CS256

Chapter 13: Introduction to Classes

Hanoi University of Science and Technology

Slide contents are mainly based on the following documents:

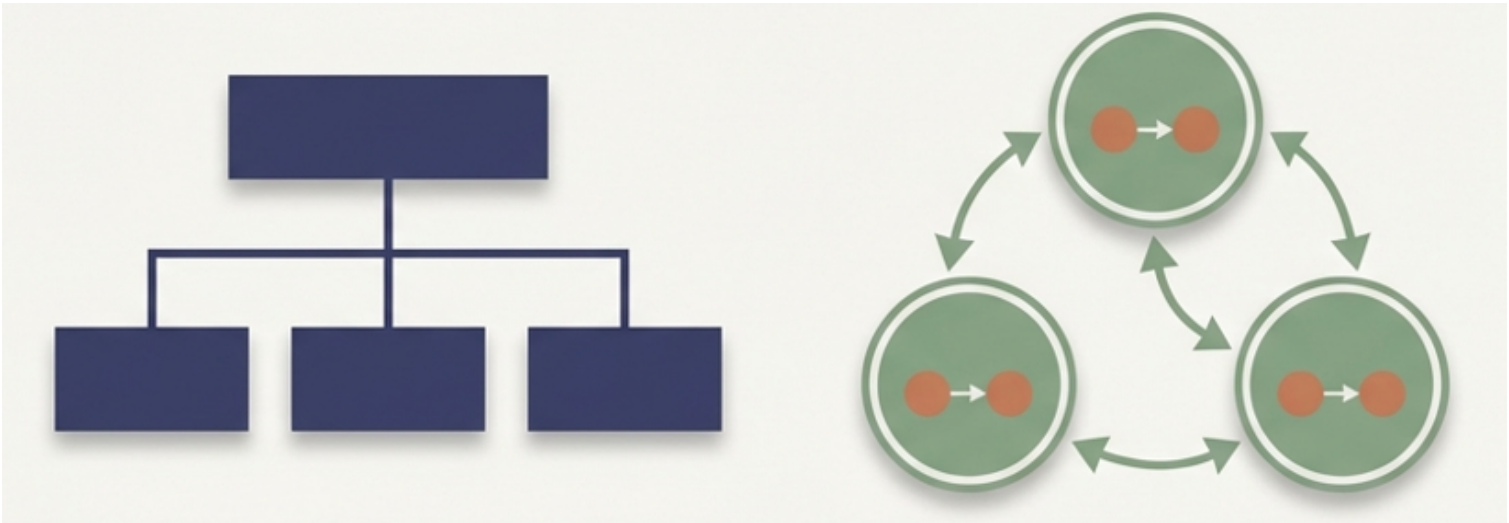
- Tony Gaddis, Starting out with C++, Eighth Edition, Pearson, 2015
- Tony Gaddis, Power Points Slides:
  `https://www.umsl.edu/~antognolij/gaddis.html`

Limitations of Procedural Programming

- If the data structures change, many functions must also be changed.
- Programs that are based on complex function hierarchies are:
  - Difficult to understand and maintain
  - Difficult to modify and extend
  - Easy to break

Procedural and Object-Oriented Programming

Procedural Oriented	**Object Oriented**
functions	data objects
top-down approach	bottom-up approach
difficult to modify, extend	easy to upgrade, modularized

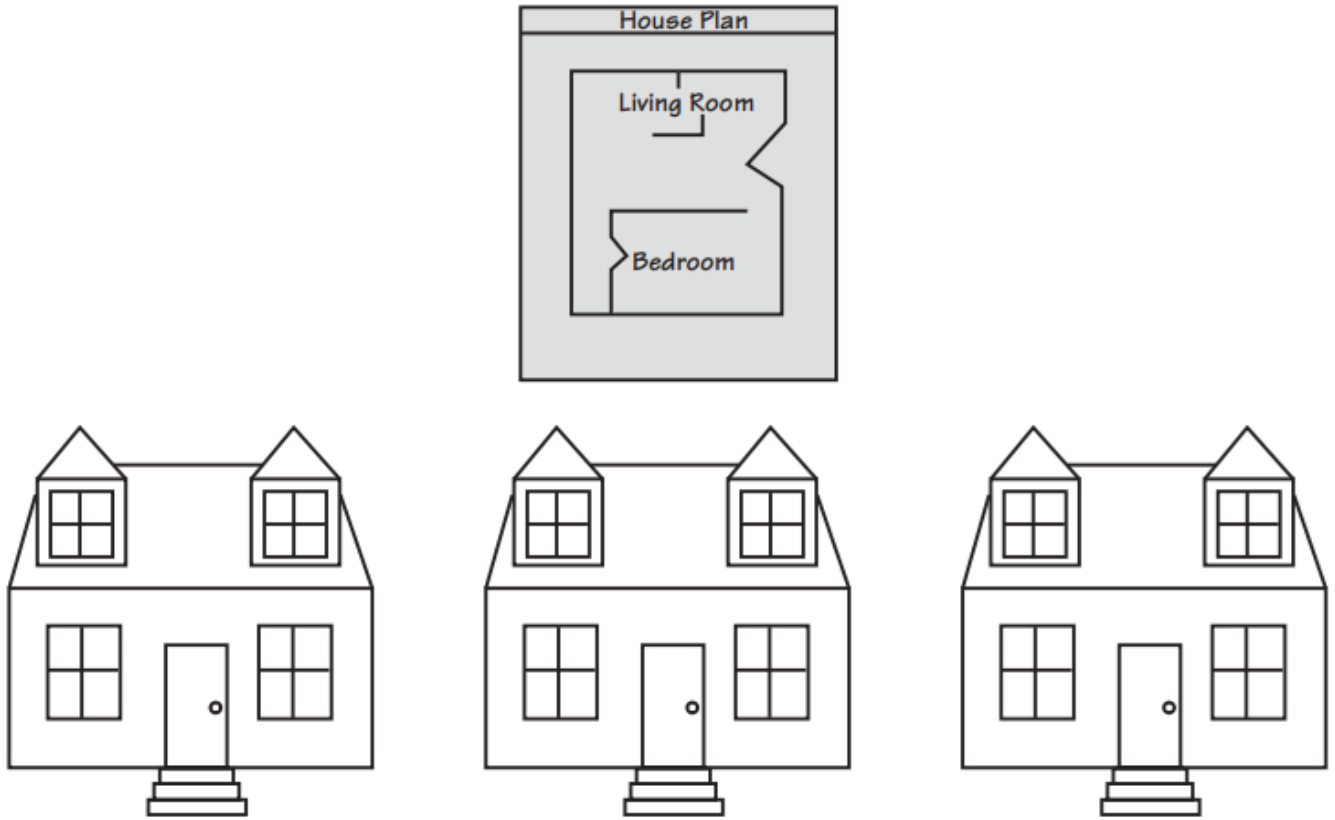


Object-Oriented Programming Terminology

- **Class**: like a `struct` (allows bundling of related variables), but variables and functions in the class can have different properties than in a `struct`.
- **Object**: An instance of a class, in the same way that a variable can be an instance of a `struct`.

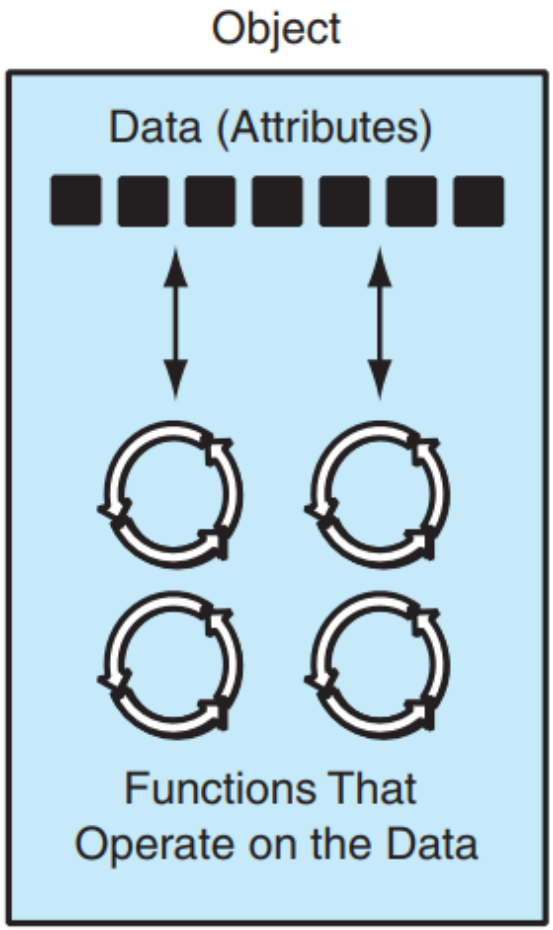
Classes and Objects

- A **Class** is like a blueprint.
- **Objects** are like the houses built from that blueprint.



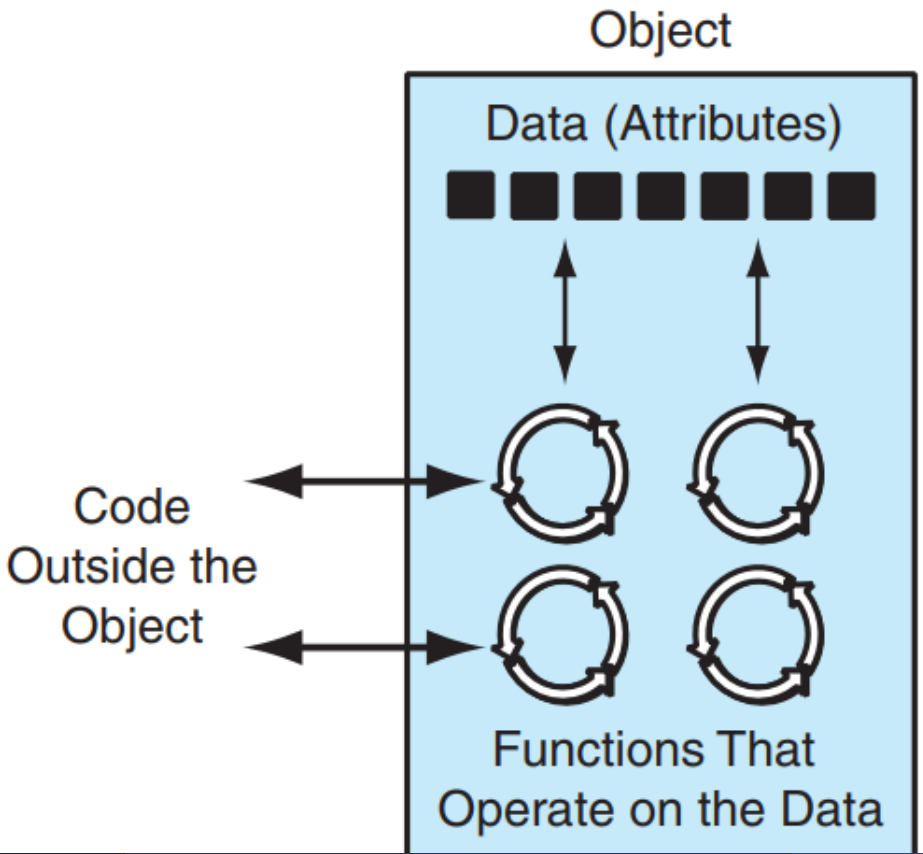
More Object-Oriented Terminology

- **Attributes**: Data that an object holds.
- **Methods or Behaviors**: Member functions of a class.



More on Objects

- **Data Hiding**: Restricting access to certain members of an object.
- **Public Interface**: Members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides protection from data corruption.



Introduction to Classes in C++

- Objects are created from a class.
- **Format**:

```
class ClassName
{
    declaration;
    declaration;
};
```

Access Specifiers

- Used to control access to members of the class.
- **public**: Can be accessed by functions outside of the class.
- **private**: Can only be called by or accessed by functions that are members of the class.

```
// Rectangle class declaration.
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```

Access Specifiers

- Can be listed in any order in a class.
- Can appear multiple times in a class.
- If not specified, the default is `private`.

Using const With Member Functions

- const appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

```
// Rectangle class declaration.
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```

Defining a Member Function

When defining a member function:

1. Put the prototype in the class declaration.
2. Define the function using the class name and the scope resolution operator (`::`).

```
int Rectangle::setWidth(double w)
{
    width = w;
}
```

Accessors and Mutators

- **Mutator**: A member function that stores a value in a private member variable, or changes its value in some way.
- **Accessor**: A function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.

Defining an Instance of a Class

- An object is an instance of a class.
- Defined like structure variables:

```
Rectangle box;
```

- Access members using the dot operator:

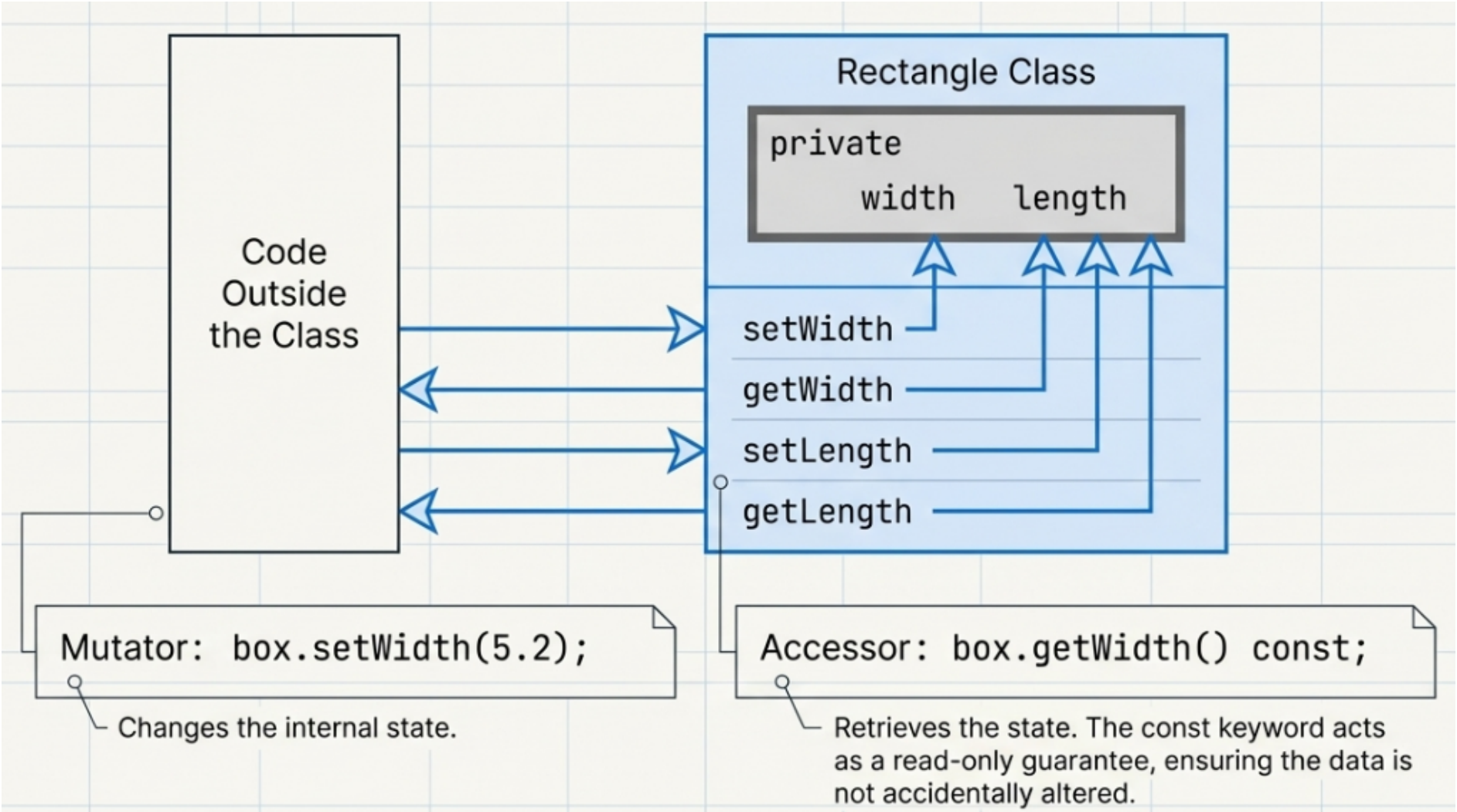
```
box.setWidth(5.2);
cout << box.getWidth();
```

- *Note:* You will get a compiler error if you attempt to access a private member using the dot operator.

Accessors and Mutators

- **Mutator**: A member function that stores a value in a private member variable, or changes its value in some way.
- **Accessor**: A function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.

Accessors and Mutators



Example - Program 9.1

Program 13-1

```
1  // This program demonstrates a simple class.
2  #include <iostream>
3  using namespace std;
4
5  // Rectangle class declaration.
6  class Rectangle
7  {
8      private:
9          double width;
10         double length;
11     public:
12         void setWidth(double);
13         void setLength(double);
14         double getWidth() const;
15         double getLength() const;
16         double getArea() const;
17 };
18
19 //*************************************************
20 // setWidth assigns a value to the width member.  *
21 //*************************************************
22
```

Example - Program 9.1

```
23  void Rectangle::setWidth(double w)
24  {
25      width = w;
26  }
27
28  //*****************************************************
29  // setLength assigns a value to the length member. *
30  //*****************************************************
31
32  void Rectangle::setLength(double len)
33  {
34      length = len;
35  }
36
37  //*****************************************************
38  // getWidth returns the value in the width member. *
39  //*****************************************************
40
41  double Rectangle::getWidth() const
42  {
43      return width;
44  }
45
46  //*****************************************************
47  // getLength returns the value in the length member. *
48  //*****************************************************
```

Example - Program 9.1

```
49
50  double Rectangle::getLength() const
51  {
52      return length;
53  }
54
55  //*******************************************************
56  // getArea returns the product of width times length. *
57  //*******************************************************
58
59  double Rectangle::getArea() const
60  {
61      return width * length;
62  }
63
64  //*******************************************************
65  // Function main                                       *
66  //*******************************************************
67
68  int main()
69  {
70      Rectangle box;      // Define an instance of the Rectangle class
71      double rectWidth;   // Local variable for width
72      double rectLength;  // Local variable for length
73
74      // Get the rectangle's width and length from the user.
75      cout << "This program will calculate the area of a\n";
76      cout << "rectangle. What is the width? ";
```

(program continues)

Example - Program 9.1

Program 13-1 *(continued)*

```
77        cin >> rectWidth;
78        cout << "What is the length? ";
79        cin >> rectLength;
80
81        // Store the width and length of the rectangle
82        // in the box object.
83        box.setWidth(rectWidth);
84        box.setLength(rectLength);
85
86        // Display the rectangle's data.
87        cout << "Here is the rectangle's data:\n";
88        cout << "Width: " << box.getWidth() << endl;
89        cout << "Length: " << box.getLength() << endl;
90        cout << "Area: " << box.getArea() << endl;
91        return 0;
92  }
```

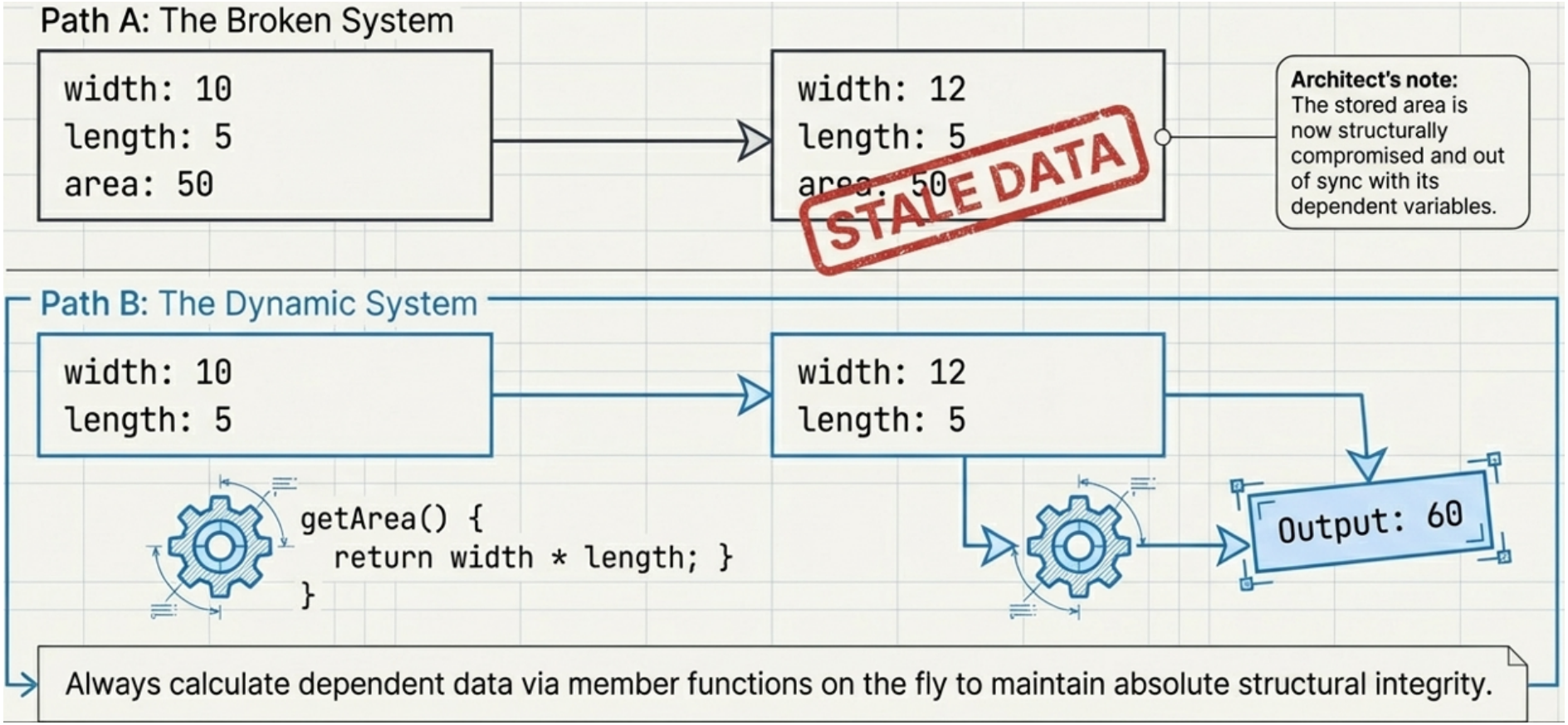
Program Output with Example Input Shown in Bold

```
This program will calculate the area of a
rectangle. What is the width? 10 [Enter]
What is the length? 5 [Enter]
Here is the rectangle's data:
Width: 10
Length: 5
Area: 50
```

Avoiding Stale Data

- Some data is the result of a calculation (e.g., the area of a rectangle is `length x width`).
- If we were to use an `area` variable in the `Rectangle` class, its value would depend on the length and the width.
- If we change the length or width without updating the area, then the area becomes **stale**.
- To avoid stale data, calculate the value of that data within a member function rather than storing it in a variable.

Defining an Instance of a Class



Pointers to Objects & Dynamic Allocation

- Can define a pointer to an object:

```
Rectangle *rPtr;
```

- Can access public members via the pointer using ->:

```
rPtr = &otherRectangle;
rPtr->setLength(12.5);
cout << rPtr->getLength() << endl;
```

- We can also use a pointer to dynamically allocate an object:

```
Rectangle *r = new Rectangle(10, 20);
```

Separating Specification from Implementation

- Place class declarations in a header file (specification file) like `Rectangle.h`.
- Place member function definitions in a `.cpp` file like `Rectangle.cpp`.
- The `.cpp` file should `#include` the class specification file.
- Programs that use the class must also `#include` the specification file and be linked with the compiled member function definitions.

Inline Member Functions

- Member functions can be defined **inline** (directly inside the class declaration).
- Inline is appropriate for short function bodies.

```
class Rectangle {
private:
    double width;
    double length;
public:
    void setWidth(double);
    void setLength(double);
    double getWidth() const { return width; }
    double getLength() const { return length; }
    double getArea() const { return width * length; }
};
```

Constructors

- A member function that is automatically called when an object is created.
- Purpose is to construct/initialize an object.
- Constructor function name is the same as the class name.
- Has **no** return type.
- If you write a class with no constructor, C++ writes a default one for you.

Constructors

Contents of Rectangle.cpp (Version 3)

```
1  // Implementation file for the Rectangle class.
2  // This version has a constructor.
3  #include "Rectangle.h"    // Needed for the Rectangle class
4  #include <iostream>       // Needed for cout
5  #include <cstdlib>        // Needed for the exit function
6  using namespace std;
7
8  //*********************************************************
9  // The constructor initializes width and length to 0.0.   *
10 //*********************************************************
11
12 Rectangle::Rectangle()
13 {
14     width = 0.0;
15     length = 0.0;
16 }
17
18 //*********************************************************
19 // setWidth sets the value of the member variable width.   *
20 //*********************************************************
21
```

Constructors

Program 13-7

```
1  // This program uses the Rectangle class's constructor.
2  #include <iostream>
3  #include "Rectangle.h" // Needed for Rectangle class
4  using namespace std;
5
6  int main()
7  {
8      Rectangle box;     // Define an instance of the Rectangle class
9
10     // Display the rectangle's data.
11     cout << "Here is the rectangle's data:\n";
12     cout << "Width: " << box.getWidth() << endl;
13     cout << "Length: " << box.getLength() << endl;
14     cout << "Area: " << box.getArea() << endl;
15     return 0;
16 }
```

Program Output

```
Here is the rectangle's data:
Width: 0
Length: 0
Area: 0
```

Passing Arguments to Constructors

- To create a constructor that takes arguments, indicate parameters in the prototype:

```
Rectangle(double, double);
```

- Use parameters in the definition:

```
Rectangle::Rectangle(double w, double len) {
    width = w;
    length = len;
}
```

- Pass arguments when creating the object:

```
Rectangle r(10, 5);
```

More About Constructors

- **Default Arguments**: If all of a constructor's parameters have default arguments, it acts as a default constructor.

```
Rectangle(double = 0, double = 0);
```

- **No Default Constructor**: When all of a class's constructors require arguments, the class has no default constructor. You *must* pass arguments when creating an object.

Overloading Constructors

- A class can have more than one constructor as long as they have different parameter lists.

```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```

- **Rule**: Do not provide more than one default constructor (e.g., one with no arguments and one where all arguments have defaults).

```
Square();
Square(int = 0);  // will not compile
```

Destructors

- A member function automatically called when an object is destroyed.
- Name is ~`classname` (e.g., ~`Rectangle`).
- Has no return type and takes no arguments.
- Only **one** destructor per class (cannot be overloaded).
- If the constructor allocates dynamic memory, the destructor should release it using `delete`.

Arrays of Objects

- Objects can be elements of an array:

```
InventoryItem inventory[40];
```

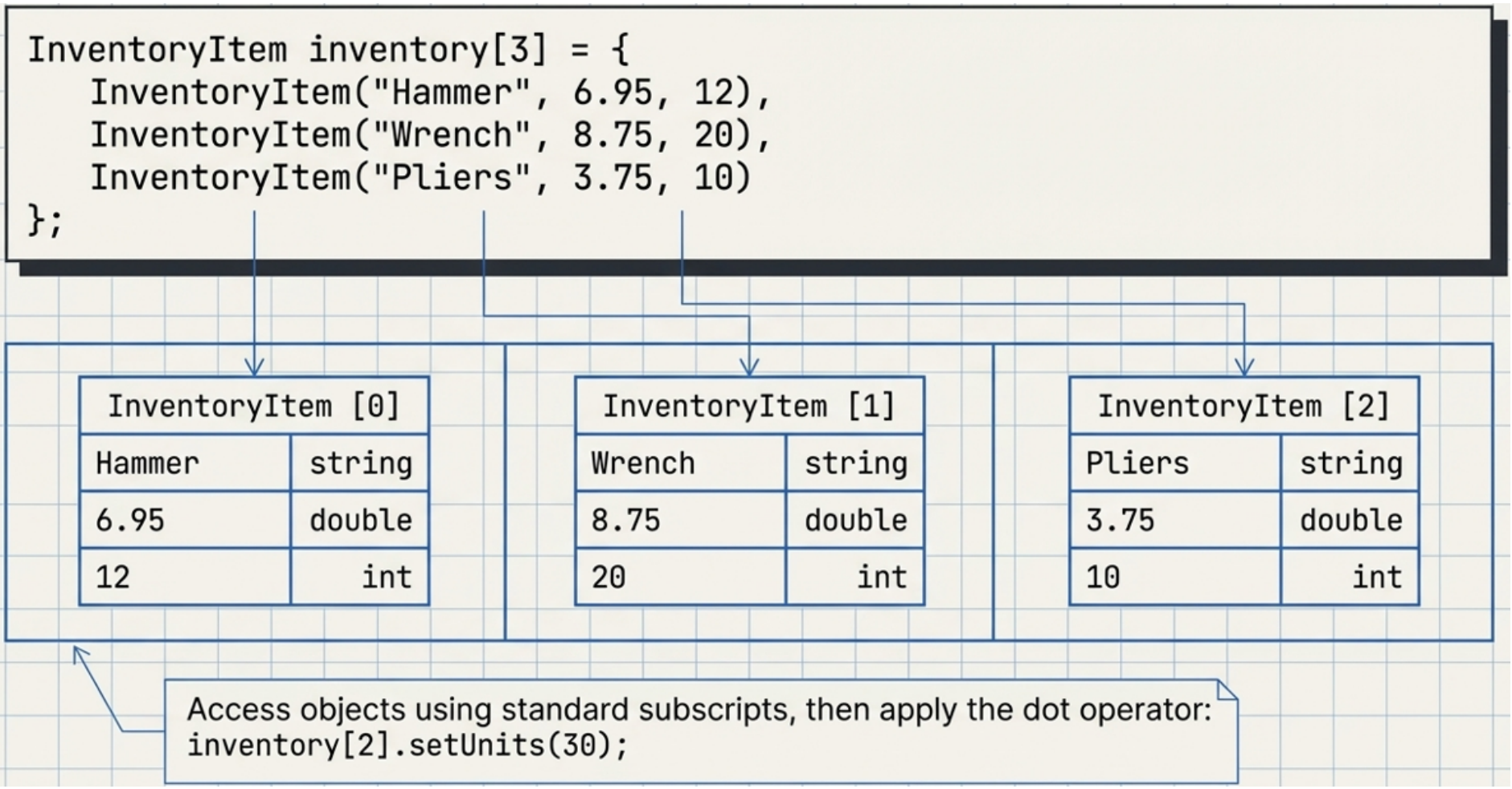
- The default constructor is used when the array is defined.
- Must use an initializer list to invoke a constructor that takes arguments:

```
InventoryItem inventory[3] =
   {InventoryItem("Hammer", 6.95, 12),
    InventoryItem("Wrench", 8.75, 20),
    InventoryItem("Pliers", 3.75, 10)};
```

- Access members using subscripts and dot notation:

```
inventory[2].setUnits(30);
```

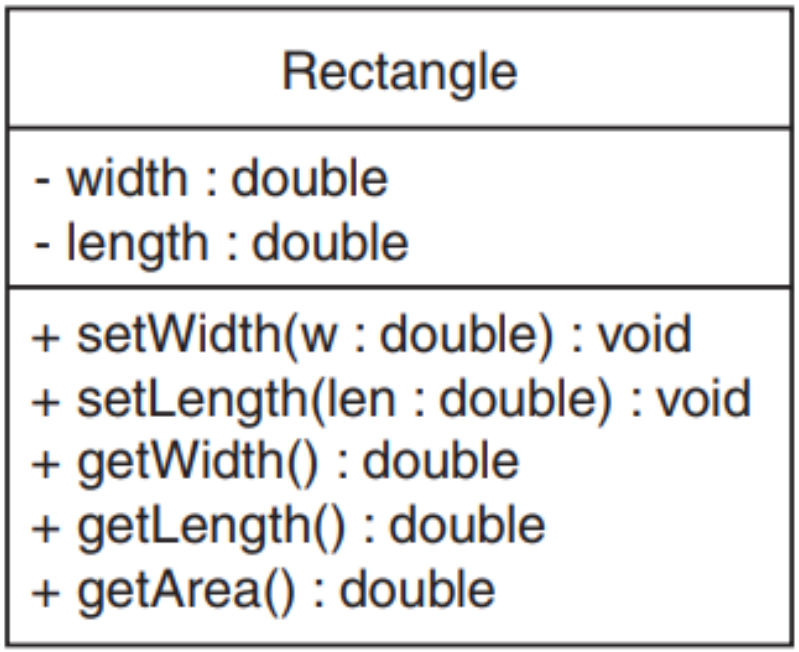
Arrays of Objects

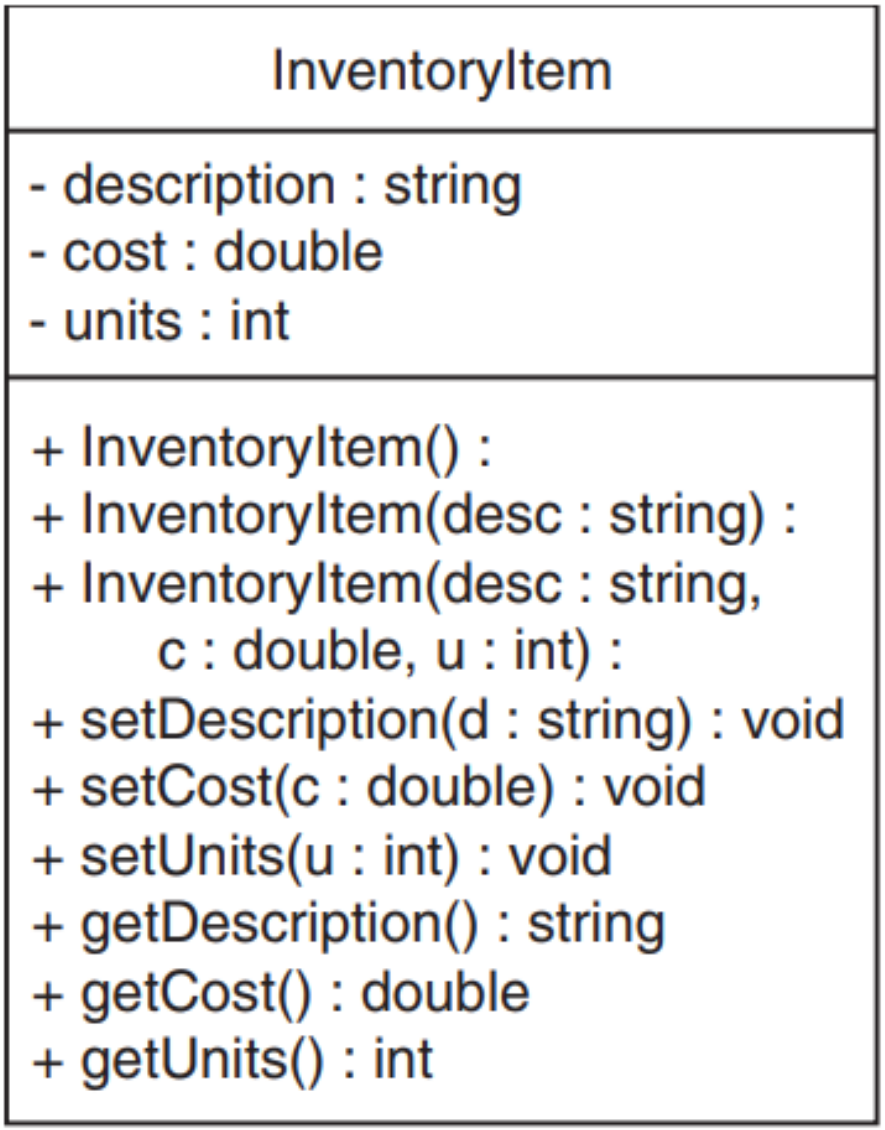


The Unified Modeling Language (UML)

- UML stands for Unified Modeling Language.
- Provides a set of standard diagrams for graphically depicting object-oriented systems.
- **UML Class Diagram**: Has three main sections (Class Name, Member Variables, Member Functions).
- **Access Specifiers**: Minus (-) for private, Plus (+) for public.
- **Data Types**: Placed after the variable or parameter name, separated by a colon.
- **Example**: `- width : double`
- **Return Types**: `+ setWidth(w : double) : void`

The Unified Modeling Language (UML)





Exercises

- Programming Challenges: 1, 2, 3, 4, 5.