Hochiminh City University of Technology
Computer Science and Engineering
OOP in C++

Basic OOP

Lecturer: Vu Van Tien

Today's outline

- Concept and Definition of Class
- Access Specifier
- Constructor / Destructor
- Operator Overloading
- Friendship
- Nested Class
- Shallow Copy & Deep Copy
- Assignment Operator
- Copy Constructor

Class

Structure versus Object-Oriented Programming

- **Structure programming** focuses on the process/actions that occur in a program. The program starts at the beginning, does something, and ends.

- **Object-Oriented programming** is based on the data and the functions that operate on it. Objects are instances of abstract data types that represent the data and its functions

Limitations of Structure 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

Class

Class

- Class: a user defined datatype which groups together related pieces of information
  - Data
  - Functions (Methods)
- **Classes** are similar to **Structure** but contain functions, as well.

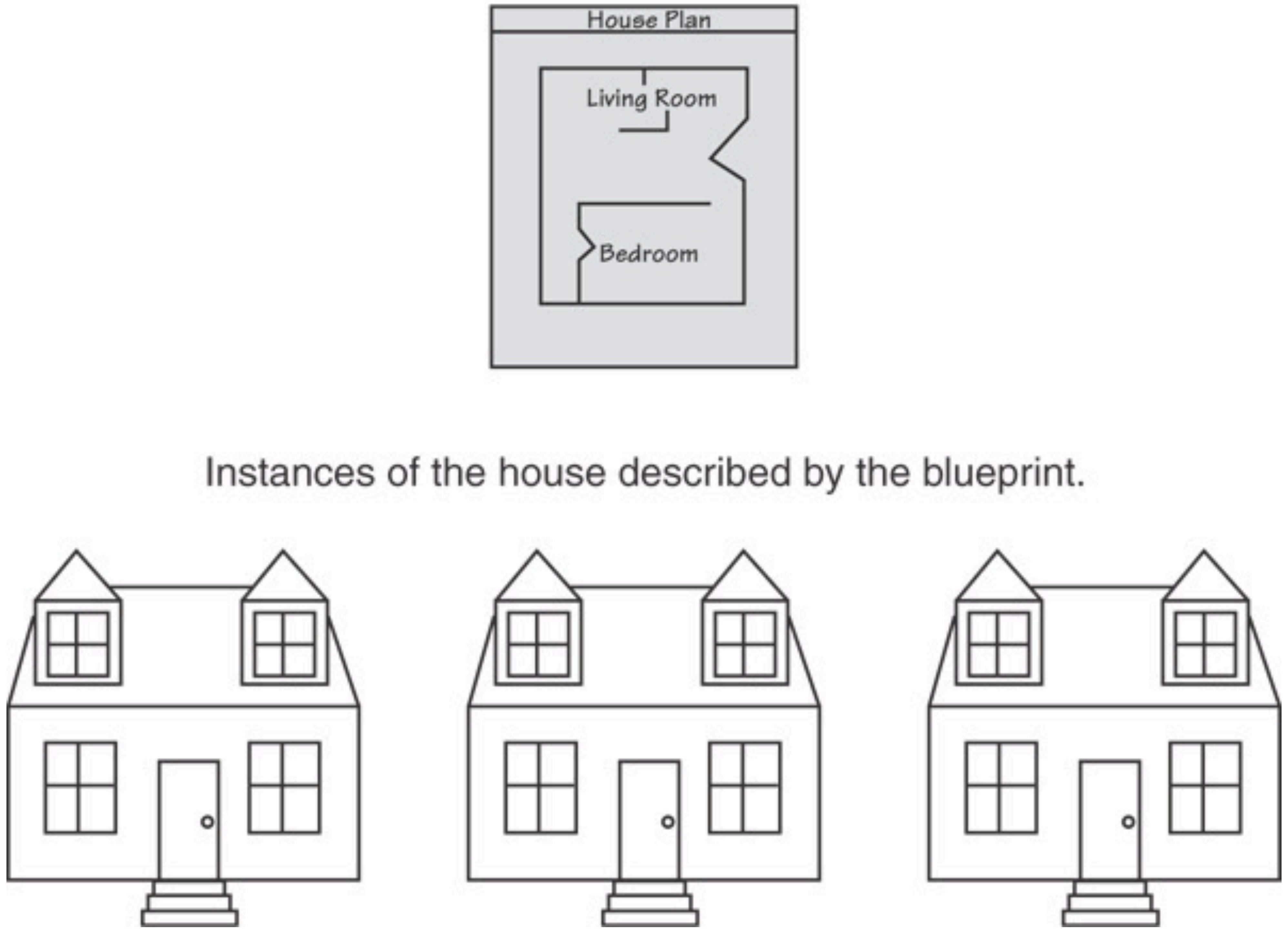
Terminologies

- Object is an instant of a particular class
- Data are known as fields, members, attributes, or properties
- Functions are known as methods

Classes and Objects

- A Class is like a blueprint and objects are like houses built from the blueprint

Blueprint that describes a house.



Instances of the house described by the blueprint.

Class Declaration

```
class <Class_Name>
{
<access_specifier>:
   member declaration;
   ...
<access_specifier>:
   member declaration;
   ...
};
```

Class Example

```
class Rectangle
{
private:
   double width;
   double height;
public:
   void setWidth(double);
   void setHeight(double);
   double getWidth();
   double getHeight();
   double getArea();
};
```

Class Access specifier

- Used to control access to members of the class:
  - private (**default**) : the members declared as private are only accessible from within the class. No outside Access is allowed.
  - public: the members declared as public are accessible from outside the Class through an object of the class.
- Can be listed in any order in a class
- Can appear multiple times in a class

Member Function Definition

- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator (::)

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

Set and Get

- Set (mutator): a member function that stores a value in a private member variable, or changes its value in some way.

```
void setWidth(double);
void setHeight(double);
```

- Get (accessor): a member function that retrieves a value from a private member variable.

```
double getWidth();
double getHeight();
```

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.
- Example

```
double getWidth() const;
double getHeight() const;
double getArea() const;
```

Scope operator

- Scope operator `::`
    - Is used in the definition of member function outside the class
    - Inline function vs. normal function
        - Member functions defined in the class definition is considered as inline function.

Constructor vs Destructor

Constructor

- Constructors: a special function that is automatically called whenever a new object is created .
  - allow the class to initialize member variables or allocate storage.
  - do not return a value, including void.
  - can not be called explicitly as member functions.

Default Constructor

- A default constructor is a constructor that takes no arguments.
- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.
- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```

Constructor Syntax

```
class <Class_Name>
{
     ...
public:
     <Class_Name>();
     ...
};
```

Constructors with Parameters

- To create a constructor that takes arguments:
  - Indicate parameters in prototype:

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

  - Use parameters in the definition:

    ```
    Rectangle::Rectangle(double w, double h)
    {
       width = w;
       height = h;
    }
    ```

- You can pass arguments to the constructor when you create an object:

  ```
  Rectangle r2(6, 4);
  ```

More About Default Constructors

- If all of a constructor's parameters have default arguments, then it is a default constructor. For example:

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

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```

Overloading Constructors

- A class can have more than one constructor. They can be overloaded.
- The compiler automatically call the one whose parameters match the arguments.

```
Rectangle();

Rectangle(double);

Rectangle(double, double);
```

Destructor

- Destructor: responsible for the necessary cleanup of a class when lifetime of an object ends.
- Destructors cannot:
  - return a value
  - accept parameters
- Destructors must have the same name as the class.
- Only one destructor per class, i.e., it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it

Destructor Syntax

```
class <Class_Name>
{
      ...
public:
      ~<Class_Name>();
      ...
};
```

Using Private Member Functions

- A `private` member function can only be called by another member function
- It is used for internal processing by the class, not for use outside of the class
- Example: If you wrote a class that had a public sort function and needed a function to swap two elements, you'd make that private

Arrays of Objects

- Objects can be the elements of an array:

```
Rectangle rooms[8];
```

- Default constructor for object is used when array is defined

Arrays of Objects

- Must use initializer list to invoke constructor that takes arguments:

```
Rectangle rectArray[3]={Rectangle(2.1,3.2),
                       Rectangle(4.1, 9.9),
                       Rectangle(11.2, 31.4)};
```

Accessing Objects in an Array

- Objects in an array are referenced using subscripts

- Member functions are referenced using dot notation:

```
rectArray[1].setWidth(11.3);
cout << rectrArray[1].getArea();
```

Pointer to Class

- Objects can also be pointed by pointers. Class is a valid type.
- Class pointers is similar to struct pointers.
- E.g.:

```
Rectangle r2(6, 4);
Rectangle* r3 = &r2;
cout << r3->getArea() << endl;
cout << (*r3).getArea() << endl;
```

Using the this Pointer

- Every object has access to its own address through a pointer called `this` (a C++ keyword)

```
void Rectangle::setWidth(double width)
{
   this->width = width;
}
```

Operator overloading

Fundamentals of Operator Overloading

- Overloading an operator
  - –Write function definition as normal
  - –Function name is keyword `operator` followed by the symbol for the operator being overloaded
  - –`operator+` used to overload the addition operator (`+`)
- Using operators
  - –To use an operator on a class object it must be overloaded unless the assignment operator `(=)` or the address operator `(&)`
    - Assignment operator by default performs memberwise assignment
    - Address operator (&) by default returns the address of an object

Restrictions on Operator Overloading

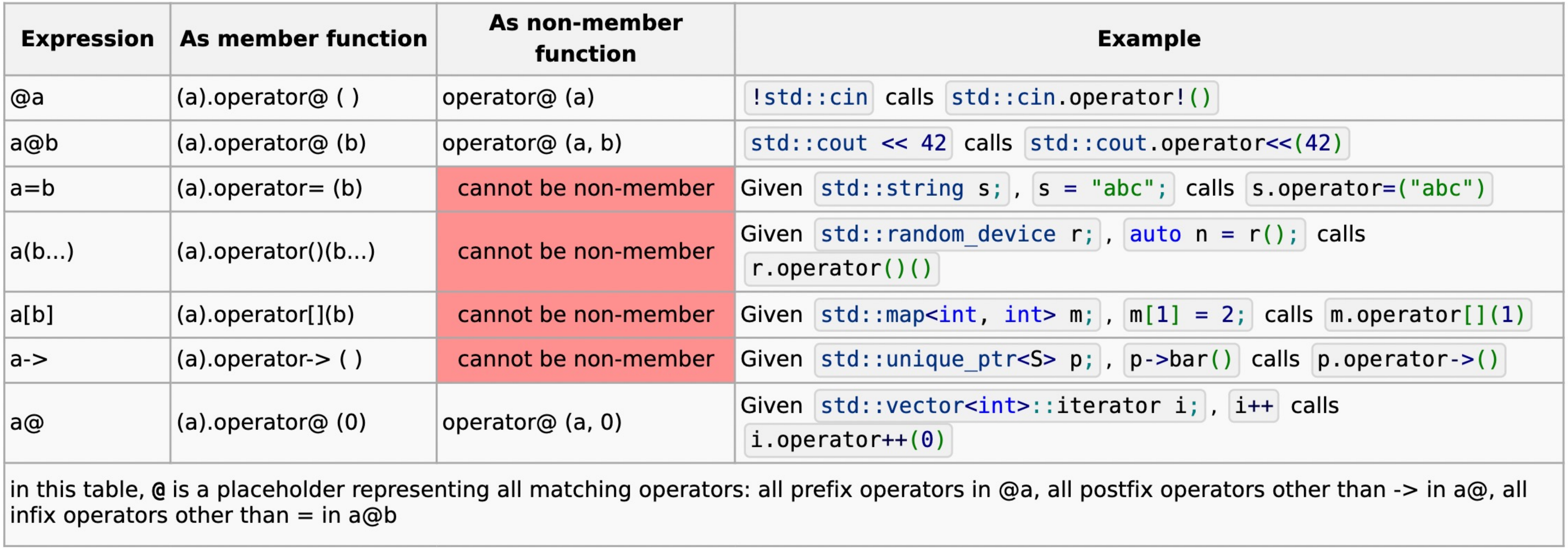
Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Restrictions on Operator Overloading

- Overloading restrictions
  - –Precedence of an operator cannot be changed
  - –Associativity of an operator cannot be changed
  - –Arity (number of operands) cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators `&`, `*`, `+` and `-` each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No new operators can be created
  - –Use only existing operators
- No overloading operators for built-in types
  - –Cannot change how two integers are added
  - –Produces a syntax error

Operator Overloading



Expression	As member function	As non-member function	Example
@a	(a).operator@ ( )	operator@ (a)	`!std::cin` calls `std::cin.operator!()`
a@b	(a).operator@ (b)	operator@ (a, b)	`std::cout << 42` calls `std::cout.operator<<(42)`
a=b	(a).operator= (b)	cannot be non-member	Given `std::string s;`, `s = "abc";` calls `s.operator=("abc")`
a(b...)	(a).operator()(b...)	cannot be non-member	Given `std::random_device r;`, `auto n = r();` calls `r.operator()()`
a[b]	(a).operator[](b)	cannot be non-member	Given `std::map<int, int> m;`, `m[1] = 2;` calls `m.operator[](1)`
a->	(a).operator-> ( )	cannot be non-member	Given `std::unique_ptr<S> p;`, `p->bar()` calls `p.operator->()`
a@	(a).operator@ (0)	operator@ (a, 0)	Given `std::vector<int>::iterator i;`, `i++` calls `i.operator++(0)`

in this table, `@` is a placeholder representing all matching operators: all prefix operators in @a, all postfix operators other than -> in a@, all infix operators other than = in a@b

*As non-member function: a friend function of class

Reference: operator overloading - cppreference.com

Friendship

Friendship

- Friends are functions or classes declared with the friend keyword.
- Using friend functions can enhance performance.

Friend function member

- A non-member function can access private and protected members of class if it is declared as a friend of class.
- E.g.:
```
class Student {
```
-
```
. . .
public:
    friend Student duplicateStudent(Student& a);
};
```

Friend class

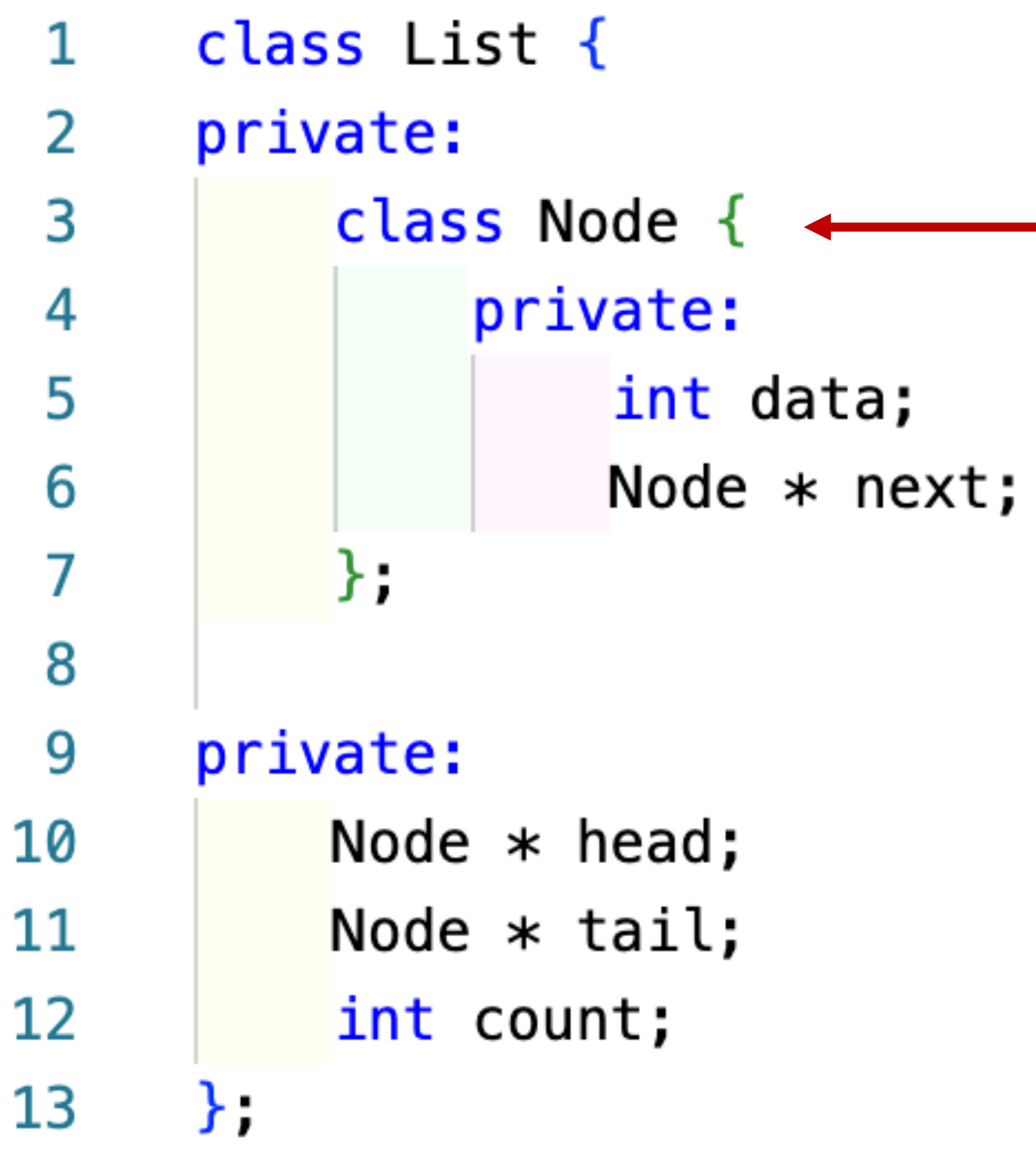
- Friend class: is a class whose members can access to private and protected members of other classes.

-
  ```
  class Lecturer;
  class Student {
      friend class Lecturer;// lecturer is a friend
      . . .
  };
  ```

Nested Class

Nested class

- Nested class: Class declared within another class



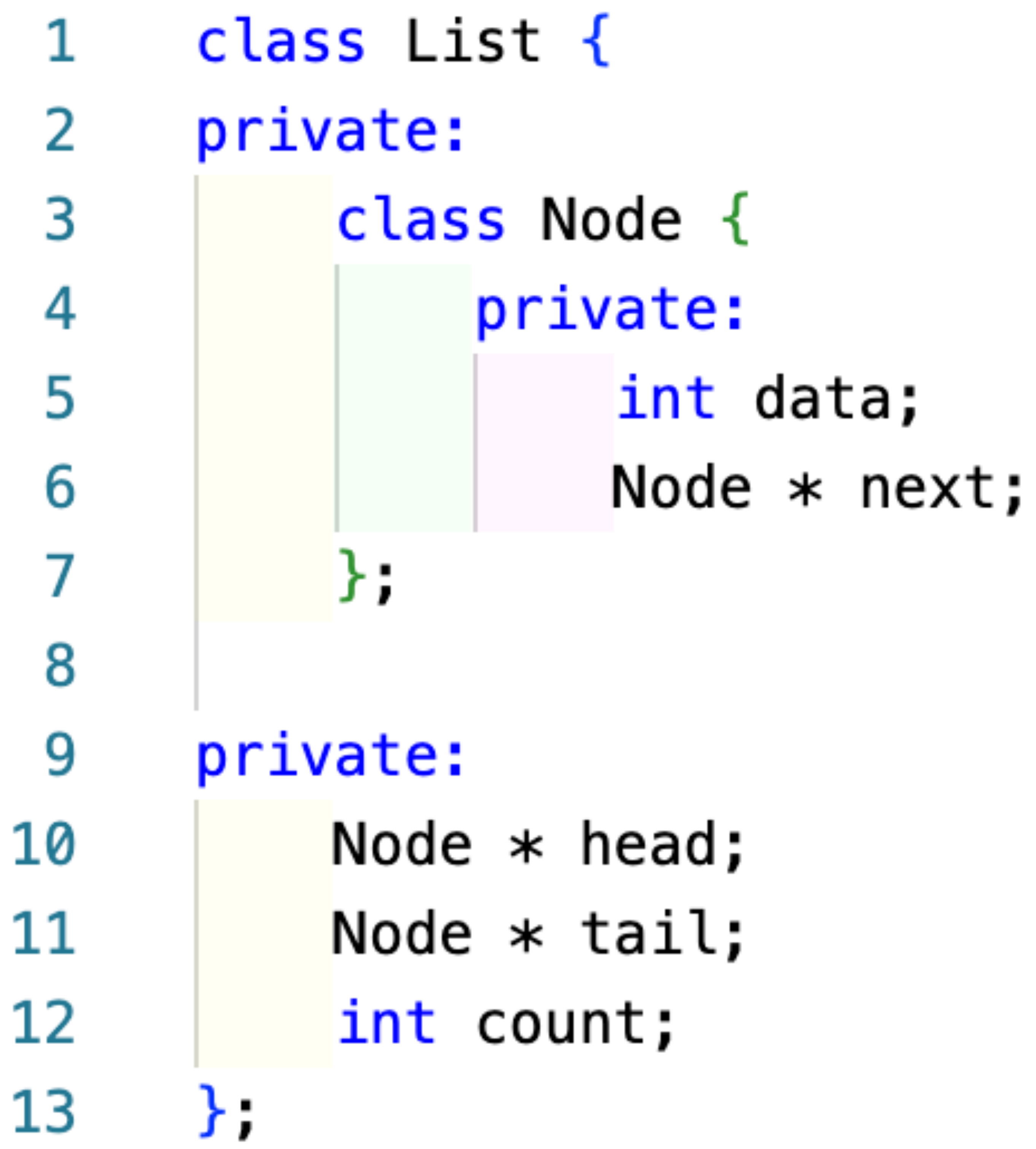
```
class List {
private:
    class Node {
        private:
            int data;
            Node * next;
    };

private:
    Node * head;
    Node * tail;
    int count;
};
```

Nested class

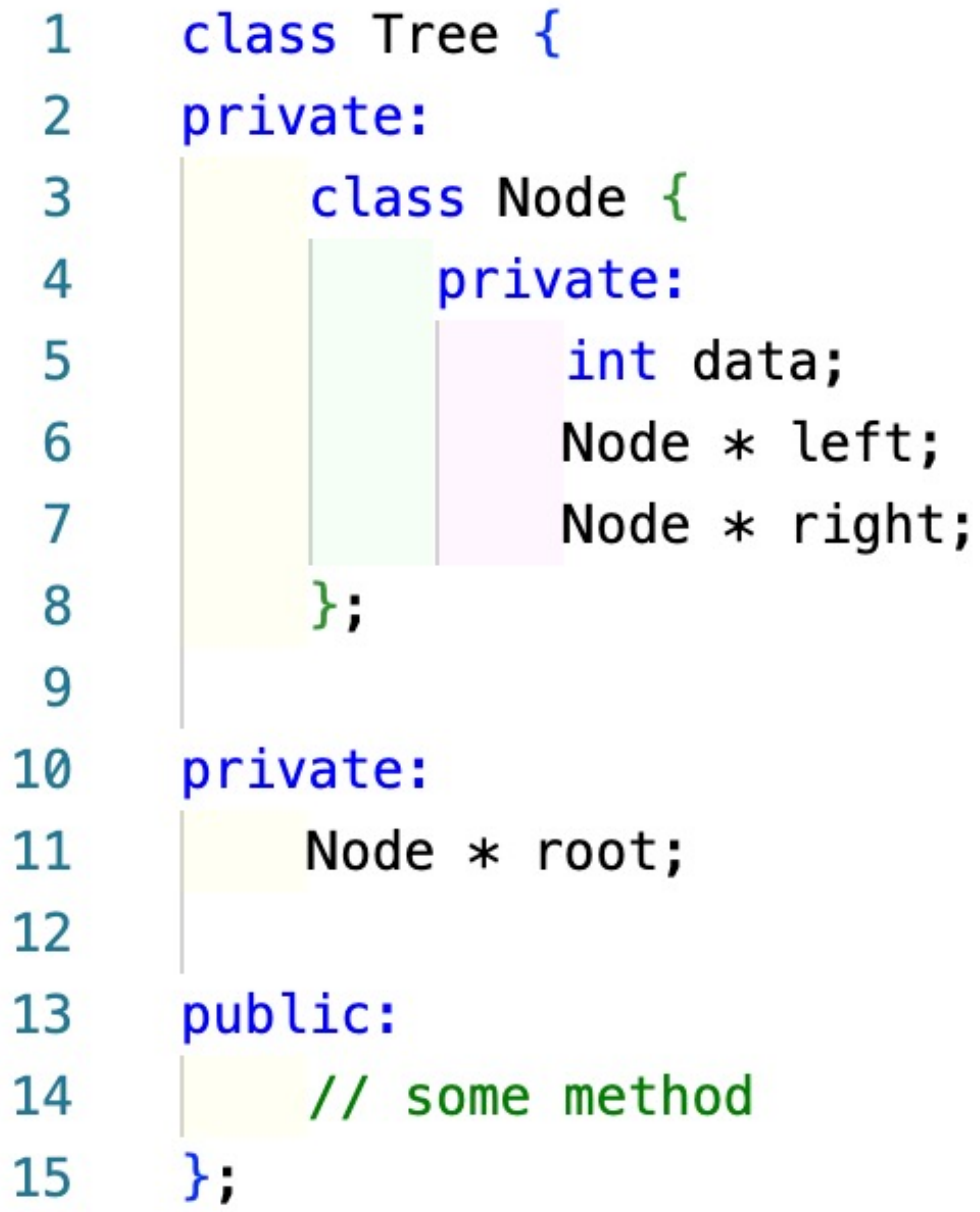
Nested class –Why?

- Avoid name conflicts while assisting the implementation of enclosed class



```cpp
class List {
private:
    class Node {
        private:
            int data;
            Node * next;
    };

private:
    Node * head;
    Node * tail;
    int count;
};
```



```cpp
class Tree {
private:
    class Node {
        private:
            int data;
            Node * left;
            Node * right;
    };

private:
    Node * root;

public:
    // some method
};
```

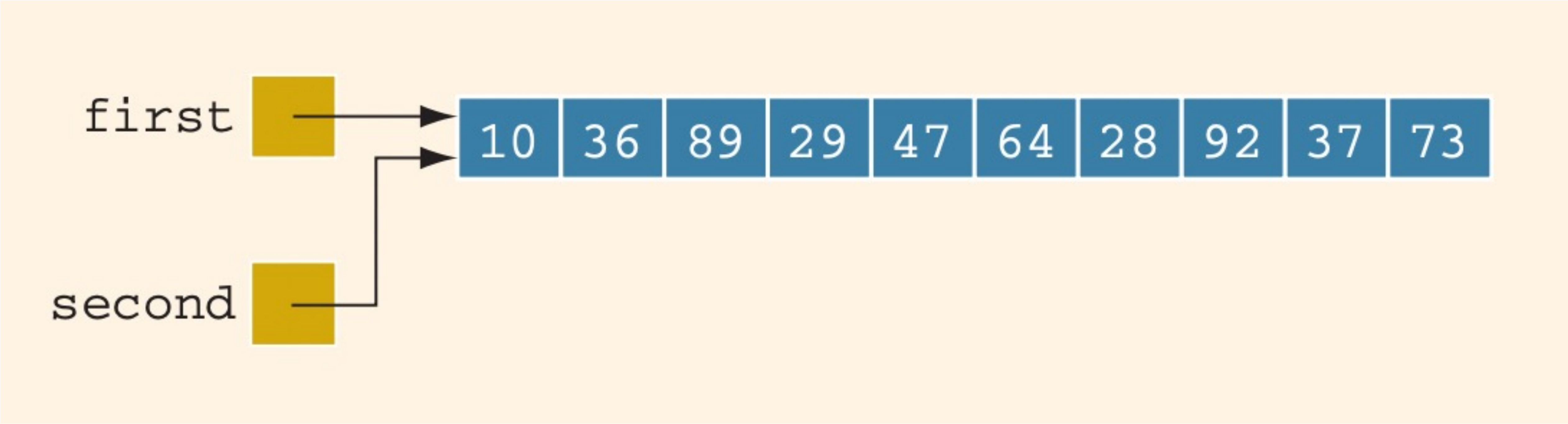
Shallow vs Deep Copy

Shallow Copy

- **Shallow copy**: two or more pointers of the same type point to the same memory

```
second = first;

delete [] first;
```



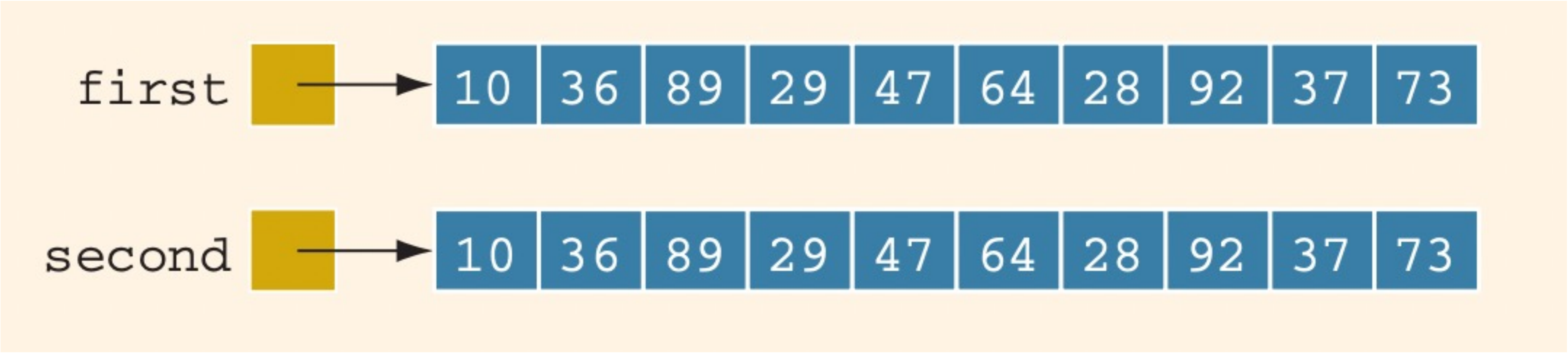
- **Problem**: If the program later tries to access the memory pointed to by second, either the program will access the wrong memory or it will terminate in an error

Deep Copy

- Deep copy: two or more pointers have their own data

```
second = new int[10]

for (int i = 0; i < 10; ++i)
  second[i] = first[i]
```



first → 10 36 89 29 47 64 28 92 37 73

second → 10 36 89 29 47 64 28 92 37 73

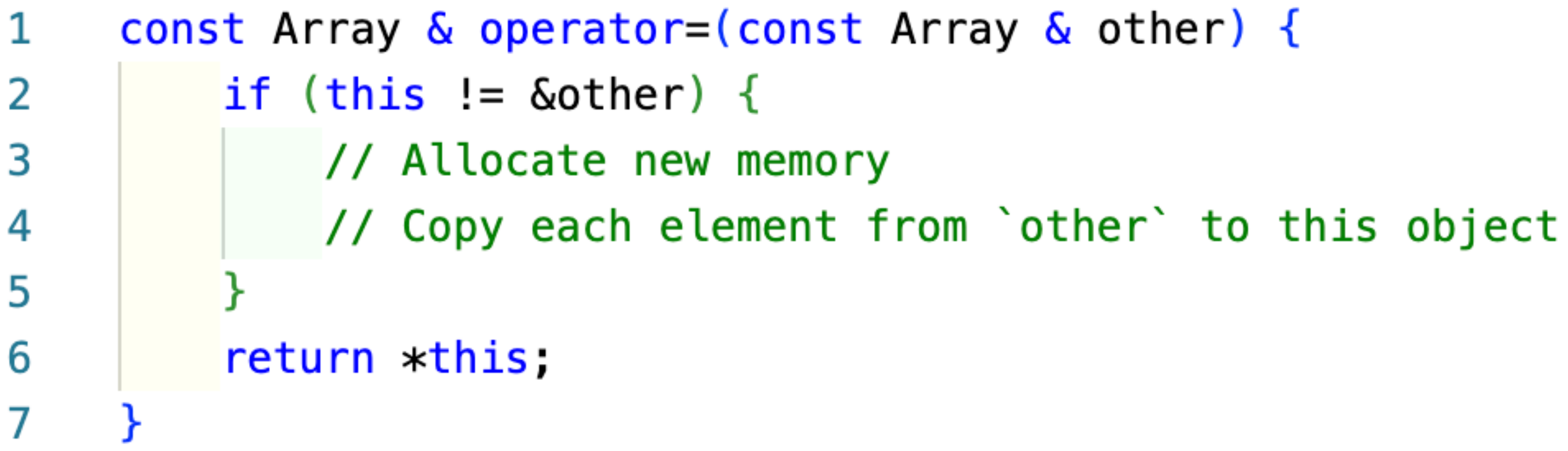
Assignment Operator

- Automatically provided by compiler
- Copies value in member variables from one object to the other
- If the member variable is a pointer, this copying would lead to Shallow Copy

```
class Array;
Array arr1;
Array arr2;
arr2 = arr1; // Assignment Operator
```

Overloading Assignment Operator

- To help Assignment Operator results in both objects have their own data (Deep Copy)
- Example Code



```
const Array & operator=(const Array & other) {
    if (this != &other) {
        // Allocate new memory
        // Copy each element from `other` to this object
    }
    return *this;
}
```

Copy Constructor

- When declaring a object, you can initialize it by using the value of an existing object of the same type

```
Array array(otherArray);
```

- Automatically provided by compiler: element-wise initialization
- Like Assignment Operator, if the member variable is a pointer, this initialization would lead to Shallow Copy

Class & Cause to Shallow Copy

- If a class has pointer member variables:
  - During object declaration, the initialization of one object using the value of another object would lead to a shallow copying of the data if the default member-wise copying of data is allowed.
  - If, as a parameter, an object is passed by value and the default member-wise copying of data is allowed, it would lead to a shallow copying of the data.

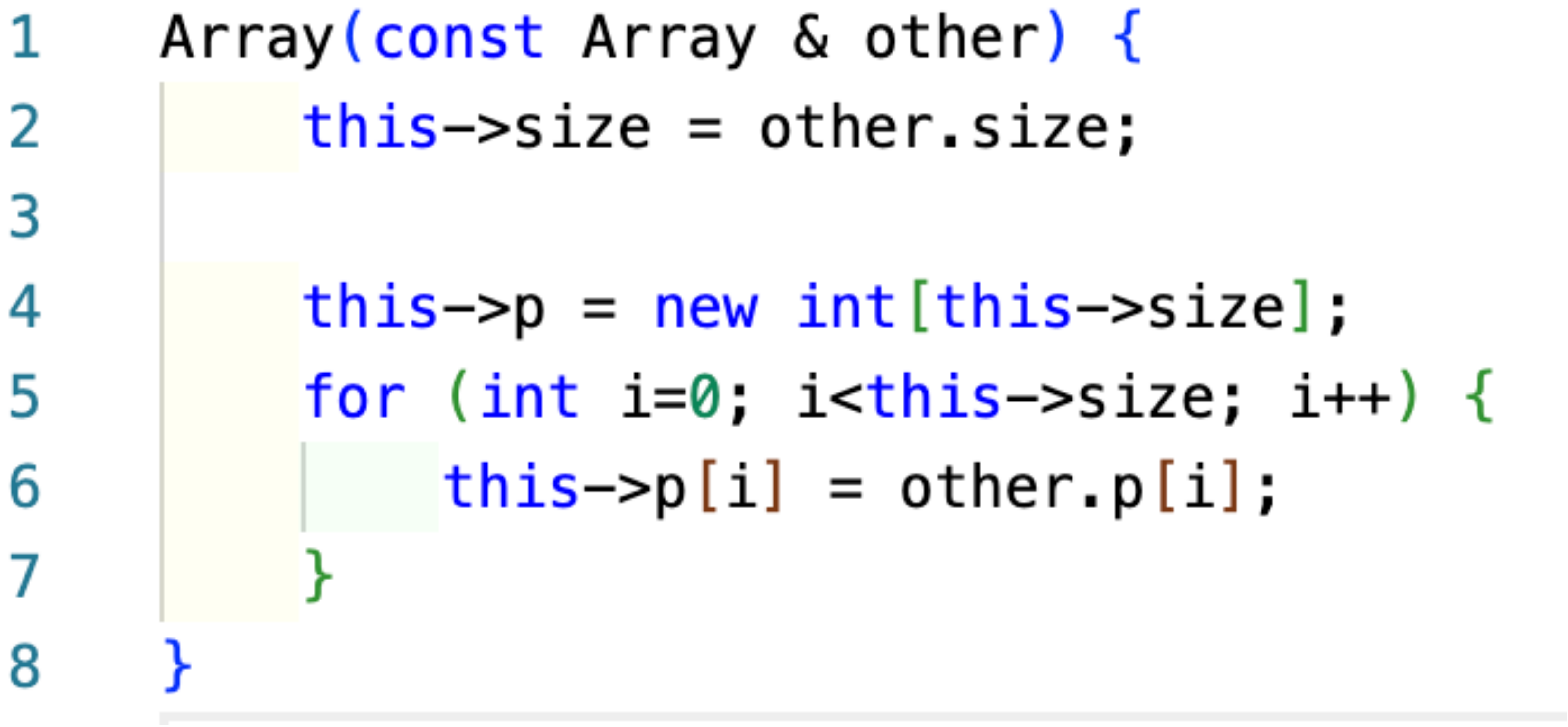
```
void display(Array array);
```

Copy Constructor Called

- The copy constructor automatically executes in three situations
  - When an object is declared and initialized by using the value of another object
  - When, as a parameter, an object is passed by value
  - When the return value of a function is an object

Overloading Copy Constructor

- To help Copy Constructor results in both objects have their own data (Deep Copy)
- Example Code:



```
Array(const Array & other) {
    this->size = other.size;

    this->p = new int[this->size];
    for (int i=0; i<this->size; i++) {
        this->p[i] = other.p[i];
    }
}
```

Summarise