CS-1201 Object-Oriented Programming

Static and Constant Members

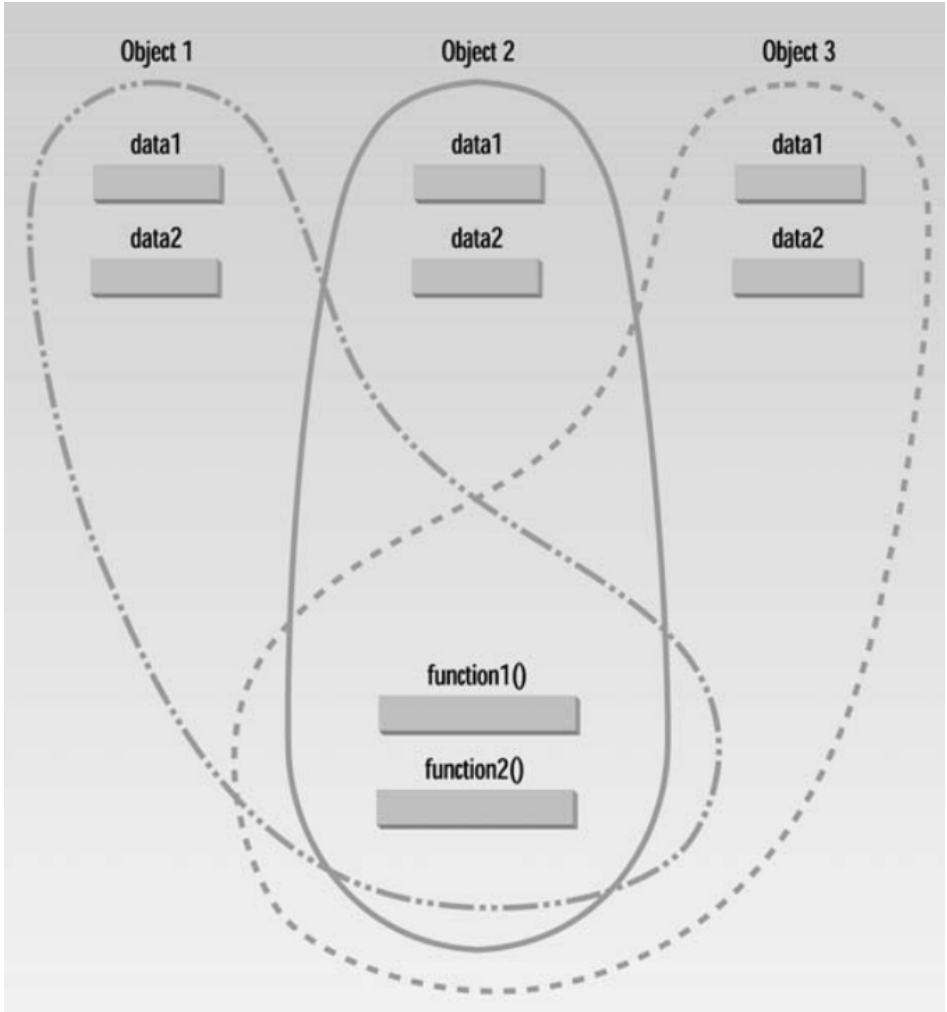
Arbish Akram

Department of Computer Science
Government College University

Objects, Classes, and Memory

- Objects are created based on a class definition.
- All objects of a class share the same member functions.
- Member functions are created once and not duplicated for each object.
- Each object has its own data members (attributes).
- Data values differ for each object, but functions remain shared.
- Separate memory is allocated for each object's data members.

Objects, Classes, and Memory



Static Data Members

- A static data member is shared by all objects of a class.
- Only one copy of the static data exists, no matter how many objects are created.
- Its lifetime spans the entire program (it exists even if no objects are created).
- Access depends on its access specifier.
- Useful for storing information common to all objects of the class.
- Static data is useful when all objects need to share the same information.
- Example: A static variable `count` can track how many objects of a class have been created.
- In a game, a race car class might use a static variable to track how many cars are still in the race.

Example: Non-static Data Members

```
1  class foo
2  {
3      private:
4          int count = 0;
5      public:
6          foo() { //increments count when object created
7              count++;
8          }
9          int getcount() {  //returns count
10             return count;
11         }
12     };
13 int main()
14 {
15     foo f1, f2, f3; //create three objects
16     cout << "count is " << f1.getcount() << endl;
17     cout << "count is " << f2.getcount() << endl;
18     cout << "count is " << f3.getcount() << endl;
19     return 0;
20 }
```

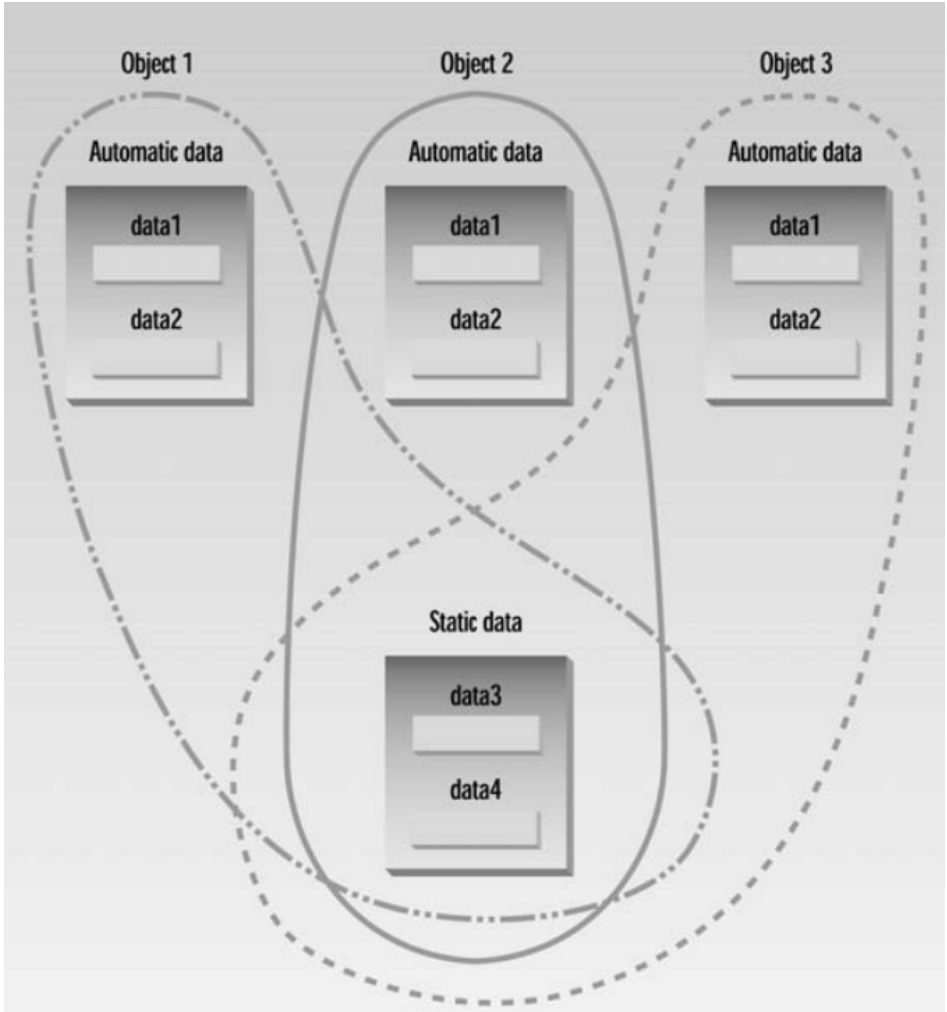
Example: Static Data Members

```
1  class foo
2  {
3      private:
4          static int count; //only one data item for all objects
5      public:
6          foo() {//increments count when object created
7              count++;
8          }
9          int getcount() { //returns count
10             return count;
11         }
12 };
13 int foo::count = 0; //*definition* of count
14 int main()
15 {
16     foo f1, f2, f3; //create three objects
17     cout << "count is " << f1.getcount() << endl;
18     cout << "count is " << f2.getcount() << endl;
19     cout << "count is " << f3.getcount() << endl;
20     return 0;
21 }
```

Static Member Declaration

- Ordinary member variables:
  - Declared and defined in one step (inside the class).
  - Compiler allocates memory automatically.
- Static member variables:
  - Require two steps: declaration and definition.
  - Declaration: inside the class definition.
  - Definition: outside the class, similar to a global variable.

Static vs Automatic Member Variables



Static Member Functions

- Declared using the `static` keyword in the class definition.
- Cannot access non-static member variables directly.
- Used mainly to access or modify static member variables.
- Can be called without creating an object of the class.

```
static ReturnType FunctionName (ParameterTypeList);
```

Why Can't Static Functions Touch Non-static Members?

- Every non-static member function is secretly given a hidden `this` pointer, it tells the function *which object* to work on.
- A static member function has *no object* to be called on, so it has *no `this` pointer*.
- Without `this`, there is no way to know *whose* non-static data to use.
- Declaring a static member function `const` is a *compilation error*, `const` promises not to modify `*this`, but a static function has no `*this` to operate on.
- Static functions can therefore only touch:
  - other static data members, and
  - other static member functions.

Static Member Functions

```
1  #include<iostream>
2  using namespace std;
3  class Budget {
4      private:
5          static double corpBudget;
6      public:
7          static void mainOffice(double amount)
8          {
9              corpBudget += amount;
10         }
11         static double getBudget()  //static accessor
12         {
13             return corpBudget;
14         }
15 };
16 double Budget::corpBudget = 0.0;
17 int main() {
18     Budget::mainOffice(50000.0);   //no object needed
19     Budget::mainOffice(25000.0);
20     cout << "Budget is " << Budget::getBudget() << endl;
21     return 0;
22 }
```

Static Members Exist Without Objects

- A class's static data members and static member functions *exist and can be used even if no objects of that class have been instantiated*.
- To access a **public** static member with no objects in existence, prefix the class name and the scope resolution operator `::`: e.g. `Budget::getBudget()`.
- A **private** or **protected** static member must be accessed through a `public` static member function, called the same way.
- A static member function is a service of the *class*, not of any particular object.

`Budget::mainOffice(50000.0)` is called **before any** `Budget` **object is ever created**, possible because `mainOffice` is static.

Static Constant Members

```
1  #include <iostream>
2  using namespace std;
3  class Example {
4  public:
5      static const int value = 100;  //integral const: OK in-class
6      int id;
7  };
8  int main() {
9      cout << "Static constant value: " << Example::value << endl;
10     Example e1;
11     cout << "id value: " << e1.id <<endl;
12     // cout << "id value: " << Example::id <<endl;  //id is per-object
13     //  Example::value = 150;                        //error: const
14     return 0;
15 }
```

Constant Member Functions

- Non-const member functions can modify data members.
- Const member functions cannot modify data members, the compiler will generate an error if attempted.
- Declared by appending the keyword `const` to the function declaration and definition.
- Const member functions can be called on both `const` and non-`const` objects.
- Ideal for functions that only read data (e.g., getters).
- Helps prevent accidental modifications and clearly communicates intent.

Syntax of Constant Member Functions

1. Declaration inside the class:

```
return_type function_name() const;
int get_data() const;
```

2. Definition inside the class:

```
return_type function_name() const { // function body
}
int get_data() const {
    // function body
}
```

3. Definition outside the class:

```
return_type ClassName::function_name() const { // function body
}
int Demo::get_data() const { // function body
}
```

Const Member Functions

- A `const` member function *guarantees* it will never modify the object's data.
- The `const` keyword goes *after* the parameter list, before the body.

```
class aClass {
private:
    int alpha;
public:
    void nonFunc() {
        alpha = 99;   // OK: non-const member function
    }
    void conFunc() const {
        alpha = 99;   // ERROR: cannot modify a member
    }                 // in a const member function
};
```

Constant Objects

- Class objects can be declared as `const`.
- A `const` object cannot be modified after it is created.
- Only `const` member functions can be called on `const` objects.
- Non-`const` objects can call both `const` and non-`const` member functions.

Example 1: Constant Objects

```
1   #include <iostream>
2   using namespace std;
3   class Rectangle {
4       private:
5           int width, height;
6       public:
7           Rectangle(int w, int h) : width(w), height(h) {}
8           int getWidth() const {
9               return width;
10          }
11          int getHeight() const {
12              return height;
13          }
14  };
15  int main() {
16      const Rectangle rect(10, 5); // Creating a const object
17      // Accessing const member functions
18      cout<<"width: "<<rect.getWidth()<<endl;  // Valid
19      cout<<"height: "<<rect.getHeight()<<endl; // Valid
20      // rect.width = 20; // Error: cannot modify a const object
21      return 0;
22  }
```

Example 2: Constant Objects

```
1  class Distance
2  {
3      private:
4          int feet;
5          float inches;
6      public:
7          Distance(int ft, float in) : feet(ft), inches(in)
8          { }
9      void getdist() {
10         cout << "\nEnter feet: "; cin >> feet;
11         cout << "Enter inches: "; cin >> inches;
12     }
13     void showdist() const
14         { cout << feet << "\'-" << inches << '\"'; }
15 };
16 int main() {
17     const Distance football(300, 0);
18     cout << "football = ";
19     football.showdist();
20     // football.getdist();  // Error: getdist() is not const
21     cout << endl;
22     return 0;
23 }
```

Initializing a `const` Data Member

- A `const` data member (like a reference member) *cannot* be assigned a value in the constructor's **body**, by the time the body runs, initialization has already happened.
- It **must** be initialized using a *member initializer list*: written between the constructor's parameter list and its opening brace, after a colon `:`.
- Multiple member initializers are separated by commas, and the list runs **before** the constructor body executes.

Example: Member Initializer List

```
1  class Increment {
2      public:
3          // member initializer list required for 'increment'
4          Increment(int c, int i) : count(c), increment(i)
5          { }
6          void addIncrement() {
7              count += increment;
8          }
9          void print() const {
10             cout << "count = " << count
11                  << ", increment = " << increment << endl;
12         }
13     private:
14         int count;              // ordinary data member
15         const int increment;    // const data member
16 };
17 int main() {
18     Increment value(10, 5);
19     value.print();           // count = 10, increment = 5
20     value.addIncrement();
21     value.print();           // count = 15, increment = 5
22     return 0;
23 }
```

Error: Assigning a const Member in the Body

```
class Increment {
public:
    Increment(int c, int i) {
        count = c;
        increment = i; // ERROR: assignment of read-only
    }                  // member 'increment'
private:
    int count;
    const int increment;
};
```

By the time the constructor **body** runs, all data members already exist. A `const` member can be given a value only once, **during initialization**, so it must go in the member initializer list, not the body.

Summary

- *Static data members* are shared by all objects; one copy, class-wide lifetime.
- *Static member functions* have no `this` pointer, so they can only use static members, and can be called without an object.
- `static const` of an integral type may be initialized in-class; ordinary statics must be defined outside.
- *Const member functions* promise not to modify the object, the compiler enforces this at compile time.
- *Const objects* can only call const member functions, the two ideas exist for each other.