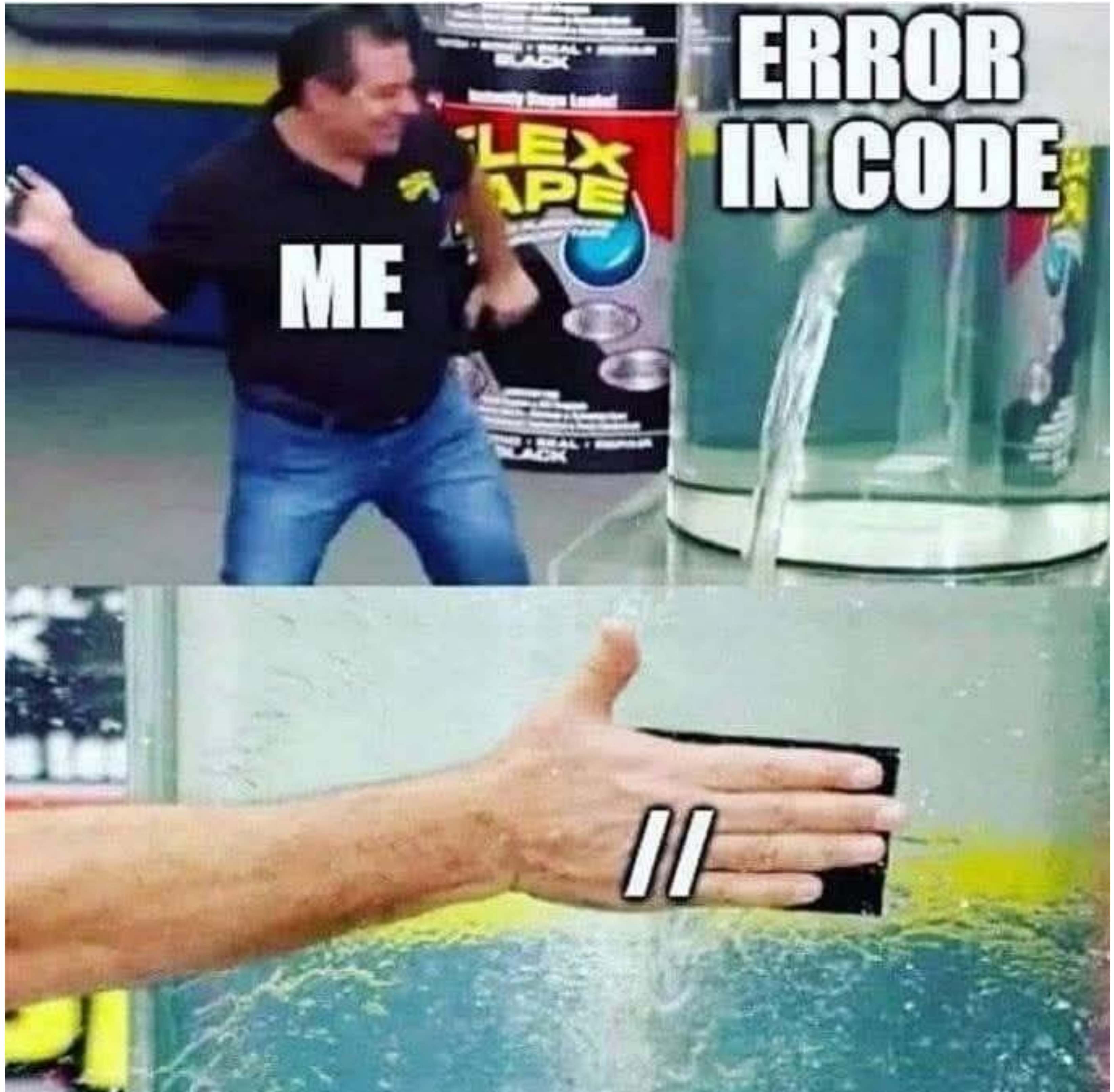
```cpp
#include <iostream>
using namespace std;

// Function declaration with default arguments
void displayMessage(string message = "Hello", int number = 3) {
    for (int i = 0; i < number; ++i) {
        cout << message << endl;
    }
}

int main() {
    displayMessage();               // Uses both defaults: prints "Hello" 3 times
    displayMessage("Hi");           // Uses default for number: prints "Hi" 3 times
    displayMessage("Hey", 2);       // Uses no defaults: prints "Hey" 2 times
    return 0;
}
```

Unary Scope Resolution Operator

- unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope.
- The unary scope resolution operator cannot be used to access a local variable of the same name in an outer block.
- A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

```
#include <iostream>
using namespace std;
int number{7}; // global variable named number

int main() {
double number{10.5}; // local variable named number
    // display values of local and global variables
    cout << "Local double value = " << number
        << "\nGlobal int value = " << ::number << endl;
}
```

```
Local double value of number = 10.5
Global int value of number = 7
```


ERROR
IN CODE
ME
//

Function Template

- Blueprint or formula for creating a family of functions. It allows you to write a generic function that can work with any data type.
- Usage:

```cpp
template <typename T>
T functionName(T parameter) {
    // function body
}
```

```cpp
// Function template to return the larger of two values
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

int main() {
    // The following calls will generate two functions automatically by the compiler:
    // max<int>(int, int) and max<double>(double, double)
    std::cout << "Max of 10 and 20 is " << max(10, 20) << std::endl;
    std::cout << "Max of 22.5 and 18.5 is " << max(22.5, 18.5) << std::endl;

    return 0;
}
```

Recursion

- Recursion is a method of solving problems where a function calls itself as a subroutine.
- A recursive function typically has two main parts:
  - Base Case: The condition under which the recursion ends. This prevents infinite loops. It's a simple case, where the answer can be provided directly without further recursion.
  - Recursive Case: The part of the function where it calls itself to work towards the base case.

```
// n! = n × (n-1) × (n-2) × … × 3 × 2 × 1
int factorial(int n) {
    if (n <= 1) { // Base case
        return 1;
    } else { // Recursive case
        return n * factorial(n - 1);
    }
}
```

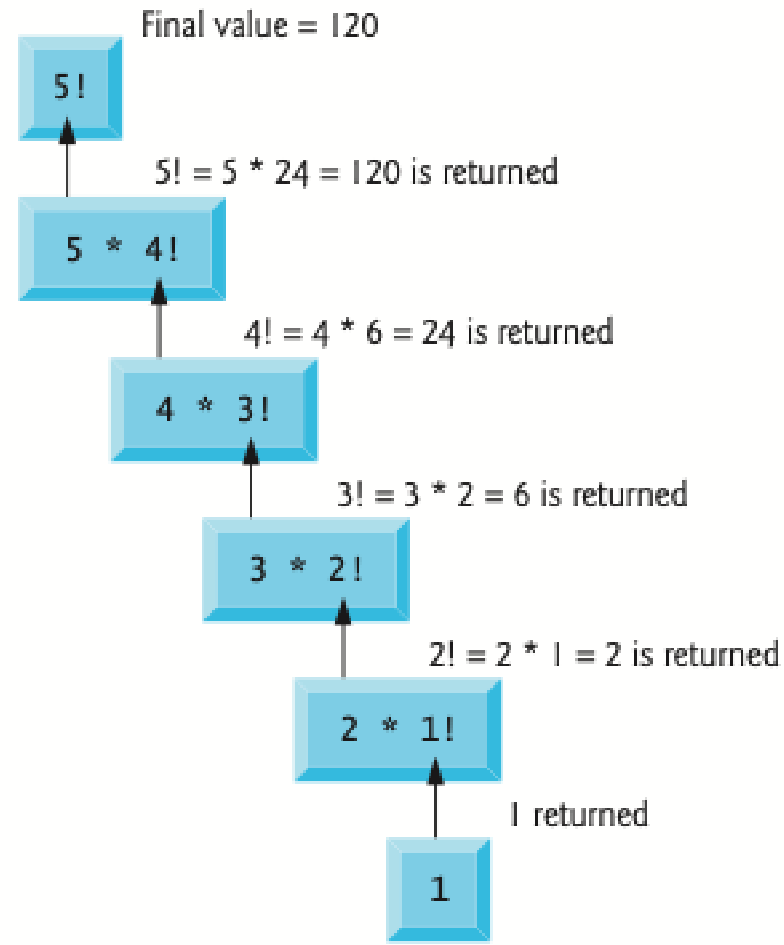
- Key Points to Remember in Recursion:
  - Always Define a Base Case: Without a base case, your recursion could go on indefinitely, leading to a stack overflow.
  - Each Recursive Call Should Progress Toward the Base Case
  - Recursion vs. Iteration: Recursion can often be replaced with iteration (loops). The choice between them depends on the specific problem and which approach is more intuitive or efficient.

Recursion

```
// n! = n × (n-1) × (n-2) × … × 3 × 2 × 1
int factorial(int n) {
    if (n <= 1) { // Base case
        return 1;
    } else { // Recursive case
        return n * factorial(n - 1);
    }
}
```

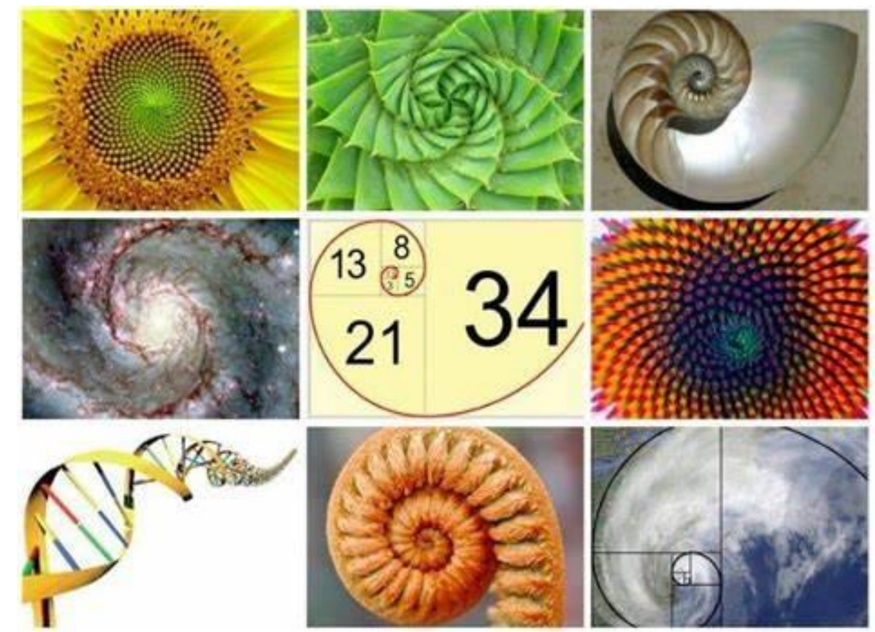
```
int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

Values returned from each recursive call



Example Using Recursion: Fibonacci Series

The Fibonacci Series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. Mathematically, it is defined by the following recurrence relation:



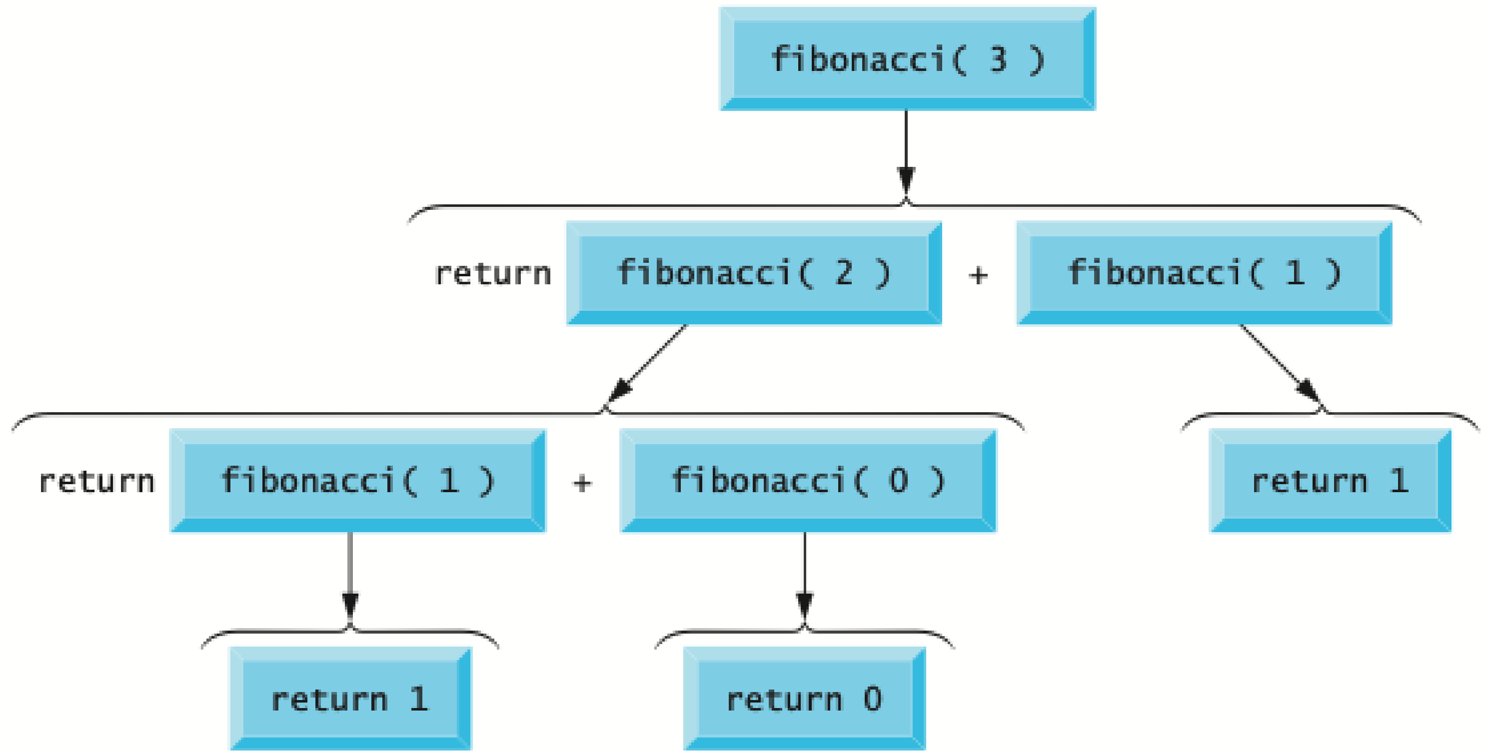
$F(0) = 0, F(1) = 1$
$F(n) = F(n-1) + F(n-2)$ for $n>1$

Example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on.

Why study Fibonacci

- The computational basis for algorithms, such as those that find the greatest common divisor.
- In computer science, it's used for data structure like the Fibonacci heap.
- It appears in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruitlets of a pineapple, and even the flowering of artichoke.

Example Using Recursion: Fibonacci Series



Lambda Function

- Lambda functions are anonymous functions that are used to define small, inline functions that can capture variables from their surrounding scope.
- Syntax:

```
[ capture_clause ] ( parameters ) -> return_type {
        function_body
};
```

- Example:

```
auto add = [](int a, int b) -> int {
        return a + b;
};
std::cout << add(3, 4) << std::endl; // Output: 7

int factor = 3;
auto multiplyByFactor = [factor](int x) {
        return x * factor;
};
std::cout << multiplyByFactor(4) << std::endl; // Output: 12
```

Lambda Function

- Example:

```
int factor = 3;
auto multiplyByFactor = [&factor](int x) {
        return x * factor;
};
factor = 5; // Change the value of factor
std::cout << multiplyByFactor(4) << std::endl; // Output: 20
```

- When capturing by value ([factor]), the lambda captures a copy of the variable's value at the time of definition.
- Changes to the original variable (factor) do not affect the captured value inside the lambda.

Standard Library header	Explanation
<iostream>	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
<iomanip>	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 4.10 and is discussed in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
<cmath>	Contains function prototypes for math library functions (Section 6.3).
<cstdlib>	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 6.7; Chapter 11, Operator Overloading; Class string; Chapter 17, Exception Handling: A Deeper Look; Chapter 22, Bits, Characters, C Strings and structs; and Appendix F, C Legacy Code Topics.
<ctime>	Contains function prototypes and types for manipulating the time and date. This header is used in Section 6.7.
<array>, <vector>, <list>, <forward_list>, <deque>, <queue>, <stack>, <map>, <unordered_map>, <unordered_set>, <set>, <bitset>	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <vector> header is first introduced in Chapter 7, Class Templates array and vector; Catching Exceptions. We discuss all these headers in Chapter 15, Standard Library Containers and Iterators. <array>, <forward_list>, <unordered_map> and <unordered_set> were all introduced in C++11.
<cctype>	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 22, Bits, Characters, C Strings and structs.
<cstring>	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 10, Operator Overloading; Class string.
<typeinfo>	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 12.9.
<exception>, <stdexcept>	These headers contain classes that are used for exception handling (discussed in Chapter 17, Exception Handling: A Deeper Look).
<memory>	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 17, Exception Handling: A Deeper Look.
<fstream>	Contains function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 14, File Processing).
<string>	Contains the definition of class string from the C++ Standard Library (discussed in Chapter 21, Class string and String Stream Processing).
<sstream>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 21, Class string and String Stream Processing).
<functional>	Contains classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 15.
<iterator>	Contains classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 15.
<algorithm>	Contains functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 15.
<cassert>	Contains macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.
<cfloat>	Contains the floating-point size limits of the system.
<climits>	Contains the integral size limits of the system.
<cstdio>	Contains function prototypes for the C-style standard input/output library functions.
<locale>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
<limits>	Contains classes for defining the numerical data type limits on each computer platform—this is C++'s version of <climits> and <cfloat>.
<utility>	Contains classes and functions that are used by many C++ Standard Library headers.

Q & A