C++ Course 10 : Operator overloading. Copy/move constructors.

By Oleksiy Grechnyev

Classes:

Before (Lecture 4): Class basics:
methods, fields, public, overloading, ctors ...

Today: Classes:

1. Operator overloading - Перегрузка операторов

==, <, +=, + , ++, [] , (), ...

2. Copy/Move operations - Операции Copy/Move

The "Rule of 5" : Dtor, Copy/Move ctor, Copy/Move assignment

Operator overloading

int a = 3 + 2; // int + int : built-in

string s = string("Hello") + "World"; // string + const char[] ???

Suppose we have class **MyClass** :

Предположим у нас есть класс **MyClass** :

MyClass m1(1), m2(2);

MyClass m = m1 + m2;

How does it work? Как это работает?

MyClass m = MyClass(1) + MyClass(2);

MyClass m = operator+(m1, m2); // Function, non-member operator

OR / ИЛИ

MyClass m = m1.operator+(m2); // Method, member operator

Operators that may be overloaded

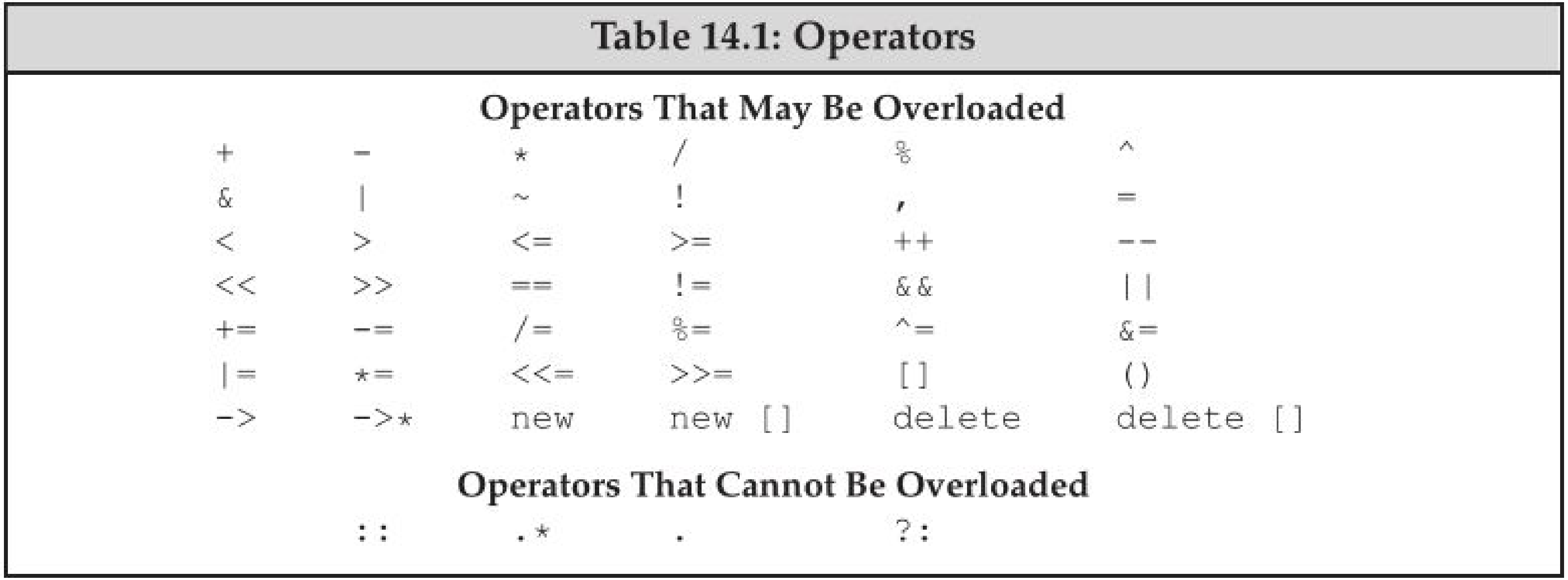


Table 14.1: Operators

Operators That May Be Overloaded

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

Operators That Cannot Be Overloaded

::	.*	.	?:

Cannot create new operators ! Нельзя создавать новые операторы !

Operators are often **inline**. **noexcept** is a good idea.

Non-member operators are often **friend**.

Operator parameter types, return type, behavior

```
??? operator+ (??? lhs, ??? rhs)  { ... }
```

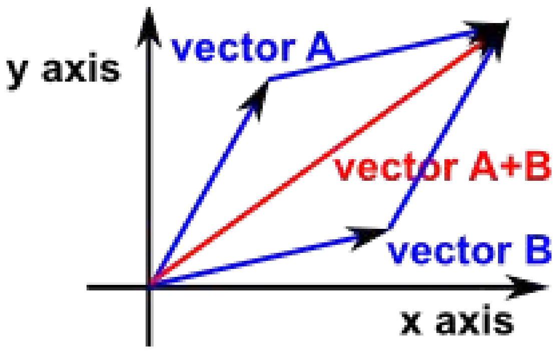
Can be (almost) anything. Может быть (почти) что угодно.

But usually we want it to behave like **+** for primitive types.

Но обычно мы хотим чтобы он вел себя как **+** для примитивных типов.

Example: Vec2 : 2-dimensional vector (x, y)

```
class Vec2{

public:  //===== Methods

        Vec2(double x, double y) : x(x), y(y) {}        /// xy ctor

        Vec2() = default;                              /// Default ctor

        ...
private: //===== Data
    double x = std::nan("");
    double y = std::nan("");
};
```



Operators: Members or Non-Members :

Operators can be members (methods) or non-members (functions) :

```
class Vec2 { ...
     Vec2 operator+(const Vec2 & rhs);   // Member
}   // End of class Vec2
Vec2 operator+(const Vec2 & lhs, const Vec2 & rhs);   // Non-Member
```

1. Must be members: **=, [], (), ->**

2. Usually members: **+=, ++,** unary ***, ...**

3. Usually non-members: **+, <, ==, ...**

Comparison : operator==, operator!=

Non-member:

```
class Vec2 { ...
     friend bool operator==(const Vec2 & lhs, const Vec2 & rhs)  noexcept;
     ...
}   // End of class Vec2

inline bool operator==(const Vec2 & lhs, const Vec2 & rhs)  noexcept {
   if (& lhs == & rhs)               // Optional, check if the same object
      return true;
   else
      return (lhs.x == rhs.x) && (lhs.y == rhs.y);
}
inline bool operator!=(const Vec2 & lhs, const Vec2 & rhs)  noexcept {
   return !(lhs == rhs);   // The proper way to define !=
}
```

Comparison: operator<

Non-member:

```
inline bool operator<(const Vec2 & lhs, const Vec2 & rhs) noexcept  {
        if (lhs.x == rhs.x)
                return lhs.y < rhs.y;          // In our example, x is more important that y
        else
                return lhs.x < rhs.x ;
}
inline bool operator>(const Vec2 & lhs, const Vec2 & rhs) noexcept {
    return rhs < lhs;
}
inline bool operator<=(const Vec2 & lhs, const Vec2 & rhs) noexcept  {
    return !(lhs > rhs);   // Like this
}
inline bool operator>=(const Vec2 & lhs, const Vec2 & rhs) noexcept  {
    return !(lhs < rhs);
}
```

istream input, ostream output

Non-member:

```
inline std::ostream & operator<< (std::ostream & os, const Vec2 & v){
    os << std::setw(10) << v.x << " " << std::setw(10) << v.y;  // No endl !
    return os;
}


inline std::istream & operator>> (std::istream & is, Vec2 & v) {
    is >> v.x >> v.y;
    if (!is)
        v = Vec2();   // Default (NaN) vector on IO error
    return is;
}
```

operator+=, operator-=, operator*=, operator/=

Member:

Add two vectors:

```
Vec2 & operator+= (const Vec2 & rhs) noexcept {
        x += rhs.x;
        y += rhs.y;
        return *this;
}
```

Multiply vector by a number:

```
Vec2 & operator*= (double rhs) noexcept {
        x *= rhs;
        y *= rhs;
        return *this;
 }
```

operator+, operator-, operator*, operator/

Non-member:

```
inline Vec2 operator+(const Vec2 & lhs, const Vec2 & rhs) noexcept {
    Vec2 temp = lhs;    // Make a copy
    temp += rhs;
    return temp;
}
inline Vec2 operator-(Vec2 lhs, const Vec2 & rhs) noexcept {
    lhs -= rhs;
    return lhs;
}
inline Vec2 operator*(Vec2 lhs, double rhs) noexcept {
    lhs *= rhs;
    return lhs;
}
```

operator++, operator--

Member:

Prefix version (**++v**):

```
Vec2 & operator++() noexcept {
        ++x;                    // Increase both x, y by 1
        ++y;
        return *this;    // Return self
}
```

Postfix version (**v++**): Dummy **(int)** argument signifies postix !

```
Vec2 operator++(int) noexcept {
        Vec2 temp{*this};            // Make a copy
        ++*this;            // Call prefix like this
        return temp;   // Return the copy
}
```

operator[]

Member:

Non-**const** version:

```
double & operator[] (int i) {
        switch (i) {
        case 0:
            return x;
        case 1:
            return y;
        default:
            throw std::out_of_range("Vec2::operator[]");
        }
 }
```

const version:

```
const double & operator[] (int i) const { ... }
```

operator()

Member:

```
void operator()(const std::string &s) noexcept  {
        std::cout << s << *this << std::endl;
}
```

Use **Vec2** as a function:

```
Vec2 a{1.0, 2.5};
a("Terrible Vector");
```

operator double (Cast to double)

Member (of class **Vec2**) :

```
double len(){   // Normal method
        return std::sqrt(x*x + y*y);
}
explicit operator double() noexcept {
        return len();
}
```

To use it:

```
Vec2 a{1.0, 2.5};
double d1 = (double) a;                  // OK
double d2 = static_cast<double>( a );     // OK
double d3 = a;                          // ERROR ! Forbidden by explicit !
double d4 = sqrt( a );                  // ERROR ! Forbidden by explicit !
```

Without **explicit** : implicit type conversions ! Dangerous !

Без **explicit** : неявные преобразования типа! Опасно !

operator bool

Member (of class **Vec2**) :

```
explicit operator bool() noexcept {
        return x || y;   // false is x == y == 0
}
```

Usage:

```
Vec2 a{1.0, 2.5};
bool b1 = (bool) a;                      // OK
bool b2 = static_cast<bool>( a );     // OK
bool b3 = a;                          // ERROR ! Forbidden by explicit !
```

But **if** and **?:** work fine with **explicit** :

```
if (a) ...     // OK

int z = a ? 13 : 25;   // OK
```

Copy/Move operations

The "*Rule Of Five*":

1. Destructor (Dtor)
2. Copy Constructor (Ctor)
3. Move Ctor
4. Copy assignment (**operator=**)
5. Move assignment (**operator=**)

Usually we don't have to implement them. Обычно не надо их реализовывать.
The compiler creates standard versions. Компилятор создает стандартные версии.
1. Empty Dtor.
2-5. Copy/ Move all fields.

No Copy operations if non-copyable fields:
Не генерируются операции копирования если есть поле, которое нельзя копировать:
references, **unique_ptr, istream, ostream, ...**

Copy and move constructors and assignment operators

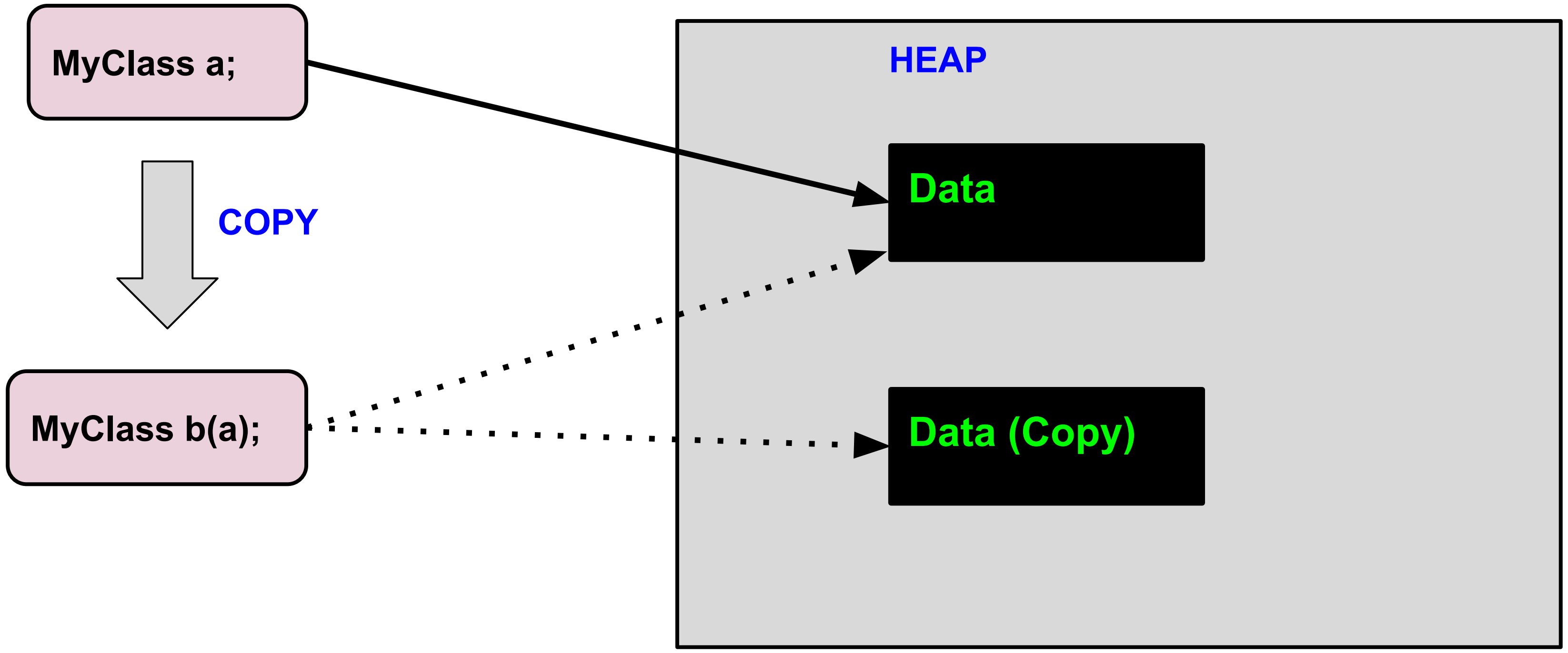
```
Tjej(const Tjej & rhs) : name(rhs.name){}              // Copy Ctor

Tjej(Tjej && rhs) : name(std::move(rhs.name)){}       // Move Ctor

Tjej & operator= (const Tjej & rhs) {              // Copy assignment
    if (this != &rhs)     // Check for self-assignment
        name = rhs.name;
    return *this;
}

Tjej & operator= (Tjej && rhs) {                   // Move assignment
    if (this != &rhs)     // Check for self-assignment
        name = std::move(rhs.name);
    return *this;
}
```

Tjej &&   is an *rvalue* reference, e.g. ref to temp object, e.g. **Tjej("Bettan")**

Terminology comes from C: **lvalue = rvalue;**

For example: **w = Tjej("Bettan");**

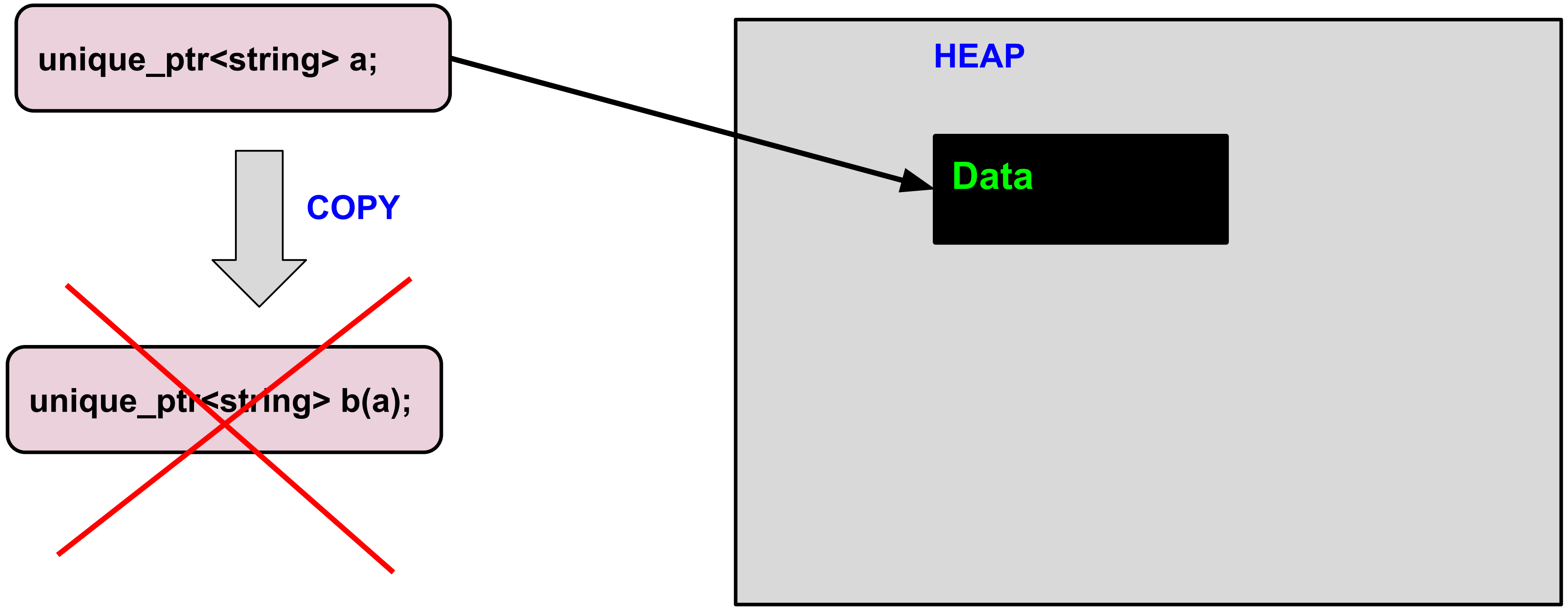
Class with heap resources:



What happens to the heap data when we copy an object ?

Что происходит с данными в хипе когда мы копируем объект?

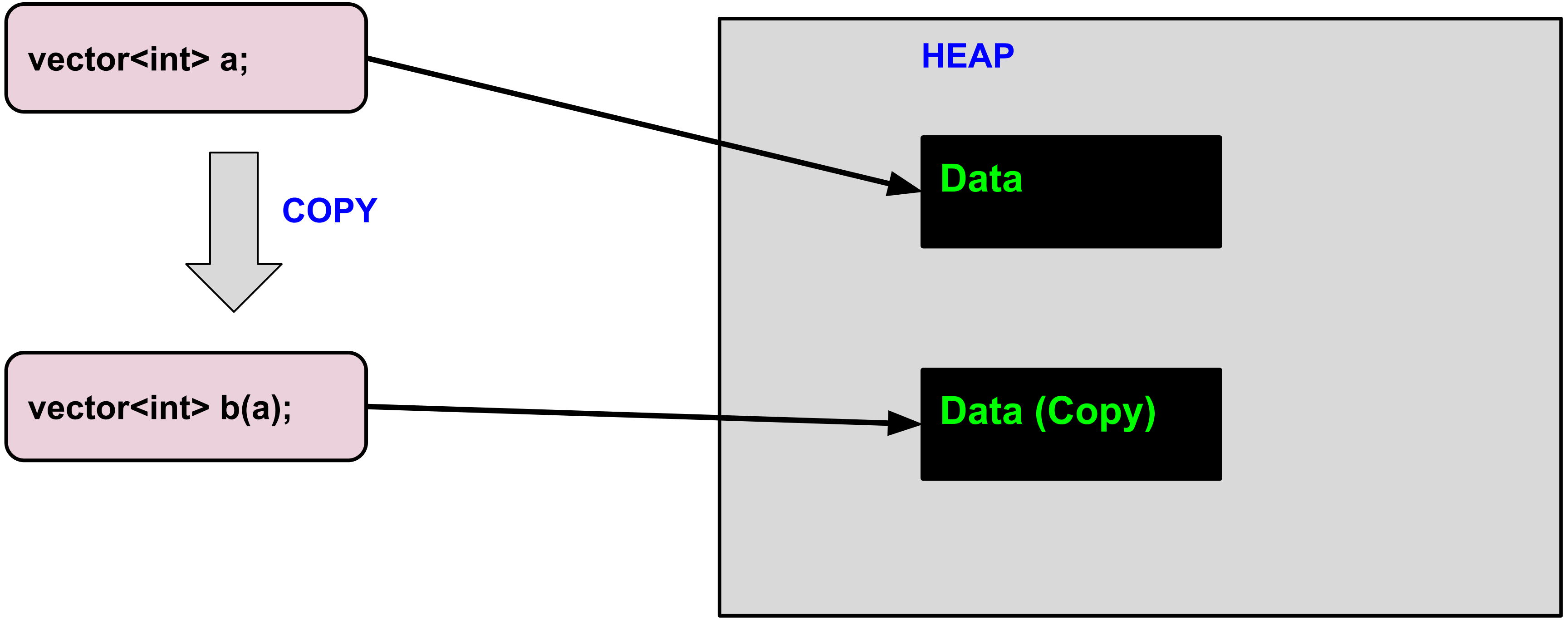
Unique object: Copying is forbidden



unique_ptr, istream, ostream

If you have a **unique_ptr** field, your class cannot be copied !

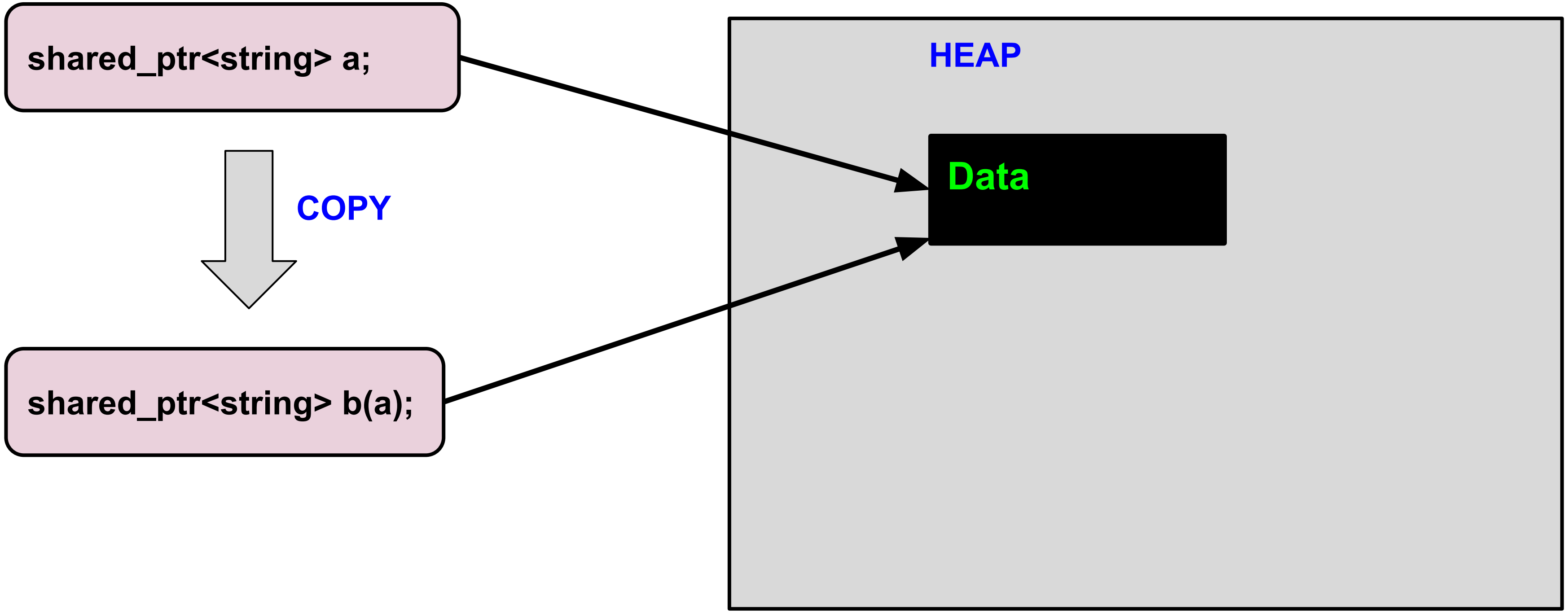
Value behavior: Copy resources



std::vector and most containers behave like this.

You can use container fields in your class for value behavior.

Pointer behavior: Do not copy resources



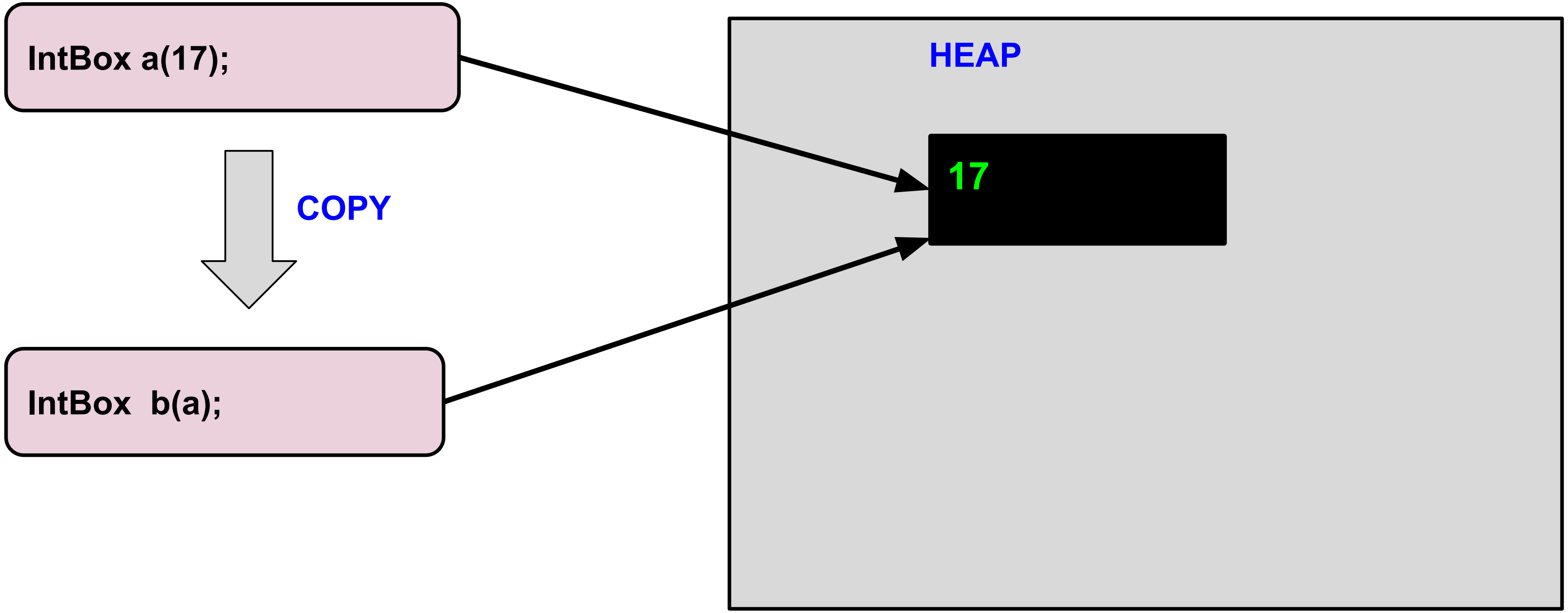
shared_ptr behaves like this. You have to count references !

Use **shared_ptr** fields in your class for pointer behavior !

Example : Implementing value behavior with pointers !

```
class IntBox{
public:
        IntBox() { }     // Empty Ctor

        IntBox(int n) :  data(new int(n)) { }     // Ctor, new heap object

        ~IntBox(){          // Dtor
                if (data)
                        delete data;
        }
        ...
private:
        int * data = nullptr;     // Pointer to data, nullptr if empty
}
```

May we copy InBox ?



Two pointers to a heap object ! Double **delete** by destructor !!!
And we wanted to COPY the heap data !

clear() method makes IntBox empty

```
class IntBox{
public:
        ..
        void clear(){
                if (data) {
                        delete data;
                        data = nullptr;
                 }
        }
        ...
private:
        int * data = nullptr;     // Pointer to data, nullptr if empty
}
```

Copy and move constructors:

Copy constructor: clone a heap object

```
IntBox(const IntBox & rhs) {     // Copy Ctor : Deep Clone
    if (rhs.data) {                      // If rhs is not empty
        data = new int(*rhs.data);      // Deep clone the heap object
    }
}
```

Move constructor:

```
IntBox(IntBox && rhs) {
    data = rhs.data;                     // Copy the pointer
    rhs.data = nullptr;                  // Set rhs to empty without delete
}
```

Copy assignment (operator=)

```
IntBox & operator=(const IntBox & rhs) {
        if (this != &rhs) {             // Check for self-assign
            clear();                    // Clear self first
            if (rhs.data) {             // If rhs is not empty
                data = new int(*rhs.data);      // Deep clone the heap object
            }
        }
        return *this;
}
```

Self - assignment: **a = a;**

Move assignment (operator=)

```
IntBox & operator=(IntBox && rhs) {
        if (this != &rhs) {              // Check for self-assign
            clear();                     // Clear self first
            data = rhs.data;             // Copy the pointer
            rhs.data = nullptr;          // Set rhs to empty without delete
        }
        return *this;
}
```

swap() function : For optimization

```
void swap(IntBox &lhs, IntBox &rhs) noexcept {
    using std::swap;
    swap(lhs.data, rhs.data);           // Swap pointers, uses std::swap
}
```

Usage :

```
using std::swap;   // Or using namespace std;
IntBox a(17), b(42);
swap(a, b);  // NOT std::swap() !!!
```

std::swap() is not very efficient.

We use **swap()** for a class specific version of swap.

We fall back to **std::swap()** if no class-specific version exists.

Thank you for your attention !

title

text