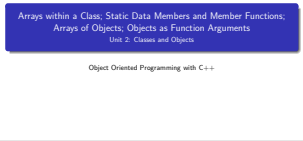
Arrays within a Class; Static Data Members and Member Functions; Arrays of Objects; Objects as Function Arguments

Unit 2: Classes and Objects

Object Oriented Programming with C++



Object Oriented Programming with C++

- Lecture 15: Arrays within a Class & Static Members
- Unit 2: Classes and Objects
- Topics: Arrays in Classes, Static Data, Static Methods, Object Arrays, Objects as Arguments

Object Oriented Programming with C++
- Lecture 15: Arrays within a Class & Static Members
- Unit 2: Classes and Objects
- Topics: Arrays in Classes, Static Data, Static Methods, Object Arrays, Objects as Arguments

Welcome students to Lecture 15. Today we are diving deeper into the capabilities of classes in C++. We will transition from simple data types to more complex structures within classes, such as arrays. We will also introduce the 'static' keyword, which is crucial for managing class-level state. Finally, we'll see how to handle multiple objects using arrays and how to pass objects around in our programs.

Agenda and Learning Objectives

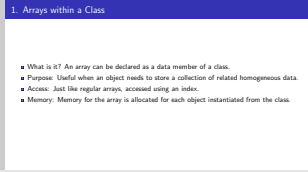
- 1. Arrays within a Class: Storing multiple values in one object.
- 2. Static Data Members: Class variables shared among all objects.
- 3. Static Member Functions: Methods operating on class level.
- 4. Arrays of Objects: Creating collections of class instances.
- 5. Objects as Function Arguments: Passing objects into methods.

Agenda and Learning Objectives
1. Arrays within a Class: Storing multiple values in one object.
2. Static Data Members: Class variables shared among all objects.
3. Static Member Functions: Methods operating on class level.
4. Arrays of Objects: Creating collections of class instances.
5. Objects as Function Arguments: Passing objects into methods.

Here is our roadmap for the next 50 minutes. We'll start by putting arrays inside classes, then move to a major OOP concept: static members. After that, we'll look at arrays of objects, which are extremely common in real-world applications like databases. Lastly, we'll discuss the nuances of passing whole objects into functions.

1. Arrays within a Class

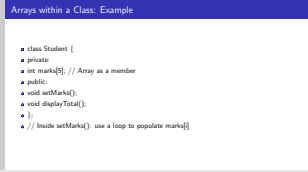
- What is it? An array can be declared as a data member of a class.
- Purpose: Useful when an object needs to store a collection of related homogeneous data.
- Access: Just like regular arrays, accessed using an index.
- Memory: Memory for the array is allocated for each object instantiated from the class.



You already know how to use arrays in structural programming. In OOP, an array can simply be a property of a class. For example, a 'Student' class might have an array to store marks for 5 different subjects. Every time you create a new Student object, a new array of 5 integers is created in memory specifically for that student.

Arrays within a Class: Example

- class Student {
- private:
- int marks[5]; // Array as a member
- public:
- void setMarks();
- void displayTotal();
- };
- // Inside setMarks(): use a loop to populate marks[i]



Here is a conceptual snippet. The 'marks' array is private, meaning external code cannot mess with the grades directly. We provide a public member function 'setMarks' to populate this array, likely using a for-loop. We also have 'displayTotal' which will iterate through the array, sum the values, and print the result. The syntax is exactly the same as normal arrays.

2. Static Data Members: Concept

- Normal variables: Each object has its own copy (Instance Variables).
- Static variables: Only ONE copy exists for the entire class (Class Variables).
- Shared: This single copy is shared by all objects of that class.
- Keyword: Declared using the 'static' keyword inside the class.

2. Static Data Members: Concept
- Normal variables: Each object has its own copy (Instance Variables).
- Static variables: Only ONE copy exists for the entire class (Class Variables).
- Shared: This single copy is shared by all objects of that class.
- Keyword: Declared using the 'static' keyword inside the class.

Now let's talk about 'static'. Normally, if I have 100 'Student' objects, I have 100 different 'marks' arrays. But what if I want to keep track of the TOTAL number of students created? If I use a normal variable, each student gets a 'count' starting at 1. That doesn't work. We need a variable that belongs to the class itself, not any specific object. This is a static data member.

Static Data Members: Characteristics

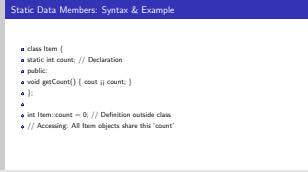
- Initialization: Initialized to zero automatically when the first object is created (if not initialized explicitly).
- Definition: Must be defined OUTSIDE the class definition to allocate memory.
- Lifetime: Exists for the entire duration of the program.
- Visibility: Follows public/private/protected access rules.

Static Data Members: Characteristics
- Initialization: Initialized to zero automatically when the first object is created (if not initialized explicitly).
- Definition: Must be defined OUTSIDE the class definition to allocate memory.
- Lifetime: Exists for the entire duration of the program.
- Visibility: Follows public/private/protected access rules.

There is a strict rule in C++: you declare a static member inside the class, but you MUST define it outside the class. Why? Because the class definition is just a blueprint; it doesn't allocate memory. Since static variables are independent of objects, their memory must be allocated separately before any objects are even created. If you forget to define it outside, you will get a linker error.

Static Data Members: Syntax & Example

- class Item {
- static int count; // Declaration
- public:
- void getCount() { cout ¡¡ count; }
- };
-
- int Item::count = 0; // Definition outside class
- // Accessing: All Item objects share this 'count'



Notice the line 'int Item::count = 0;'. This is where the memory is actually reserved. The 'Item::' tells the compiler that this 'count' belongs to the 'Item' class. Inside the class, you just write 'static int count;'. If three Item objects are created, and one increments count, all three will see the updated value.

3. Static Member Functions

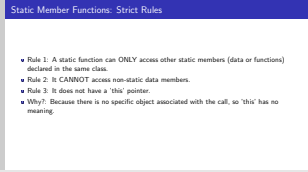
- Just like data members, functions can also be declared static.
- Purpose: To access and manipulate static data members.
- Invocation: Can be called using the class name, without needing an object.
- Syntax: ClassName::FunctionName();

3. Static Member Functions
- Just like data members, functions can also be declared static.
- Purpose: To access and manipulate static data members.
- Invocation: Can be called using the class name, without needing an object.
- Syntax: ClassName::FunctionName();

If we have static data, it makes sense to have static functions to manage that data. The beauty of a static member function is that you do not need an object to call it. Because it belongs to the class, you can invoke it directly using the class name and the scope resolution operator.

Static Member Functions: Strict Rules

- Rule 1: A static function can ONLY access other static members (data or functions) declared in the same class.
- Rule 2: It CANNOT access non-static data members.
- Rule 3: It does not have a 'this' pointer.
- Why?: Because there is no specific object associated with the call, so 'this' has no meaning.



These rules are critical and often tested in exams. A static function cannot read an instance variable. Think about it: if I call 'Item::printData()', and it tries to print an item's price, which item's price should it print? There is no object! Therefore, the compiler strictly forbids static functions from accessing non-static data. They also lack the 'this' pointer for the exact same reason.

Static Member Functions: Example

- class Demo {
- static int staticVal;
- public:
- static void showStatic() {
- cout ¡¡ staticVal; // Valid
- }
- };
- int Demo::staticVal = 10;
- int main() {
- Demo::showStatic(); // Called using Class Name
- }

Static Member Functions: Example
class Demo {
static int staticVal;
public:
static void showStatic() {
cout ¡¡ staticVal; // Valid
}
};
int Demo::staticVal = 10;
int main() {
Demo::showStatic(); // Called using Class Name
}

Here we see the full implementation. 'showStatic' is declared static. In main, we don't even create a 'Demo' object. We just type 'Demo::showStatic()'. This proves that the method exists independently of any instances.

4. Arrays of Objects

- Concept: Just like arrays of standard types (int, float), we can create arrays of user-defined class types.
- Usage: Storing data for multiple entities (e.g., 50 Employees, 100 Books).
- Memory: Contiguous memory blocks are allocated to hold the objects sequentially.

4. Arrays of Objects
- Concept: Just like arrays of standard types (int, float), we can create arrays of user-defined class types.
- Usage: Storing data for multiple entities (e.g., 50 Employees, 100 Books).
- Memory: Contiguous memory blocks are allocated to hold the objects sequentially.

Moving on to arrays of objects. In real life, you rarely deal with a single employee or a single book. You deal with hundreds. Instead of writing 'Employee e1, e2, e3... e100;', we create an array: 'Employee emp[100];'. This allocates contiguous memory for 100 Employee objects.

Arrays of Objects: Implementation

```
class Employee {
int id; char name[30];
public:
void getData(); void putData();
};
int main() {
Employee emp[3]; // Array of 3 objects
for(int i=0; i¡3; i++) {
emp[i].getData(); // Accessing methods
}
}
```


Arrays of Objects: Implementation
class Employee {
int id; char name[30];
public:
void getData(); void putData();
};
int main() {
Employee emp[3]; // Array of 3 objects
for(int i=0; i¡3; i++) {
emp[i].getData(); // Accessing methods
}
}

To access the members of each object in the array, we use the array index followed by the dot operator. For instance, 'emp[i].getData()'. The loop iterates through the array, calling the 'getData()' method on each specific object in turn. It behaves exactly like an array of structs in C.

5. Objects as Function Arguments

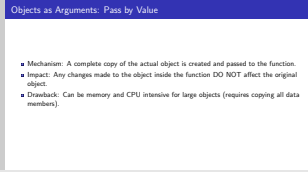
- Objects can be passed as parameters to functions (both member and non-member functions).
- Three ways to pass:
- 1. Pass by Value
- 2. Pass by Reference
- 3. Pass by Pointer (Address)

5. Objects as Function Arguments
- Objects can be passed as parameters to functions (both member and non-member functions).
- Three ways to pass:
- 1. Pass by Value
- 2. Pass by Reference
- 3. Pass by Pointer (Address)

The last topic for today is passing objects to functions. Just like we can pass an 'int' to a function, we can pass a whole 'Student' object. A common use case is a function that compares two objects or adds the values of two objects together. We must choose the right passing mechanism for performance and desired behavior.

Objects as Arguments: Pass by Value

- Mechanism: A complete copy of the actual object is created and passed to the function.
- Impact: Any changes made to the object inside the function DO NOT affect the original object.
- Drawback: Can be memory and CPU intensive for large objects (requires copying all data members).



In Pass by Value, the compiler makes a bit-by-bit copy of the object. If the object has huge arrays or complex data, this copying takes time and memory. Also, if the function modifies the object, it's only modifying the local copy. The original object in 'main' remains untouched. While safe, it's often inefficient for large objects.

Objects as Arguments: Pass by Reference/Pointer

- Pass by Reference: Passes an alias of the object. (Syntax: void func(ClassType &obj))
- Pass by Pointer: Passes the memory address. (Syntax: void func(ClassType *obj))
- Advantage: Highly efficient. No copy is made.
- Impact: Changes made inside the function directly affect the original object.

Objects as Arguments: Pass by Reference/Pointer
- Pass by Reference: Passes an alias of the object. (Syntax: void func(ClassType &obj))
- Pass by Pointer: Passes the memory address. (Syntax: void func(ClassType *obj))
- Advantage: Highly efficient. No copy is made.
- Impact: Changes made inside the function directly affect the original object.

To avoid the overhead of copying, we use pass by reference or pass by pointer. We are simply telling the function 'here is where the object lives in memory'. It's very fast. However, with great power comes great responsibility: if the function alters the object, the original object is permanently altered. We can use the 'const' keyword if we want to pass by reference for speed but prevent modification.

Objects as Arguments: Example (Adding Complex Numbers)

- class Complex { int real, imag; ...
- void add(Complex c1, Complex c2) {
- real = c1.real + c2.real;
- imag = c1.imag + c2.imag;
- }
- };
- // In main:
- Complex cA, cB, cC;
- cC.add(cA, cB); // cA and cB are passed as objects

Objects as Arguments: Example (Adding Complex Numbers)
class Complex { int real, imag; ...
void add(Complex c1, Complex c2) {
real = c1.real + c2.real;
imag = c1.imag + c2.imag;
}
};
// In main:
Complex cA, cB, cC;
cC.add(cA, cB); // cA and cB are passed as objects

This is a classic exam question: write a program to add two complex numbers. We have a 'Complex' class. The 'add' function takes two 'Complex' objects as arguments. We call it using a third object 'cC'. So 'cC.add(cA, cB)' means cC's 'real' part becomes the sum of cA's real and cB's real. Here we used pass by value, which is fine since the object only holds two integers.

Summary & Key Takeaways

- Arrays in classes group related data for an object.
- Static members belong to the CLASS, not instances. Remember the initialization rule!
- Static functions can only access static data and lack a 'this' pointer.
- Arrays of objects allow for managing bulk entities efficiently.
- Objects passed to functions can be passed by value (copy) or reference (alias).

Summary & Key Takeaways
- Arrays in classes group related data for an object.
- Static members belong to the CLASS, not instances. Remember the initialization rule!
- Static functions can only access static data and lack a 'this' pointer.
- Arrays of objects allow for managing bulk entities efficiently.
- Objects passed to functions can be passed by value (copy) or reference (alias).

To summarize, we expanded our OOP toolkit today. We saw how arrays and objects interact. The most important conceptual hurdle today was 'static'. Remember: normal members = instance members, static members = class members. And when passing objects, be mindful of pass-by-value vs pass-by-reference regarding performance.

Q&A and Next Steps

- Questions?
- Readings: Text book Chapter 5, Sections 5.3 to 5.7.
- Practice: Try writing the 'Complex number addition' program using Pass by Reference.
- Next Lecture: Constructors and Destructors (Unit 2.3).

Q&A and Next Steps
- Questions?
- Readings: Text book Chapter 5, Sections 5.3 to 5.7.
- Practice: Try writing the 'Complex number addition' program using Pass by Reference.
- Next Lecture: Constructors and Destructors (Unit 2.3).

Are there any questions on static members or passing objects? I highly recommend you go home and code the Complex number example, but change it to pass-by-reference to see the syntax differences. In our next class, we will solve the problem of initializing objects automatically using Constructors. Class dismissed.