CS4414 Recitation 5

Continue with containers and classes

09/2024

Alicia Yang

What is C++?

A federation of related languages, with four primary sublanguages

- **C**: C++ is based on C, while offering approaches superior to C. Blocks, statements, processor, built-in data types, arrays, pointers, etc., all come from C
- **Object-Oriented C++:** "C with Classes", classes including constructor, destructors, inheritance, virtual functions, etc.
- **Template C++:** generic programming language. Gives a template, define rules and pattern of computation, to be used across different classed.
- **STL(standard template library):** a special template library with conventions regarding containers, iterators, algorithms, and function objects

Overview

- C++ classes
  - Copy constructor, move construction
  - Operator overload
  - C++ objects and containers

Recap: C++ Classes

- Once a class is defined, you can define instances(called objects)
- Unlike JAVA, class objects are NOT null references in C++
- This means that when you create an object, all of its internal must be initialized. When the object goes out of scope, it is destroyed (deconstructed).
- Each class has at least one constructor and one destructor

More on Constructor Destructor

Recap: Constructors

- Constructor are used to **initialize** object of the class type
- A constructor has the **same name** as the class and **no** return type. It can have as many argument as needed
- e.g.,
  - myClass(); // default constructor
  - myClass(int x, std::string str); // Parameterized constructor
  - …

Constructor: construct and initialize objects of that class

- Default constructor: a constructor can be called with no argument

```
Rectangle::Rectangle():
        width(0),
        length(0),
        area(0)
{
    // Constructor body (can be empty or contain additional logic)

}
```

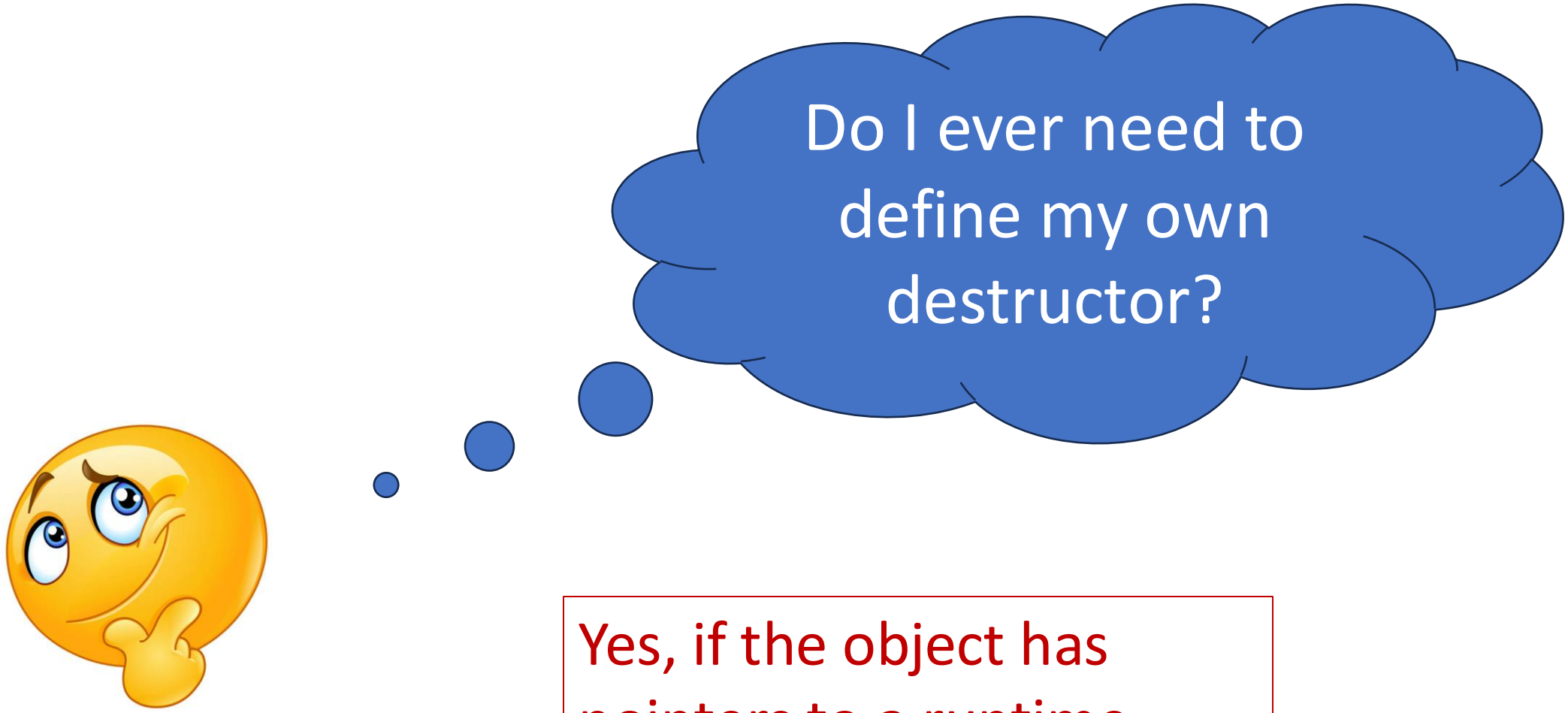
Recap: Destructor

- Destructor is called when the lifetime of an object **ends**
- It is used to **free** the resources that the object acquired during its lifetime
- e.g.,
  - ~myClass();

Implicit constructor and destructor

- Implicit default constructor:
  - If there is **no user-declared constructor** for a class type, **the compiler** implicitly provides a (public) default constructor.
  - If there **is** user-declared constructor, **the compiler** will NOT implicitly create the default constructor
- Implicit default destructor
  - The implicitly-defined destructor defined by the compiler has an empty body

Implicit constructor and destructor



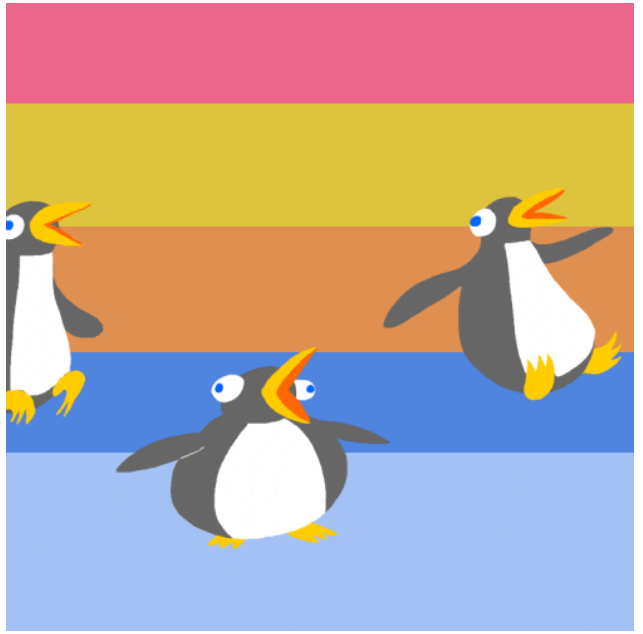
Yes, if the object has pointers to a runtime allocation of resources

MyIntVector example

Demo

```
class MyIntVector {
private:
    int* data;          // Pointer to dynamically allocated array
    size_t size;    // Number of elements in the vector
public:
     MyIntVector(size_t s) : size(s), data(new int[s]) {
          for (size_t i = 0; i < size; ++i) {
               data[i] = 0;
          }
     }       ……
};
```

Copying in C++



- Copying data, copying memory from one place to another
- Copying takes time

Copying in C++

```
int a = 5;
int b = a;
```

// creating a copy of int a

```
Rectangle obj1 = Rectangle(10.0, 11.0);
Rectangle obj2 = obj1;
```

// obj2 is a copy of object obj1

Copying in C++

```
Rectangle obj1 = Rectangle(10.0, 11.0);
Rectangle obj2 = obj1;
obj2.width = 100.0;
```

Now, what's the value of obj1.wdith?

Copying in C++

```
Rectangle* obj1 = new Rectangle(10.0, 11.0);
Rectangle* obj2 = obj1;
Obj2->width = 100.0;
```

Now, what's the value of obj1.wdith?

Copy constructor

```
class Rectangle{

        Rectangle(const Rectangle& other);

}
```

// Construct an object of class Rectangle by copying Rectangle object, other, passing by reference.

Copy constructor

Demo

- **Create** a new object by **initializing** it with an object of the same class
- Called when
  - Initialization `Rectangle obj2 = obj1;`
  - Function argument passing by value `func(Rectangle obj);`
  - Function return by value `return obj;`

// for c++ 17+, there is RVO(return value optimization) that could avoid copy and use move instead

Implicitly-defined default copy-constructor

- If no user-defined copy con… …d define a copy constructo…
  - It performs member-wise … …and …s to the new object it initializes

Do I ever need to define my own copy-constructor?

Yes, if an object has pointers or any runtime allocation of resources

Implicitly-defined default copy-constructor

- If no user-defined copy constructor, the compiler declare and define a copy constructor
  - It performs member-wise copy of the object's bases and members to the new object it initializes
  - Default constructor does only **shallow** copy

myIntVector example

Demo

```
class myIntVector{
public:
        int* data;
        size_t size;
        size_t capacity;

        myIntVector();
        myIntVector(size_t s);
        ~myIntVector();
        ...
};
```

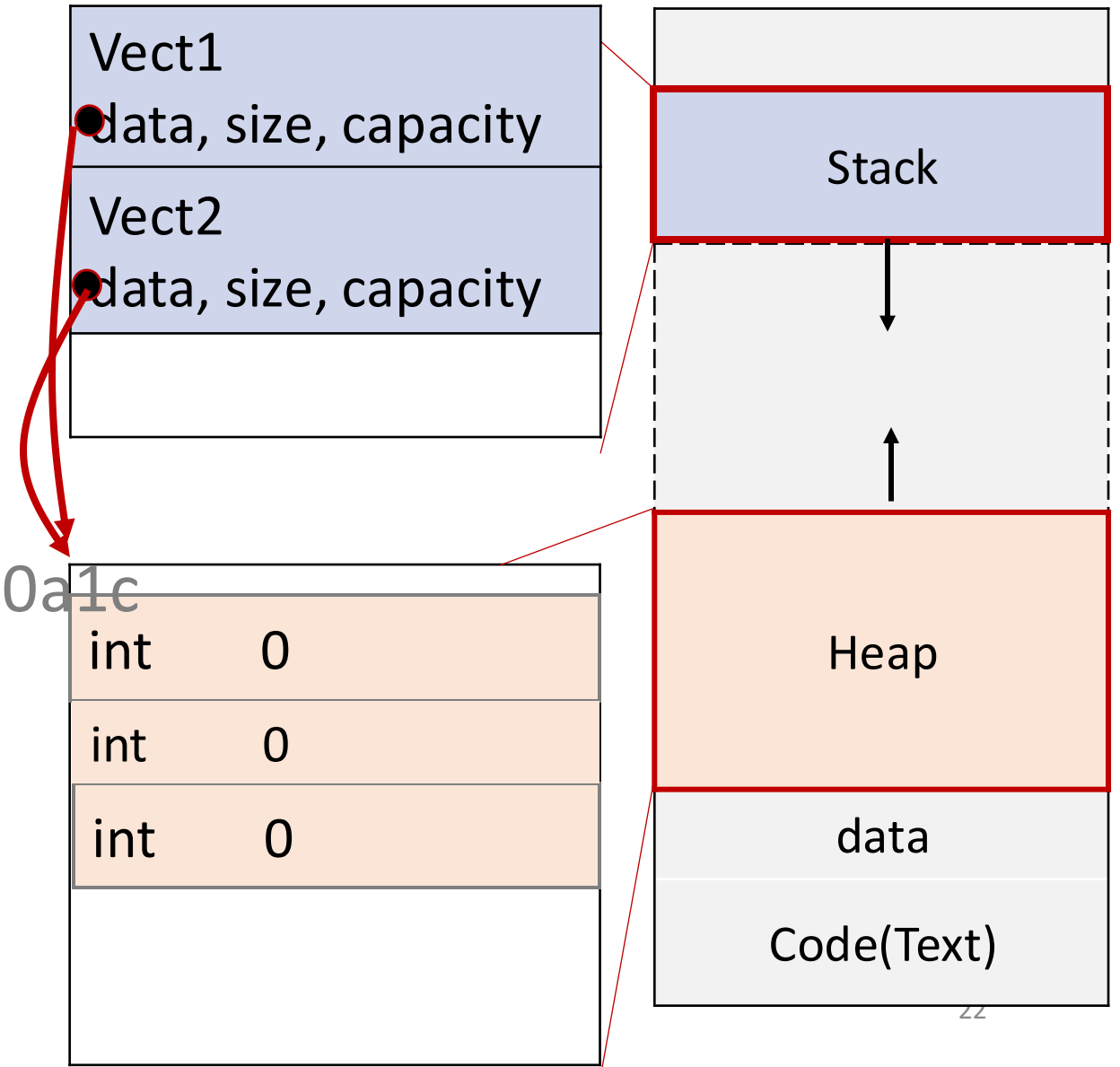
```
myIntVector::myIntVector(size_t s) {
        size = s;
        capacity = s;
        data = new int[capacity];
        for (size_t i = 0; i < size; ++i) {
            data[i] = 0;
        }
}
myIntVector::~myIntVector(){
          delete[] data;
}
```

Default copy-constructor

shallow copy

```
myIntVector vect1 = myIntVector(3);

myIntVector vect2 = vect1;

vect1.data = 5;
```

Not ideal, because changing only one changes both of them. Want two identical independent objects

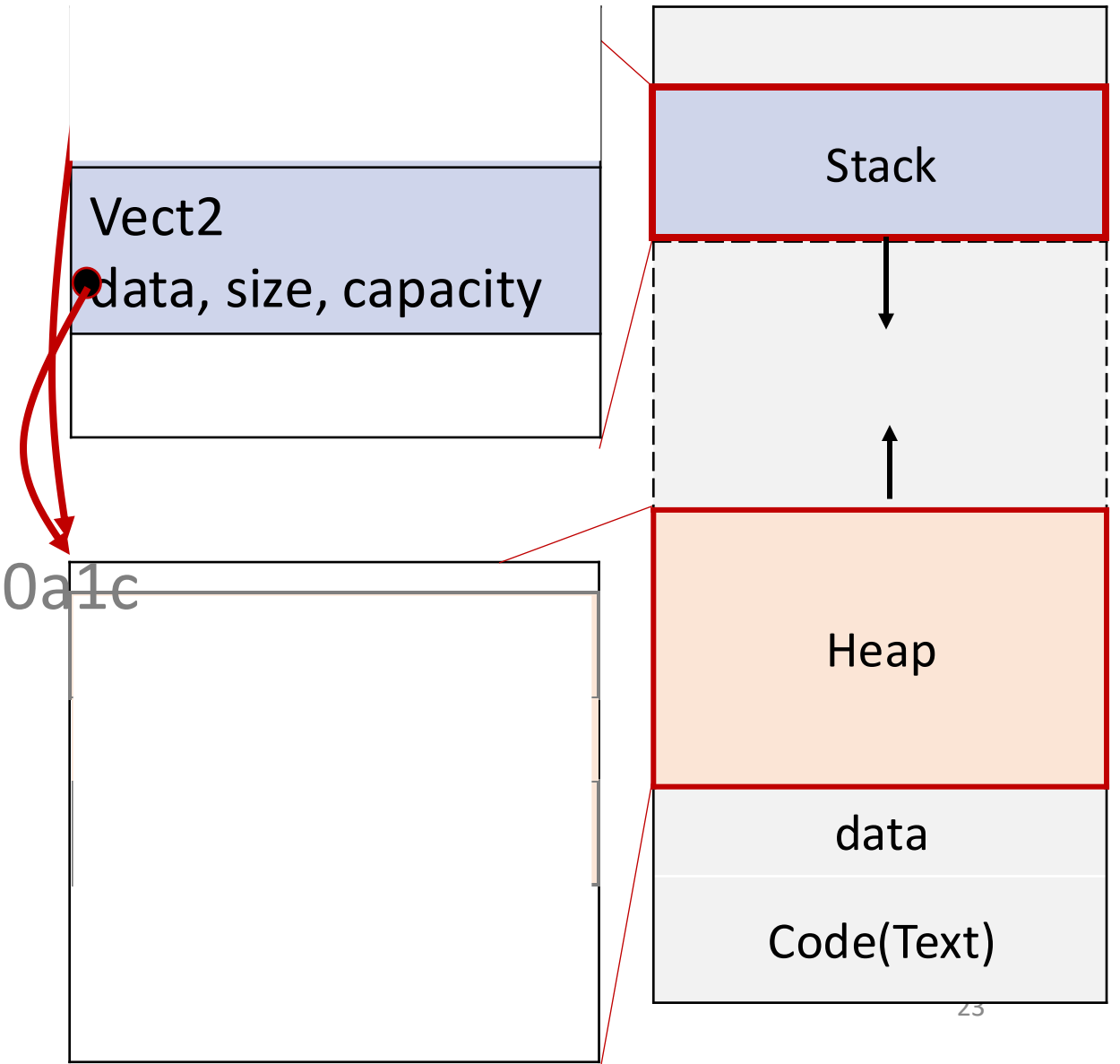


Default copy-constructor

shallow copy

```
myIntVector func(){

  myIntVector vect1 = myIntVector(3);

  myIntVector vect2 = vect1;

}

myIntVector vect = func();
```

Bad, because now vect2.data points to nothing



Fix: User-defined copy constructor

```
myIntVector(const myIntVector& other) :
    size(other.size), data(new int[other.size]) {
      for (size_t i = 0; i < size; ++i) {
          data[i] = other.data[i];
      }
}
```

Deep copy the object's members

Move constructor

```
class myIntVector{

        myIntVector(myIntVector && other);

}
```

// Transfer the ownership of the resources from the object, other, to the new object

Move constructor

- Transfer the ownership of resources from one object to another, instead of making a copy
- Called when
  - Initialization
  - Function argument passing
  - Function return with Return Value Optimization(RVO)

```
Rectangle obj2 = std::move(obj1);
```

```
func(std::move(obj));
```

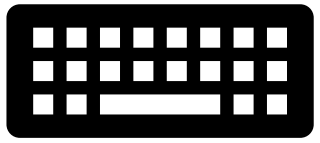
Why use move constructor?

Demo

- Improve the performance of the program by avoiding the overhead caused by unnecessary copying.

```
myIntVector(myIntVector&& other)  : size(0), data(nullptr) {
    data = other.data;          // copy the pointer of the memory address of other.data
    size = other.size;
    other.data = nullptr;       // Transfer the ownership of other's resource to this new object
    other.size = 0;
}
```

C++ Class keyword



this keyword

the address of the implicit object parameter (object on which the implicit object member function is being called)

```
class T {
        int x;
        void foo() {
                x = 6;       // same as this->x = 6;
                this->x = 5;      // explicit use of this->
         }
        …
};
```

https://en.cppreference.com/w/cpp/language/this

default & **delete** keyword

- Using the keywords **default** and **delete**, you can enable or disable a constructor
- When to disable the copy constructor?
  - When you want unique ownership of a resource and disallow it duplicated. E.g std::unique_ptr
  - myClass(const myClass& other) = **delete**;
- If you write a parameterized constructor, but still want to keep a default constructor
  - myClass() = **default**;

Static keyword

declarations of class members not bound to specific instances

Static data members of a class

- A data member that is **shared** by all objects of the class
- Static data members **cannot** be **initialized** in **constructors** (because they don't exist per class object)

Rectangle example

Demo

```
class Rectangle{
        float width;
        float length;
        static int count;
public:
        Rectangle();
        ...
};
int Rectangle::count = 0;
```

```
int main(){
        Rectangle::count ++ ;
        std::cout << " rectangle count is: " << Rectangle::count << std::endl;
}
```

Question:
Which memory segment does static member data, count live in member after initialized?

Static data members of a class

- Prefer static class member over global
  - Better encapsulation
  - Avoiding name collisions
  - Improve maintainability

Static member functions of a class

Demo

- A member function **independent** of any instance of the class
- Scope
  - Accessed using the **class name** through the scope resolution operator
- Class member access
  - Can access **static** (data/function) members
  - **Cannot** access **non-static** (data/function) **members**

Class operator overloading

Operator overload

- Customizes the C++ operators for operands of user-defined types.

Operators that can be overloaded	Examples
Binary Arithmetic	+, -, *, /, %
Assignment	=, +=,*=, /=,-=, %=
Bitwise	& , | , << , >> , ~ , ^
Subscript	[ ]
Function call	()
Relational	>, < , = =, <=, >=
…	…

Operator overload

- Customizes the C++ operators for operands of user-defined types.

```
std::string str = "Hello, ";
str.operator+=("world");
// same as str += "world";
operator<<(operator<<(std::cout, str), '\n');
// same as std::cout << str << '\n';
```

Operator overload

Example overloading < operator

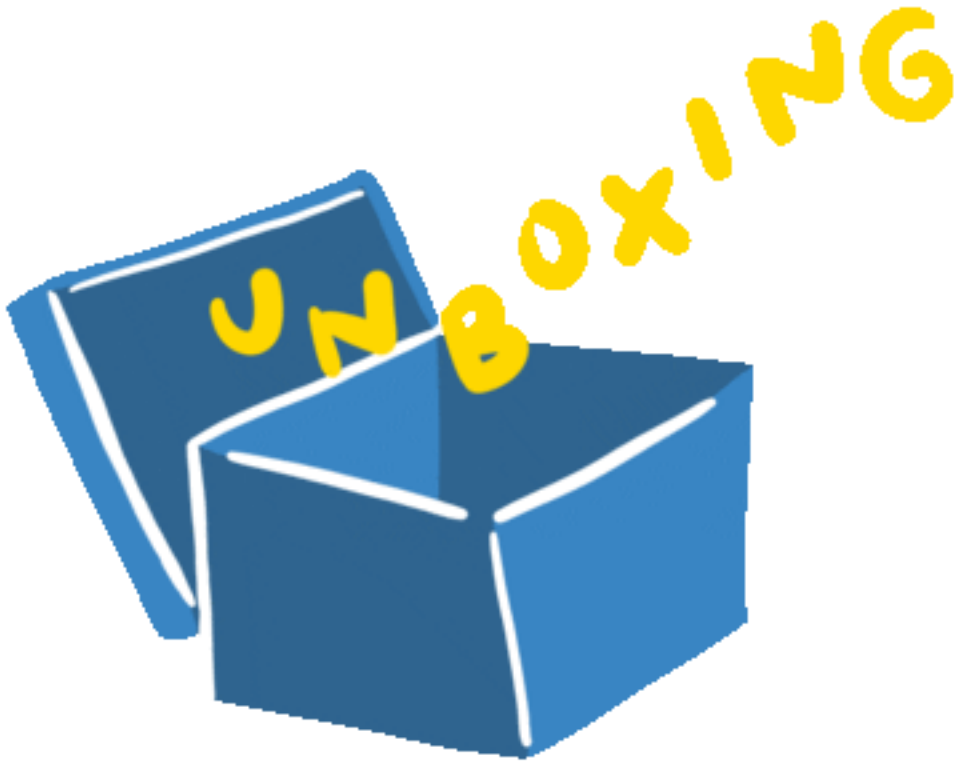
```
bool Rectangle::operator<(const Rectangle& other) const {
        return this->area() < other.area();
}

    // Overload the < operator to compare rectangles
    based on their area
```

Putting it all together

Demo

What is happening under the hood for C++ standard library?

Fun activity!



Combining what we learnt about classes with vector

std::vector

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

https://en.cppreference.com/w/cpp/container/vector

push_back

- Appends the given element value to the end of the container.
- The new element could be constructed via

```
std::vector<Rectangle> rec_vec;

Rectangle rec1;

rec_vec.push_back(rec1);                    // Copy constructor

rec_vec.push_back(std::move(rec1));   // Move constructor
```


https://en.cppreference.com/w/cpp/container/vector

emplace_back

- Appends a new element to the end of the container
- Besides the capabilities of push_back, it allows construct the new element **in-place**

```
std::vector<Rectangle> rec_vec;

rec_vec.emplace_back(10.0, 11.0);      // parameterized constructor

Rectangle rec1;

rec_vec.emplace_back(rec1);            // Copy constructor
```


https://en.cppreference.com/w/cpp/container/vector

Exercise: Find the error

```
class myClass {
public:
    myClass(int x) {}
private:
     int myInt;
};
std::vector<myClass> myObjects(4);
```

Exercise: Find the error

```
class myClass {
public:
    myClass(int x) {}
private:
     int myInt;
};
std::vector<myClass> myObjects( 4 );
```

Compiler no longer provides default constructor, because of user-defined constructor

std::vector needs a way to create default-constructed elements when resizing or initializing the vector with a specified size.

Exercise: Find the error

Fix

```
std::vector<myClass> myObjects; // size 0
myClass obj1(5);
                                    // constructed elements
myClass obj2(7);
myObjects.push_back(obj1);
myObjects.push_back(obj2);          // push_back invokes the
                                    copy constructor to copy
                                    the object into the vector
```

Where to find the resources?

- Copy constructor: https://www.geeksforgeeks.org/copy-constructor-in-cpp/
- Move semantics: https://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html
- Operator overload: https://www.geeksforgeeks.org/operator-overloading-cpp/
- Effective C++: 55 specific ways to improve your programs and designs, Scott Meyers, 3rd edition
- A Tour of C++, Bjarne Stroustrup