C++ Programming For Absolute Beginners

Sahas Talasila

Contents

Introduction

In this guide, we will combine three major areas in C++ programming for beginners:

- **Object-Oriented Programming (OOP)** concepts
- **Dynamic Memory Allocation** (using `new` and `delete`)
- **Basic Algorithms** (searching and sorting)

By understanding how these components fit together, you can design more powerful and flexible classes that manage data and implement algorithms to handle that data.

Review of OOP Concepts

2.1 OOP Pillars (Recap)

1. **Encapsulation:** Grouping data and functions (methods) into a single unit (class), hiding implementation details.
2. **Abstraction:** Exposing only essential features and hiding complex details.
3. **Inheritance:** Reusing code by forming a relationship between base and derived classes.
4. **Polymorphism:** Providing different implementations (many forms) under a single interface, especially via virtual functions.

Dynamic Memory Allocation in C++

3.1 Why Do We Need Dynamic Memory?

Sometimes, we do not know the amount of memory needed at compile time. For example, a program that asks the user for how many elements to store and then allocates exactly that amount during run time.

3.2 Using `new` and `delete`

- `new`: allocates memory on the *heap* for a variable or array.
- `delete`: frees memory previously allocated with `new` (for single objects).
- `delete[]`: frees memory previously allocated with `new[]` (for arrays).

```cpp
// Example: dynamic allocation
#include <iostream>


int main() {
    int n;
    std::std::cout << "Enter number of elements: ";
    std::cin >> n;

    // Dynamically allocate an array of size n
    int* arr = new int[n];

    // Input elements
    for(int i = 0; i < n; i++){
        std::cin >> arr[i];
    }

    // Output elements
    for(int i = 0; i < n; i++){
        std::cout << arr[i] << " ";
```

```
    }
    std::cout << std::endl;

    // Free the allocated memory
    delete[] arr;

    return 0;
}
```

Review of Basic Algorithms

4.1 Searching

- **Linear Search:** Checks each element sequentially until the target is found or we reach the end.
- **Binary Search:** Requires a sorted array. Repeatedly divides the search space in half until the target is found or the subarray size is zero.

4.2 Sorting

- **Bubble Sort:** Compares adjacent pairs, swapping if out of order, repeated until no swaps.
- **Selection Sort:** Finds the minimum (or maximum) in the unsorted portion and places it in its correct position.
- **Merge Sort:** Uses divide and conquer: splits the array, recursively sorts each half, then merges the halves.

Combining OOP, Dynamic Memory, and Algorithms

5.1 Designing a Class with Dynamic Memory

Let us create a class called `DynamicArray` that:

- Allocates memory dynamically for an array of integers.
- Stores the current size (number of elements).
- Provides methods for insertion, searching, sorting, etc.

Step 1: Class Definition and Constructor

- Private members:
  - `int *data;` (pointer to an integer array)
  - `int size;` (number of elements)
- Public methods:
  - A constructor to allocate memory
  - A destructor to free memory
  - Methods for searching (e.g., `linearSearch`, `binarySearch`)
  - Methods for sorting (e.g., `bubbleSort`, `mergeSort`)

```
#include <iostream>


class DynamicArray {
```

```
private:
    int* data;
    int size;

    // Helper function for merge sort
    void merge(int left, int mid, int right);

    // Recursive merge sort function
    void mergeSortRec(int left, int right);

public:
    // Constructor
    DynamicArray(int n);

    // Destructor
    ~DynamicArray();

    // Set element at index
    void setElement(int index, int value);

    // Get element at index
    int getElement(int index) const;

    // Get array size
    int getSize() const;

    // Linear Search
    int linearSearch(int target) const;

    // Binary Search
    int binarySearch(int target) const;

    // Sorting: Bubble Sort
    void bubbleSort();

    // Sorting: Merge Sort
    void mergeSort();
};

// Implementation of methods will follow
```

Step 2: Implementing the Constructor and Destructor

- **Constructor:** Allocates a dynamic array of size `n`.
- **Destructor:** Frees that array.

```
// Constructor
DynamicArray::DynamicArray(int n) {
    size = n;
    data = new int[size];
    // Optionally initialize elements to 0
    for(int i = 0; i < size; i++) {
        data[i] = 0;
    }
}

// Destructor
DynamicArray::~DynamicArray() {
    delete[] data;
}
```

Step 3: Basic Get/Set Methods

```
void DynamicArray::setElement(int index, int value) {
    if(index >= 0 && index < size) {
        data[index] = value;
    } else {
        std::cout << "Index out of bounds!\n";
    }
}

int DynamicArray::getElement(int index) const {
    if(index >= 0 && index < size) {
        return data[index];
    } else {
        std::cout << "Index out of bounds!\n";
        return -1; // or some error code
    }
}

int DynamicArray::getSize() const {
    return size;
}
```

Step 4: Searching Methods

```
int DynamicArray::linearSearch(int target) const {
    for(int i = 0; i < size; i++) {
        if(data[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Binary Search We assume the array is already sorted for binary search to work correctly.

```
int DynamicArray::binarySearch(int target) const {
    int left = 0;
    int right = size - 1;

    while(left <= right) {
        int mid = (left + right) / 2;
        if(data[mid] == target) {
            return mid;
        } else if(data[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Step 5: Sorting Methods

```
void DynamicArray::bubbleSort() {
    bool swapped;
    do {
        swapped = false;
        for(int i = 0; i < size - 1; i++) {
```

```
            if(data[i] > data[i+1]) {
                // swap
                int temp = data[i];
                data[i] = data[i+1];
                data[i+1] = temp;
                swapped = true;
            }
        }
    } while(swapped);
}
```

```
void DynamicArray::merge(int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temporary arrays
    int* L = new int[n1];
    int* R = new int[n2];

    // Copy data
    for(int i = 0; i < n1; i++) {
        L[i] = data[left + i];
    }
    for(int j = 0; j < n2; j++) {
        R[j] = data[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while(i < n1 && j < n2) {
        if(L[i] <= R[j]) {
            data[k] = L[i];
            i++;
        } else {
            data[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining
    while(i < n1) {
        data[k++] = L[i++];
    }
    while(j < n2) {
        data[k++] = R[j++];
    }

    delete[] L;
    delete[] R;
}

void DynamicArray::mergeSortRec(int left, int right) {
    if(left < right) {
        int mid = (left + right) / 2;
        mergeSortRec(left, mid);
        mergeSortRec(mid + 1, right);
        merge(left, mid, right);
    }
}

void DynamicArray::mergeSort() {
```

```
    mergeSortRec(0, size - 1);
}
```

Step 6: Putting It All Together (Example Main)

```
#include <iostream>

int main() {
    // Ask user for size
    int n;
    std::cout << "Enter size of dynamic array: ";
    std::cin >> n;

    // Create a DynamicArray object
    DynamicArray myArray(n);

    // Let the user fill the array
    std::cout << "Enter " << n << " elements:\n";
    for(int i = 0; i < n; i++) {
        int value;
        std::cin >> value;
        myArray.setElement(i, value);
    }

    // Demonstrate linear search
    std::cout << "Enter a value to search (linear): ";
    int target;
    std::cin >> target;
    int index = myArray.linearSearch(target);
    if(index != -1) {
        std::cout << "Found at index: " << index << std::endl;
    } else {
        std::cout << "Not found.\n";
    }

    // Sort using bubble sort
    myArray.bubbleSort();
    std::cout << "Array after bubble sort: ";
    for(int i = 0; i < myArray.getSize(); i++) {
        std::cout << myArray.getElement(i) << " ";
    }
    std::cout << std::endl;

    // Now that it's sorted, do binary search
    std::cout << "Enter a value to search (binary): ";
    std::cin >> target;
    index = myArray.binarySearch(target);
    if(index != -1) {
        std::cout << "Found at index: " << index << std::endl;
    } else {
        std::cout << "Not found.\n";
    }

    // Let's also demonstrate merge sort
    // (We need to shuffle the elements to show the effect again, or re-input them)
    // ... For simplicity, let's assume the user re-enters them:
    std::cout << "\nRe-enter " << n << " elements for merge sort:\n";
    for(int i = 0; i < n; i++) {
        int value;
        std::cin >> value;
        myArray.setElement(i, value);
```

```
    }
    myArray.mergeSort();
    std::cout << "Array after merge sort: ";
    for(int i = 0; i < myArray.getSize(); i++) {
        std::cout << myArray.getElement(i) << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

In this example, we have combined:

- **OOP Concepts** (class, methods, private data)
- **Dynamic Memory** (using `new` and `delete`)
- **Algorithms** (searching and sorting)

into a single cohesive program.

6.1 1. Multiple Choice Questions (10 Questions)

1. Which of the following correctly releases dynamically allocated memory for a single integer?
   (a) (A) `delete ptr;`
   (b) (B) `free(ptr);`
   (c) (C) `delete[] ptr;`
   (d) (D) `ptr = nullptr;`
2. Which of these best describes a **destructor** in a class that uses dynamic memory?
   (a) (A) It is called to construct an object.
   (b) (B) It is used for memory management when an object goes out of scope or is deleted.
   (c) (C) It is an operator that must return a pointer.
   (d) (D) It can only be invoked manually.
3. **Linear search** on an array of size n has a worst-case time complexity of:
   (a) (A) $O(\log n)$
   (b) (B) $O(n)$
   (c) (C) $O(n \log n)$
   (d) (D) $O(1)$
4. Which of the following sorting algorithms is generally the most efficient for large data sets?
   (a) (A) Bubble Sort
   (b) (B) Selection Sort
   (c) (C) Merge Sort
   (d) (D) Linear Sort
5. **Binary search** requires:
   (a) (A) The array to be sorted
   (b) (B) The array to be reversed
   (c) (C) The array to have only positive integers
   (d) (D) The array to have distinct elements
6. In an OOP context, **encapsulation** means:
   (a) (A) Exposing all variables to the outside
   (b) (B) Grouping data and methods into a class and restricting direct access to data
   (c) (C) Using only static memory
   (d) (D) Removing pointers from your code
7. When allocating an array of 10 integers `ptr = new int[10];`, the correct way to free this memory is:
   (a) (A) `delete ptr;`

(b) (B) `free(ptr);`

(c) (C) `delete[] ptr;`

(d) (D) `ptr = 0;`

8. Which of the following is an advantage of using **merge sort**?

   (a) (A) It does not require extra space.

   (b) (B) It has a worst-case time complexity of $O(n \log n)$.

   (c) (C) It is straightforward to implement in-place without additional arrays.

   (d) (D) It is always faster than any other sorting algorithm.

9. If a class needs to manage resources (e.g., dynamic memory), which special member functions are critical to define properly (Rule of Three / Rule of Five in C++)?

   (a) (A) Default constructor, copy constructor, copy assignment operator (and possibly move constructor, move assignment operator)

   (b) (B) Constructors only

   (c) (C) Destructors only

   (d) (D) No special member functions are needed

10. Which searching algorithm would be more suitable for a **linked list** if you need to search an element?

    (a) (A) Binary Search

    (b) (B) Linear Search

    (c) (C) Merge Search

    (d) (D) Quick Search

6.2 2. Short Answer Questions (10 Questions)

Answer each in 1–3 sentences:

1. What does the `this` pointer in C++ represent?
2. Why is **dynamic memory allocation** sometimes preferred over static allocation?
3. State one advantage and one disadvantage of **bubble sort**.
4. How does a **binary search** decide which half of the array to continue searching in?
5. What is the purpose of a **destructor** in a class using dynamic memory?
6. Which searching algorithm should you use if your data is *not* sorted and cannot be sorted quickly?
7. Explain the term **encapsulation** in your own words.
8. In **selection sort**, how is the minimum (or maximum) element handled each pass?
9. What is the time complexity of **merge sort** in the worst case?
10. Why might you want a **copy constructor** in a class that uses dynamic memory?

6.3 3. Open-Ended Coding Questions (5 Questions)

Write C++ code or relevant snippets to solve:

1. **Write a class** `IntArray` that dynamically allocates an integer array, includes a constructor, destructor, and a `pushBack()` method to add a new element at the end (increasing the array size by one).
2. **Create a class** `SearchableArray` that inherits from `IntArray` and adds a `binarySearch()` method. Assume the array is sorted before calling `binarySearch()`.
3. **Implement a function template** `bubbleSort` that can sort an array of any type (int, double, string, etc.), demonstrating OOP plus template usage. Show how you would call it in `main()`.
4. **Implement a small program** that demonstrates reading a list of student IDs from the user, storing them in a dynamically allocated array, and then using `selection sort` to sort them. Finally, print the sorted IDs.
5. **Write a program** that includes a `mergeSort()` function (recursive) to sort an array of floats. The program should prompt the user to enter the number of floats, allocate memory, read them, sort them, and finally display them.

Conclusion

In this document, we demonstrated how to:

- Use **object-oriented programming** principles in C++ to create robust classes.
- Manage memory dynamically using `new` and `delete`.
- Implement **searching** and **sorting** algorithms (linear search, binary search, bubble sort, merge sort, etc.).

These skills are highly valuable for designing efficient and flexible programs. By encapsulating both data and algorithmic behavior inside classes, you can maintain cleaner, more modular code. Continue experimenting with various data structures, exploring new algorithms, and refining your OOP designs.