Array ADT (C++)

This C++ file represents a significant step up from procedural programming by re-implementing the array Abstract Data Type (ADT) as a complete **C++ class**. This object-oriented approach provides superior organization and safety by encapsulating the array's data and the functions that operate on it into a single, cohesive unit.

The array class neatly bundles the core data members (A, size, length) and makes them private. This is a key feature of **encapsulation**, as it prevents direct, uncontrolled access to the array's internal state, reducing the risk of bugs. All interactions with the array must go through the public member functions (methods), which act as a well-defined and safe interface.

The class includes **constructors** (array() and array(int sz)) for flexible object creation and a **destructor** (~array()) to automatically handle memory deallocation, which helps prevent memory leaks.

The main function demonstrates how to use this class by creating a menu-driven program. This interactive menu allows the user to call the various public methods of the array object (arr1) to perform a wide range of operations, from basic insertions and searches to more complex set operations like Union and Intersection. Overall, this file is a perfect example of how to build a reusable and robust data structure using the principles of Object-Oriented Programming in C++.

**Class Definition**

This section defines the blueprint for the array class. It separates the internal data and helper functions (private) from the functions that the user will interact with (public). This is the core of the object-oriented design.

* private: int \*A; int size; int length; These are the **private data members**. By making them private, we ensure they can only be modified by the class's own methods, which provides data integrity. A is the pointer to the heap memory, size is the capacity, and length is the current number of elements.
* public: array(){...} array(int sz){...} ~array(){...} These are special class methods.
  + array() and array(int sz) are **constructors**. They are called automatically when an array object is created to initialize its private members.
  + ~array() is the **destructor**. It's called automatically when the object is destroyed (e.g., when the program ends) to clean up resources, specifically by deallocating the heap memory using delete []A;.
* void display(); void append(int x); ... These are the **public member functions** (or methods). They form the public interface of the class, providing a controlled way to access and manipulate the private data.

**Append & Insert**

These methods handle adding new elements to the array. append adds to the end, while insert adds at a specific position.

* void array::append(int x){ ... } The :: is the **scope resolution operator**, indicating that this append function belongs to the array class. It adds the element x to the first available spot and increments the length.
* void array::insert(int index, int x){ ... } This method first **shifts all elements** from the target index one position to the right to make space, then places the new element x in the newly created gap.

**Delete**

This method removes an element from a specified index.

* void array::del(int index){ ... } To remove the element, it **shifts all subsequent elements** one position to the left, effectively overwriting the value at the target index. It then decrements the length.

**Linear and Binary Search**

These methods are for finding elements within the array.

* int array::LinearSearch(int x){ ... } This performs a sequential search. The line swap(&A[i], &A[i-1]); is an optimization known as **Transposition**, which moves a found element one step closer to the front, making future searches for it faster.
* int array::BinarySearch(int key){ ... } This implements the efficient **Binary Search** algorithm for sorted arrays. It repeatedly divides the search space in half until the key is found or the search space is empty.

**Get, Set, Max, Min, Sum, Avg**

This is a suite of utility methods for accessing and calculating information about the array's data.

* int array::get(int index){ ... } Safely **retrieves** the element at a given index, returning -1 if the index is invalid.
* int array::set(int index, int value){ ... } Safely **updates** the element at a given index with a new value.
* float array::avg(){ return (float) sum() / length; } This method calculates the average. It demonstrates **method chaining** by calling another member function (sum()) to help with its calculation. The (float) type cast ensures accurate floating-point division.

**Reverse, Swap & Left Shift**

These methods perform in-place modifications of the array's structure.

* void array::reverse(){ ... } This method uses an **auxiliary array (B)** to store the elements in reverse order before copying them back.
* void array::swapping(){ ... } This is a more memory-efficient method for reversing the array. It performs an **in-place swap** of elements from the outside in, only requiring a single temporary variable.
* void array::leftShift(){ ... } This performs a **circular left shift** by saving the first element, shifting all other elements one position to the left, and then placing the saved element at the end.

**Sorting and Partitioning**

These methods deal with the order and arrangement of elements.

* void array::insertSort(int x){ ... } This method inserts an element into a **sorted array** while maintaining the sorted order by shifting larger elements to the right.
* bool array::isSorted(){ ... } This method efficiently **checks if the array is sorted** by returning false as soon as it finds a single pair of adjacent elements that are out of order.
* void array::negativeCheck(){ ... } This method **partitions the array** using a two-pointer approach, ensuring all negative numbers are moved to the left side and positive numbers to the right, without necessarily sorting them.

**Set Operations (Union, Intersection, Difference)**

These methods perform set operations on two sorted arrays, returning a new array object with the result.

* array\* array::ArrayUnion(array arr2){ ... } This method computes the **Union** of two sets. It merges the two arrays and handles duplicate elements by copying the element once and advancing the pointers for both arrays.
* array\* array::ArrayIntersection(array arr2){ ... } This computes the **Intersection**, which includes only the elements present in *both* arrays. An element is added to the result only when a match is found.
* array\* array::ArrayDifference(array arr2){ ... } This computes the **Difference** (this array minus arr2). It adds an element from the first array only if it's smaller than the current element in the second array or if it's left over after the second array is exhausted.

**Main Function and Menu**

The main function serves as the driver for the program, creating an instance of the array class and providing a user-friendly, interactive menu to test its functionality.

* array \*arr1; arr1 = new array(sz); This dynamically allocates an array **object on the heap** using the new keyword and the class constructor that takes a size parameter. arr1 is a pointer to this object.
* do { ... } while(ch != 0); This **do-while loop** keeps the menu running until the user chooses to exit by entering 0.
* switch(ch) { ... } The **switch statement** is an organized way to handle the user's menu choice. Each case corresponds to a menu option and calls the appropriate public method on the arr1 object (e.g., arr1->append(x);).
* delete arr1; When the user exits, the delete keyword is called. This frees the memory allocated for the arr1 object and, importantly, automatically triggers the class's **destructor (~array())** to free the internal A array, preventing memory leaks.