



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

200
1822 - 2022

Functions in C++

Man Geni Addysg Uwch yng Nghymru
Where Higher Education Began in Wales

www.uwtsd.ac.uk

Learning Objectives

- Functions- pass by value, pass by reference
- Pointers
- Constructors

Pass by Value

- When you pass an argument by value to a function, a copy of the original value is made and passed to the function.
- Any modifications made to the parameter inside the function do not affect the original value outside of the function.
- This method is generally used for basic data types like integers, floats, and characters, as well as user-defined data types when you want to work with a local copy of the data.

Example:

Imagine you are developing a program to calculate the sales tax for items in a shopping cart. Each item in the cart has its own price, and you want to calculate the sales tax for each item based on the item's price. However, you want to keep the original prices intact and return the sales tax for each item.

Example:

Imagine you are developing a program to calculate the sales tax for items in a shopping cart. Each item in the cart has its own price, and you want to calculate the sales tax for each item based on the item's price. However, you want to keep the original prices intact and return the sales tax for each item.

Passby Value

```
#include <iostream>
#include <vector>

// Function to calculate sales tax for an item by value
double calculateSalesTax(double itemCost)
{
    // Sales tax rate (10%)
    const double taxRate = 0.10;
    double taxAmount = itemCost * taxRate;

    return taxAmount;
}
```

```
int main() {
    // List of item costs in a shopping cart
    std::vector<double>
    cartItemCosts = {10.0, 20.0, 30.0};

    // Calculate & display the sales tax for each
    item
    for (double cost : cartItemCosts) {
        double tax = calculateSalesTax(cost);
        // Pass item cost by value
        std::cout << "Item Cost: £" <<
        cost << ", Sales Tax: £" << tax <<
        std::endl;
    }
    return 0;
}
```

Pass by Reference

```
int main()
```

```
{ . . .
```

```
    // Call the swap function, passing the addresses of 'x' and 'y'
```

```
    swap(&x, &y);
```

```
}
```

```
// Function to swap two integers
```

```
void swap(int* a, int* b)
```

```
{
```

```
-----
```

```
----- }
```

Pointers

- every piece of data stored at a specific memory address
- pointers allow you to store and manipulate these memory addresses.
- Memory addresses are essential for dynamic memory allocation, where you can allocate and deallocate memory as needed during program execution
- linked lists, trees, graphs, and hash tables, where elements often contain references to other elements.

Pointer Declaration and Initialization

- Declaration:

```
int* myPointer;
```

Pointers are declared with data types and initialized with addresses.

```
myPointer = &x;
```

 & is the "address-of" operator

Examples: **int* pInt = &myInt; char* pChar = &myChar;**

Simple Example: Swapping two numbers

```
#include <iostream>

// Define a NumberSwapper class
class NumberSwapper {
public:
    // Function to swap the two numbers using pointers
    void swapNumbers(int* a, int* b) {
        int temp = *a;
        // Dereference 'a' to access its value
        *a = *b;
        // Assign the value of 'b' to 'a'
        *b = temp;
        // Assign the stored value of 'a' (temp) to 'b'
    }
};
```

```
int main()
{
    int x = 5;
    int y = 10;

    std::cout << "Before swapping: x = " << x <<
    ", y = " << y << std::endl;

    NumberSwapper swapper;
    // Create a NumberSwapper object

    // Call the swapNumbers function using pointers
    swapper.swapNumbers(&x, &y);

    std::cout << "After swapping: x = " << x <<
    ", y = " << y << std::endl;
    return 0;
}
```

Simple Example: Swapping two numbers

```
#include <iostream>
```

```
// Define a NumberSwapper class
```

```
class NumberSwapper {
```

```
public:
```

```
// Function to swap the 2 numbers using pass by reference
```

```
void swapNumbers(int& a, int& b)
```

```
{
```

```
    int temp = a;
```

```
    a = b;
```

```
    b = temp;
```

```
}
```

```
};
```

```
int main()
```

```
{
```

```
    int x = 5;
```

```
    int y = 10;
```

```
    std::cout << "Before swapping: x = " << x << ",  
y = " << y << std::endl;
```

```
    NumberSwapper swapper;
```

```
// Create a NumberSwapper object
```

```
// Call the swapNumbers function using pass by reference  
swapper . swapNumbers(x, y);
```

```
    std::cout << "After swapping: x = " << x << ", y  
= " << y << std::endl;
```

```
    return 0;
```

```
}
```

Difference between references and raw pointers

References	Raw pointers
Syntax and usage: declared using the & symbol and are used like regular variables	Syntax and usage: declared using the * symbol and require explicit dereferencing with * to access the value they point to
Null or Uninitialized Values: must always be initialized with a valid object when declared	Null or Uninitialized: Pointers can be null (pointing to nothing) or uninitialized
Reassignment: A reference remains tied to the same object throughout its lifetime	Reassignment: Pointers can be reassigned to point to different objects during their lifetime.
Memory Management: References do not manage memory. They do not affect memory allocation or deallocation.	Memory Management: Pointers can be used for dynamic memory allocation and deallocation
Compatibility: used with objects of classes or structures	Compatibility: used with a wider range of data types, including built-in types, arrays, and user-defined types

Difference between references and raw pointers

References are often preferred for their simplicity, safety, and use with objects,

while raw pointers are more versatile and are required for tasks involving dynamic memory allocation and array manipulation.

Tutorial

Suppose you are designing a banking application for a local bank. Customers have savings accounts, and you are tasked with implementing a function that deducts a specific withdrawal amount from a customer's account balance. To ensure the account balance is updated directly and reflected outside the function, you decide to use pass by reference. Can you outline the key steps and code for this function, keeping in mind the principles of pass by reference and the need to handle potential scenarios such as insufficient funds?

```
class Customer {  
public:  
    std::string name;  
    double balance;  
  
    Customer(const std::string& customerName, double initialBalance)  
        : name(customerName), balance(initialBalance) {}  
  
    bool deductFromBalance(double amount) {  
        if (amount <= balance) {  
            balance -= amount;  
            return true;  
        } else {  
            std::cout << "Insufficient funds in " << name << "'s account." ;  
            return false;  
        } }  
};
```

```
int main()
{
    Customer custobj("Alice", 1000.0);

    // Deduct an amount from the balance
    double withdrawalAmount = 500.0;
    bool success = custobj.deductFromBalance(withdrawalAmount);

    if (success) {
        std::cout << "Withdrawal of £" << withdrawalAmount << " was successful.";
    } else {
        std::cout << "Withdrawal failed.";
    }
    return 0;
}
```

Constructors are special member functions used for initializing objects in C++.

Initialization: The primary purpose of a constructor is to initialize the attributes (also known as data members or fields) of an object to a **valid initial state**. This ensures that an object is ready for use as soon as it is created.

Without proper initialization, objects might contain unpredictable or garbage data, which can lead to unexpected behavior and errors in your program.

constructors have the same name as the class and are executed automatically when an object is created.

No Return Type: Constructors don't have a return type, not even void. They are automatically invoked when an object is created, and their main job is to set up the object's state.

Types

- **Default Constructor:** Initializes the object with default values if no arguments are provided during object creation.
- **Parameterized Constructor:** Accepts one or more arguments during object creation, allowing customization of the object's initial state.
- **Copy Constructor:** Creates a new object as a copy of an existing object, often used for object duplication.

Default Constructor

- what a default constructor is.
- how to create a class with a default constructor.
- when and why you might want to use a default constructor.

Parameterized Constructors

- create constructors with parameters.
- benefits of parameterized constructors in object initialization.

Imagine you're designing a **Person** class to represent individuals in a database. A **Person** has attributes such as a name, age, and address. In this scenario, you want to provide a default constructor to create instances of the **Person** class with reasonable default values when specific information is not provided.

Here's a C++ example with a **Person** class:

```

#include <iostream>
#include <string>

class Person {
public:
    // Default constructor
    Person() {
        name = "John Doe";
        age = 30;
        address = "123 Main St";
    }

    // Parameterized constructor
    Person(const std::string& name, int age, const std::string&
address) {
        this->name = name;
        this->age = age;
        this->address = address;
    }

    // Member function to display person's information
    void displayInfo() {
        std::cout << "Name: " << name << ", Age: " << age << ",
Address: " << address << std::endl;
    }

```

```

int main() {
    // Create a person with a default constructor
    Person person1;

    std::cout << "Person 1 (Default
Constructor)";
    person1.displayInfo();

    // Create a person with a parameterized constructor

    Person person2("Alice", 25, "456 Elm St");
    std::cout << "\nPerson 2
(Parameterized Constructor)";

    person2.displayInfo();
    return 0;
}

```

Copy constructor

```
// Parameterized constructor
```

```
Person(const std::string& n) : name(n) {}
```

```
// Copy constructor
```

```
Person(const Person& other) {
```

```
    name = other.name;
```

```
}
```

```
};
```

```
// in main()
Person person1("Alice");
Person person2 = person1;
```

Constructor Overloading

- concept of constructor overloading.
- how to have multiple constructors with different parameter lists.
- advantages of constructor overloading in creating flexible objects.

- Constructor overloading is a technique in object-oriented programming where a class has multiple constructors, each with a different parameter list.
- The choice of which constructor to use depends on the arguments provided when creating an object

Example:

Imagine you are designing a class called Book to represent books in a library. A book can have different types of attributes, and you want to create Book objects with various sets of information based on the available data

```
#include <iostream>
#include <string>
```

```
class Book {
public:
    // Default constructor
    Book() {
        title = "Unknown";
        author = "Unknown";
        publicationYear = 0;
    }
}
```

```
2. // Constructor with essential
information
    Book(std::string bookTitle, std::string
        bookAuthor)
    {
        title = bookTitle;
        author = bookAuthor;
        publicationYear = 0;
        // Initialize to unknown year
    }
}
```

```
3.
    // Constructor with full
information
    Book(std::string bookTitle,
        std::string bookAuthor, int year)
    {
        title = bookTitle;
        author = bookAuthor;
        publicationYear = year;
    }
}
```

```
4. void displayInfo() {
    std::cout << "Title: " << title << std::endl;
    std::cout << "Author: " << author <<
std::endl;
    std::cout << "Publication Year: " <<
publicationYear << std::endl;
}
```

```
5. private:
    std::string title;
    std::string author;
    int publicationYear;
};
```

```
6. int main() {  
    // Create Book objects using different constructors  
    Book book1; // Using the default constructor  
    Book book2("To Kill a Mockingbird", "Harper Lee");  
    // Using the 2-parameter constructor  
    Book book3("The Great Gatsby", "F. Scott Fitzgerald", 1925);  
    // Using the 3-parameter constructor  
  
    // Display book information  
    std::cout << "Book 1 (Default Constructor):" << std::endl;  
    book1.displayInfo();  
  
    std::cout << "\nBook 2 (2-Parameter Constructor):" << std::endl;  
    book2.displayInfo();  
  
    std::cout << "\nBook 3 (3-Parameter Constructor):" << std::endl;  
    book3.displayInfo();  
  
    return 0;  
}
```



output

Book 1 (Default Constructor):

Title: Unknown

Author: Unknown

Publication Year: 0

Book 2 (2-Parameter Constructor):

Title: To Kill a Mockingbird

Author: Harper Lee

Publication Year: 0

Book 3 (3-Parameter Constructor):

Title: The Great Gatsby

Author: F. Scott Fitzgerald

Publication Year: 1925

Tutorial

Imagine you are developing a class called **Car** to represent vehicles. The **Car** class has two attributes: **brand** and **model**. You want to provide constructors to initialize **Car** objects in various ways.

Can you provide an example of a constructor that initializes a **Car** object with the brand and model, and another constructor that initializes a **Car** object with just the brand?

```
#include <iostream>
#include <string>
class Car {
public:
    // Constructor with brand and model
    Car(const std::string& carBrand, const std::string& carModel) {
        brand = carBrand;
        model = carModel;
    }

    // Constructor with brand only
    Car(const std::string& carBrand) {
        brand = carBrand;
        model = "N/A"; // Set the model to "N/A" (Not Available) by default
    }

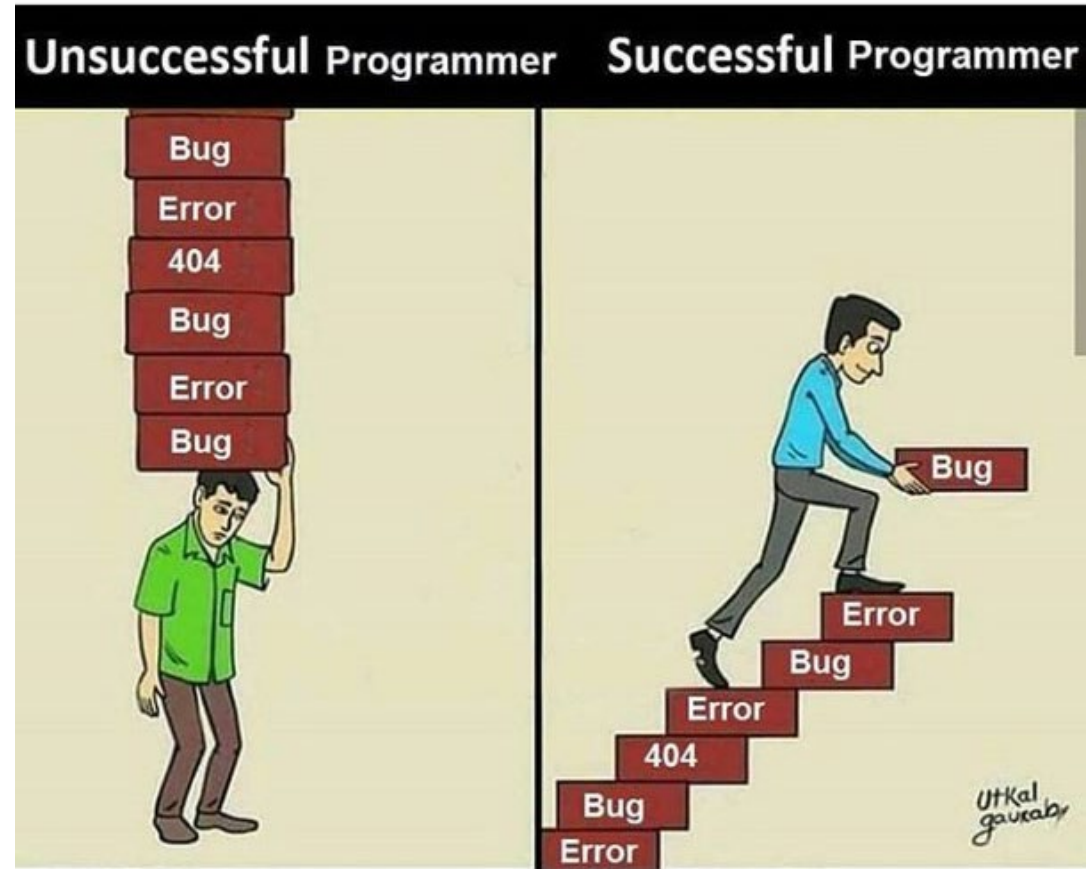
    void displayInfo() {
        std::cout << "Brand: " << brand << std::endl;
        std::cout << "Model: " << model << std::endl;
    }
private:
    std::string brand;
    std::string model;
};
```

```
int main()
{
    // Create Car objects using different constructors
    Car car1("Toyota", "Camry"); // Initialize with brand and model
    Car car2("Honda"); // Initialize with brand only

    // Display car information
    std::cout << "Car 1 (Brand and Model):" << std::endl;
    car1.displayInfo();

    std::cout << "\nCar 2 (Brand Only):" << std::endl;
    car2.displayInfo();

    return 0;
}
```



Happy Coding!!!

Next Session.....

Inheritance