Modern Programming Principles & Practice

Feb-May 2025

sruthi.padathara@dorset.ie

Contents

- Arrays: Definition, advantages
- Array types
- Single dimension
- Double dimension
- Declaration, accessing array data
- Implementation of array operations
- Definition, advantages, types of functions, classification
- Implementing various kinds of functions
- Inline functions
- Defining a Class ,creating Objects
- Accessing Data Members using objects
- Calling Member Functions using objects
- Implementing Array of Objects
- Scope resolution operator
- access specifiers(private, public,protected)
- Implementing Static Data Members
- Implementing Static Member Functions

Arrays: Definition, advantages

What are C++ Arrays?

- **Definition**: A collection of similar data types stored in contiguous memory locations.
- **Purpose**: Efficient for storing and manipulating large datasets.
- **Can store**: Integers, characters, strings, or complex structures.

Uses of C++ Arrays

- **Store multiple values** in a single variable.
- **Sorting**: Algorithms like selection sort and bubble sort.
- **Searching**: Linear search, binary search.

Declaration in C++

```
data_type array_name[size];

int arr[5]; // Array of 5 integers
```

Array Initialization in C++

```
data_type array_name[] = {value1, value2, value3, ...};

int arr[] = {1, 2, 3, 4, 5};
```

Accessing Array Elements

Using index:
Arrays are accessed using the index starting from 0.

Syntax:

```
array_name[index]
int arr[5] = {1, 2, 3, 4, 5};
cout << arr[0];  // Outputs 1
```

Advantages of C++ Arrays

- **Easy to Use**: Store multiple values in one variable.
- **Random Access**: Access any element directly using the index.
- **Memory Efficiency**: Contiguous memory block allocation.
- **Performance**: Fast element access with minimal overhead.

Disadvantages of C++ Arrays

- **Fixed Size**: Size is fixed at compile-time and cannot be resized dynamically.
- **Lack of Flexibility**: Cannot resize or modify array size during runtime.
- **Overhead**: Fixed memory allocation may lead to wasted space.

Array types

One-Dimensional Array (1D Array)

Stores elements in a single row.

```
data_type array_name[size];
```

```
int arr[5] = {1, 2, 3, 4, 5};
```

Two-Dimensional Array (2D Array)

Stores data in rows and columns (matrix format).

```
data_type array_name[rows][columns];
```

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Multi-Dimensional Array (3D and higher)

```
data_type array_name[size1][size2][size3];
```

```
int cube[2][2][2] = {
    {{1, 2}, {3, 4}},
    {{5, 6}, {7, 8}} };
```

Accessing Array Data Using Length in C++

In C++, the length of an array can be determined using `sizeof(array) / sizeof(array[0])`. This helps iterate over the array dynamically instead of using a hardcoded size.

```
int length = sizeof(array) / sizeof(array[0]);

int numbers[] = {10, 20, 30, 40, 50};

// Calculate length of array

int length = sizeof(numbers) / sizeof(numbers[0]);

for (int i = 0; i < length; i++) {

cout << numbers[i] << " ";

}
```

Method 2:

```
// Pass array length to function

int length = sizeof(numbers) / sizeof(numbers[0]);

printArray(numbers, length);
```

```
int main() {
    int arr[] = {10, 20, 30, 40, 50};

    cout << "Size of entire array: " << sizeof(arr) << "
bytes" << endl;
    cout << "Size of one element: " << sizeof(arr[0]) << "
bytes" << endl;

    int length = sizeof(arr) / sizeof(arr[0]);
    cout << "Number of elements in array: " << length
<< endl;

    return 0;
}
```

Output

Size of entire array: 20 bytes
Size of one element: 4 bytes
Number of elements in array: 5

Accessing Multi-Dimensional Arrays Using Length in C++

For a **2D array**:

```
int rows = sizeof(array) / sizeof(array[0]);  // Number of rows

int cols = sizeof(array[0]) / sizeof(array[0][0]);  // Number of columns
```

For a **3D array**:

```
int depth = sizeof(array) / sizeof(array[0]);

int rows = sizeof(array[0]) / sizeof(array[0][0]);

int cols = sizeof(array[0][0]) / sizeof(array[0][0][0]);

for (int i = 0; i < depth; i++) { cout << "Depth " << i + 1 << ":\n";
        for (int j = 0; j < rows; j++) {
                for (int k = 0; k < cols; k++) {
                        cout << cube[i][j][k] << " "; }
                cout << endl; }
        cout << endl; }
```

Write a C++ program to perform the following array operations:

1. **Display the array elements.**
2. **Insert a new element** at a specific position in the array.
3. **Delete an element** from a given position.
4. **Search for an element** in the array and display its index if found.
5. **Update an element** at a specific position with a new value.

The program should be **menu-driven**, allowing the user to repeatedly choose operations until they decide to exit. The array should have a fixed maximum size, and the program should ensure that insertions and deletions are handled correctly.

How should it work?

`displayArray()` – Prints the current elements in the array.

`insertElement()` – Inserts an element at a specific position and shifts elements to the right.

`deleteElement()` – Removes an element from a specific position and shifts elements to the left.

`searchElement()` – Searches for a given element and returns its index.

`updateElement()` – Updates an existing element at a specific index.

Menu-driven system – Allows users to perform operations interactively.

Functions in C++: Definition, Advantages, Types, and Implementation

A **function** in C++ is a self-contained block of code that performs a specific task. It helps in breaking down complex problems into smaller, manageable tasks, improving code reusability and readability.

Syntax of a Function in C++:

```
return_type function_name(parameters) {
    // Function body
    return value; // If required
}
```

Advantages of Functions in C++

Code Reusability – Functions prevent code duplication by allowing reuse.

Modularity – Breaking down the program into smaller functions improves organization.

Ease of Debugging – Since functions work independently, debugging is simpler.

Improved Readability – Function-based programs are easier to read and maintain.

Efficient Memory Usage – Reduces memory consumption by avoiding redundant code.

Types of Functions in C++

Library Functions – Predefined functions in C++ standard libraries, e.g., `sqrt()`, `abs()`, `pow()`, etc.
User-Defined Functions – Functions written by programmers for custom tasks.

User-defined functions can be further classified into:

Functions with No Arguments and No Return Value

Functions with Arguments and No Return Value

Functions with No Arguments but a Return Value

Functions with Arguments and a Return Value

Function Overloading (Multiple functions with the same name but different parameters)

Inline Functions (Optimized functions for small operations)

Recursive Functions (Functions that call themselves)

Friend Functions (Functions that access private members of a class)

Inline Function (For Small, Frequently Used Functions)

An **inline function** in C++ is a function that is defined with the `inline` keyword. The compiler replaces the function call with the actual code of the function during compilation. This is done to eliminate the overhead of function calls, such as stack operations, and to improve the performance of small functions by avoiding function call overhead.

```
inline return_type function_name(parameters) {

    // function body

    return value;  // if applicable

}
```

```
inline int multiply(int a, int b) {
    return a * b;  // Simple inline function to multiply two numbers
}
int main() {
    int result = multiply(4, 5);  // Function call to inline function
    cout << "The result of multiplication is: " << result << endl;
    return 0;
}
```

The multiply function is marked as inline.

When the program runs, instead of calling the function, the compiler will replace the call to multiply(4, 5) with 4 * 5 directly in the main() function.

This reduces function call overhead and enhances performance.

Advantages of Inline Functions

Performance Improvement:
Inline functions can improve performance by eliminating the function call overhead. The function code is inserted directly into the place where it is called, which can lead to faster execution, especially for small functions.

No Function Call Overhead:
Traditional functions have overhead due to function call setup and return. Inline functions are expanded in place, eliminating this overhead.

Better Code Optimization:
The compiler has a better understanding of the inline function and can optimize the code more efficiently, often resulting in faster execution.

Code Simplification:
When a small function is defined inline, it can be written and used in the same way as a normal function, making the code easier to read and understand.

Disadvantages of Inline Functions

- **Code Bloat:**
  If an inline function is called many times, it increases the size of the compiled code, known as "code bloat." This can lead to a larger binary size, which is especially problematic for memory-constrained applications.
- **Compilation Time:**
  Since the code is substituted at each call site, it can increase the compilation time and the size of the object file.
- **Limited Use:**
  Inline functions should only be used for small, frequently called functions (e.g., simple getter functions). Complex functions can lead to inefficient use of inline and may not provide the desired performance benefit.
- **Compiler Decision:**
  The inline keyword is a **suggestion** to the compiler, not a command. The compiler may choose not to inline a function if it deems it inappropriate, such as for large functions.

When to Use Inline Functions?

- **Short Functions:**
  Functions that are small and perform simple tasks (like addition, multiplication, etc.), which are called frequently, are ideal candidates for inlining.
- **Getters and Setters:**
  Functions that only retrieve or set the values of member variables in classes can be efficiently inlined.
- **Performance-Critical Sections:**
  If you have a section of code that will be executed repeatedly in a loop or other frequently called functions, making the function inline can help with performance.

Inline Member Function

An **inline member function** in C++ is a function that is defined inside the class definition itself. The `inline` keyword suggests to the compiler to insert the function's body directly at the point of function call, avoiding the overhead of function calls. This can improve performance for small functions, especially those that are called frequently.

Inside the Class Definition:

```
class ClassName {
public:
    inline return_type function_name(parameters) {
        // function body
        return value;
    }
};
```

Outside the Class Definition:

```
class ClassName {
public:
    return_type function_name(parameters); // declaration
};

// Outside the class definition
inline return_type ClassName::function_name(parameters) {
 // function body
    return value;
}
```

Recursive Function in C++

A **recursive function** is a function that **calls itself** directly or indirectly to solve a problem. Recursion is often used to break down complex problems into smaller, more manageable subproblems.

```
return_type function_name(parameters) {
    if (base_condition) {
        return result;  // Base case: Stops recursion
    } else {
        return function_name(modified_parameters);  // Recursive call
    }
}
```

Base Case: A condition that stops the recursion.

Recursive Call: The function calls itself with modified arguments to move towards the base case.\

Advantages of Recursion

Simplifies complex problems like tree traversal, backtracking, etc.
Reduces code length by eliminating loops and repetitive code.
Better readability and **natural problem-solving** approach.

Disadvantages of Recursion

Consumes more memory due to function call stack.
Slower execution compared to iteration due to overhead.
Risk of infinite recursion if the base case is missing or incorrect.

Key Takeaways

A **recursive function** is a function that calls itself.
Recursion requires a **base case** to stop execution.
Direct recursion occurs when a function calls itself, while **indirect recursion** happens when two or more functions call each other.
Use recursion wisely, as excessive recursion can lead to **stack overflow**.

Friend Function (Accessing Private Data of a Class)

A **friend function** in C++ is a function that is **not a member** of a class but is allowed to access the private and protected members of the class. It is declared **inside the class** using the `friend` keyword.

```
class ClassName {

private:

    int privateData;  // Private member


public:

    friend void friendFunction(ClassName obj);  // Friend function declaration

};
```

The **friend function is declared inside the class** but is **defined outside** the class.

It does **not use the dot operator (.)** to access class members directly; instead, it accesses them through the object.

Key Features of Friend Functions

Not a member function of the class.
Declared **inside the class** but defined **outside** it.
Uses the friend keyword.
Can **access private and protected members** of the class.
It **cannot be called using an object**, but it takes an object as an argument.

Advantages of Friend Functions

Access to private data of a class without being a member.
Improves encapsulation by providing controlled access to private members.
Can **be used to access multiple classes' data** in a single function.

Disadvantages of Friend Functions

Breaks data encapsulation, as it allows access to private members.
Not bound to class scope, meaning it doesn't have this pointer.
Can lead to security risks if misused, as private members become accessible.

Defining a Class ,creating Objects, Accessing Data Members using objects,Calling Member Functions using objects

A **class** in C++ is a user-defined data type that serves as a blueprint for creating objects. It groups related variables (data members) and functions (member functions) together.

```
class ClassName {

public:

    // Data members (variables)

    // Member functions (methods)

};
```

Key Points

Defining a Class: Use the class keyword to define a class with data members and member functions.
Creating Objects: An object is an instance of a class.
ClassName objectName;
Accessing Data Members: Use the dot (.) operator to assign or retrieve values.
`objectName.dataMember = value;`
Calling Member Functions: Use the dot (.) operator to call functions within the class.
objectName.memberFunction();

Implementing Array of Objects

An **array of objects** in C++ is a collection of multiple objects of the **same class** stored in a **contiguous memory location**. It allows us to manage multiple objects efficiently using array indexing.

```
class ClassName {

    // Class members

};


int main() {

    ClassName objArray[size];  // Declaring an array of objects

}
```

Here, objArray is an array that holds multiple **instances of a class**.

The size of the array defines **how many objects** it contains.

Scope Resolution Operator (::)

The **scope resolution operator (::)** in C++ is used to access **global variables, class members, and namespace members** when there are conflicts with local variables or when defining functions outside a class.

```
class ClassName {

public:

    void functionName();  // Function declaration

};



// Using scope resolution operator to define the function outside the class

void ClassName::functionName() {

    // Function definition

}
```

Use Cases of Scope Resolution Operator

Accessing Global Variables When They Conflict with Local Variables

If a local variable has the same name as a global variable, the :: operator is used to access the **global** version.

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

int main() {
    int num = 20;  // Local variable
    cout << "Local num: " << num << endl;      // Output: 20
    cout << "Global num: " << ::num << endl;  // Output: 10
    return 0;
}
```

Defining Class Functions Outside the Class

```
class Car {
public:
    void show();  // Function declaration
};

// Function definition outside the class
void Car::show() {
    cout << "This is a car!" << endl;
}
```

```
int main() {
    Car myCar;
    myCar.show();
    return 0;
}
```

Accessing Namespace Members

```
namespace First {
    int value = 10;
}

namespace Second {
    int value = 20;
}

int main() {
    cout << "First namespace value: " << First::value << endl;
    cout << "Second namespace value: " << Second::value << endl;
    return 0;
}
```

access specifiers(private, public,protected)

Access Specifier	Accessible inside class	Accessible outside class	Accessible in derived class
public	Yes	Yes	Yes
private	Yes	No	No
protected	Yes	No	Yes

Why Use `private`?

- **Encapsulation:** Prevents direct access to data.
- **Data security:** Users cannot modify the balance directly.

Why Use `protected`?

- Useful in **inheritance** when child classes need access but external functions should not.

Implementing Static Data Members

- **Declared inside the class using static keyword**.
- **Defined outside the class** (memory allocated only once).
- **Shared among all objects** of the class.

```
class Counter {

private:
    static int count;  // Static data member
public:

    void increment() {
        count++;
    }
    void showCount() {
        cout << "Count: " << count << endl;

    }};

// Define static member outside the class

int Counter::count = 0;
```

```
int main() {

    Counter obj1, obj2;

    obj1.increment();

    obj1.showCount();  // Output: Count: 1



    obj2.increment();

    obj2.showCount();  // Output: Count: 2 (same
static variable shared)



    return 0;

}
```

Static Member Functions

- **Declared inside the class using `static` keyword**.
- **Can access only `static` data members** (because non-static members belong to objects).
- **Can be called using the class name (no need for an object).**

```
class Math {
private:
    static int value;  // Static variable

public:
    static void setValue(int v) {  // Static function
        value = v;
    }

    static void showValue() {  // Static function
        cout << "Value: " << value << endl;
    }
};

// Define static member outside the class
int Math::value = 0;
```

```
int main() {
    Math::setValue(10);  // Calling static function without
object
    Math::showValue();   // Output: Value: 10

    return 0;
}
```

Key Differences

Feature	Static Data Members	Static Member Functions
Memory Allocation	Stored in **class memory**, shared among objects	No memory for each object
Accessibility	Accessed via **objects** or **class name**	Accessed via **class name only**
Access to Members	Can be accessed by **all** class members	Can access **only static data members**
Initialization	**Declared in class**, **defined outside**	Defined normally inside the class

Static Data Members: Shared across all objects, defined outside the class.
Static Member Functions: Can access only static members, called using class name.

Why sizeof(array) Gives the Wrong Size in C++?

```
void sizeArray(int array[]) {
    int length = sizeof(array);  // Wrong size
}
```

Problem: sizeof(array) does **not** return the correct array size.

Why? When you pass an array to a function, it **decays into a pointer** (int*), losing size information.

sizeof(array) in this function actually returns the size of a pointer, not the actual array size.

Correct Way to Find Array Size

1. Pass Array with Its Size

A common fix is to pass the array **with its size**:

```
void sizeArray(int array[], int size) {
    cout << "Array size: " << size << endl;
}
```

Use std::vector Instead

```
#include <vector>

void sizeArray(const std::vector<int>& arr) {
    cout << "Array size: " << arr.size() << endl;
}

int main() {
    std::vector<int> array = {1, 2, 3, 4, 5};
    sizeArray(array);  // No need to pass size separately
}
```

Don't use sizeof(array) inside a function for a parameter array → It gives the size of a pointer.
Pass the array with its size explicitly → sizeof(array) / sizeof(array[0]) before passing.
Use std::vector for a more modern approach with automatic size handling.