Programming Technique II SECJ1023

Semester 2, 2024/2025

Faculty of Computing, UTM

Course Overview

- The course covers another concept of programming: **Object-Oriented Programming (OOP)**

- Language used: **C++**

Course Topics

- Overview of Programming Paradigms
- Introduction to OOP

- Introduction to Classes and Objects
- Constructors and Destructors
- Class and Object Manipulations

- String Manipulations

- Associations, Aggregations and Compositions
- Inheritance
- Polymorphisms

- Exceptions and Templates
- Containers and Iterators

01: Introduction to Object-oriented Programming

Programming Technique II

(SECJ1023)

Adapted from Tony Gaddis and Barret Krupnow (2016), Starting out with C++: From Control Structures through Objects

Procedural Programming (PP)

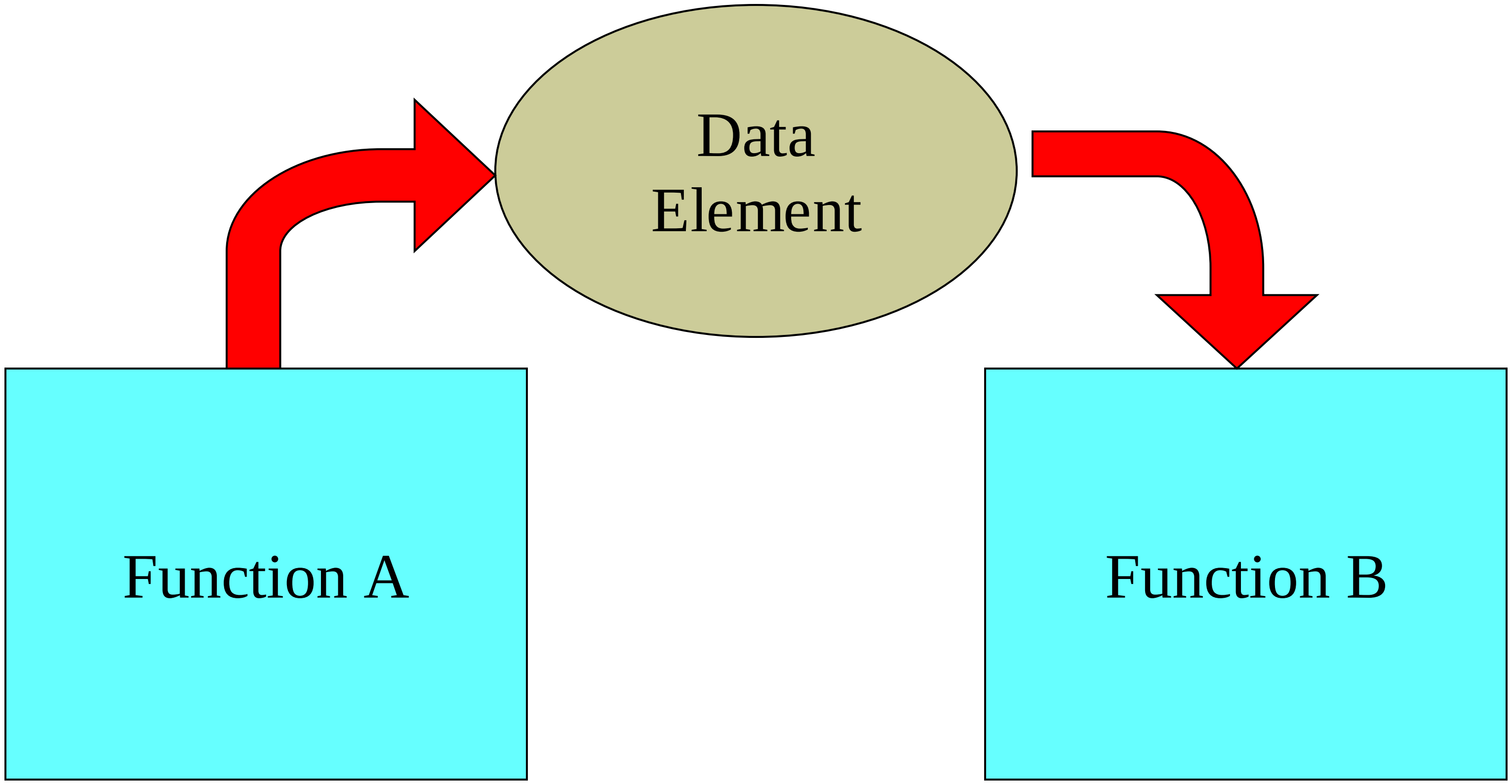
- Traditional programming languages were procedural.
  - C, Pascal, BASIC, Ada and COBOL

- Programming in procedural languages involves choosing data structures (appropriate ways to store data), designing algorithms, and translating algorithm into code.

- In procedural programming, data and operations are separated.

- This methodology requires sending data to procedure/functions

Procedural Programming



Functional Programming (FP)

- FP is a programming paradigm where programs are constructed by applying and composing functions
- Functions are treated as first-class citizen. They can be:
  - bound to names
  - passed as parameters to other functions
  - returned from other functions

- FP uses declarative programming style
  - expresses the logic of WHAT the program should accomplish without specifying how it should achieve that.

Note: PP and OOP use imperative programming style. This style focuses on describing HOW a program operates

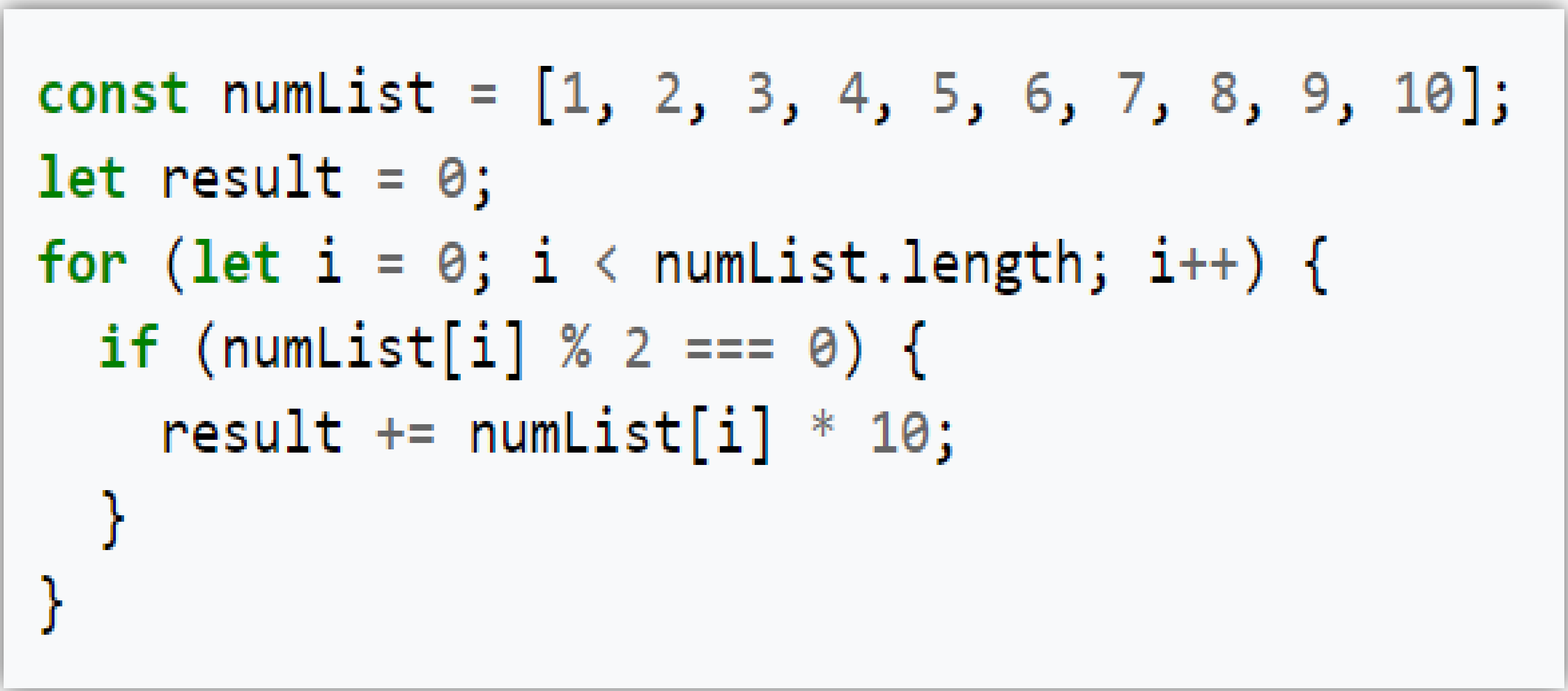
Comparison of FP to Imperative Programming

Example Problem:

Multiply all even numbers in an array by 10 and add them all, storing the final sum in the variable "result".

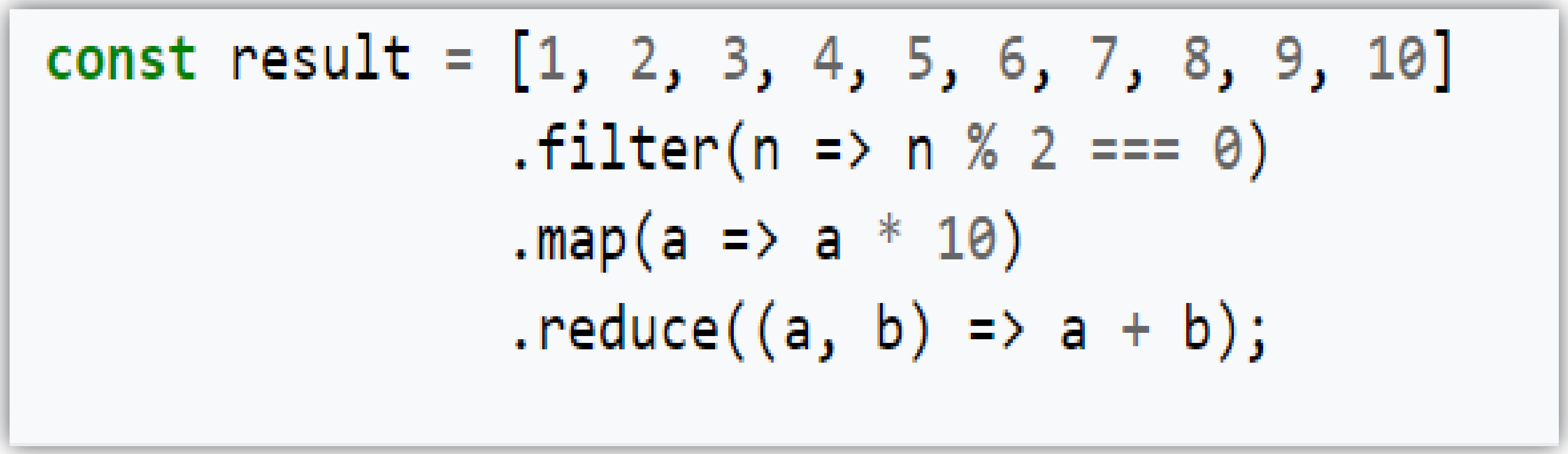
Both solution on the next slides are written in JavaScript, but with different programming paradigms

Traditional imperative loop



```javascript
const numList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let result = 0;
for (let i = 0; i < numList.length; i++) {
  if (numList[i] % 2 === 0) {
    result += numList[i] * 10;
  }
}
```

Functional Programming with high-order functions



```
const result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
                .filter(n => n % 2 === 0)
                .map(a => a * 10)
                .reduce((a, b) => a + b);
```

FP Concepts

- High-order and Callback Functions
- Pure Functions
- Lambda Functions
- Currying
- Recursion
- Function composition

Further readings:

Functional programming
https://en.wikipedia.org/wiki/Functional_programming

A Comprehensive Look at Functional Programming (FP)
https://medium.com/swlh/a-comprehensive-look-at-functional-programming-fp-4a87629ecaed

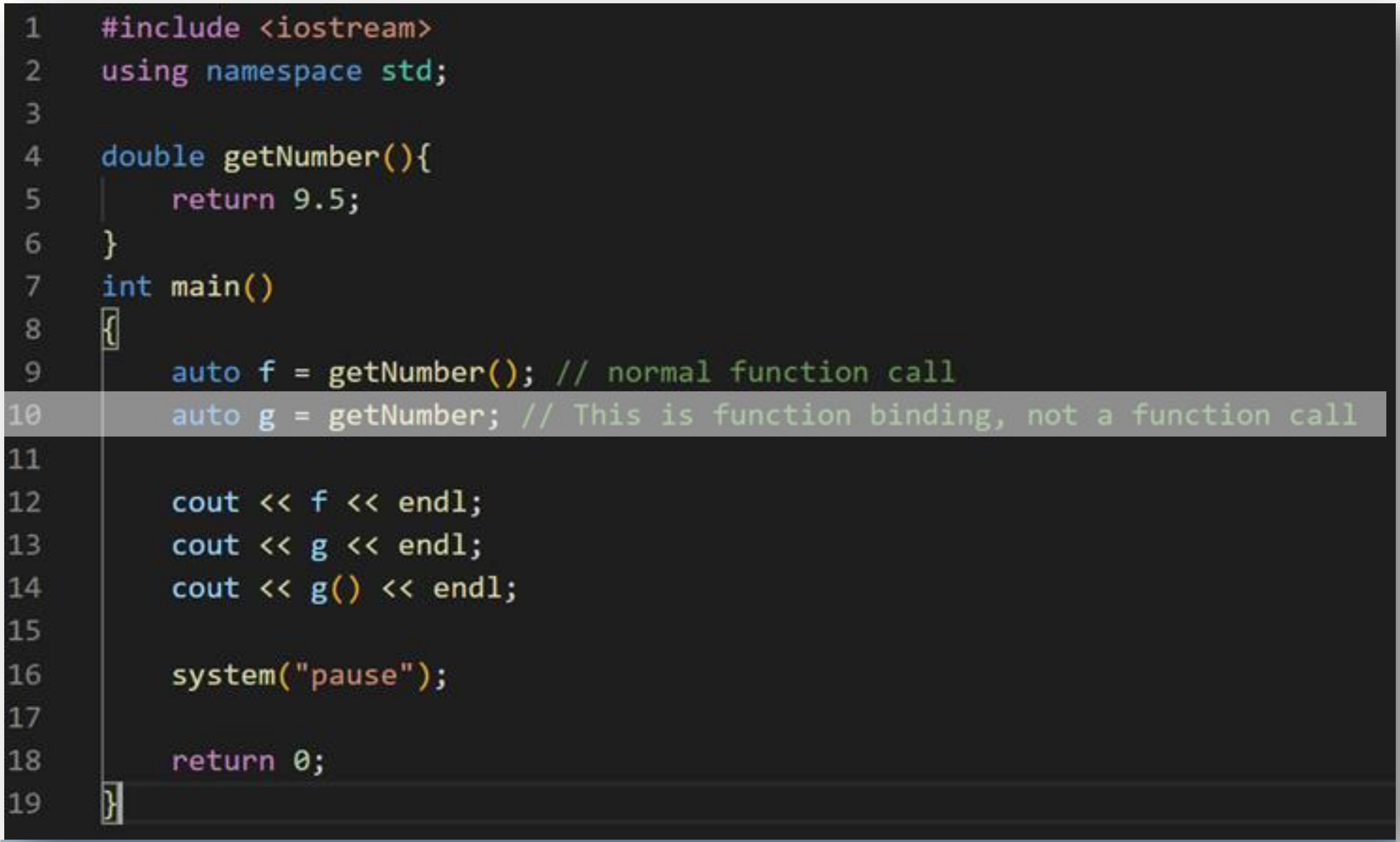
High-order functions

- A High-order function (HOF) is a function that meets either one of these requirements:
  - It accepts other functions as parameters
  - It returns a function *(this kind of HOF is called a factory function)*

- The functions that are sent as parameters are called callback functions
- Note that the sent functions will be bound rather than called to (execute)

Function Binding

A variable can hold a function



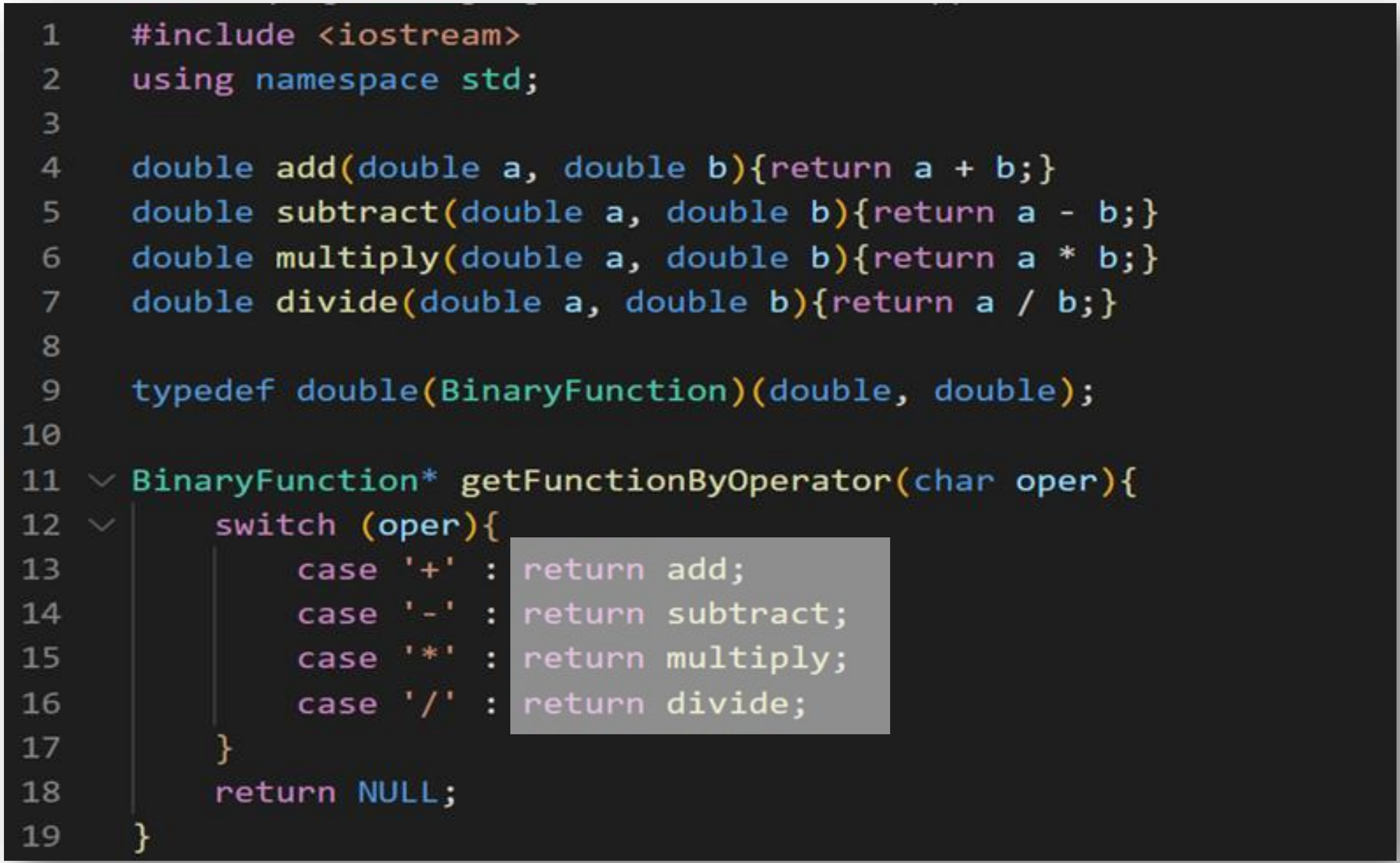
```cpp
#include <iostream>
using namespace std;

double getNumber(){
    return 9.5;
}
int main()
{
    auto f = getNumber(); // normal function call
    auto g = getNumber; // This is function binding, not a function call

    cout << f << endl;
    cout << g << endl;
    cout << g() << endl;

    system("pause");

    return 0;
}
```

Output:

```
9.5
1
9.5
```

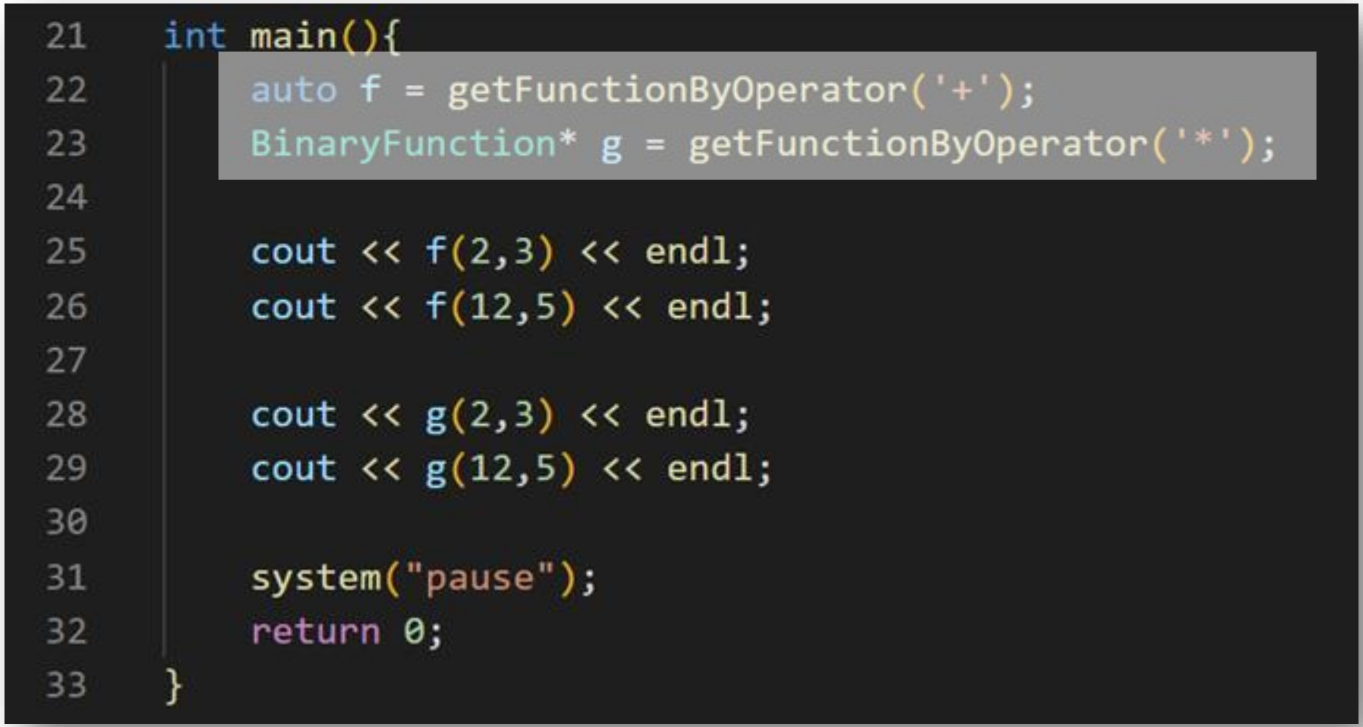
Returning Functions from a Function

In the following example, getFunctionByOperator is a high-order function



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

double add(double a, double b){return a + b;}
double subtract(double a, double b){return a - b;}
double multiply(double a, double b){return a * b;}
double divide(double a, double b){return a / b;}

typedef double(BinaryFunction)(double, double);

BinaryFunction* getFunctionByOperator(char oper){
    switch (oper){
        case '+' : return add;
        case '-' : return subtract;
        case '*' : return multiply;
        case '/' : return divide;
    }
    return NULL;
}
```

Return Function from Another Function



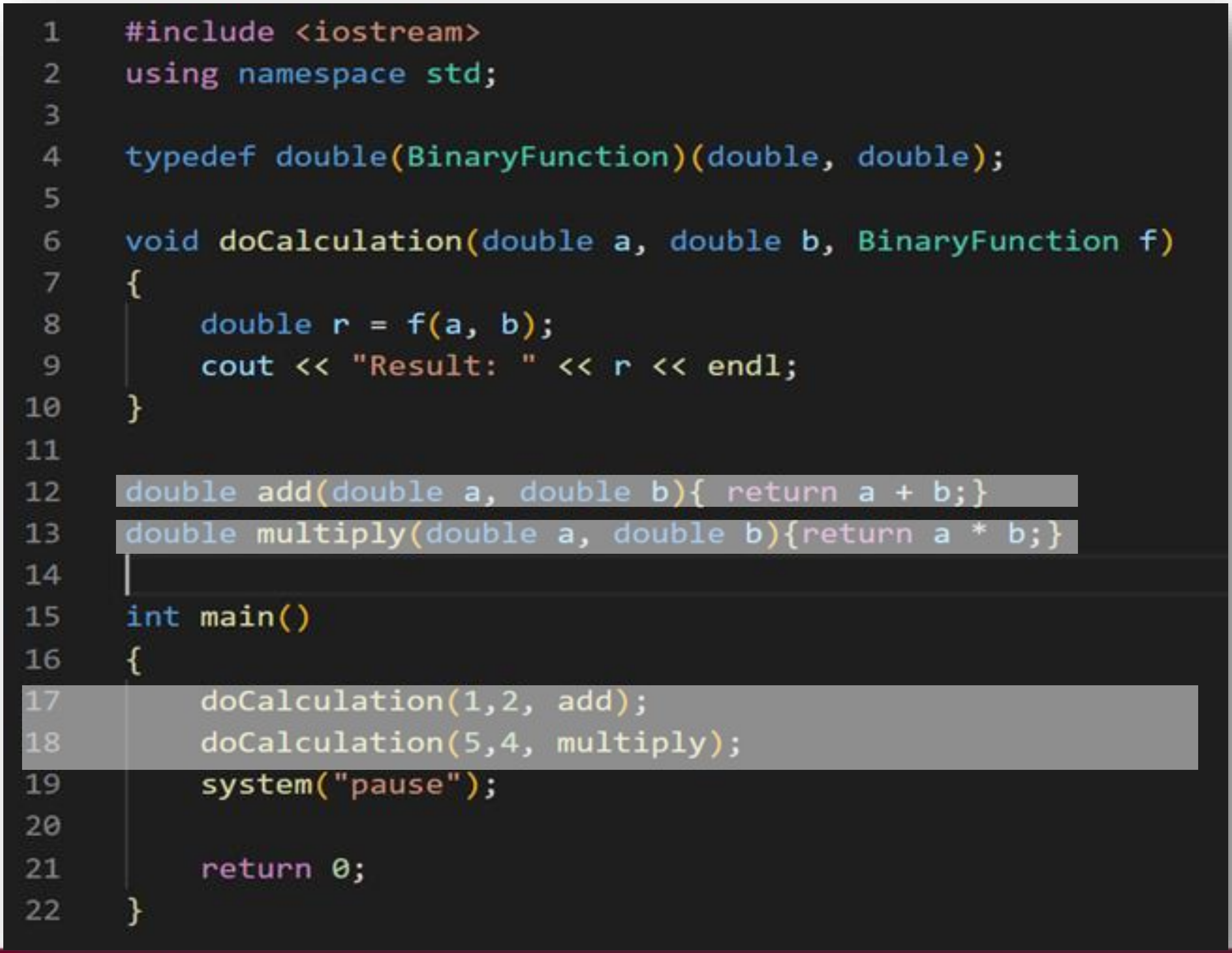
```
int main(){
    auto f = getFunctionByOperator('+');
    BinaryFunction* g = getFunctionByOperator('*');

    cout << f(2,3) << endl;
    cout << f(12,5) << endl;

    cout << g(2,3) << endl;
    cout << g(12,5) << endl;

    system("pause");
    return 0;
}
```

Output:

```
5
17
6
60
```

High-order Functions and Callbacks

In the following example, add and multiply are callback functions and doCalculation is a high-order function



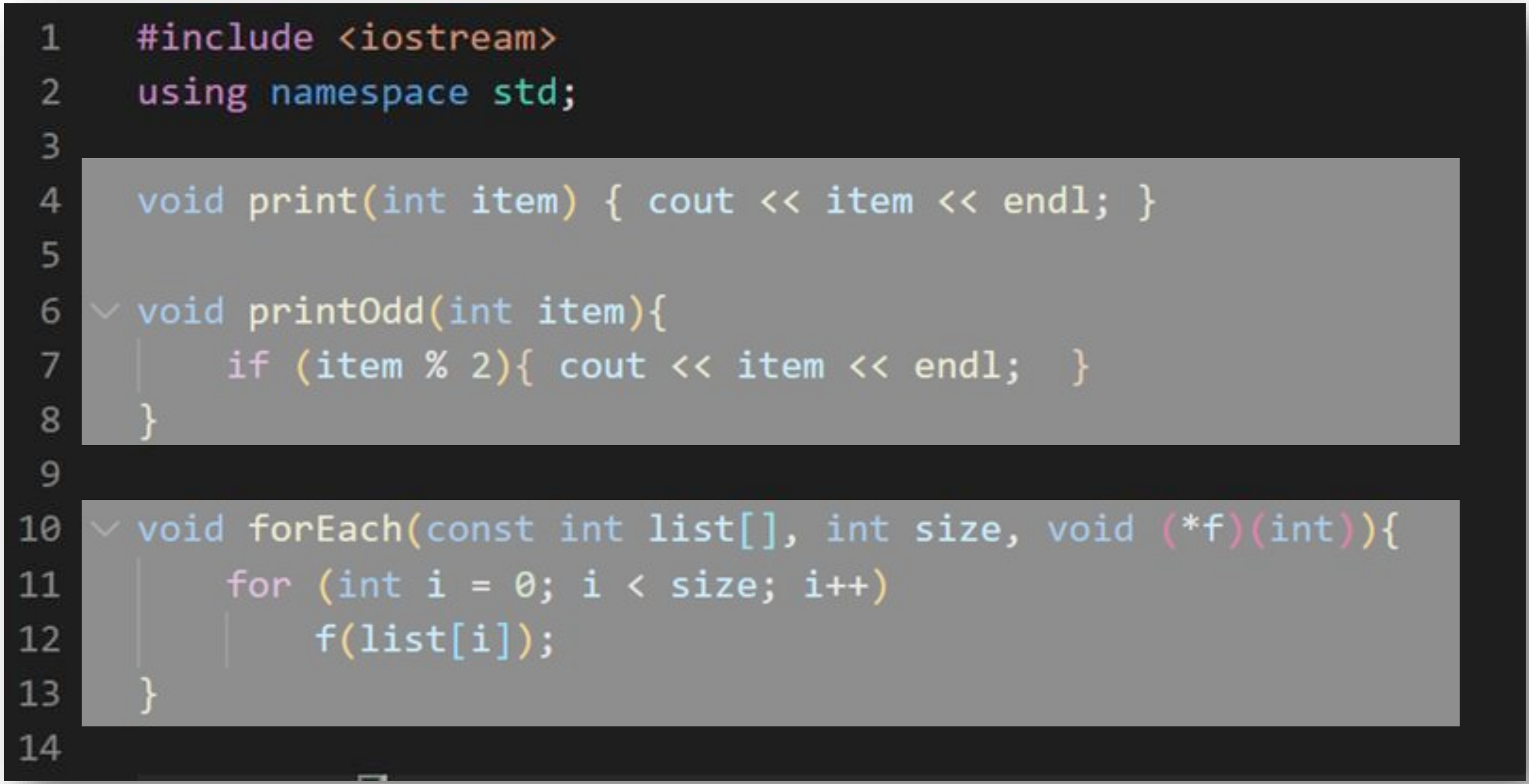
```cpp
#include <iostream>
using namespace std;

typedef double(BinaryFunction)(double, double);

void doCalculation(double a, double b, BinaryFunction f)
{
    double r = f(a, b);
    cout << "Result: " << r << endl;
}

double add(double a, double b){ return a + b;}
double multiply(double a, double b){return a * b;}

int main()
{
    doCalculation(1,2, add);
    doCalculation(5,4, multiply);
    system("pause");

    return 0;
}
```

Output:

```
Result: 3
Result: 20
```

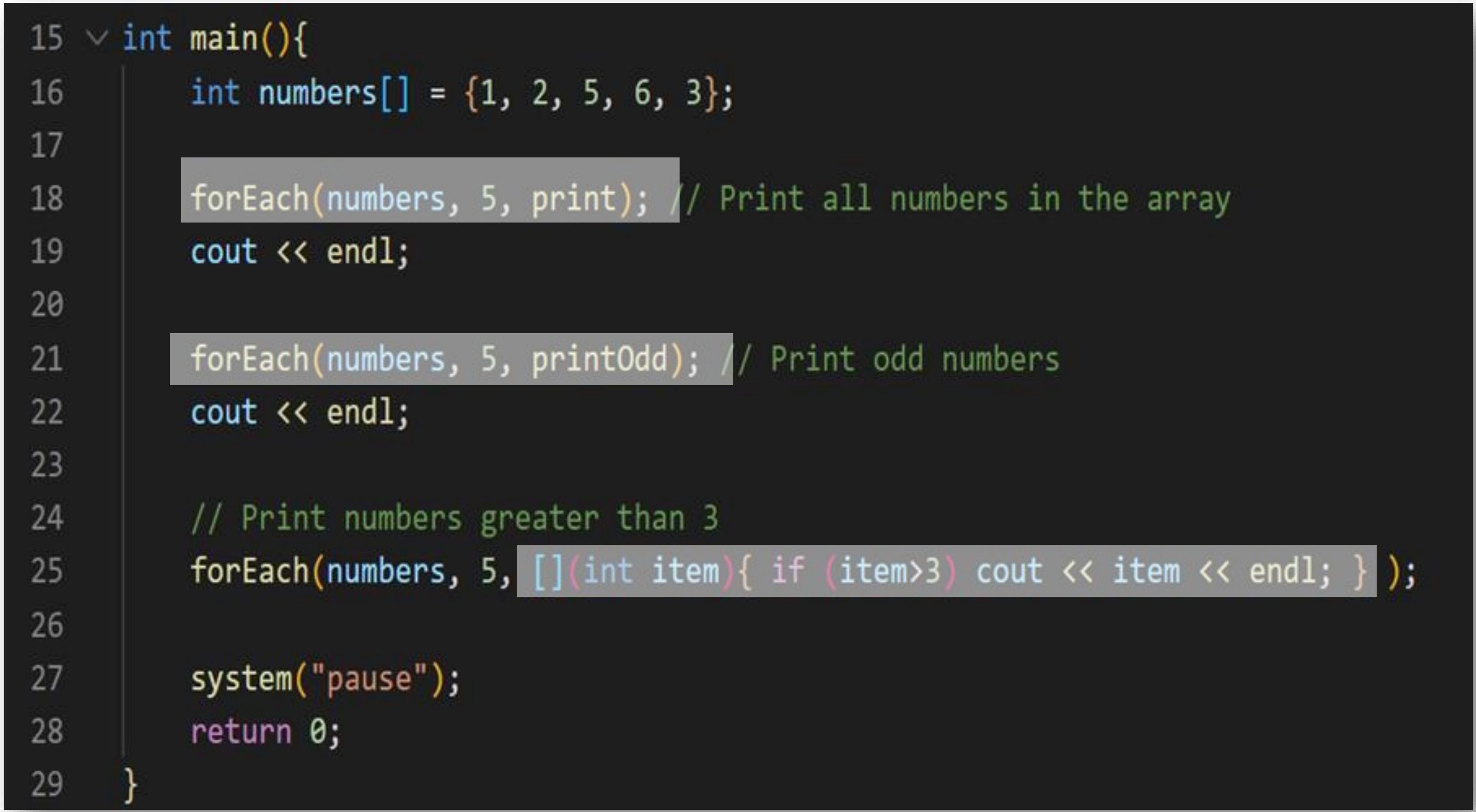
Practical Use Case of High-order Functions

To manipulate arrays



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

void print(int item) { cout << item << endl; }

void printOdd(int item){
    if (item % 2){ cout << item << endl;  }
}

void forEach(const int list[], int size, void (*f)(int)){
    for (int i = 0; i < size; i++)
        f(list[i]);
}

```

Practical Use Case of High-order Functions



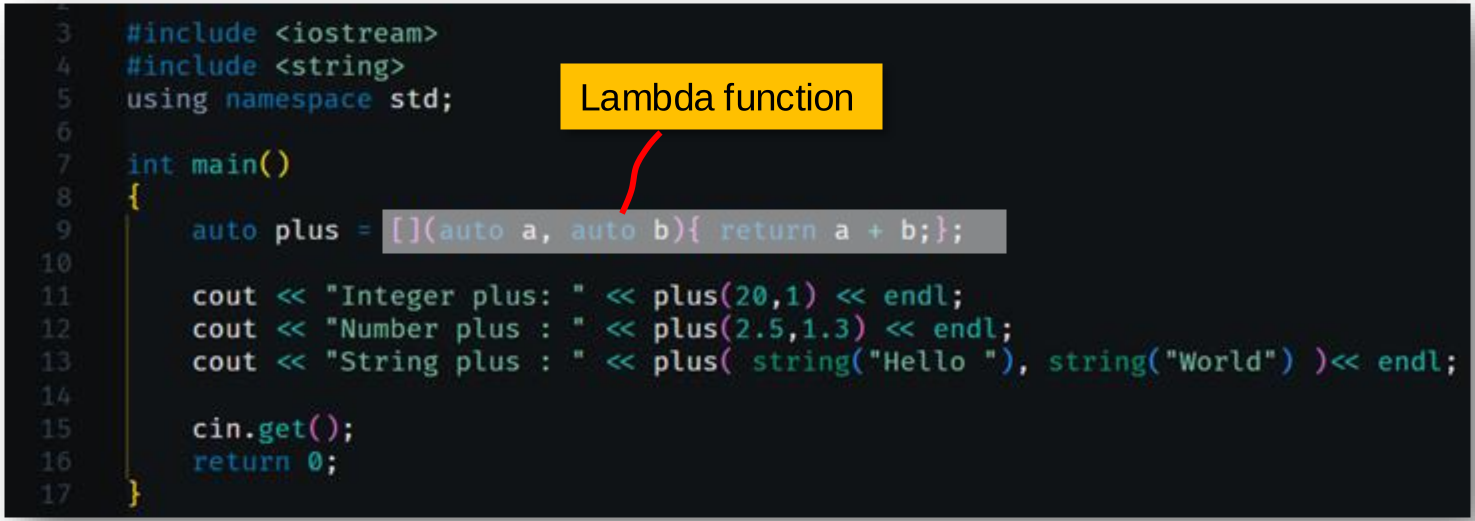
```cpp
int main(){
    int numbers[] = {1, 2, 5, 6, 3};

    forEach(numbers, 5, print); // Print all numbers in the array
    cout << endl;

    forEach(numbers, 5, printOdd); // Print odd numbers
    cout << endl;

    // Print numbers greater than 3
    forEach(numbers, 5, [](int item){ if (item>3) cout << item << endl; } );

    system("pause");
    return 0;
}
```

Lambda Functions

- Lambda functions are functions without names.
- They can be written inline (no need declaration)
- They cannot be re-used (i.e. they execute only once)
- Some use cases of Lambda functions:
  - Function binding
  - Passing functions as parameters to another function.
  - Returning a function from another.

Lambda Function Example 1: Function Binding

- You can assign a variable with a function directly or inline (i.e., no need to declare the function first)



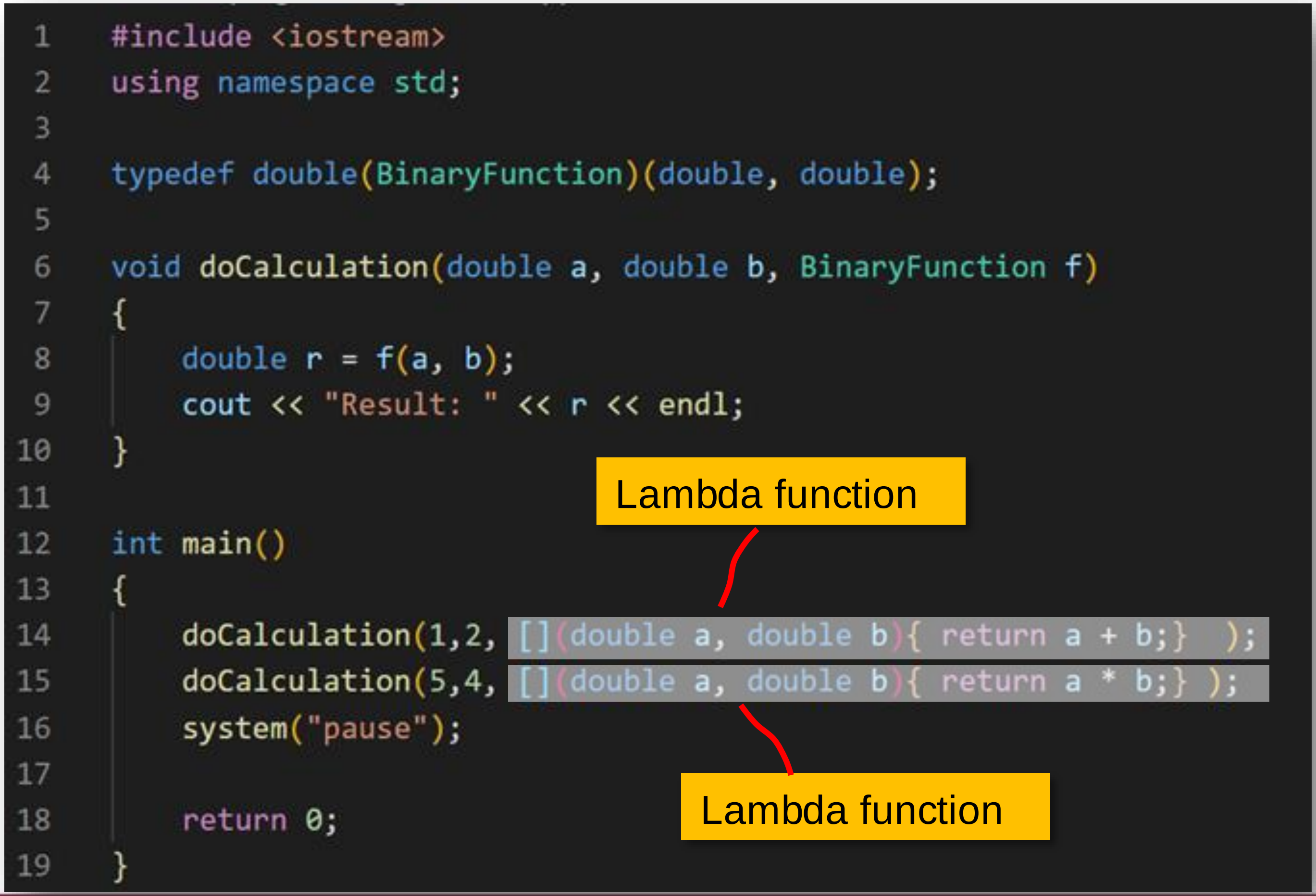
```cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
    auto plus = [](auto a, auto b){ return a + b;};

    cout << "Integer plus: " << plus(20,1) << endl;
    cout << "Number plus : " << plus(2.5,1.3) << endl;
    cout << "String plus : " << plus( string("Hello "), string("World") )<< endl;

    cin.get();
    return 0;
}
```

Output:

```
Integer plus: 21
Number plus: 3.8
String plus: Hello World
```

Lambda Function Example 2: Passing functions as parameters

- A callback can be directly written to the high-order function (without having it defined first).
- This is called Lambda function (or Anonymous function, i.e. nameless or no name function)



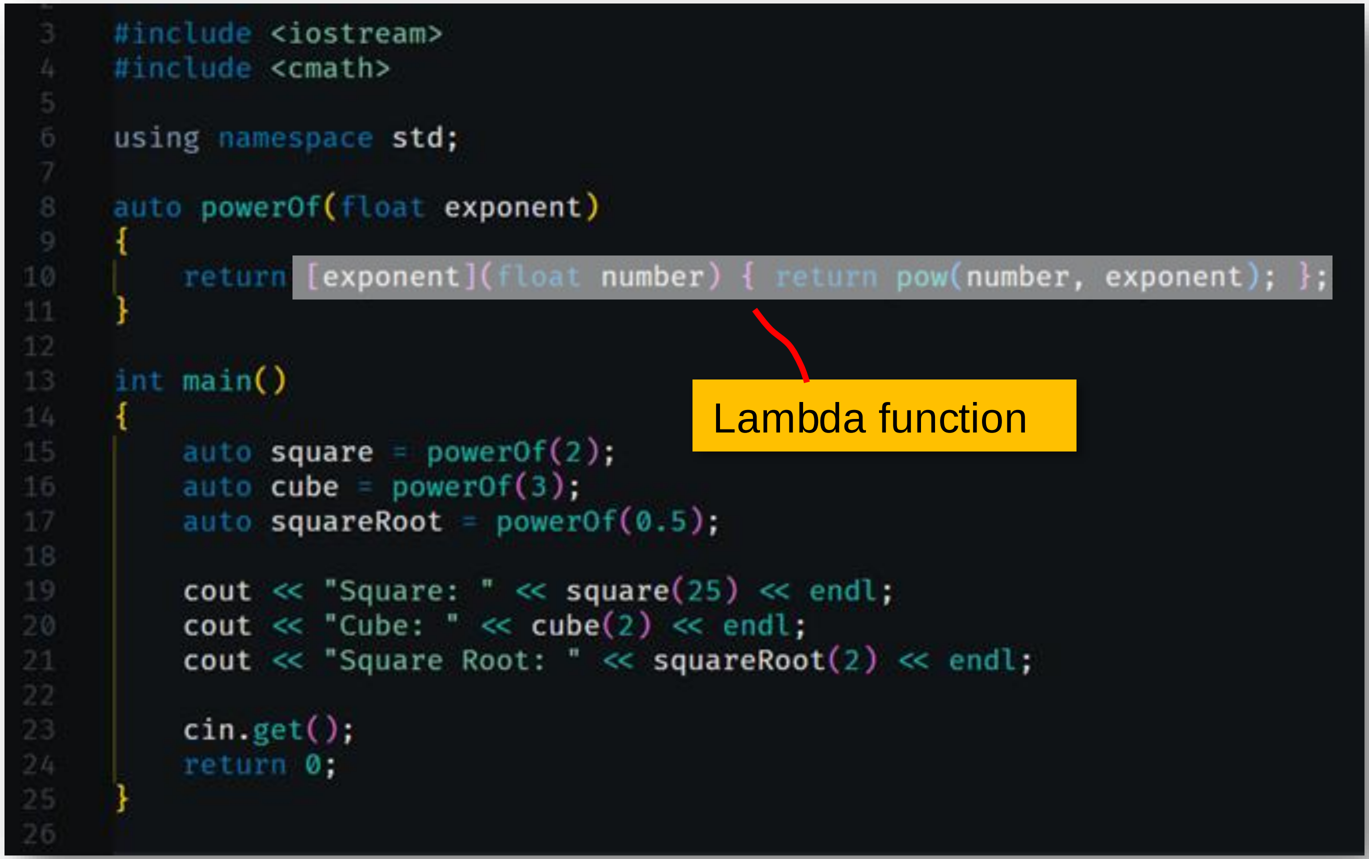
```
#include <iostream>
using namespace std;

typedef double(BinaryFunction)(double, double);

void doCalculation(double a, double b, BinaryFunction f)
{
    double r = f(a, b);
    cout << "Result: " << r << endl;
}

int main()
{
    doCalculation(1,2, [](double a, double b){ return a + b;}  );
    doCalculation(5,4, [](double a, double b){ return a * b;} );
    system("pause");

    return 0;
}
```

Output:

```
Result: 3
Result: 20
```

Lambda Function Example 3: Returning a function from another function

- In the following example, powerOf is a function factory (a function that returns another function).
- square, cube and squareRoot are functions generated from the function factory.



```cpp
#include <iostream>
#include <cmath>

using namespace std;

auto powerOf(float exponent)
{
    return [exponent](float number) { return pow(number, exponent); };
}

int main()
{
    auto square = powerOf(2);
    auto cube = powerOf(3);
    auto squareRoot = powerOf(0.5);

    cout << "Square: " << square(25) << endl;
    cout << "Cube: " << cube(2) << endl;
    cout << "Square Root: " << squareRoot(2) << endl;

    cin.get();
    return 0;
}
```

Lambda function

Output:

```
Square: 625
Cube: 8
Square Root: 1.41421
```

Currying

- Currying is a technique from Functional Programming that transforms a function accepting multiple parameters into several functions accepting single parameter

- The translation looks something like this:

```
originalFunction(param1, param2, param3)
curriedFunction(param1)(param2)(param3)
```

- Currying is implemented by wrapping a function inside a function, i.e., the function returns the other function.

- The parent function takes the first parameter and returns the function that takes the next parameter and this repeats until the last parameter.

Currying

- From the previous example, there will be three function calls at the **`curriedFunction:`**
  - Call to the parent function, i.e., **`curriedFunction`** which takes `param1` and returns another function (we call this function, **a**).

    ```
    a = curriedFunction(param1)
    ```

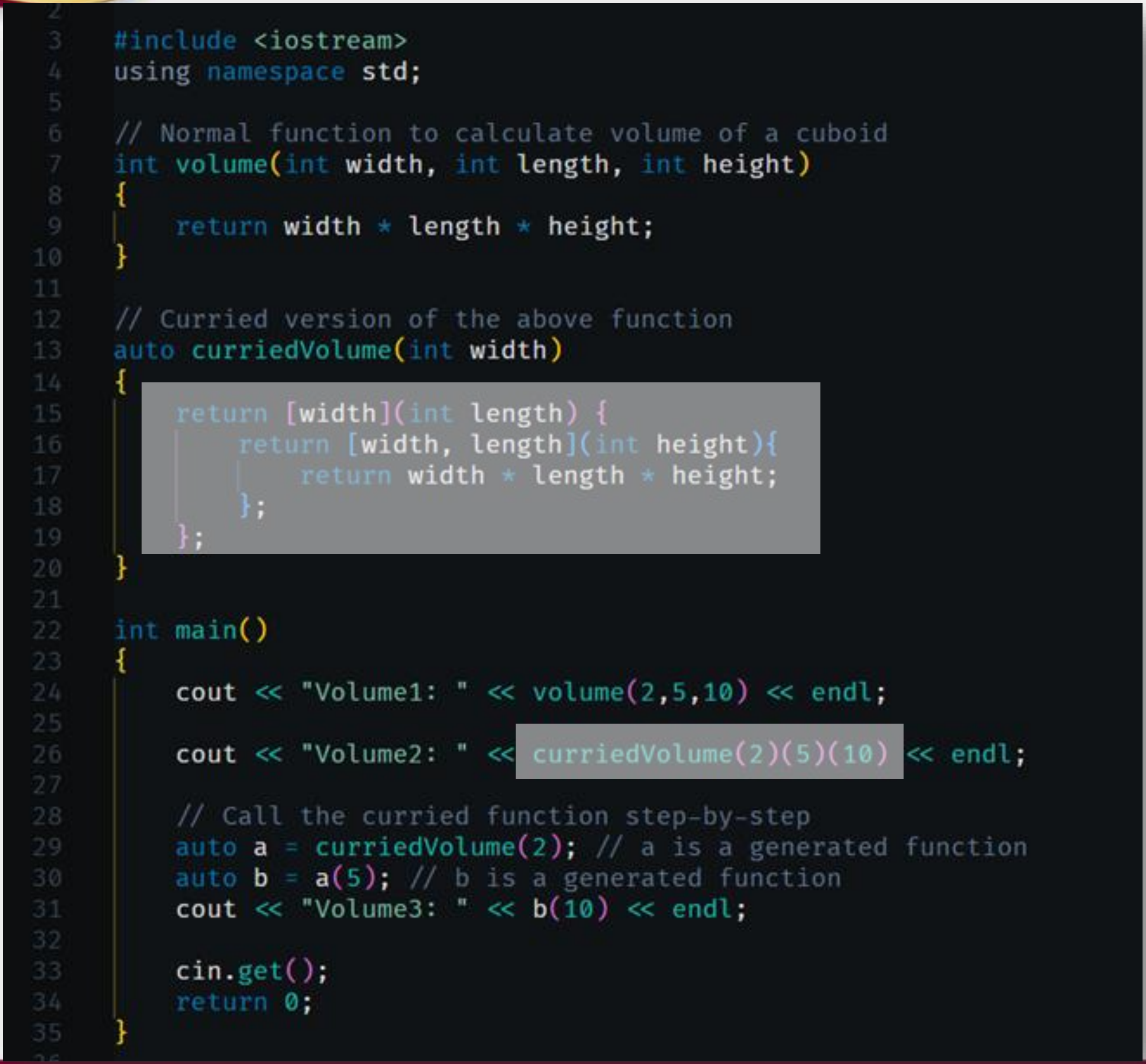
  - Call to the function **a** which takes `param2` and returns another function (we call this function, **b**).

    ```
    b = a(param2)
    ```

  - Call to the function **b** which takes `param3` and returns the result.

    ```
    result = b(param3)
    ```

Currying Example:



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

// Normal function to calculate volume of a cuboid
int volume(int width, int length, int height)
{
    return width * length * height;
}

// Curried version of the above function
auto curriedVolume(int width)
{
    return [width](int length) {
        return [width, length](int height){
            return width * length * height;
        };
    };
}

int main()
{
    cout << "Volume1: " << volume(2,5,10) << endl;

    cout << "Volume2: " << curriedVolume(2)(5)(10) << endl;

    // Call the curried function step-by-step
    auto a = curriedVolume(2); // a is a generated function
    auto b = a(5); // b is a generated function
    cout << "Volume3: " << b(10) << endl;

    cin.get();
    return 0;
}
```

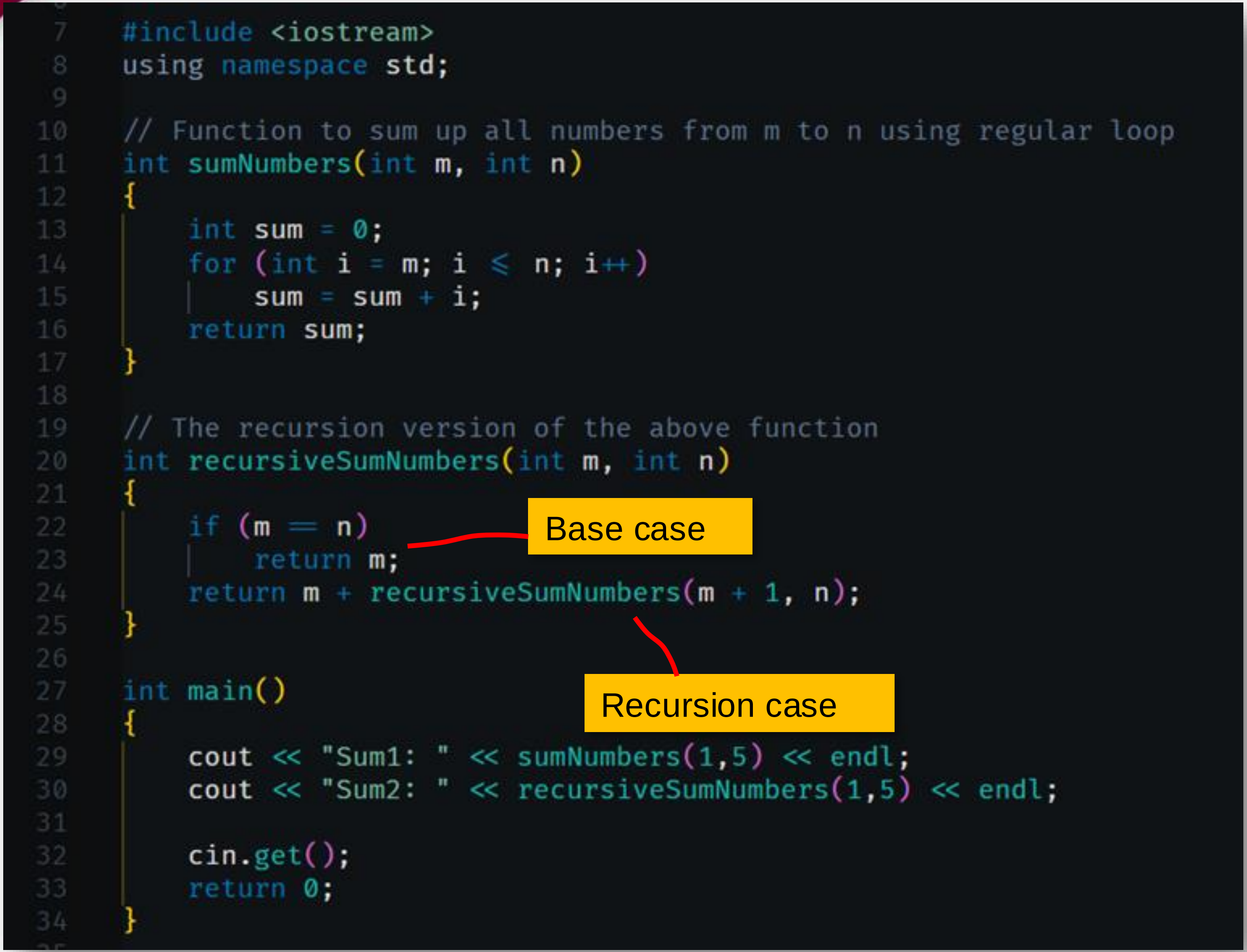
Output:

```
Volume1 : 100
Volume2 : 100
Volume3 : 100
```

Recursion

- Recursion is a technique that uses functions to call to themselves.
- Such functions are called recursive functions.
- A recursion instruction continues until another instruction prevents it.
- Iterations in PP paradigm are done with loops. In FP paradigm, mostly using recursion to perform repeating tasks.
- A recursion code normally consists two parts:
  - Recursion. Parts where the function calls to itself
  - Base case (or termination condition). To end the recursion.

Recursion Example



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

// Function to sum up all numbers from m to n using regular loop
int sumNumbers(int m, int n)
{
    int sum = 0;
    for (int i = m; i <= n; i++)
        sum = sum + i;
    return sum;
}

// The recursion version of the above function
int recursiveSumNumbers(int m, int n)
{
    if (m == n)
        return m;
    return m + recursiveSumNumbers(m + 1, n);
}

int main()
{
    cout << "Sum1: " << sumNumbers(1,5) << endl;
    cout << "Sum2: " << recursiveSumNumbers(1,5) << endl;

    cin.get();
    return 0;
}
```

Output:

```
Sum1: 15
Sum2: 15
```

Object-Oriented Programming (OOP)

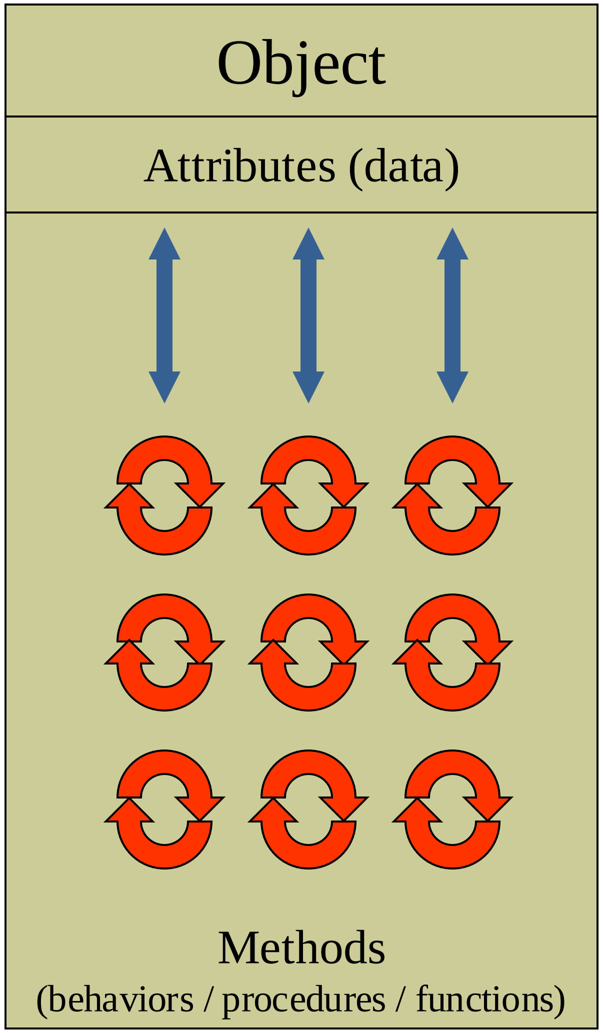
- OOP is centred on **objects** rather than procedures / functions.

- Objects are a melding of **data and operations** that manipulate that data.

- Data in an object are known as **properties** or **attributes** .

- Operations/functions in an object are known as **methods**.

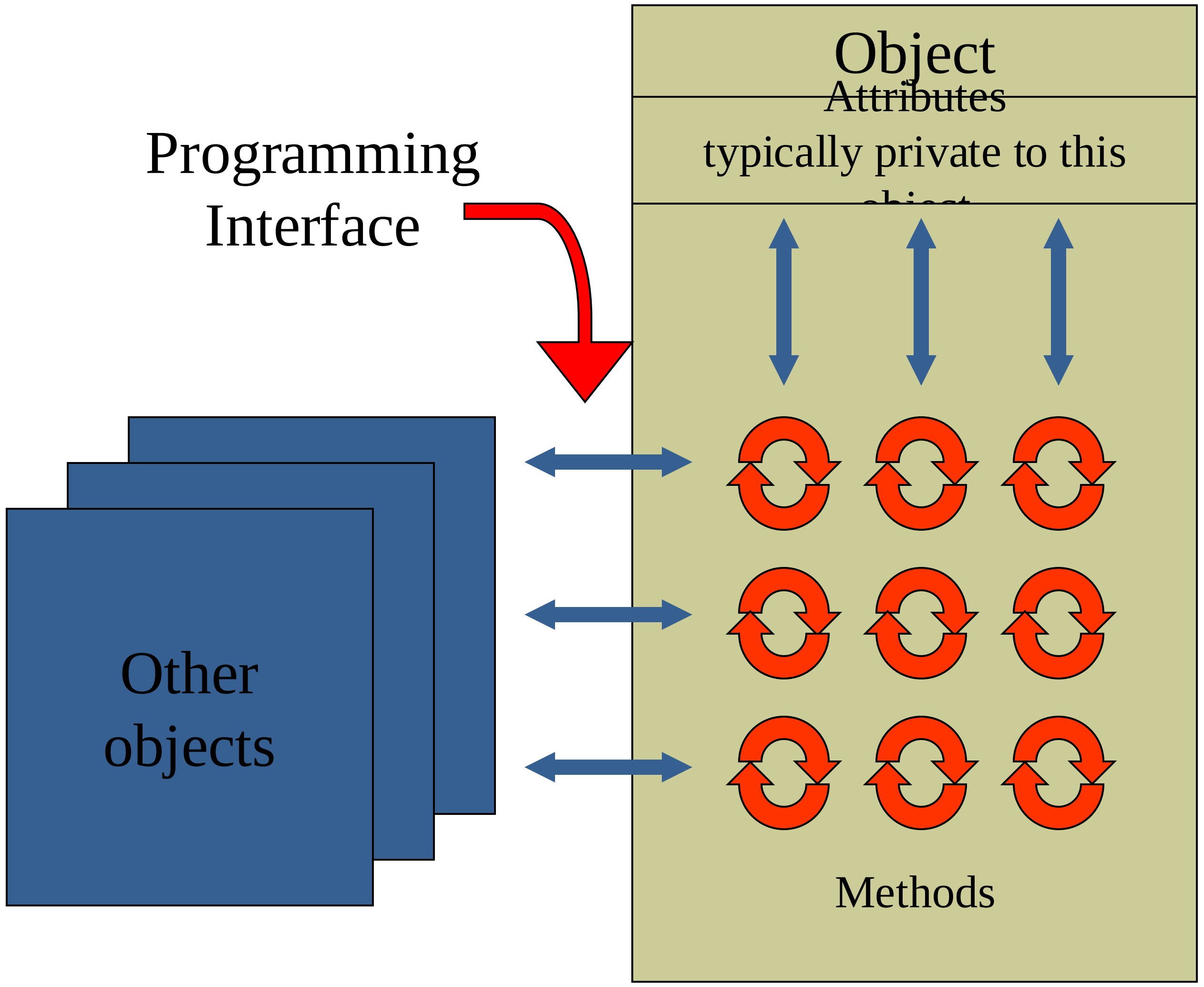
Object-Oriented Programming



Object-Oriented Programming

- Object-oriented programming combines data and methods via **encapsulation**.
- **Data hiding** is the ability of an object to hide data from other objects in the program
- Only object’s methods should be able to directly manipulate its attributes
- Other objects are allowed to manipulate object’s attributes via the object’s methods.
- This indirect access is known as a **programming interface**

Object-Oriented Programming



Object-Oriented Programming

- Object-oriented programming combines data and methods via **encapsulation**.
- **Data hiding** is the ability of an object to hide data from other objects in the program
- Only object’s methods should be able to directly manipulate its attributes
- Other objects are allowed to manipulate object’s attributes via the object’s methods.
- This indirect access is known as a **programming interface**

OOP Principles: Classes

- A class is the **template** or mould or blueprint from which objects are actually made.

- A class **encapsulates** the attributes and actions that characterizes a certain type of object.

OOP Principles: Objects

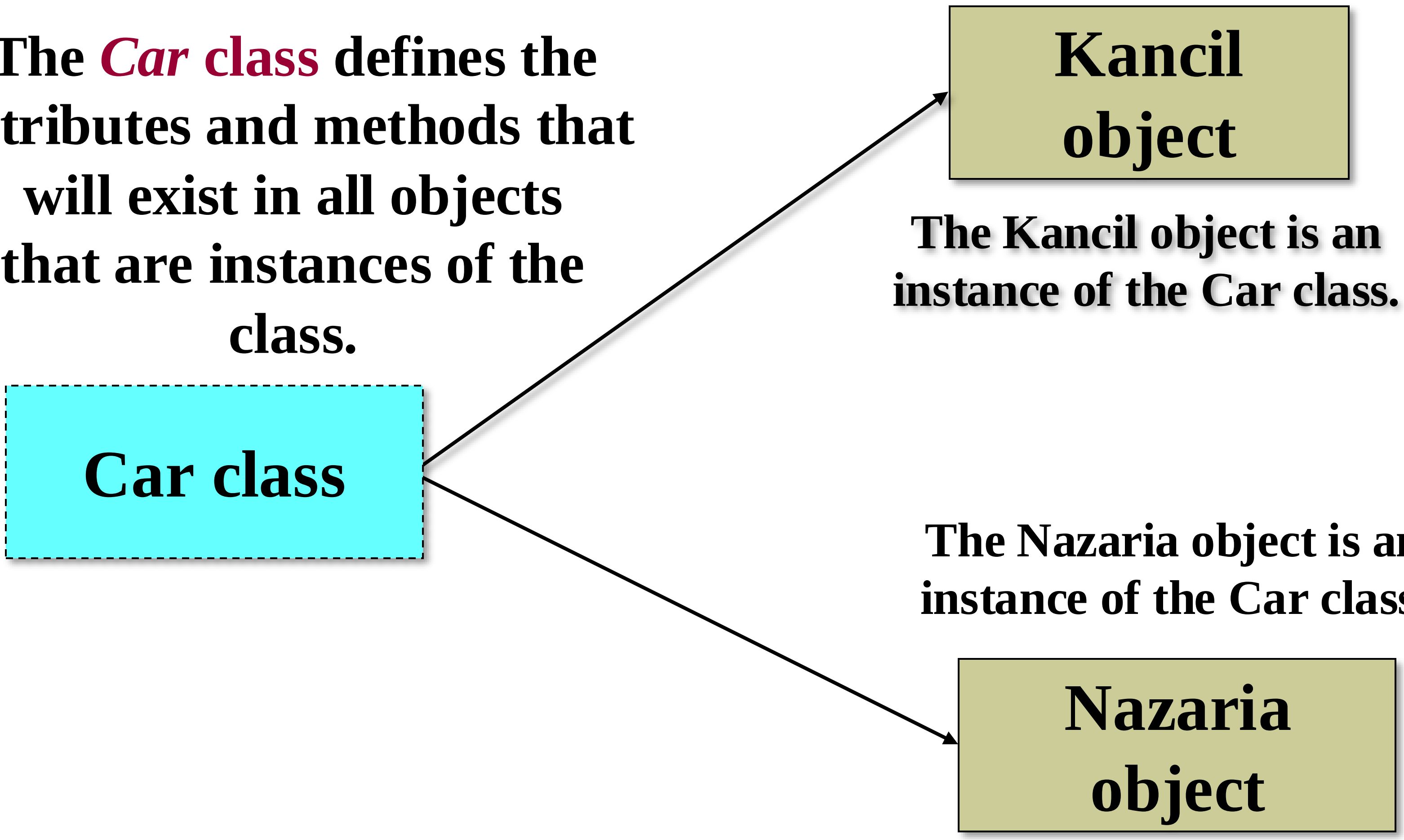
- Classes can be used to **instantiate** as many objects as are needed.

- Each object that is created from a class is called an **instance** of the class.

- A program is simply a collection of objects that interact with each other to accomplish a goal.

Classes and Objects

The *Car* class defines the attributes and methods that will exist in all objects that are instances of the class.



The Kancil object is an instance of the Car class.

The Nazaria object is an instance of the Car class.

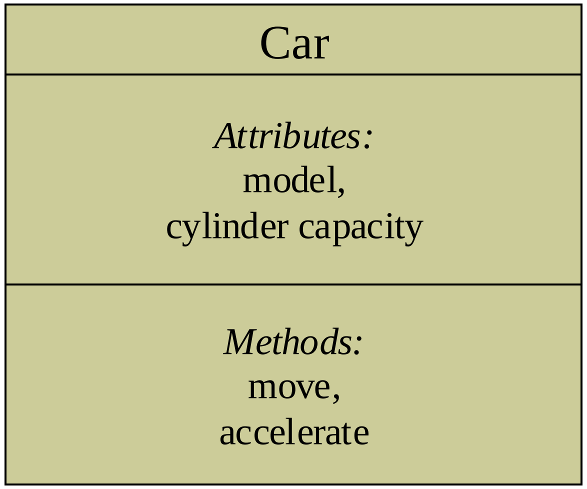
OOP Principles: Encapsulation

- **Encapsulation** is a key concept in working with objects: **Combining attributes and methods** in one package and hiding the implementation of the data from the user of the object.

Encapsulation:

Attributes/data
+
Methods/functions = Class

Example:
a car has attributes and methods below.



OOP Principles: Data Hiding

- **Data hiding** ensures methods **should not directly access** instance attributes in a class other than their own.

- Programs should interact with object attributes only through the object's methods.

- Data hiding is important for several reasons.
  - It protects attributes from accidental corruption by outside objects.
  - It hides the details of how an object works, so the programmer can concentrate on using it.
  - It allows the maintainer of the object to have the ability to modify the internal functioning of the object without “breaking” someone else's code.

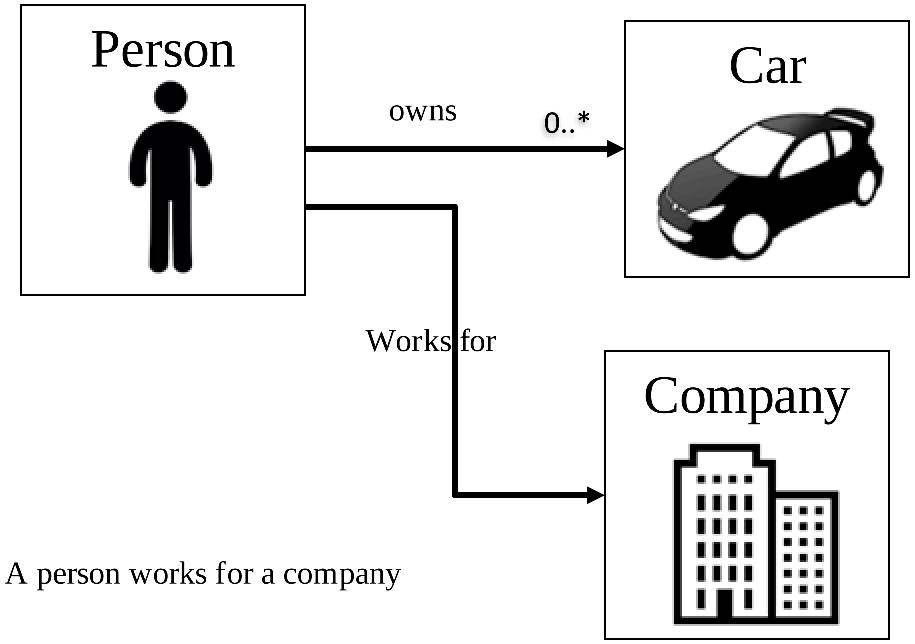
OOP Principles: Associations

- **Association**: relates classes to each other through their objects.

- Association can be, one to one, one to many, many to one, or many to many relationships.

Example:

A person can own several cars



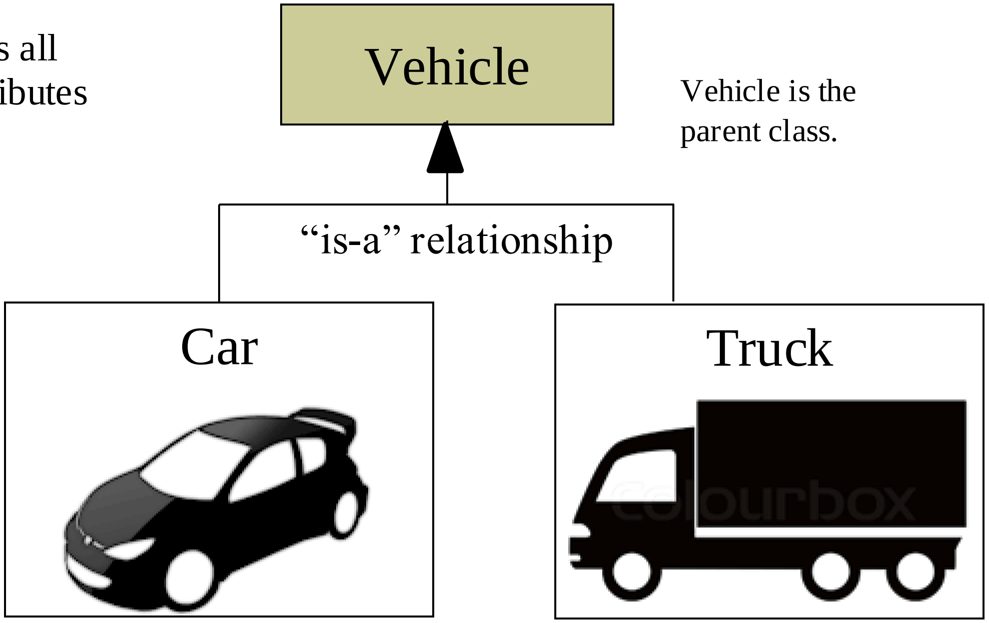
A person works for a company

OOP Principles: Inheritance

- Inheritance is the ability of one class to **extend** the capabilities of another.
  - it allows code defined in one class to be reused in other classes

Example:

Vehicle represents all of the generic attributes and methods of a vehicle.



Vehicle is the parent class.

Car and Truck are Specialized versions of a Vehicle.

Car and Truck are child classes of Vehicle.

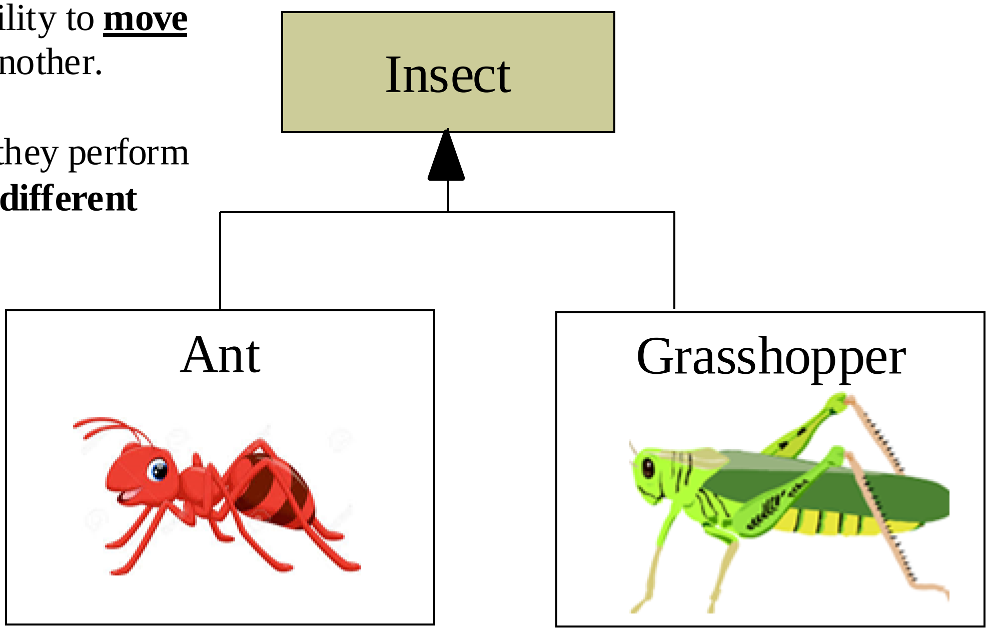
OOP Principles: Polymorphism

- Polymorphism is the ability of objects **performing the same actions differently**.

Example:

Insects have the ability to **move** from one point to another.

However, the way they perform their **movement is different**



Ants **move** by **crawling**

Grasshoppers **move** by **jumping**

Self-test: Introduction to Object Oriented Programming

- State the differences between procedural programming and Object Oriented Programming.
- What is an Object and what is a Class? What is the difference between them?
- What is an Attribute?
- What is a Method?
- What is encapsulation? How it relates to data hiding?
- What is association?
- What is inheritance? How it relates to polymorphism?

The Unified Modeling Language

Programming Technique II

(SCSJ1023)

Adapted from Tony Gaddis and Barret Krupnow (2016), Starting out with C++: From Control Structures through Objects

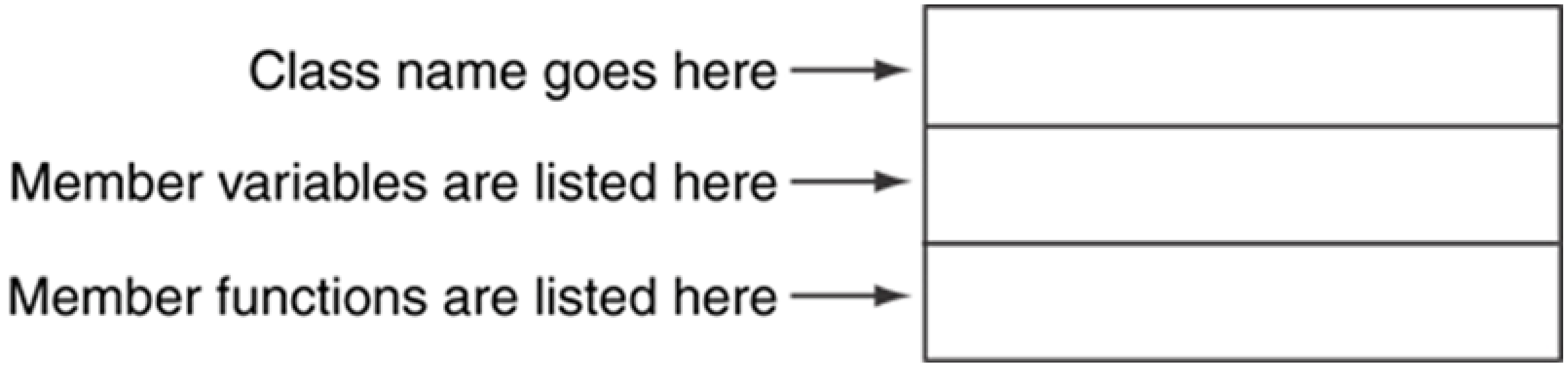
The Unified Modelling Language

- UML stands for Unified Modelling Language.

- The UML provides a set of standard diagrams for graphically depicting object-oriented systems

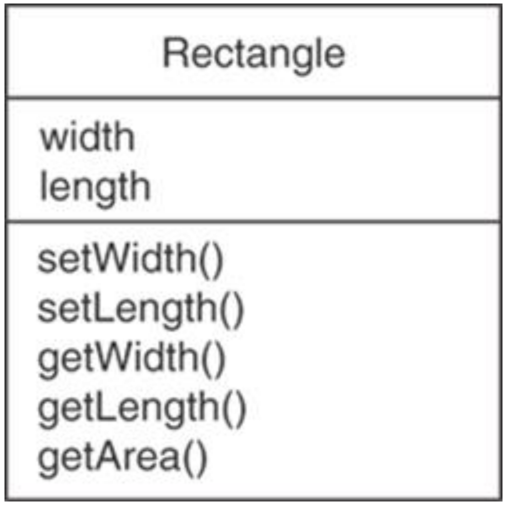
UML Class Diagram

- A UML diagram for a class has three main sections.


Class name goes here
Member variables are listed here
Member functions are listed here

Example: A Rectangle Class

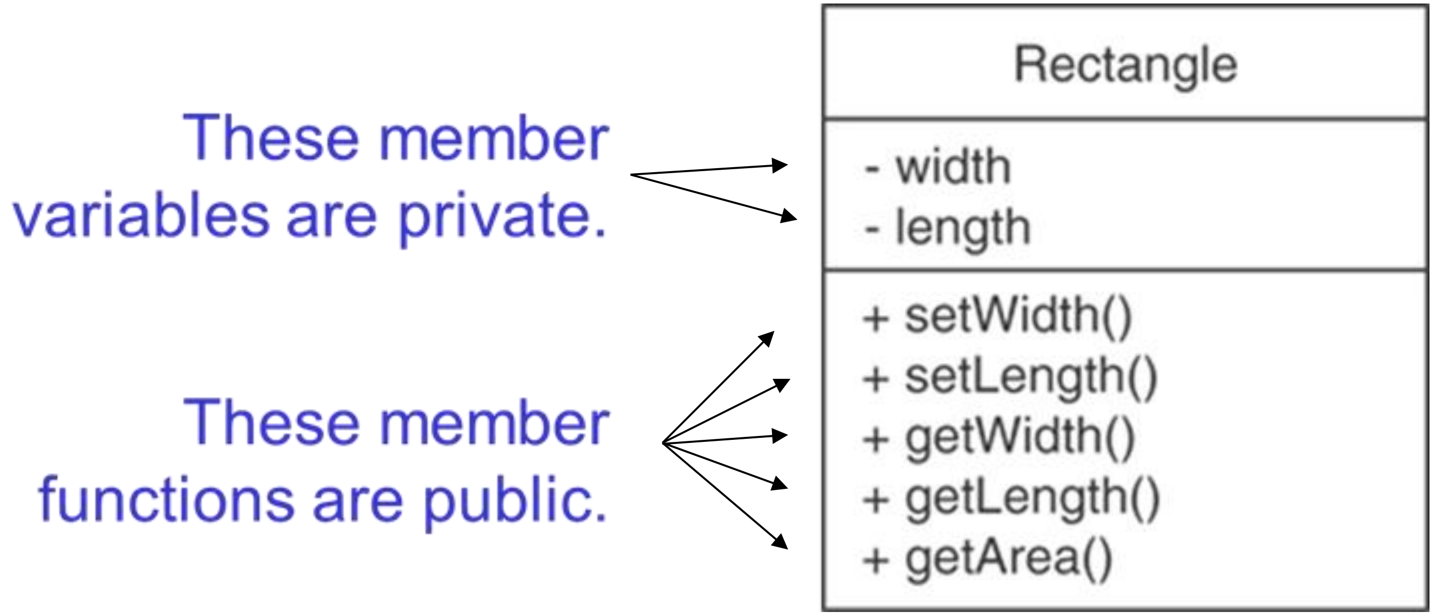
- A UML diagram for a class has three main sections.



```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      bool setWidth(double);
      bool setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

UML Access Specification Notation

- In UML you indicate a private member with a minus (-) and a public member with a plus(+).



UML Data Type Notation

- To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable.

```
- width : double
- length : double
```

UML Parameter Type Notation

- To indicate the data type of a function's parameter variable, place a colon followed by the name of the data type after the name of the variable.

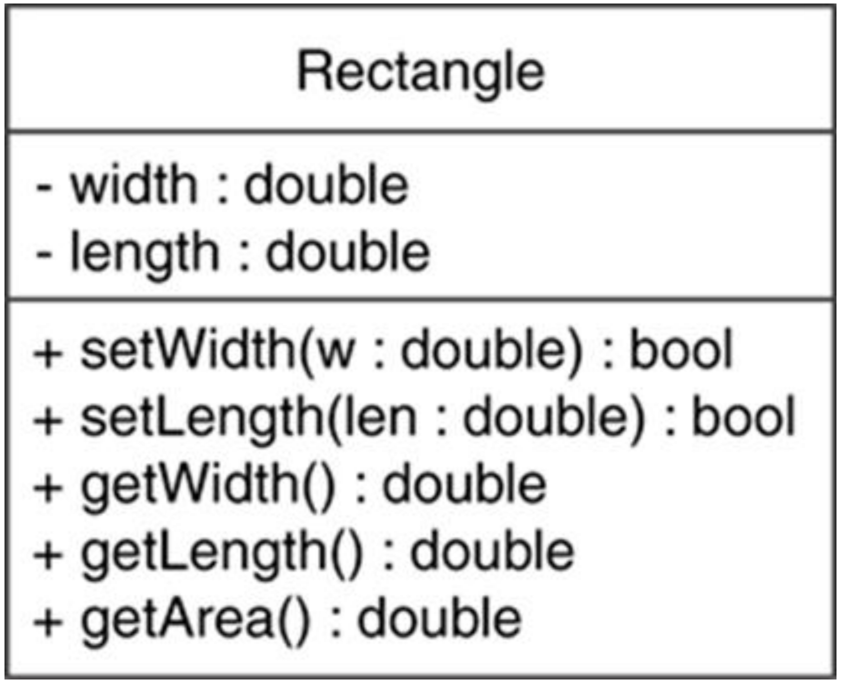
```
+ setWidth(w : double)
```

UML Function Return Type Notation

- To indicate the data type of a function’s return value, place a colon followed by the name of the data type after the function’s parameter list.

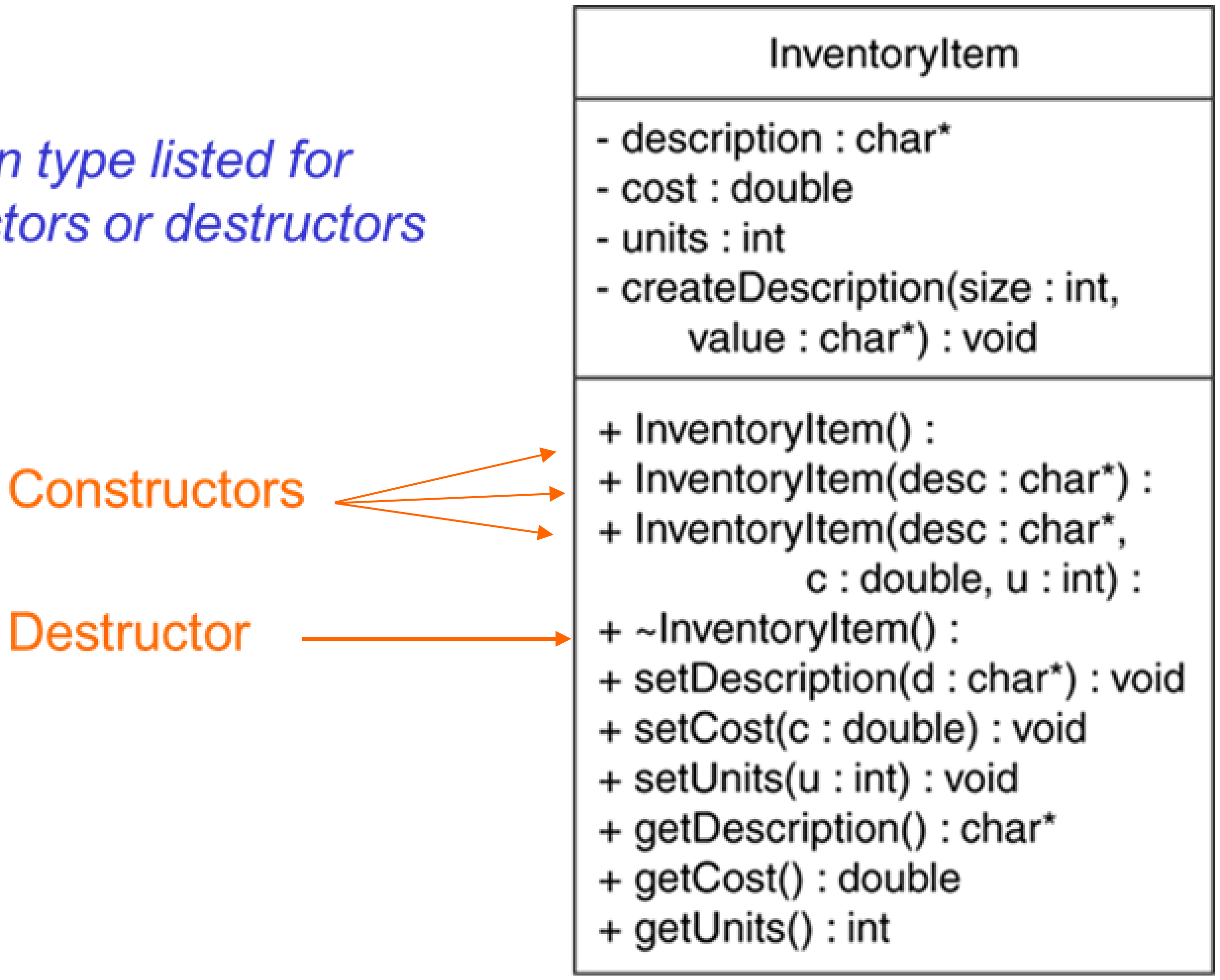
```
+ setWidth(w : double) : void
```

The Rectangle Class



Showing Constructors and Destructors

No return type listed for constructors or destructors



```
InventoryItem
- description : char*
- cost : double
- units : int
- createDescription(size : int,
      value : char*) : void
+ InventoryItem() :
+ InventoryItem(desc : char*) :
+ InventoryItem(desc : char*,
            c : double, u : int) :
+ ~InventoryItem() :
+ setDescription(d : char*) : void
+ setCost(c : double) : void
+ setUnits(u : int) : void
+ getDescription() : char*
+ getCost() : double
+ getUnits() : int
```