CS-1201 Object-Oriented Programming

Object Manipulation

Arbish Akram

Department of Computer Science
Government College University

How Do Objects Move Through a Program?

- So far, every object we have built has just sat inside `main()`.
- Real programs move objects *around*: into functions, back out of functions, and into collections.
- Today we ask three very practical questions:
  1. What happens when I *pass* an object into a function?
  2. What happens when a function *returns* an object?
  3. How do I work with *many* objects of the same class at once?

Passing Objects to Functions

- An object can be passed as an argument to a member function or a non-member function, just like an `int` or a `double`.
- This lets a function use the data inside one or more objects to compute something new.
- C++ passes objects *by value*. The function receives a new *copy* of your object, not the original.

Example 1: Passing Objects to a Function

```cpp
// C++ program to calculate the average marks of two students
#include <iostream>
using namespace std;

class Student {
    public:
        double marks;
        // constructor to initialize marks
        Student(double m) : marks(m) { }
};

// function that takes objects as parameters
double average_marks(Student s1, Student s2) {
    // s1 and s2 are COPIES of the arguments passed in
    return (s1.marks + s2.marks) / 2;
}
```

Example 1: Passing Objects to a Function

```
#include<iostream>

class Student {...};

void calculateAverage(Student s1, Student s2) {
    // code
}

int main() {
    ... ...
    calculateAverage(student1, student2);
    ... ...
}
```

```
1  int main() {
2      Student student1(88.0), student2(56.0);
3      // pass the objects as arguments -- copies are made
4      cout << "Average Marks = "
5           << average_marks(student1, student2) << "\n";
6      return 0;
7  }
```

Trace It Through

- `student1` and `student2` live in `main()`.
- When we call `average_marks(student1, student2)`, C++ builds two **new** `Student` objects, `s1` and `s2`, copying the `marks` values in.
- `average_marks` computes using `s1` and `s2`, `student1` and `student2` are never touched.
- If `average_marks` modified `s1.marks`, would `student1.marks` change back in `main()`? **No**, `s1` is a separate copy.
- This is exactly the same rule as passing an `int` by value; objects don't get special treatment.

Example 2: Passing Objects to a Function

```
1class Distance {
2  private:
3      int feet;
4      float inches;
5  public:
6      Distance() : feet(0), inches(0.0)
7      { }
8      Distance(int ft, float in) : feet(ft), inches(in)
9      { }
10     void getdist() {
11         cout << "\nEnter feet: "; cin >> feet;
12         cout << "Enter inches: "; cin >> inches;
13     }
14     void showdist() {
15         cout << feet << "\'-" << inches << '\"';
16     }
17     void add_dist(Distance, Distance);
18 };
```

Example 2: Passing Objects to a Function

```
1  void Distance::add_dist(Distance d2, Distance d3)
2  {
3   inches = d2.inches + d3.inches; //add the inches
4   feet = 0;
5   if (inches >= 12.0) {
6       inches -= 12.0;
7       feet++;
8   }
9   feet += d2.feet + d3.feet;
10 }
11 int main() {
12      Distance dist1, dist3;
13      Distance dist2(11, 6.25);
14      dist1.getdist();          // e.g. user enters 5 ft, 8.5 in
15      dist3.add_dist(dist1, dist2);
16      cout << "\ndist1 = "; dist1.showdist();
17      cout << "\ndist2 = "; dist2.showdist();
18      cout << "\ndist3 = "; dist3.showdist();
19      cout << endl;
20      return 0;
21 }
```

Returning Objects from Functions

- A function's return type can be a class type, just like *int* or *double*.
- The function builds an object *locally*, then returns it, the caller receives a *copy* of that object.
- This mirrors passing objects *in*: value semantics apply on the way *out* too.

Example 1: Returning an Object from a Function

```
1  class Student {
2     public:
3      double marks1, marks2;
4  };
5  // function that returns a Student object
6  Student createStudent() {
7      Student student;
8      // Initialize member variables of Student
9      student.marks1 = 96.5;
10     student.marks2 = 75.0;
11     // print member variables of Student
12     cout << "Marks 1 = " << student.marks1 << endl;
13     cout << "Marks 2 = " << student.marks2 << endl;
14     return student;   // a copy of 'student' is returned
15 }
16 int main() {
17     Student student1;
18     // Call function
19     student1 = createStudent();
20     return 0;
21 }
```

Key Insight: Local Object, Returned Copy

- `student` inside `createStudent()` is a *local* object, it is destroyed the moment the function ends.
- So how does `main()` still see its data?
- Before `student` is destroyed, its values are *copied* into the returned object, which then gets assigned to `student1`.

Example 2: Returning an Object from a Function

```
1  Distance Distance::add_dist(Distance d2)
2  {
3      Distance temp;                          // temporary local object
4      temp.inches = inches + d2.inches;       // add the inches
5      temp.feet = 0;
6      if (temp.inches >= 12.0)
7      {
8          temp.inches -= 12.0;
9          temp.feet = 1;
10     }
11     temp.feet += feet + d2.feet;            // add the feet
12     return temp;                            // a copy of temp is returned
13 }
14 int main() {
15     Distance dist1, dist3;
16     Distance dist2(11, 6.25);
17     dist1.getdist();
18     dist3 = dist1.add_dist(dist2);
19     cout << "\ndist1 = "; dist1.showdist();
20     cout << "\ndist2 = "; dist2.showdist();
21     cout << "\ndist3 = "; dist3.showdist();
22     cout << endl;
23     return 0;
24 }
```

Array of Objects

- An array of objects is a collection of instances of a class stored in contiguous memory locations.
- Syntax: `ClassName arrayName[arraySize];`
- Each element is a full, independent object, with its own copy of every data member.

Array of Objects

```
1  class Employee
2  {
3      int id;
4      int salary;
5      public:
6          void setId() {
7              salary = 122;
8              cout << "Enter the id of employee" << endl;
9              cin >> id;
10         }
11         void getId() {
12             cout << "The id of this employee is " << id << endl;
13         }
14 };
15 int main() {
16     // compiler-generated default constructor runs 4 times here
17     Employee arr[4];
18     for (int i = 0; i < 4; i++) {
19         arr[i].setId();
20         arr[i].getId();
21     }
22     return 0;
23 }
```