Object-Oriented Programming Approach

黃世強 (Sai-Keung Wong)

National Yang Ming Chiao Tung University

Taiwan

Intended Learning Outcomes

- List the key ideas on the object-oriented programming approach
- Distinguish between class variables and instance variables

Using C++ string class

Two ways to process strings in C++:

1) Treat a string as an array ending with the null terminator (‘\0’).

char s[100] = “here”;

2) Use the string class

string myStr = “here”;

Using C++ string class

Two ways to process strings in C++:

1) Treat a string as an array ending with the null terminator ('\0').

```
//C-string
char s[100] = "here";
```

Approach 1: A1 to know how the string A2

2) Use the string class

```
// C++ string
string myStr = "here";
```

Approach 2: A3 to know how the string is stored.

Constructing a string object

We can create an empty string using the string's no-arg constructor:

```
string newString;
```

Constructing a string object

We can create an empty string using the string's no-arg constructor:

```
string newString;
```

We can create a string object from a string value or from an array of characters.

To create a string from a string literal:

```
#define  stringLiteral    "My string"

string newString( stringLiteral ); // create a string object with an argument
```

Constructing a string object

We can create an empty string using the string's no-arg constructor:

```
string newString;
```

We can create a string object from a string value or from an array of characters.

To create a string from a string literal:

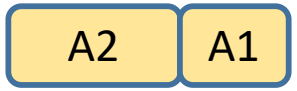
```
#define  stringLiteral    "My string"

string newString( stringLiteral ); // create a string object with an argument
```

Assigning to a C-string structure

To create a C-string?

Assigning to a C-string structure

To create a C-string?



NUM_CHARS = 32;

char myString[NUM_CHARS];

Assigning to a C-string structure

To create a C-string?

```
const int NUM_CHARS = 32;
char myString[ NUM_CHARS ]; // is this size enough?
```

Assigning to a C-string structure

To create a C-string?

```
const int NUM_CHARS = 32;
char myString[ NUM_CHARS ]; // is this size enough?

#define  stringLiteral        “My string”

myString = stringLiteral;
```

 //This is an assignment. It does not work to a C-string.

Assigning to a C-string structure

To create a C-string?

```
const int NUM_CHARS = 32;
char myString[ NUM_CHARS ]; // is this size enough?

#define  stringLiteral        “My string”

myString = stringLiteral;
```

//This is an assignment. It does not work to a C-string.

Need to define a function and then use the function to set the string.

e.g., assignString(myString, **size**, stringLiteral);

Assigning to a C-string structure

To create a C-string?

```
const int NUM_CHARS = 32;
char myString[ NUM_CHARS ]; // is this size enough?
```

```
#define  stringLiteral        “My string”
```

```
myString = stringLiteral;
```
 //This is an assignment. It does not work to a C-string.

Need to define a function and then use the function to set the string.

e.g., assignString(myString, **size**, stringLiteral);

In C++:

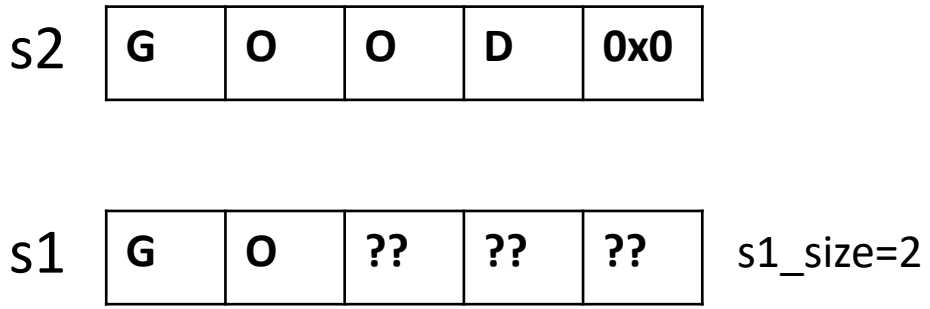
```
string myString;
myString = stringLiteral;
```

assignString(myString, size, stringLiteral);

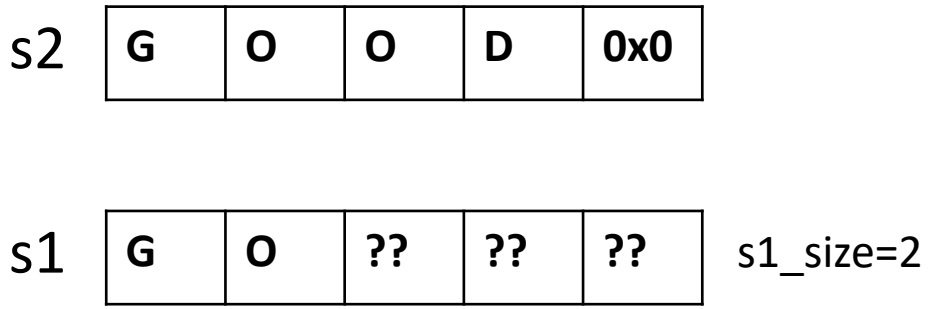
s2

G	O	O	D	0x0

assignString(myString, size, stringLiteral);

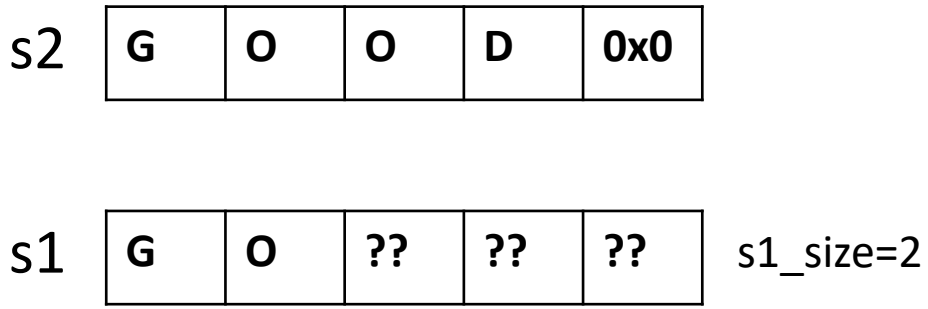


assignString(myString, size, stringLiteral);



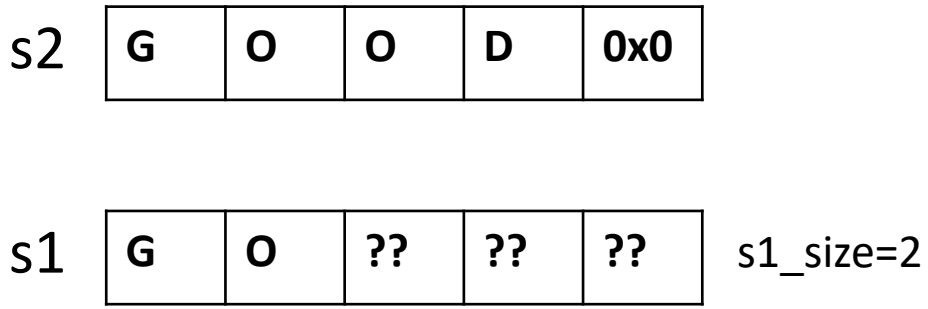
assignString(myString, size, stringLiteral);

```
void assignString( char *s1, int s1_size, const char *s2 ) {
        int i = 0;
        while ( i < s1_size ) {
                if ( s2[i] == 0 ) {break; }
                s[i] = s2[i];
                ++i;
        }
         s1[i] = 0x0;            // the null terminator for a string
}
```



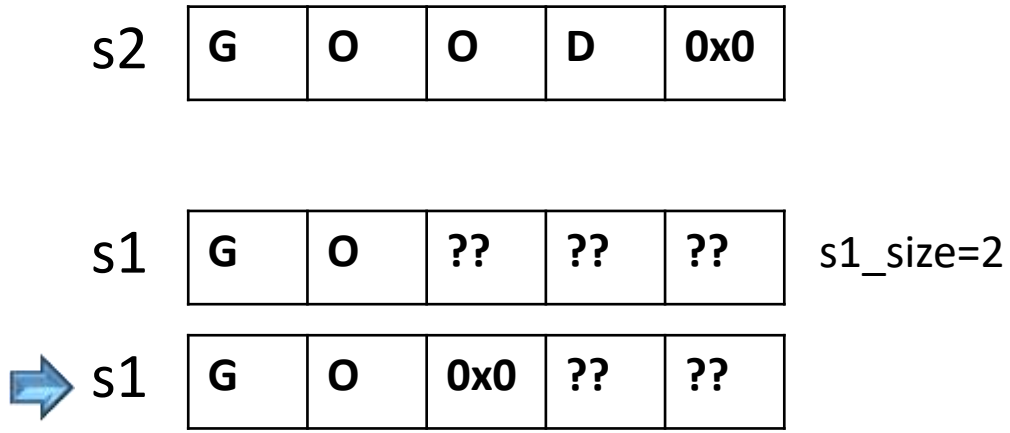
assignString(myString, size, stringLiteral);

```
void assignString( char *s1, int s1_size, const char *s2 ) {
        int i = 0;
        while ( i < s1_size ) {
                if ( s2[i] == 0 ) {break; }
                s[i] = s2[i];
                ++i;
        }
         s1[i] = 0x0;            // the null terminator for a string
}
```



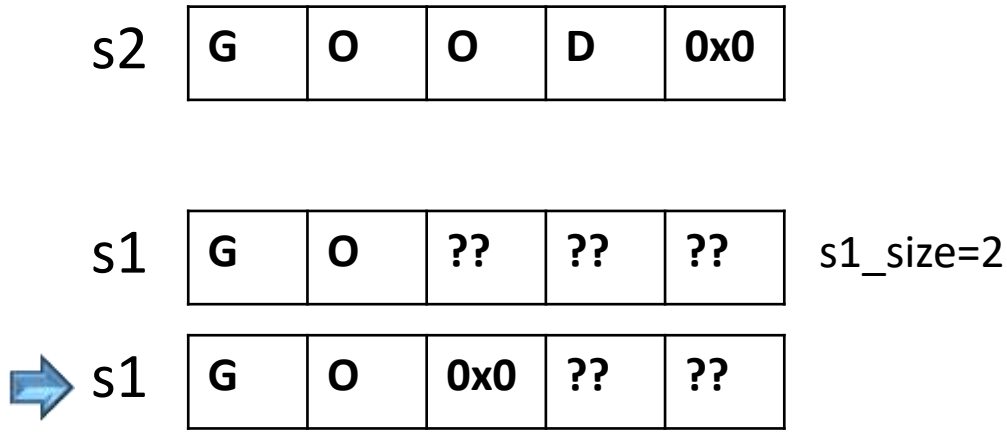
assignString(myString, size, stringLiteral);

```
void assignString( char *s1, int s1_size, const char *s2 ) {
        int i = 0;
        while ( i < s1_size ) {
                if ( s2[i] == 0 ) {break; }
                s[i] = s2[i];
                ++i;
        }
    ➡ s1[i] = 0x0;          // the null terminator for a string
}
```



assignString(myString, size, stringLiteral);

```
void assignString( char *s1, int s1_size, const char *s2 ) {
        int i = 0;
        while ( i < s1_size ) {
                if ( s2[i] == 0 ) {break; }
                s[i] = s2[i];
                ++i;
        }
    ➡ s1[i] = 0x0;          // the null terminator for a string
}
// s1: the memory space allocated for s1 must be large enough.
// Otherwise: missing data
```



Appending a String Examples

We can use several overloaded functions to add new contents to a string.

```
string s1("Good ");

s1.append(" Programming");
cout << s1 << endl;

string s2("Good");
s2.append(" to learn C++", 0, 5);
cout << s2 << endl;
```

Functions of string

assign	length	substr		
append	size	insert		
at	capacity	replace		
clear	c_str			
erase	data			
empty	compare			

Examples

```
string s("C++ is a high-level language.");
cout << s.length() << endl;
cout << s.size() << endl;
cout << s.capacity() << endl;

s.erase(1, 5);

cout << s.length() << endl;
cout << s.size() << endl;
cout << s.capacity() << endl;
cout << s.substr(3) << endl;          // substring
cout << s.substr(3, 5) << endl;
```

Examples

```
string s("C++ is a high-level language.");
cout << s.length() << endl;
cout << s.size() << endl;
cout << s.capacity() << endl;

s.erase(1, 5);

cout << s.length() << endl;
cout << s.size() << endl;
cout << s.capacity() << endl;
cout << s.substr(3) << endl;          // substring
cout << s.substr(3, 5) << endl;
```

size()
not include 0x0

Examples

```
string s("C++ is a high-level language. Good! Good!");
cout << s.find("od") << endl;
cout << s.find("hi", 9) << endl;
cout << s.find('o') << endl;

string s2 = s1;
s2.insert(2, 5, 'H’);
cout << s2 << endl;
s2.replace(6,9, "Here");
cout << s2 << endl;
```

Converting Numbers to Strings

itoa: convert an integer to a string.

Alternatively, we can do the following for conversion:

```
#include <sstream>


stringstream ss;
ss << 2.7182;
string s = ss.str(); // convert a number to a string
```

Reading Strings

```
string city;
cout << "Enter a city: ";
cin >> city; // Read to array city
cout << "You entered " << city << endl;
```

```
string city;
cout << "Enter a city: ";
getline(cin, city, '\n'); // Same as
getline(cin, city)
cout << "You entered " << city << endl;
```

Object-Oriented Programming Approach

- The users focus on the usage of the class.
  - Learn the purposes of the methods
  - Learn how to call the methods
  - Develop algorithms
- Need not to worry about the details of the data structures.

Exercise: Implement a simple class for string

- When we (as programmers/developers) develop our classes, we must handle the details. So that the clients can use our classes easily.

```
myString( )
myString( const char *s)
~ myString( )
int length( const char *s) const
int length( ) const

private:
char *ptr;        // store the characters of a string
int maxSize;   // the memory space allocated to ptr
int size;         // the current size
```

Passing Objects to Functions

We can pass objects by value or by reference.

```
void foo( Square c) { ……}
```

Pass-by-value:
The copy-constructor is invoked.

If the copy-constructor is undefined, we do a shallow-copy. That's, copy from the actual parameter to the formal parameter.

```
Square a;
foo( a );
```

```
void foo( Square &c) { ……}
```

Pass-by-reference:
The formal parameter is the alias of the actual parameter.

```
Square a;
foo( a );
```

Passing Objects to Functions

```
void foo(Circle c) {

//pass-by-value

}
```

```
void foo(Circle &c) {

//pass-by-reference

}
```

```
void foo(const Circle &c) {

//pass-by-reference
//c cannot be changed

}
```

Array of Objects and Pointers to Objects

```
Square squares[4] = {Square(1), Square(2), Square(), Square()};

Square *squares[4] = {
                        new Square(1),
                        new Square(2),
                        new Square(),
                        new Square()
                        };
```

Instance and static members

```
class Square{
    private:
       static int numberOfObjects;          // it is a class variable
       double side;                         //Data Fields
    public:
       Square( );                           //Constructor. Increase numOfObjects
       Square( double side);                //Constructor. Increase numOfObjects
       double getArea( ) const;             //Function
       static int getNumberOfObjects( ) const;                //
};
….
Square a(5.0), b(3.0), c;
```

Instance and static members

```
class Square{
    private:
       static int numberOfObjects;              // it is a class variable
       double side;                             //Data Fields
    public:
       Square( ) { ++ numberOfObjects; }    //Constructor. Increase numOfObjects
       Square( double side) {{ ++ numberOfObjects;  this->side = side; }       //Constructor. Increase numOfObjects
       double getArea( ) const;                 //Function
       static int getNumberOfObjects( ) const;                  //
};


Square a(5.0), b(3.0), c;
```

Instance and static members

```
class Square{
    private:
       static int numberOfObjects;            // it is a class variable
       double side;                           //Data Fields
    public:
       Square( ) { ++ numberOfObjects; }   //Constructor. Increase numOfObjects

       double getArea( ) const;               //Function
       static int getNumberOfObjects( ) const;                //
};

Square a(5.0), b(3.0), c;
```

Instance and static members

```
class Square{
    private:
       static int numberOfObjects;              // it is a class variable
       double side;                             //Data Fields
    public:
       Square( ) { ++ numberOfObjects; }    //Constructor. Increase numOfObjects
       Square( double side) {{ ++ numberOfObjects;  this->side = side; }       //Constructor. Increase numOfObjects
       double getArea( ) const;                 //Function
       static int getNumberOfObjects( ) const;                  //
};


Square a(5.0), b(3.0), c;
```

Instance and static members

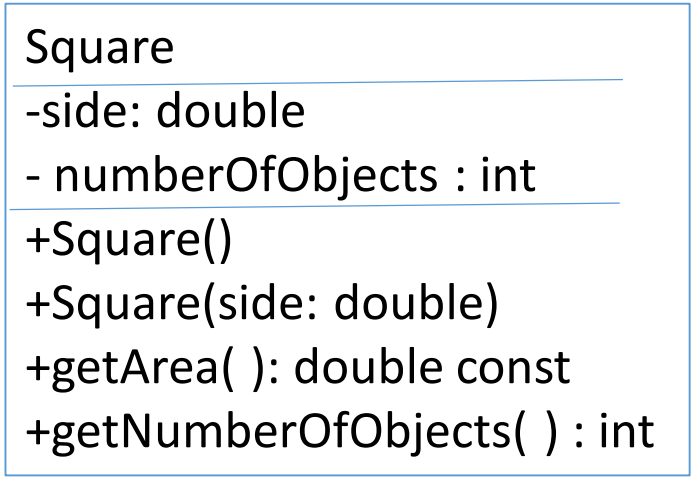
```
class Square{
    private:
       static int numberOfObjects;            // it is a class variable
       double side;                           //Data Fields
    public:
       Square( ) { ++ numberOfObjects; }    //Constructor. Increase numOfObjects
       Square( double side) {{ ++ numberOfObjects;  this->side = side; }       //Constructor. Increase numOfObjects
       double getArea( ) const;               //Function
       static int getNumberOfObjects( ) const;                  //
};
int Square::numOfObjects = 0; // initialize the static variable
Square a(5.0), b(3.0), c;
```

Instance and static members

```
class Square{
    private:
        static int numberOfObjects;
        double side;
    public:
        Square( );
        Square( double side);
        double getArea( ) const;
        static int getNumberOfObjects( );
};
```

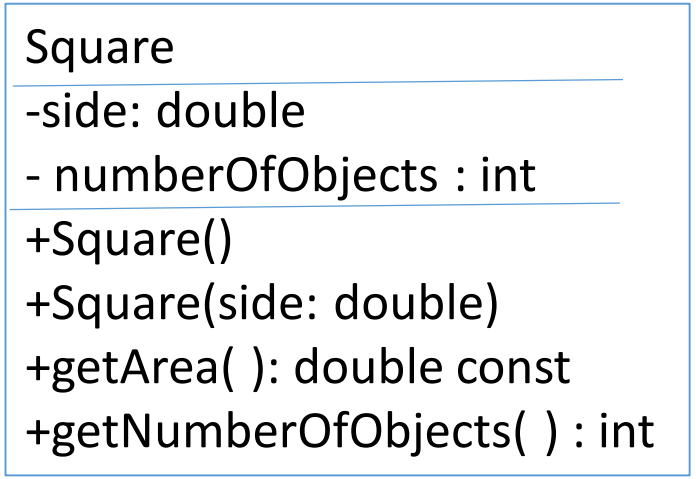
Instance and static members

```
class Square{
    private:
        static int numberOfObjects;
        double side;
    public:
        Square( );
        Square( double side);
        double getArea( ) const;
        static int getNumberOfObjects( );
};
```



Instance and static members

```
class Square{
    private:
        static int numberOfObjects;
        double side;
    public:
        Square( );
        Square( double side);
        double getArea( ) const;
        static int getNumberOfObjects( );
};
```



Square a(5.4), b(6.2)

Instance and static members

```
class Square{
    private:
        static int numberOfObjects;
        double side;
    public:
        Square( );
        Square( double side);
        double getArea( ) const;
        static int getNumberOfObjects( );
};
```

Square
-side: double
- numberOfObjects : int
+Square()
+Square(side: double)
+getArea( ): double const
+getNumberOfObjects( ) : int

Instantiate an object a

a: Square
side = 5.4
numberOfObjects = 1

Square a(5.4), b(6.2)

Instance and static members

```
class Square{
    private:
        static int numberOfObjects;
        double side;
    public:
        Square( );
        Square( double side);
        double getArea( ) const;
        static int getNumberOfObjects( );
};
```

Square
-side: double
- numberOfObjects : int
+Square()
+Square(side: double)
+getArea( ): double const
+getNumberOfObjects( ) : int

Instantiate an object a

a: Square
side = 5.4
numberOfObjects = 1

b: Square
side = 6.2
numberOfObjects = 2

Instantiate an object b

Square a(5.4), b(6.2)

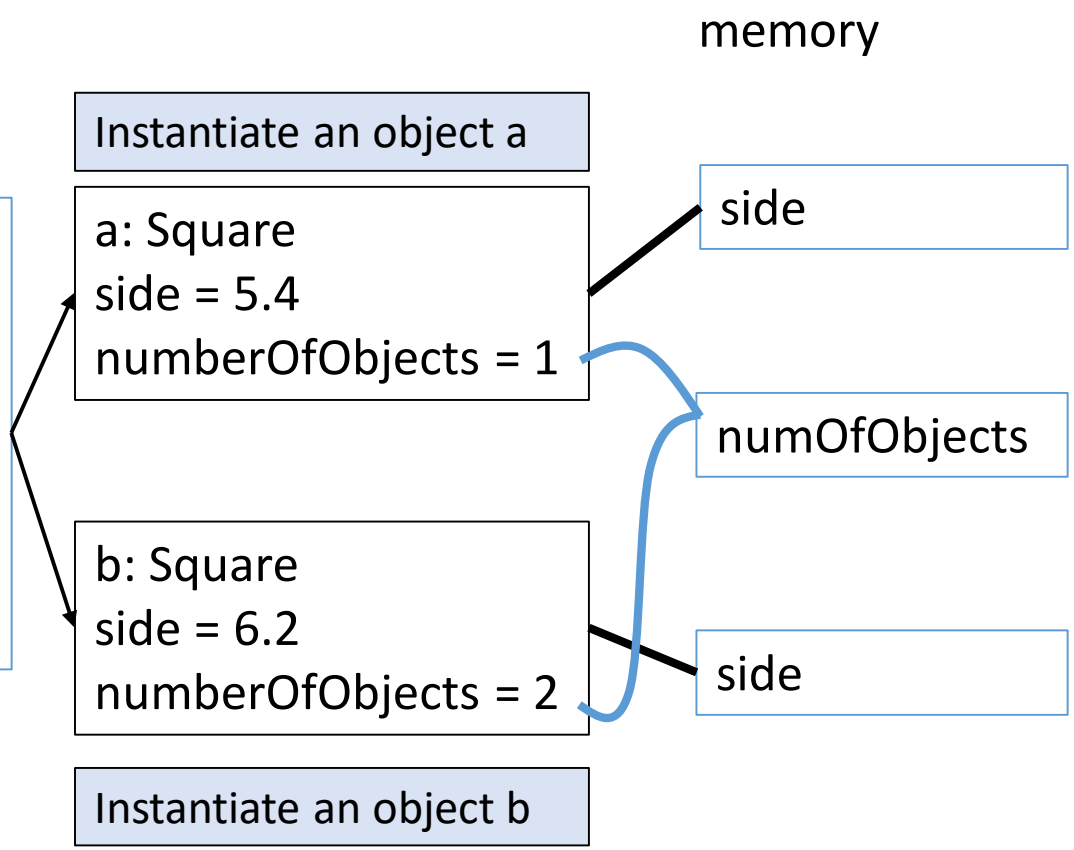
Instance and static members

```
class Square{
   private:
      static int numberOfObjects;
      double side;
   public:
      Square( );
      Square( double side);
      double getArea( ) const;
      static int getNumberOfObjects( );
};
```

static member

instance member

Square
-side: double - numberOfObjects : int
+Square() +Square(side: double) +getArea( ): double const +getNumberOfObjects( ) : int



Square a(5.4), b(6.2)

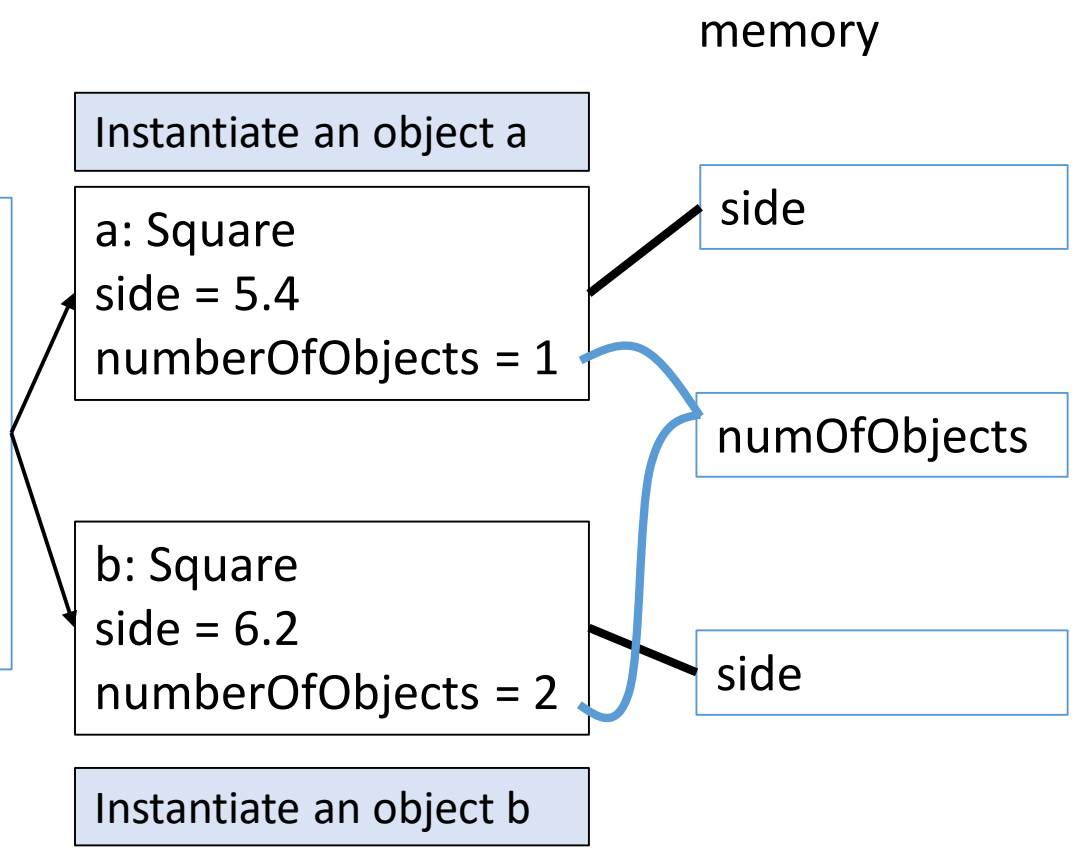
Instance and static members

```
class Square{
   private:
      static int numberOfObjects;
      double side;
   public:
      Square( );
      Square( double side);
      double getArea( ) const;
      static int getNumberOfObjects( );
};
```

static member

instance member

Square
-side: double - numberOfObjects : int
+Square() +Square(side: double) +getArea( ): double const +getNumberOfObjects( ) : int



Square a(5.4), b(6.2)

Invoke static functions

```
class Square{
    private:
        static int numberOfObjects;
        double side;
    public:
        Square( );
        Square( double side);
        double getArea( ) const;
        static int getNumberOfObjects( );
};
```

Invoke static functions

```
class Square{
    private:
       static int numberOfObjects;
       double side;
    public:
       Square( );
       Square( double side);
       double getArea( ) const;
       static int getNumberOfObjects( );
};
```

Use

ClassName::functionName (arguments) to invoke a static function

Invoke static functions

```
class Square{
    private:
        static int numberOfObjects;
        double side;
    public:
        Square( );
        Square( double side);
        double getArea( ) const;
        static int getNumberOfObjects( );
};
```

Use

ClassName::functionName (arguments) to invoke a static function

Square::getNumberOfObjects( )

To access a static variable, use

Invoke static functions

```
class Square{
    private:
       static int numberOfObjects;
       double side;
    public:
       Square( );
       Square( double side);
       double getArea( ) const;
       static int getNumberOfObjects( );
};
```

Use

ClassName::functionName (arguments) to invoke a static function

Square::getNumberOfObjects( )

To access a static variable, use

ClassName::staticVariable

This **improves readability** because the user can easily recognize the static function and data in the class.

Constant Member Functions

```
return_type A::func( ) const

void A::func( ) const
```

Constant Member Functions

constant member function

```
return_type A::func( ) const

void A::func( ) const
```

Constant Member Functions

const keyword: we use it to specify a constant parameter. So that the compiler knows that the parameter must not be changed.

constant member function

```
return_type A::func( ) const

void A::func( ) const
```

We can also specify **a constant member function** to tell the compiler that the function **should not change** the value of **any data fields** in the object.

Constant Member Functions

const keyword: we use it to specify a constant parameter. So that the compiler knows that the parameter must not be changed.

constant member function

```
return_type A::func( ) const

void A::func( ) const
```

We can also specify **a constant member function** to tell the compiler that the function **should not change** the value of **any data fields** in the object.

Constant Member Functions

const keyword: we use it to specify a constant parameter. So that the compiler knows that the parameter must not be changed.

We can also specify **a constant member function** to tell the compiler that the function **should not change** the value of **any data fields** in the object.

constant member function

```
return_type A::func( ) const

void A::func( ) const
```

```
class Square{
    private:
        double area, side;
    public:
        …
        double getArea( ) const {
            side = 10;
            return area;
        }
};
```

Constant Member Functions

const keyword: we use it to specify a constant parameter. So that the compiler knows that the parameter must not be changed.

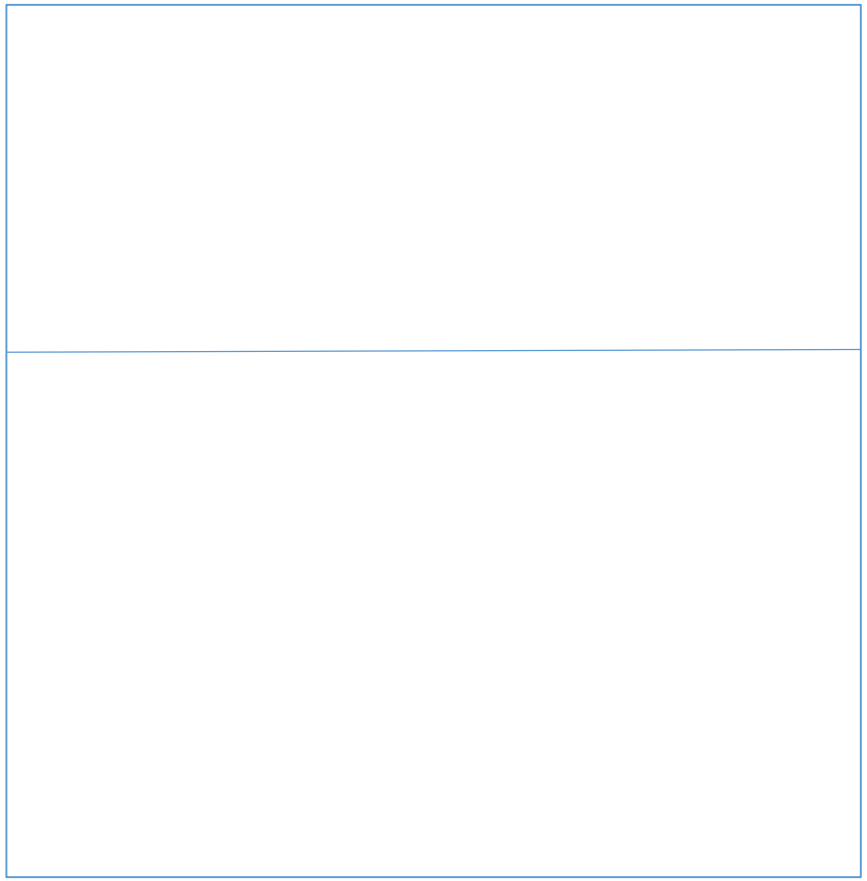
We can also specify **a constant member function** to tell the compiler that the function **should not change** the value of **any data fields** in the object.

constant member function

```
return_type A::func( ) const

void A::func( ) const
```

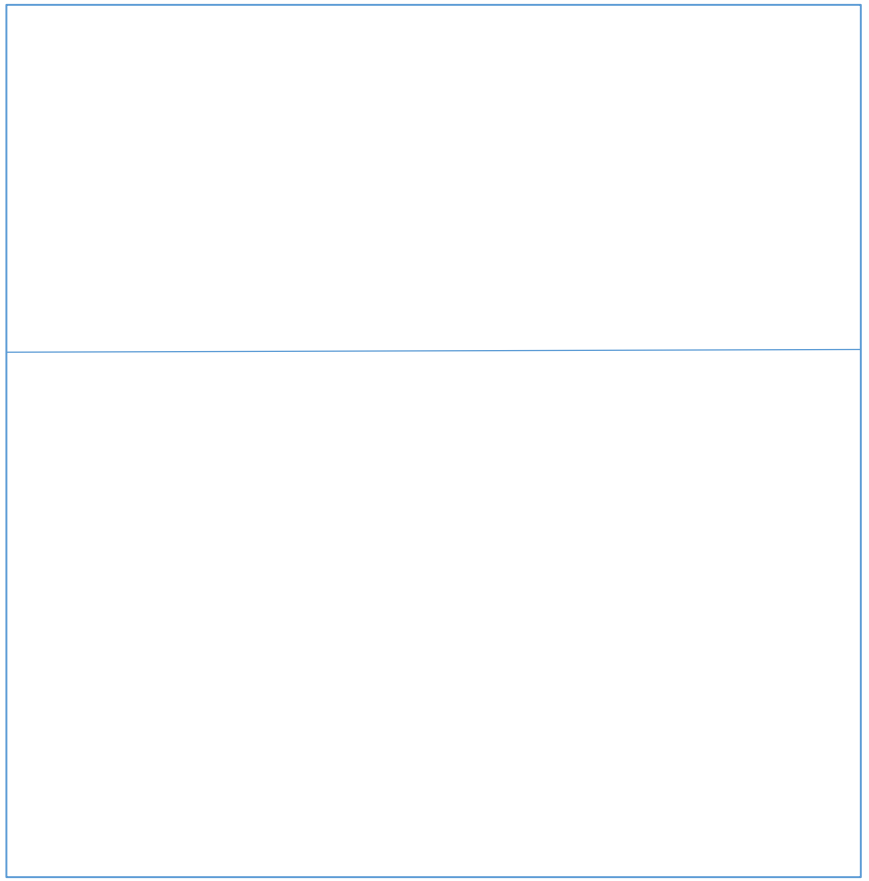
```
class Square{
    private:
        double area, side;
    public:
        …
        double getArea( ) const {
            side = 10; // not allowed
            return area;
        }
};
```

Object-Oriented Thinking



Object-Oriented Thinking

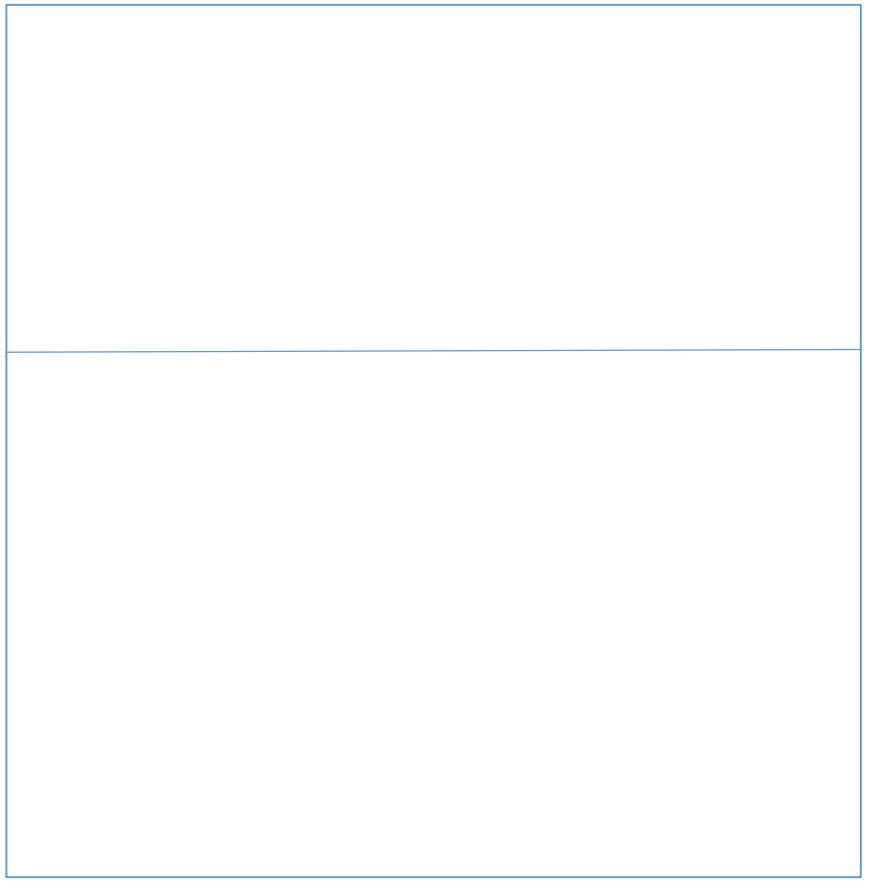
Square class



Object-Oriented Thinking

Square
class

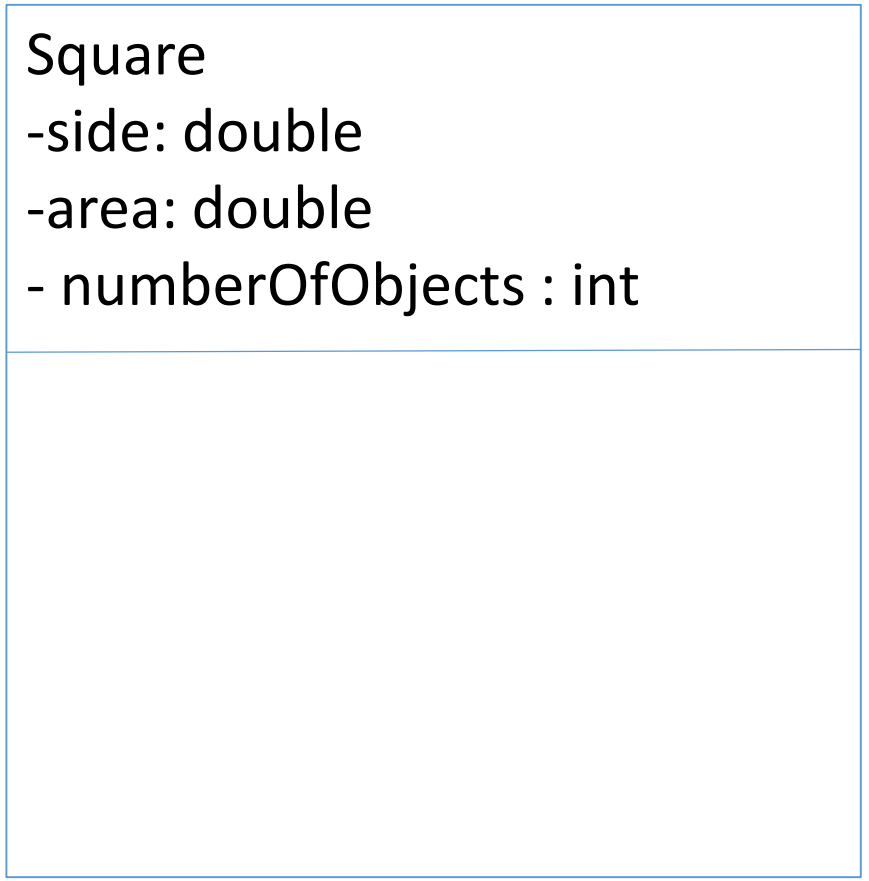
Attributes?



Object-Oriented Thinking

Square class

Attributes?

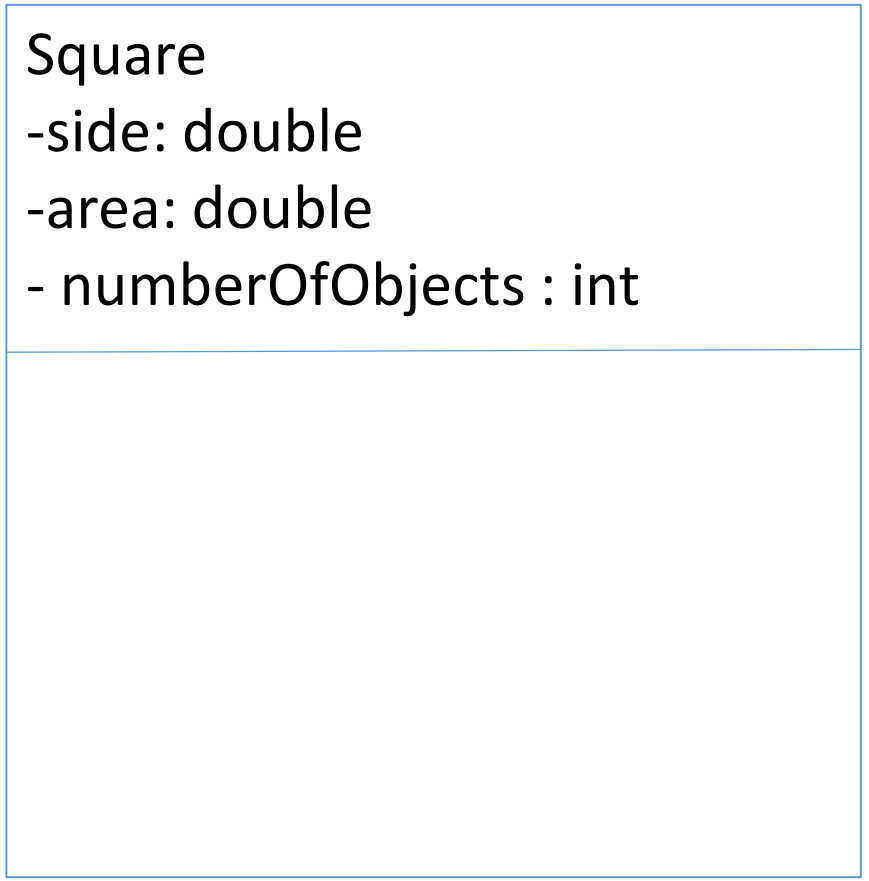


Object-Oriented Thinking

Square class

Attributes?

Actions?
Behavior?



Object-Oriented Thinking

Square class

Attributes?

Actions?
Behavior?

Square
-side: double -area: double - numberOfObjects : int
+Square() +Square(side: double) +Square(const Square &) +operator=(const Square &); +getArea( ) : double const +getNumberOfObjects( ) : int

Object-Oriented Thinking

- What kind of object do we create?
- What are the attributes of the object?
- What actions can we perform on the object?

Square class

Attributes?

Actions?
Behavior?

Square
-side: double -area: double - numberOfObjects : int
+Square() +Square(side: double) +Square(const Square &) +operator=(const Square &); +getArea( ) : double const +getNumberOfObjects( ) : int

Object-Oriented Thinking

- What A1 object do we create?
- What are the A2 of the object?
- What A3 can we perform on the object?

Square class

Attributes?

Actions?
Behavior?

Square
-side: double -area: double - numberOfObjects : int
+Square() +Square(side: double) +Square(const Square &) +operator=(const Square &); +getArea( ) : double const +getNumberOfObjects( ) : int

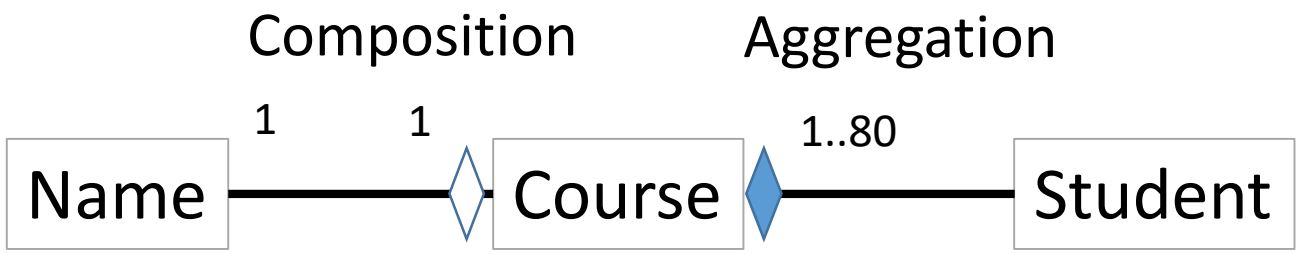
Object Composition

Aggregation models *has-a* relationships and represents an ownership relationship between two objects.

The owner object is called an *aggregating object* and its class an *aggregating class*.

The subject object is called an *aggregated object* and its class an *aggregated class*.

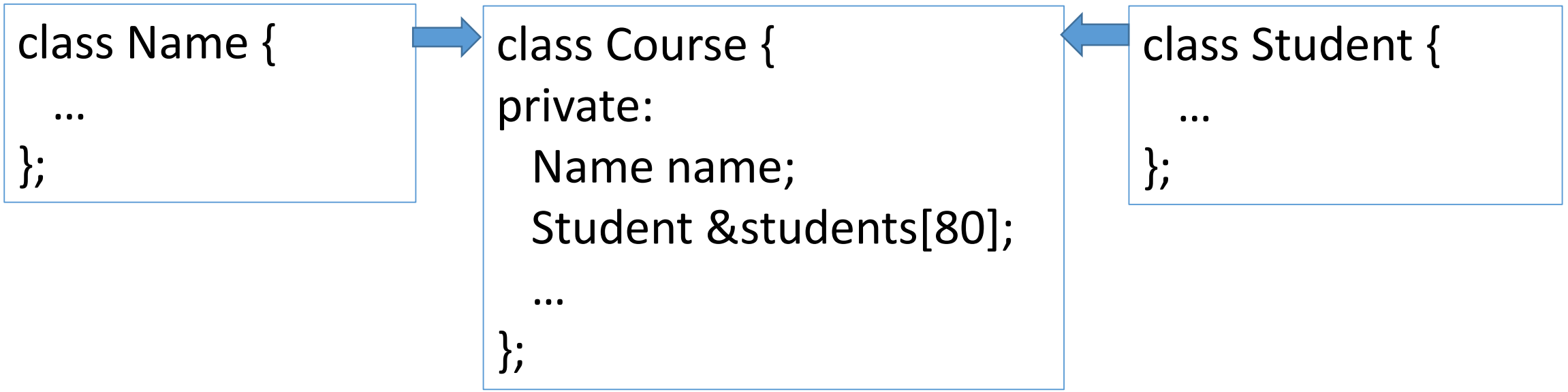
Composition: an object can contain another object (not shared). Composition is actually a special case of the aggregation relationship.



"Has-a" student

Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.



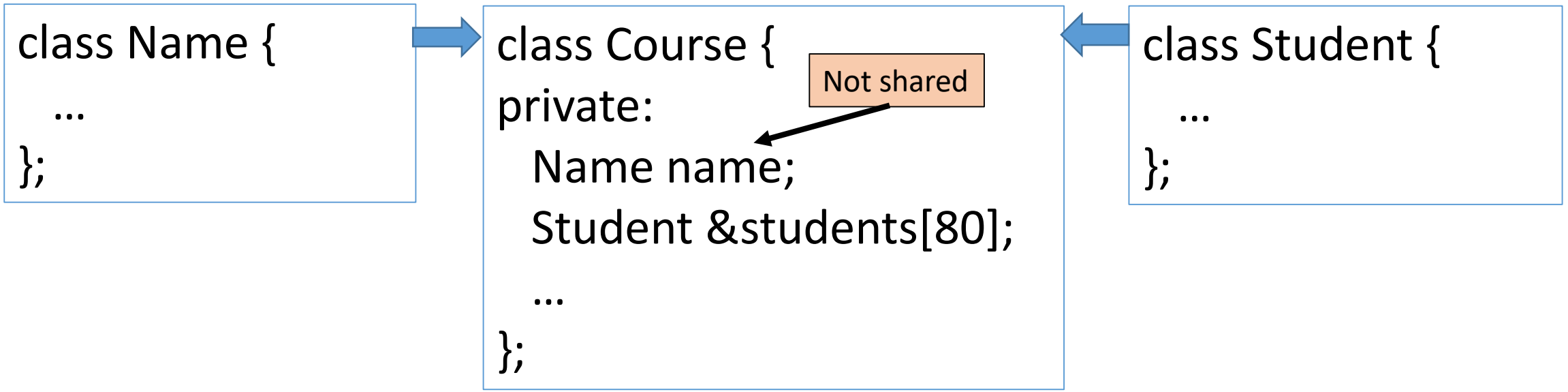
Aggregated class

Aggregating class

Aggregated class

Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.



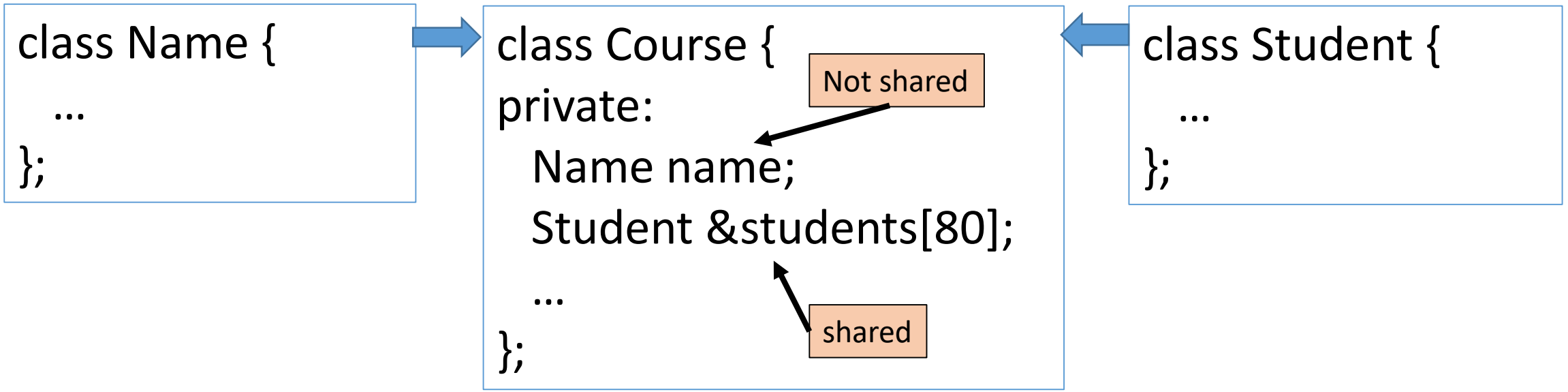
Aggregated class

Aggregating class

Aggregated class

Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.



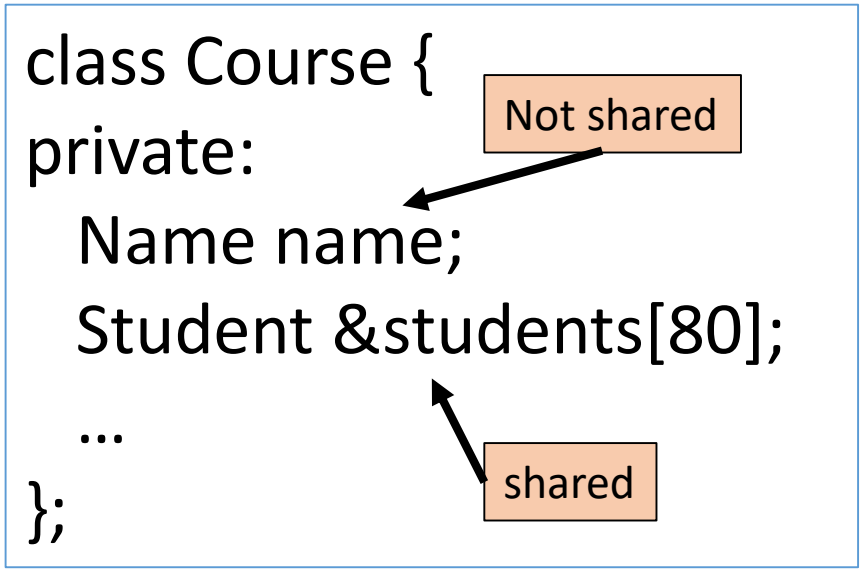
Aggregated class

Aggregating class

Aggregated class

Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.



Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.

```
class Course {
private:
   Name name;
   Student &students[80];
   …
};
```

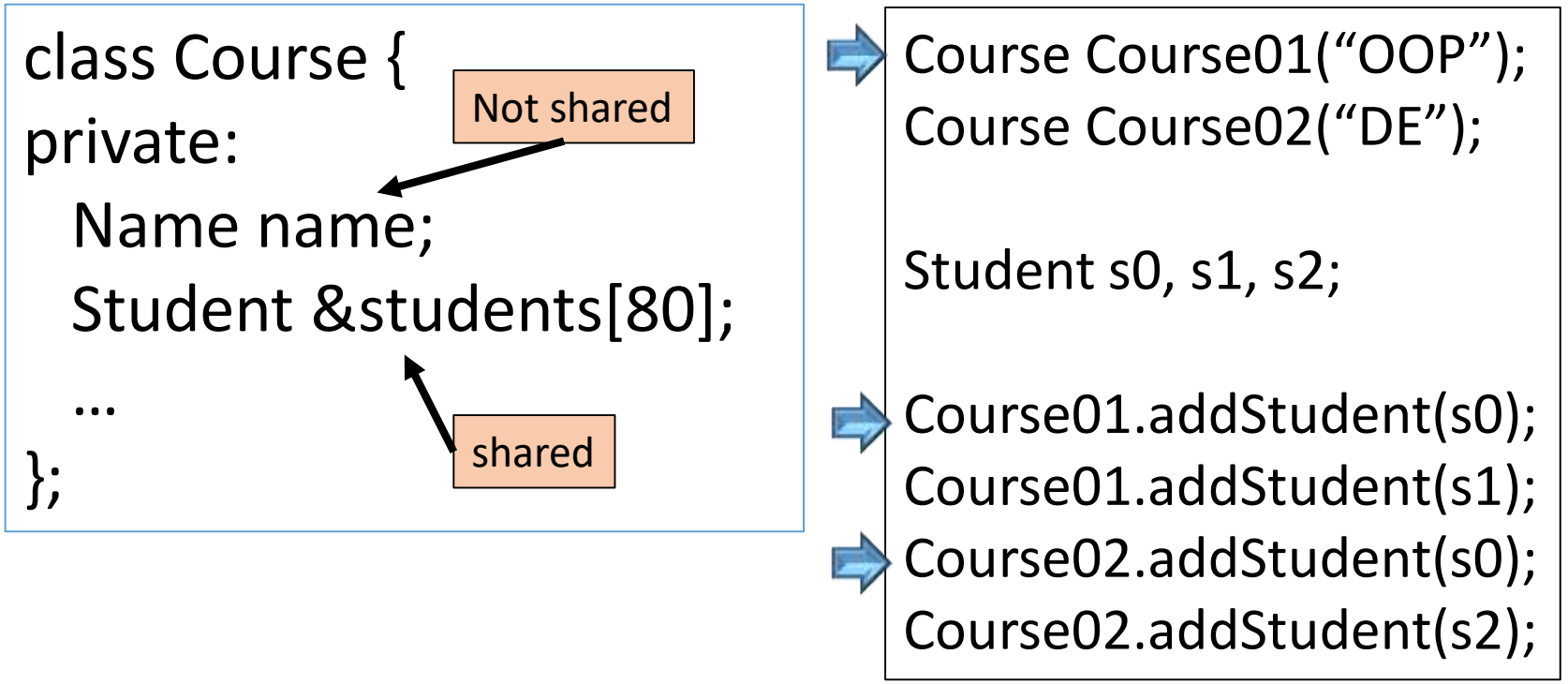
Not shared

shared

```
Course Course01("OOP");
Course Course02("DE");

Student s0, s1, s2;

Course01.addStudent(s0);
Course01.addStudent(s1);
Course02.addStudent(s0);
Course02.addStudent(s2);
```

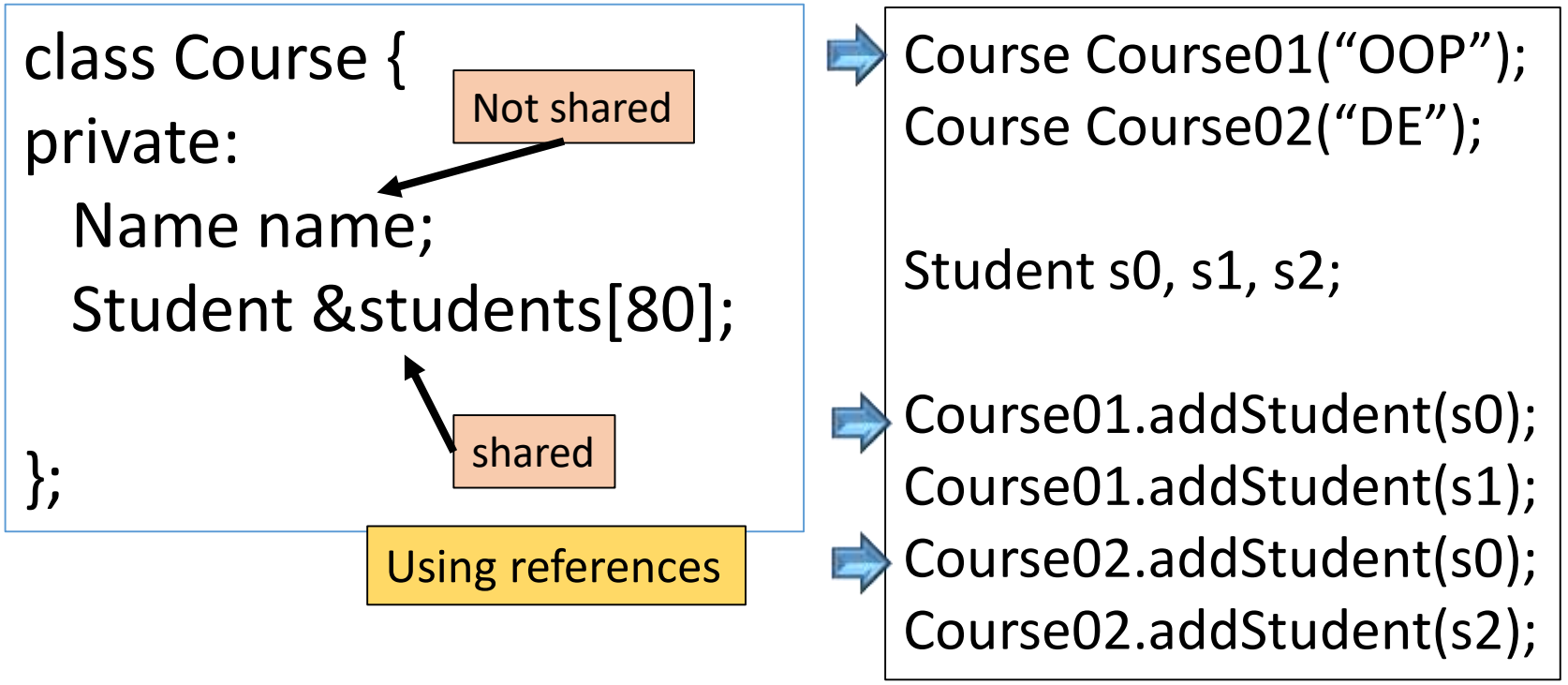
Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.



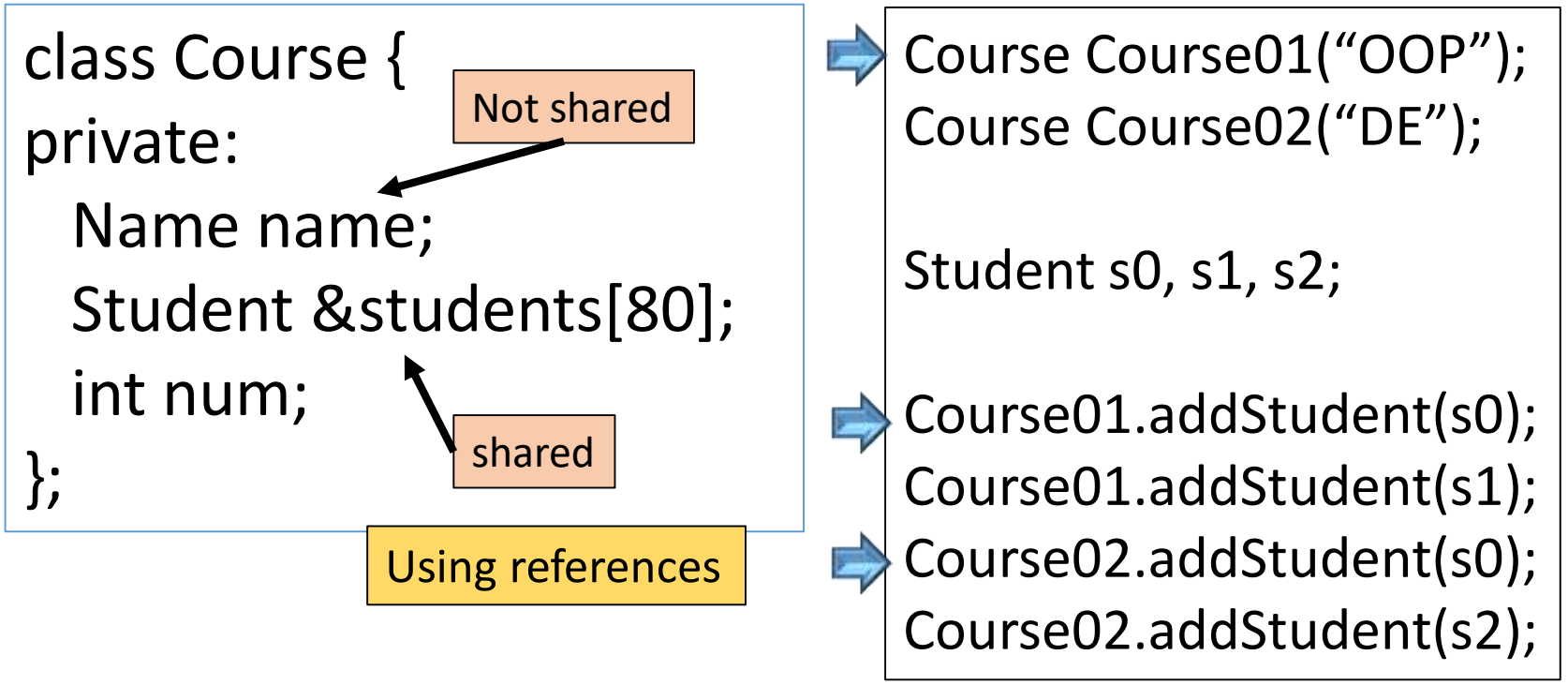
Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.



Class Representation

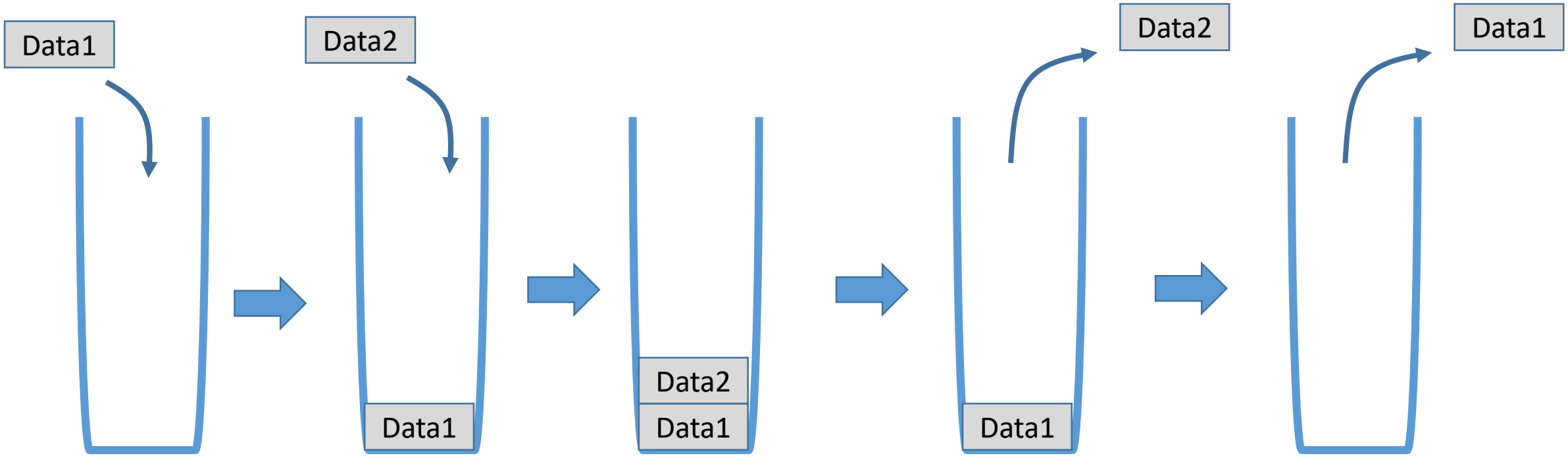
An aggregation relationship is usually represented as a data field in the aggregating class.



Example: Stack

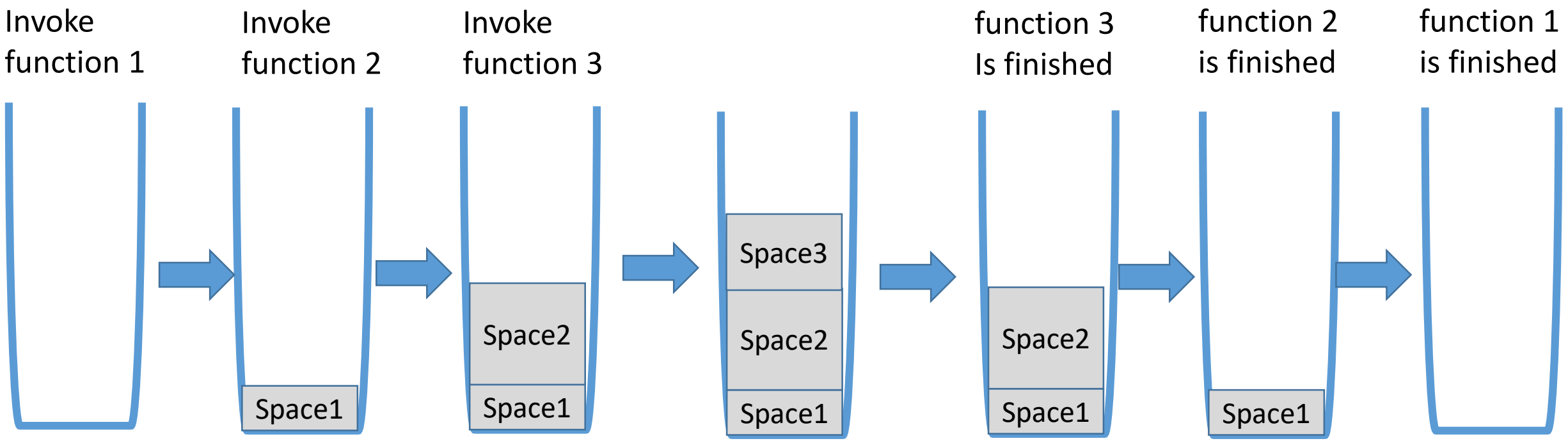
A stack is a data structure.

The last object is selected first, i.e., last-in first-out (LIFO).



Stack

- The compiler uses a stack to process **function invocations**.
- A call stack is maintained for local variables and parameters of functions.



Memory space for stack

Implementation of Stacks

IsEmpty	getSize		
push	peek		
pop			

Implementation of Stacks

```
class MyStack {
protected:
   int size;
   int capacity;
   int *arr;
   …
public:
   MyStack( );
   MyStack( int capacity);
   int pop( );
   void push(int);
   int getSize( ) const;
   bool isEmpty( ) const;
   int peek() const;
   …
};
```

IsEmpty	getSize		
push	peek		
pop			

Class Design: Cohesion

- For example, design a class that can compute the areas of squares and areas of circles.

```
class CircleSquare {
   double side;
   double radius;
   double area;
   int type;
   void computeArea_Circle( );
   void computeArea_Square( );
   void computeArea( ) {
      if (type == …) …
   }
};
```

Weak cohesion

```
class Circle {
   double radius;
   double area;
   void computeArea( );
};
```

```
class Square {
   double side;
   double area;
   void computeArea( );
};
```

Class Design: Cohesion

- A class should describe a single entity or a set of similar operations. An entity performs tasks that are related to each other.
- We should decompose a single entity into several classes if the entity has too many responsibilities.

```
class CircleSquare {
   double side;
   double radius;
   double area;
   int type;
   void computeArea_Circle( );
   void computeArea_Square( );
   void computeArea( ) {
      if (type == …) …
   }
};
```

Weak cohesion

```
class Circle {
   double radius;
   double area;
   void computeArea( );
};
```

```
class Square {
   double side;
   double area;
   void computeArea( );
};
```

Class Design: Consistency

- Follow **standard programming style** and **naming conventions**.
- Choose informative names for classes, data fields, and functions.
- A popular style in C++ is to **place the data declaration** after **the functions**, and place **constructors before functions**.

```
class MyStack {
public:
  MyStack( );
  MyStack( int capacity);
  int pop( );
  void push(int);
  int getSize( ) const;
  bool isEmpty( ) const;
  int peek() const;
  …
protected:
  int size;
  int capacity;
  int *arr;
  …
};
```

Class Design: Clarity

- Users can incorporate classes **in many different combinations, orders, and environments**.
- We should design a class that **imposes no restrictions** on what or when the user can do with it.
- Design the properties so that the user can set them **in any order** and with **any combination of values**, and design functions **independently of their order of occurrence**.

```
class MyStack {
public:
  MyStack( );
  MyStack( int capacity);
  int pop( );
  void push(int);
  int getSize( ) const;
  bool isEmpty( ) const;
  int peek() const;
  …
protected:
  int size;
  int capacity;
  int *arr;
  …
};
```

Class Design: Completeness

- Classes are designed for use by many different clients.
- To **be useful** in a **wide range of applications**, a class should provide **a variety of ways** for customization through properties and functions.
- For example, the string class contains **more than 20** functions.

```
class MyStack {
public:
  MyStack( );
  MyStack( int capacity);
  int pop( );
  void push(int);
  int getSize( ) const;
  bool isEmpty( ) const;
  int peek() const;
  …
protected:
  int size;
  int capacity;
  int *arr;
  …
};
```

Intended Learning Outcomes

- List the key ideas on the object-oriented programming approach
- Distinguish between class variables and instance variables

Supplemental Materials

Operators of string

[ ]	==	>		
=	!=			
+=	<			
<<	<=			
>>	>=			

Instance or Static?

How do we decide whether a variable or function should be instance or static?

An instance variable or function: The variable or function that is dependent on a specific instance of the class.

A static variable or function: The variable or function that is not dependent on a specific instance of the class.

Instance or Static?

Every square has its own ***side***. **The side length** is **dependent on** a **specific square**. Therefore, ***side*** is an **instance** variable of the Square class.

Because the **getArea** function is **dependent on** a **specific square**, it is an **instance** function.

Since **numberOfObjects** is **not dependent on** any specific instance, it should be declared **static**.

Class Design: Instance vs. Static

- **A variable or function** that is **dependent on** a **specific instance** of the class should be an **instance** variable or function.

- Static variables: **shared by** all the instances of a class.