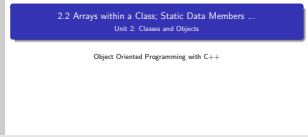
2.2 Arrays within a Class; Static Data Members ...

Unit 2: Classes and Objects

Object Oriented Programming with C++



Arrays within a Class; Static Members; Arrays of Objects; Objects as Arguments

- Unit 2: Classes and Objects
- Lecture 11
- Course: Object Oriented Programming with C++

Arrays within a Class; Static Members; Arrays of Objects; Objects as Arguments
- Unit 2: Classes and Objects
- Lecture 11
- Course: Object Oriented Programming with C++

Welcome students to Lecture 11. Today we are expanding our knowledge of classes in C++. We will explore advanced ways to store data inside classes using arrays, how to share data between objects using static members, and how to work with multiple objects and pass them around.

Lecture Outline

Lecture Outline
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 with arrays inside classes, move on to the static keyword for data and functions, look at how to create arrays of objects, and finally discuss how to pass objects to functions.

1. Arrays within a Class

- Arrays can be declared as private or public data members of a class.
- Useful when a class needs to store a collection of similar data types (e.g., a student's grades, a shopping cart's items).
- Member functions can perform operations on these array elements.
- Memory for the array is allocated for every single object created from the class.

1. Arrays within a Class
- Arrays can be declared as private or public data members of a class.
- Useful when a class needs to store a collection of similar data types (e.g., a student's grades, a shopping cart's items).
- Member functions can perform operations on these array elements.
- Memory for the array is allocated for every single object created from the class.

Just like we use primitive data types like int or float inside a class, we can also use arrays. If every object needs its own list of items, an array within a class is the perfect solution. Keep in mind that the array's size must be known at compile time when declared normally.

Arrays within a Class: Code Example

```
class ShoppingList {
private:
int itemCode[50];
float itemPrice[50];
int count;
public:
void initialize() { count = 0; }
void getitem();
void displaySum();
};
```


Arrays within a Class: Code Example
class ShoppingList {
private:
int itemCode[50];
float itemPrice[50];
int count;
public:
void initialize() { count = 0; }
void getitem();
void displaySum();
};

In this ShoppingList class, we have two parallel arrays: one for item codes and one for prices. We also have a 'count' variable to keep track of how many items are currently in the list. The initialize function sets the count to zero. Each ShoppingList object created will have its own 50-element arrays.

2. Static Data Members

- Normally, each object maintains its own independent copy of data members.
- A 'static' data member is shared among ALL objects of that specific class.
- It is automatically initialized to zero when the first object is created (if no other initialization is provided).
- Only one copy of the static data member exists in memory, regardless of how many objects are instantiated.

2. Static Data Members
- Normally, each object maintains its own independent copy of data members.
- A 'static' data member is shared among ALL objects of that specific class.
- It is automatically initialized to zero when the first object is created (if no other initialization is provided).
- Only one copy of the static data member exists in memory, regardless of how many objects are instantiated.

What if we want all objects of a class to share a single variable? For example, keeping track of the total number of objects created. We use static data members. The 'static' keyword tells the compiler to allocate memory for this variable only once, and all objects will access this exact same memory location.

Static Data Members: Rules & Characteristics

- Must be declared inside the class body.
- Must be defined outside the class body to allocate memory.
- Syntax for definition: DataType ClassName::staticVarName = Value;
- Their scope is restricted to the class, but their lifetime spans the entire program execution.
- Often used as counters or to store shared configuration settings.

Static Data Members: Rules & Characteristics
- Must be declared inside the class body.
- Must be defined outside the class body to allocate memory.
- Syntax for definition: DataType ClassName::staticVarName = Value;
- Their scope is restricted to the class, but their lifetime spans the entire program execution.
- Often used as counters or to store shared configuration settings.

A crucial rule in C++ is that declaring a static member inside the class does not allocate memory for it. You must explicitly define it outside the class using the scope resolution operator. If you forget this step, you will encounter a linker error during compilation.

Static Data Members: Code Example

- class Item {
- static int count; // Declaration
- int number;
- public:
- void getData(int a) { number = a; count++; }
- void getCount() { cout ¡¡ count; }
- };
- // Definition outside class
- int Item::count = 0;

Static Data Members: Code Example
class Item {
static int count; // Declaration
int number;
public:
void getData(int a) { number = a; count++; }
void getCount() { cout ¡¡ count; }
};
// Definition outside class
int Item::count = 0;

Here, 'count' is a static integer. Whenever the getData function is called on ANY object, the shared 'count' is incremented. Notice the definition 'int Item::count = 0;' outside the class. If we create three Item objects and call getData on each, calling getCount on any of them will output 3.

3. Static Member Functions

- Just like data members, member functions can also be declared static.
- A static member function can ONLY access static data members or invoke other static member functions.
- It cannot access non-static data members because it lacks a 'this' pointer.
- It can be called directly using the class name: ClassName::functionName();

3. Static Member Functions
- Just like data members, member functions can also be declared static.
- A static member function can ONLY access static data members or invoke other static member functions.
- It cannot access non-static data members because it lacks a 'this' pointer.
- It can be called directly using the class name: ClassName::functionName();

If we have static data, it is a good architectural practice to use static functions to manage that data. This is especially useful before any objects are even created. Because static functions don't belong to any specific object instance, they don't have a 'this' pointer, meaning they physically cannot access standard, non-static instance variables.

Static Member Functions: Code Example

- class Test {
- static int objCount;
- public:
- Test() { objCount++; }
- static void showCount() {
- cout ¡¡ "Total objects: " ¡¡ objCount ¡¡ endl;
- }
- };
- int Test::objCount = 0;
- // In main():
- Test::showCount(); // Outputs 0

Static Member Functions: Code Example
class Test {
static int objCount;
public:
Test() { objCount++; }
static void showCount() {
cout ¡¡ "Total objects: " ¡¡ objCount ¡¡ endl;
}
};
int Test::objCount = 0;
// In main():
Test::showCount(); // Outputs 0

In this example, showCount is a static function. Notice how we can call Test::showCount() in main() even before we instantiate any Test objects, and it will correctly print 0. Once we create objects, the constructor increments objCount, and calling showCount again will reflect the newly updated total.

4. Arrays of Objects

- We can create arrays of variables of a class type, which are arrays of objects.
- Syntax: ClassName arrayName[size];
- Useful for storing and managing a collection of objects (e.g., 50 students, 100 employees).
- Accessed and manipulated using standard array index notation.

4. Arrays of Objects
- We can create arrays of variables of a class type, which are arrays of objects.
- Syntax: ClassName arrayName[size];
- Useful for storing and managing a collection of objects (e.g., 50 students, 100 employees).
- Accessed and manipulated using standard array index notation.

Just as we can have an array of integers or floats, we can declare an array of objects. When we declare an array of objects, the default constructor is called automatically for every single element in the array. This is extremely useful for database-like applications in memory, such as an employee management system.

Arrays of Objects: Memory Layout & Iteration

- class Employee {
- char name[30];
- float salary;
- public:
- void getData();
- void displayData();
- };
- // In main():
- Employee manager[3]; // Array of 3 Employee objects
- for(int i = 0; i ¡ 3; i++) {
- manager[i].getData();
- }

Arrays of Objects: Memory Layout & Iteration
class Employee {
char name[30];
float salary;
public:
void getData();
void displayData();
};
// In main():
Employee manager[3]; // Array of 3 Employee objects
for(int i = 0; i ¡ 3; i++) {
manager[i].getData();
}

Here, 'manager' is an array of 3 distinct Employee objects. Each element 'manager[i]' acts as an independent object. We use a standard for-loop to iterate through the array and call the 'getData' member function for each specific object instance. In memory, the data for these three objects is stored sequentially.

5. Objects as Function Arguments

- Like any built-in data type, objects can be passed into functions as arguments.
- Objects can be passed to both standalone functions and member functions of a class.
- Three main ways to pass objects:
- 1. Pass by Value
- 2. Pass by Reference
- 3. Pass by Address (using Pointers)

5. Objects as Function Arguments
- Like any built-in data type, objects can be passed into functions as arguments.
- Objects can be passed to both standalone functions and member functions of a class.
- Three main ways to pass objects:
- 1. Pass by Value
- 2. Pass by Reference
- 3. Pass by Address (using Pointers)

Often, we need to compare, merge, or evaluate data from multiple objects at once. We can achieve this by passing objects into functions. An object can be passed to a non-member function, or it can be passed to a member function of another object of the same or a different class.

Pass by Value vs. Pass by Reference

- Pass by Value:
- - Function receives a duplicate copy of the object.
- - Changes inside the function do NOT affect the original object.
- - Can be computationally slow and memory-intensive for large objects.
- Pass by Reference (&):
- - Function receives a reference (alias) to the original object.
- - Changes inside the function WILL affect the original object (unless marked const).
- - Highly efficient (no copying overhead).

Pass by Value vs. Pass by Reference
- Pass by Value:
- - Function receives a duplicate copy of the object.
- - Changes inside the function do NOT affect the original object.
- - Can be computationally slow and memory-intensive for large objects.
- Pass by Reference (&):
- - Function receives a reference (alias) to the original object.
- - Changes inside the function WILL affect the original object (unless marked const).
- - Highly efficient (no copying overhead).

Passing by value invokes the class copy constructor, duplicating all data members. For large objects, this creates a performance bottleneck. Passing by reference is almost always preferred in professional C++ for objects. If you don't want the receiving function to modify the object, you pass it as a 'const reference'.

Objects as Arguments: Code Example (Adding Time)

- class Time {
- int hours, minutes;
- public:
- void setTime(int h, int m) { hours = h; minutes = m; }
- 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;
- }
- };

Objects as Arguments: Code Example (Adding Time)
class Time {
int hours, minutes;
public:
void setTime(int h, int m) { hours = h; minutes = m; }
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 Time class, the 'sum' member function takes two entirely separate Time objects as arguments. It accesses the minutes and hours of t1 and t2, calculates the total sum, and stores the resulting time in the calling object's own variables. For example, calling T3.sum(T1, T2) sets T3's internal state to the sum of T1 and T2.

Lecture Summary

- Arrays within a class allow managing grouped data points per object.
- Static data members allow sharing a single piece of data across all class instances.
- Static member functions operate on static data without requiring an object instance.
- Arrays of objects enable efficient, loop-driven management of multiple instances.
- Objects can be passed to functions by value or by reference to enable complex object interaction.

Lecture Summary
- Arrays within a class allow managing grouped data points per object.
- Static data members allow sharing a single piece of data across all class instances.
- Static member functions operate on static data without requiring an object instance.
- Arrays of objects enable efficient, loop-driven management of multiple instances.
- Objects can be passed to functions by value or by reference to enable complex object interaction.

To wrap up: we have greatly expanded how we handle state and data in Object-Oriented Programming. We can now store arrays inside objects, share global state across objects using 'static', create bulk instances using object arrays, and pass objects into functions to make them collaborate. Ensure you practice these syntax rules in the lab.