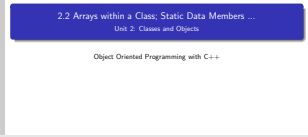
2.2 Arrays within a Class; Static Data Members ...

Unit 2: Classes and Objects

Object Oriented Programming with C++



Lecture 19: Arrays within a Class, Static Members, and Objects

- Course: Object Oriented Programming with C++
- Unit 2: Classes and Objects

Lecture 19: Arrays within a Class, Static Members, and Objects
- Course: Object Oriented Programming with C++
- Unit 2: Classes and Objects

Welcome everyone to Lecture 19. Today we are diving deeper into classes and objects in C++. We will explore some advanced membership concepts that allow us to structure our data and functions more powerfully.

Agenda for Today's Lecture

- 1. Arrays within a Class
- 2. Static Data Members
- 3. Static Member Functions
- 4. Arrays of Objects
- 5. Objects as Function Arguments

Agenda for Today's Lecture
- 1. Arrays within a Class
- 2. Static Data Members
- 3. Static Member Functions
- 4. Arrays of Objects
- 5. Objects as Function Arguments

Here is our roadmap for the next 50 minutes. We'll start by seeing how arrays can be encapsulated inside classes. Then, we will look at class-level variables and functions, known as static members. Finally, we'll discuss how to group objects into arrays and how to pass objects to functions.

Arrays within a Class

- Arrays can be declared as private or public data members of a class.
- Used when a class needs to store a collection of elements of the same data type.
- Provide a way to bind a list of items together within a single object.
- Memory for the array is allocated when the object is instantiated.

Arrays within a Class
- Arrays can be declared as private or public data members of a class.
- Used when a class needs to store a collection of elements of the same data type.
- Provide a way to bind a list of items together within a single object.
- Memory for the array is allocated when the object is instantiated.

Just like we use basic types like int and float as data members, we can also use arrays. If an object needs to represent an entity containing multiple similar data points, like a student with multiple test scores, arrays within a class are the perfect solution. The array is encapsulated and its memory is allocated on a per-object basis.

Example: Arrays within a Class

- class Student {
- int roll_no;
- int marks[5]; // Array as a data member
- public:
- void get_data();
- void display_total();
- };

Example: Arrays within a Class
class Student {
int roll_no;
int marks[5]; // Array as a data member
public:
void get_data();
void display_total();
};

In this example, the Student class contains an integer array 'marks' of size 5. Every time we create a Student object, it will have its own independent copy of the 'marks' array. We can then define member functions like get_data() and display_total() to manipulate this array.

Introduction to Static Data Members

- Normally, each object has its own separate copy of data members.
- Static data members are shared among all objects of a class.
- Only a single copy of a static data member is created for the entire class.
- Initialized to zero when the first object of its class is created (if not explicitly initialized).

Introduction to Static Data Members
- Normally, each object has its own separate copy of data members.
- Static data members are shared among all objects of a class.
- Only a single copy of a static data member is created for the entire class.
- Initialized to zero when the first object of its class is created (if not explicitly initialized).

Now let's talk about static data members. Sometimes, we need a variable that is shared by all objects of a class, like a counter to keep track of how many objects have been created. This is where static data members come in. They are class variables, not object variables. A single copy exists, regardless of how many objects you instantiate.

Characteristics of Static Data Members

- Declaration: Declared inside the class using the 'static' keyword.
- Definition: Must be defined outside the class definition.
- Scope: Visible only within the class, but its lifetime is the entire program.
- Accessibility: Can be accessed by any object of the class, or using the class name and scope resolution operator (::).

Characteristics of Static Data Members
- Declaration: Declared inside the class using the 'static' keyword.
- Definition: Must be defined outside the class definition.
- Scope: Visible only within the class, but its lifetime is the entire program.
- Accessibility: Can be accessed by any object of the class, or using the class name and scope resolution operator (::).

It's critical to understand that declaring a static member inside the class only tells the compiler about its existence. You must also define it outside the class to actually allocate memory for it. This is a common pitfall for C++ beginners. The lifetime of a static variable is the duration of the program, even if no objects currently exist.

Example: Static Data Members

- class Item {
- static int count; // Declaration
- int number;
- public:
- void get_data(int a) {
- number = a; count++;
- }
- };
-
- // Definition outside the class
- int Item::count = 0;

Example: Static Data Members
class Item {
static int count; // Declaration
int number;
public:
void get_data(int a) {
number = a; count++;
}
};
// Definition outside the class
int Item::count = 0;

Here we see the declaration of 'count' inside the Item class and its definition outside. Notice `int Item::count = 0;`. This allocates memory. Every time get_data is called on ANY object, the shared 'count' variable is incremented. If we create three items and call get_data for each, 'count' will be 3.

Static Member Functions

- A member function that is declared static.
- Can be called using the class name instead of an object.
- Can ONLY access static data members and other static member functions.
- They do not have access to the 'this' pointer.

Static Member Functions
- A member function that is declared static.
- Can be called using the class name instead of an object.
- Can ONLY access static data members and other static member functions.
- They do not have access to the 'this' pointer.

Just as we have static data, we can have static member functions. These functions operate at the class level. Because they don't belong to any specific object, they don't have a 'this' pointer. Consequently, a static function cannot access non-static data members—it wouldn't know which object's data to access!

Example: Static Member Functions

```
class Test {
static int count;
public:
static void showCount() {
cout ¡¡ "Count: " ¡¡ count ¡¡ endl;
}
};
int Test::count = 5;

int main() {
Test::showCount(); // Called without an object
return 0;
}
```


Example: Static Member Functions
class Test {
static int count;
public:
static void showCount() {
cout ¡¡ "Count: " ¡¡ count ¡¡ endl;
}
};
int Test::count = 5;
int main() {
Test::showCount(); // Called without an object
return 0;
}

Look at how showCount is called in main. We use `Test::showCount()`. We didn't even create an object of type Test! This is incredibly useful for utility functions or for accessing static data before any objects are instantiated.

Arrays of Objects

- Just like arrays of basic data types (int, float), we can create arrays of user-defined types (classes).
- An array of objects stores multiple objects of the same class in contiguous memory.
- Useful for managing collections of entities, like a list of employees or students.
- Accessed using an index, just like standard arrays.

Arrays of Objects
- Just like arrays of basic data types (int, float), we can create arrays of user-defined types (classes).
- An array of objects stores multiple objects of the same class in contiguous memory.
- Useful for managing collections of entities, like a list of employees or students.
- Accessed using an index, just like standard arrays.

Moving on to arrays of objects. Once a class is defined, it becomes a new data type. Therefore, you can create an array of this type. If you need to process data for 50 employees, instead of creating 50 separate variables, you create an array of Employee objects. Each element of the array is an individual object with its own data members.

Example: Arrays of Objects

```
class Employee {
char name[30];
float age;
public:
void getData();
void putData();
};

int main() {
Employee managers[3]; // Array of 3 Employee objects
for(int i = 0; i ¡ 3; i++)
managers[i].getData(); // Accessing object via index
return 0;
}
```


Example: Arrays of Objects
class Employee {
char name[30];
float age;
public:
void getData();
void putData();
};

int main() {
Employee managers[3]; // Array of 3 Employee objects
for(int i = 0; i ¡ 3; i++)
managers[i].getData(); // Accessing object via index
return 0;
}

Here we declare an array named 'managers' that holds 3 Employee objects. Using a loop, we can iterate through the array, calling getData() for managers[0], managers[1], and managers[2]. The syntax is intuitive: array_name[index].member_function().

Objects as Function Arguments

- Objects can be passed to functions just like standard variables.
- Three ways to pass objects:
- 1. Pass-by-Value: A copy of the object is created.
- 2. Pass-by-Reference: The memory address (reference) is passed.
- 3. Pass-by-Pointer: A pointer to the object is passed.

Objects as Function Arguments
- Objects can be passed to functions just like standard variables.
- Three ways to pass objects:
- 1. Pass-by-Value: A copy of the object is created.
- 2. Pass-by-Reference: The memory address (reference) is passed.
- 3. Pass-by-Pointer: A pointer to the object is passed.

In object-oriented programming, objects frequently need to interact with each other. This is often done by passing objects as arguments to functions. You can pass them by value, where a duplicate is made, or by reference/pointer, where the original object is accessed directly. We will focus primarily on pass-by-value and pass-by-reference.

Pass-by-Value vs Pass-by-Reference

- Pass-by-Value:
- - Creates a complete copy of the object.
- - Changes made inside the function do NOT affect the original object.
- - Can be slow and memory-intensive for large objects.
-
- Pass-by-Reference:
- - Passes an alias (address) of the original object.
- - Changes made inside the function DO affect the original object.
- - Efficient, as no copying is required. (Use 'const' if you want to prevent modification).

Pass-by-Value vs Pass-by-Reference
- Pass-by-Value:
- - Creates a complete copy of the object.
- - Changes made inside the function do NOT affect the original object.
- - Can be slow and memory-intensive for large objects.
-
- Pass-by-Reference:
- - Passes an alias (address) of the original object.
- - Changes made inside the function DO affect the original object.
- - Efficient, as no copying is required. (Use 'const' if you want to prevent modification).

When should you use which? If the object is large, pass-by-value is highly inefficient because copying takes time and memory. Pass-by-reference is generally preferred for objects to avoid overhead. If you want the efficiency of passing by reference but want to protect the object from being modified, use a const reference.

Example: Objects as Arguments (Pass-by-Value)

- class Time {
- int hours, minutes;
- public:
- void sum(Time t1, Time t2) { // Passed by value
- minutes = t1.minutes + t2.minutes;
- hours = minutes / 60;
- minutes = minutes % 60;
- hours = hours + t1.hours + t2.hours;
- }
- };

Example: Objects as Arguments (Pass-by-Value)
class Time {
int hours, minutes;
public:
void sum(Time t1, Time t2) { // Passed by value
minutes = t1.minutes + t2.minutes;
hours = minutes / 60;
minutes = minutes % 60;
hours = hours + t1.hours + t2.hours;
}
};

In this example, the sum function takes two Time objects, t1 and t2, as arguments. Because they are passed by value, copies of the arguments provided by the caller are made. The function then calculates the total time and stores it in the calling object's own members. This is a classic pattern for adding custom data types.

Example: Objects as Arguments (Pass-by-Reference)

```
class Account {
int balance;
public:
// Passed by reference using &
void transfer(Account &from, int amount) {
balance += amount;
from.balance -= amount; // Modifies the original object
}
};
```


Example: Objects as Arguments (Pass-by-Reference)
class Account {
int balance;
public:
// Passed by reference using &
void transfer(Account &from, int amount) {
balance += amount;
from.balance -= amount; // Modifies the original object
}
};

Here we have a bank account transfer scenario. The 'transfer' function takes another Account object by reference, indicated by the ampersand (&). When we subtract from 'from.balance', we are directly modifying the balance of the actual object that was passed in. If we had passed by value, the deduction would only happen to a temporary copy, resulting in a logical error in our bank system!

Lecture Summary

- Arrays can be encapsulated within classes to group data.
- Static data members belong to the class and are shared across all instances.
- Static member functions can only access static data and are called using the class name.
- We can create arrays of objects to manage collections of custom types.
- Objects can be passed to functions by value (copy) or by reference (efficient, allows modification).

Lecture Summary
- Arrays can be encapsulated within classes to group data.
- Static data members belong to the class and are shared across all instances.
- Static member functions can only access static data and are called using the class name.
- We can create arrays of objects to manage collections of custom types.
- Objects can be passed to functions by value (copy) or by reference (efficient, allows modification).

To wrap up, today we covered several powerful features of C++ classes. We learned how to manage multiple data points using arrays within classes and arrays of objects. We explored class-level properties with static members. Lastly, we saw how objects interact by being passed as arguments to functions. Mastering these concepts is essential for building robust C++ applications.