CS-1201 Object-Oriented Programming

Copy Constructor

Arbish Akram

Department of Computer Science
Government College University

Assigning Objects with =

- Like ordinary variables (but unlike arrays), objects can be assigned to one another using the = operator.
- By *default*, this performs memberwise assignment: each member of the source object is copied into the corresponding member of the destination object.
- Memberwise copying happens in two different situations:
  - Assignment, both objects already exist: `box2 = box1;`
  - Initialization, a new object is being created and given another object's values: `Rectangle box2 = box1;`

Memberwise Assignment in Action

```
1  class Rectangle {
2      private:
3          double width, length;
4      public:
5          Rectangle(double w, double l) : width(w), length(l) {}
6          double getWidth() const  { return width; }
7          double getLength() const { return length; }
8  };
9  int main() {
10     Rectangle box1(10.0, 10.0);
11     Rectangle box2(20.0, 20.0);
12     // Memberwise assignment: copies width and length from box1 into box2
13     box2 = box1;
14     cout << box2.getWidth() << " " << box2.getLength();
15 }
```

- Before `box2 = box1;`, box2 holds 20, 20.
- After it, box2 holds 10, 10, box1's values were copied member by member into box2.
- This works perfectly here because `Rectangle` owns only plain `doubles`. As we'll see shortly, a pointer member changes everything.

Types of Constructors

- Default Constructor
  - A constructor with no parameters.
  - Automatically provided by the compiler if no constructors are defined.
- Parameterized Constructor
  - A constructor that takes arguments to initialize an object with specific values.
- Copy Constructor
  - A constructor that creates a new object as a copy of an existing object.

Example: StudentTestScores

```
1  const double DEFAULT_SCORE = 0.0;
2  class StudentTestScores {
3      private:
4          string studentName;    // The student's name
5          double* testScores;    // Points to array of test scores
6          int numTestScores;     // Number of test scores
7
8          void createTestScoresArray(int size) {
9              numTestScores = size;
10             testScores = new double[size];
11             for (int i = 0; i < size; i++)
12                 testScores[i] = DEFAULT_SCORE;
13         }
14     public:
15         StudentTestScores(string name, int numScores) {
16             studentName = name;
17             createTestScoresArray(numScores);
18         }
19         ~StudentTestScores() {
20             delete[] testScores;
21         }
```

Example: StudentTestScores

```
1       void setTestScore(double score, int index) {
2           testScores[index] = score;
3       }
4       void setStudentName(string name) {
5           studentName = name;
6       }
7       string getStudentName() const {
8           return studentName;
9       }
10      int getNumTestScores() const {
11          return numTestScores;
12      }
13      double getTestScore(int index) const {
14          return testScores[index];
15      }
16  };
```

Pointer-Based Member in a Class

- A potential issue with this class is that one of its members, `testScores`, is a *pointer*.
- `createTestScoresArray` dynamically allocates memory for the `testScores` array and initializes its elements.
- For example, the following line creates a `StudentTestScores` object named `student1`, whose `testScores` points to dynamically allocated memory holding 5 `doubles`:

```
StudentTestScores student1("Maria Jones Tucker", 5);
```

Copy Constructor:

- A copy constructor is a special constructor that is called whenever a new object is created and initialized with another object's data.
- Every class in C++ has a copy constructor, either one the compiler writes for you, or one you write yourself.
  - Default (compiler-generated) copy constructor
  - User-defined copy constructor

When Is the Copy Constructor Called?

- When an object is initialized from another of the same type:

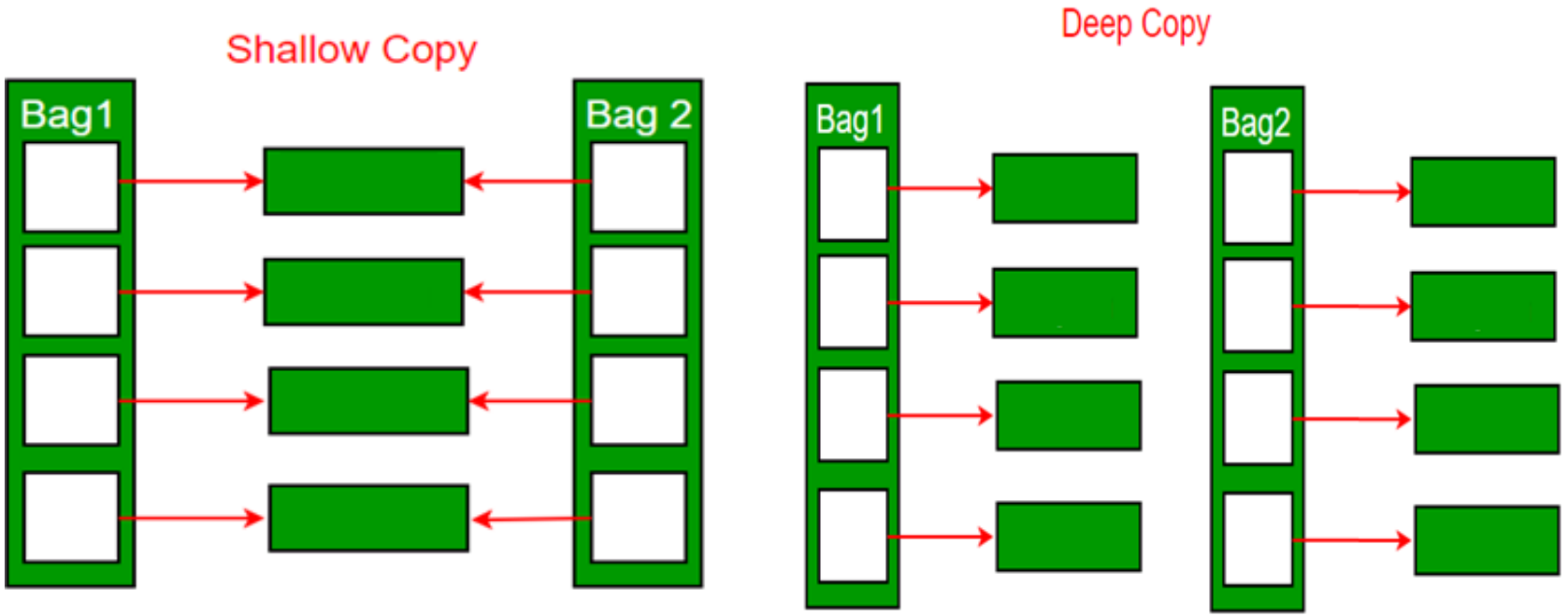
  `MyClass obj2 = obj1;` or `MyClass obj2(obj1);`
- When an object is passed by value to a function.
- When an object is returned by value from a function.
- When an object is thrown or caught by value in exceptions.

Default Copy Constructor

- Automatically provided by the compiler.
- Performs a *shallow copy*: each member is copied, value by value (this is just memberwise assignment applied to initialization).
- Fine for plain data like `int`, `double`, or `string`, there is nothing to share.

```
1    class MyClass {
2        public:
3            int value;
4    };
5
6    MyClass obj1;
7    obj1.value = 10;
8    MyClass obj2 = obj1; // Default copy constructor is called
```

Deep vs Shallow Copy



- Shallow copy: both objects' pointers hold the same address, one locker, two keys.
- Deep copy: the pointer's target is duplicated too, each object gets its own locker.

Why Is Deep Copy Needed?

- The default copy constructor performs a shallow copy.
- Shallow copy simply copies pointer values, multiple objects end up pointing to the same memory.
- Problem: this creates shared ownership, which can cause:
  - Memory corruption (one object's change affects the other)
  - *Double deletion* when both destructors run
- A deep copy allocates new memory and copies the actual data into it.
- Deep copy is essential whenever a class owns:
  - Pointers to dynamically allocated memory
  - File handles or other system resources

User-Defined Copy Constructor: Syntax

```
ClassName(const ClassName &other) {
    // Copy member variables from 'other'
}
```

- `ClassName`: name of the class.
- `const ClassName &other`:
  - Passed by reference to avoid an infinite loop of copies.
  - Marked `const` so the source object cannot be modified, there's no reason a copy constructor should change the data it's copying from.
- The body performs a shallow or deep copy, depending on what the data members require.

Closing the Loop: Deep-Copying StudentTestScores

- `testScores` is a pointer owned by the class, allocated in the constructor and freed in the destructor.
- If we relied on the default copy constructor here, copying a `StudentTestScores` object would copy the pointer's address, both objects would share one array, and both destructors would eventually try to `delete[]` it.
- The fix: write our own copy constructor.

```
1  // Deep-copy constructor for StudentTestScores
2  StudentTestScores(const StudentTestScores &other) {
3      studentName   = other.studentName;      // string handles its own copy
4      numTestScores = other.numTestScores;
5      // allocate NEW memory -- do not reuse other's pointer
6      testScores = new double[numTestScores];
7      // copy the actual values, not the address
8      for (int i = 0; i < numTestScores; i++)
9          testScores[i] = other.testScores[i];
10 }
```

Why This Matters

- Without this user-defined copy constructor, writing `StudentTestScores s2 = s1;` would leave `s1` and `s2` pointing at the same array.
- Changing a score through `s2` would silently change `s1`'s data too.
- When `s1` and `s2` both go out of scope, their destructors would both call `delete[] testScores` on the same address, undefined behavior.
- The deep-copy constructor prevents all of this by giving `s2` its own array with the same values.

Example: Student (No Pointers)

```
1  // C++ program to illustrate the use of a copy constructor
2   #include <iostream>
3   using namespace std;
4   class Student {
5       int rollNumber;
6       string Name;
7       public:
8           Student(int, string);
9           // Copy constructor
10          Student(const Student &t) {
11              rollNumber = t.rollNumber;
12              Name = t.Name;
13              cout << "Copy Constructor Called" << endl;
14          }
15          void display();
16  };
17
18  // Parameterized constructor
19  Student::Student(int number, string name) {
20      rollNumber = number;
21      Name = name;
22  }
```

Contrast Example: Student (continued)

```
1  void Student::display() {
2      cout << rollNumber << "\t" << Name << endl;
3  }
4
5  int main() {
6      Student s1(501, "Ali");
7      s1.display();
8
9      Student s2(s1);   // copy constructor called
10     s2.display();
11
12     return 0;
13 }
```

- Student has no pointer members, int and string each manage their own memory correctly.
- Here, writing a copy constructor is optional: the compiler-generated shallow copy would already do the right thing.
- It's defined here mainly to show the syntax and print a message so you can see exactly when it runs.

Summary

- The = operator performs memberwise assignment by default, for existing objects that's assignment; for a new object being created, it's initialization, and that's what calls the copy constructor.
- The compiler-generated copy constructor copies members value by value (shallow copy).
- That's correct for plain data, but wrong for a pointer member, it copies the address, not what it points to.
- Two objects then share one resource, risking memory corruption and double deletion.
- A deep-copy constructor allocates new memory and copies the actual contents, giving each object independent ownership.
- If your class owns a pointer, a file handle, or any dynamically-managed resource, write your own copy constructor. If it only holds plain values, the default one is already correct.