

C++ Course 8 : Classes 2

2020 by Oleksiy Grechnyev

Operator overloading

Operator **+** :

int **a** = **3** **+** **2**; // int + int : built-in, no questions

string **s1** = "Hello " **s** **+** "World"**s**; // string + string, how does it work ?

string **s2** = **string**("Hello ") **+** "World"; // string + const char[] ???

Suppose we have created our own class **MyClass**.

How can we implement operator **+** for it?

MyClass **m1**(1), **m2**(2);

MyClass **m3** = **m1** **+** **m2**; // How to do this?

MyClass **m4** = **MyClass**(1) **+** **MyClass**(2); // Or like this

Solution C++ *operator overloading*: **m1** **+** **m2** is a shortcut for either

MyClass **m** = **operator+**(**m1**, **m2**); // Overloaded function, a "non-member operator:

OR

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

Operators that can be overloaded

| Table 14.1: Operators | | | | | |
|-------------------------------------|-----|-----|--------|--------|-----------|
| Operators That May Be Overloaded | | | | | |
| + | - | * | / | % | ^ |
| & | | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | |
| += | -= | /= | %= | ^= | &= |
| = | *= | <<= | >>= | [] | () |
| -> | ->* | new | new [] | delete | delete [] |
| Operators That Cannot Be Overloaded | | | | | |
| | :: | .* | . | ?: | |

Cannot create new operators !

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

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

Operator parameter types, return type, behavior

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

Operator parameter types and return type can be (almost) anything.

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

Typically parameters and the return type are of the same type:

MyClass operator+ (**const MyClass & lhs**, **const MyClass & rhs**) { ... }

Note: Overloaded operators often include versions for implicit type conversion.

For example, for **string**, there are several overloaded versions of **operator+**:

string operator+ (**const string & lhs**, **const string & rhs**);

string operator+ (**const string & lhs**, **const char * rhs**);

string operator+ (**const char * lhs**, **const string & rhs**);

You can add a **std::string** to a C-string (but not two C-strings !)

Remember, functions **operator+** are all overloads of the same name!

Beware of *ambiguity* !

Example 1: Vec2 : 2-dimensional vector (x, y)

Let us design our own class: a 2D vector.

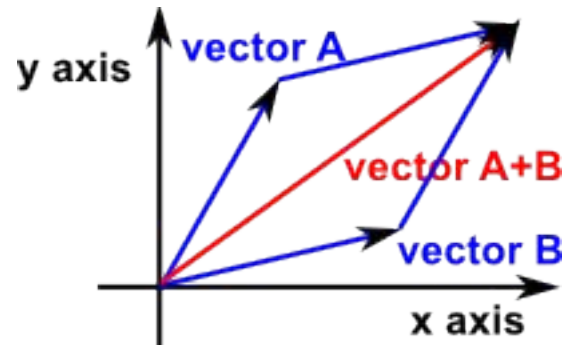
Let us overload various operators (+, == etc.).

Two private fields : **x**, **y**.

Getters and setters are not shown (see the full example in the repo).

Note: No custom copy/move operation in this class ! Standard copy behavior is OK.

```
class Vec2{  
public: //===== Methods  
    Vec2(double x, double y) : x(x), y(y) {}    /// xy ctor  
    Vec2() = default;                          /// Default (no-par) ctor  
    ...  
private: //===== Data  
    double x = std::nan(""); // Two private fields  
    double y = std::nan("");  
};
```



Operators: Members or Non-Members :

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

```
class Vec2 { ...
```

```
    Vec2 operator+(const Vec2 & rhs); // Member
```

```
    friend Vec2 operator-(const Vec2 & lhs, const Vec2 & rhs); // Declare as friend
```

```
}; // End of class Vec2
```

```
Vec2 operator-(const Vec2 & lhs, const Vec2 & rhs); // Non-Member
```

1. Must be members: =, [], (), ->
2. Usually members: special assignment and unary: +=, ++, unary *, - ...
3. Usually non-members: regular binary: +, <, ==, ...

Comparison : operator==, operator!=

Non-member:

```
class Vec2 { ... // Binary operators are usually non-members (friend functions)
    friend bool operator==(const Vec2 & lhs, const Vec2 & rhs) noexcept; // Returns bool
    ...
}; // End of class Vec2

inline bool operator==(const Vec2 & lhs, const Vec2 & rhs) noexcept {
    if (& lhs == & rhs)           // Optional, check if it is the same object
        return true;
    else // Compare all fields
        return (lhs.x == rhs.x) && (lhs.y == rhs.y);
} // reflexive, symmetric and transitive

inline bool operator!=(const Vec2 & lhs, const Vec2 & rhs) noexcept {
    return !(lhs == rhs); // The proper way to define operator!= is via operator==
} // Now operator== and operator!= are consistent !
```

Note : *inheritance* and **operator==** is a *bad* combination!

Comparison: operator<

Non-member binary operators: should be consistent with each other !

```
inline bool operator<(const Vec2 & lhs, const Vec2 & rhs) noexcept {  
    if (lhs.x == rhs.x) // The only comparison actually implemented !  
        return lhs.y < rhs.y; // In our example, x is more important than y  
    else  
        return lhs.x < rhs.x ;  
}  
// I know 2D vectors cannot be compared, just an example !  
inline bool operator>(const Vec2 & lhs, const Vec2 & rhs) noexcept {  
    return rhs < lhs; // Use existing operator<  
}  
inline bool operator<=(const Vec2 & lhs, const Vec2 & rhs) noexcept {  
    return !(lhs > rhs); // Use existing operator>  
}  
inline bool operator>=(const Vec2 & lhs, const Vec2 & rhs) noexcept {  
    return !(lhs < rhs); // Use existing operator<  
}
```


istream input, ostream output: overload bit shifts

Write a vector to a stream.

Note: Non-member function with different argument types.

```
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;  
}
```

Read a vector from a stream.

```
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;  
}
```

Do this for all your classes, if you want to read/write them !

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

Binary + assign operators, member operators by convention.

Add two vectors:

```
Vec2 & operator+= (const Vec2 & rhs) noexcept {  
    x += rhs.x;  
    y += rhs.y;  
    return *this; // Return the object by reference !  
}
```

Multiply vector by a number:

```
Vec2 & operator*= (double rhs) noexcept {  
    x *= rhs;  
    y *= rhs;  
    return *this; // Return the object by reference !  
}
```

Note: Always return ***this**, so that assignments can be chained (**a=b=c=d+=Vec2(3, 4)**).

operator+, operator-, operator*, operator/

Non-member binary operators. Use existing **operator+=** etc.

```
inline Vec2 operator+(const Vec2 & lhs, const Vec2 & rhs) noexcept {
```

```
    Vec2 temp = lhs;    // Make a copy of lhs
```

```
    temp += rhs;    // Based on the existing operator+ ! Add rhs to it.
```

```
    return temp;    // Return the result by value
```

```
} // Better pass lhs by value and avoid the copy !
```

```
inline Vec2 operator-(Vec2 lhs, const Vec2 & rhs) noexcept {
```

```
    lhs -= rhs;    // DO like this !
```

```
    return lhs;
```

```
}
```

```
inline Vec2 operator*(Vec2 lhs, double rhs) noexcept {
```

```
    lhs *= rhs;    // Multiply vector by double: different types
```

```
    return lhs;
```

```
}
```

```
inline Vec2 operator*(double lhs, const Vec2 & rhs) noexcept { return rhs*lhs; }
```

operator++, operator--

Member operators (Note: Not a proper vector operation !):

Prefix version (++v):

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

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

```
Vec2 operator++(int) noexcept {  
    Vec2 temp{*this};    // Make a copy of self  
    ++*this;             // Call prefix like this  
    return temp;         // Return the copy by value. Less efficient !  
}
```

operator[]: Index vector components !

Member operator:

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, needed for indexing const objects!

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

operator()

Member operator, prints the vector to console:

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

Now we can use **Vec2** as a function:

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

Any better uses for **operator()**? OpenCV and Eigen use it often.

operator double (Cast operator to double)

Member operator (of class **Vec2**) : Casts **Vec2** to **double**, "inverse constructor".

```
double len(){ // A regular method: Norm of the vector
```

```
    return std::sqrt(x*x + y*y);
```

```
}
```

```
explicit operator double() noexcept {
```

```
    return len();
```

```
}
```

explicit version can only be used in explicit casts:

```
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** : can be used for implicit type conversions ! Dangerous !

operator bool

Member operator (of class **Vec2**) : Casts **Vec2** to **bool**.

```
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 !
```

Principle function of **operator bool** : use in **if** statements.

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

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

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

Vec2 is now complete! Hurray!

The rule of five (previously "three")

Normally, a class has the following default methods (auto-created by the compiler)

- copy constructor (copy all fields)
- move constructor (move all fields)
- copy assignment (copy all fields)
- move assignment (move all fields)
- destructor (empty)

Rule of five: If you implement *any* of these methods, the defaults are no longer generated, you will have to implement *all* of them. Idea: you might have custom memory management, or some external resources (streams, sockets), thus default behavior is wrong.

Use **default** + **delete** to override this behavior if needed.

Note: This has nothing to do with the default (no-param) Ctor, like **Warrior()**.

It is generated if and only if there are no other (explicitly defined) constructors.

Nothing like this in our **Vec2** ! Only default copy behavior which copies two fields !

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");

Classes with custom memory usage?

Our first example **Vec2** had no custom memory management.

Many classes do though. How to create one?

sizeof(C) is determined at compile time for every C++ class **C**!

You cannot change the size of the object or set it at runtime!

But dynamic data size is possible with the heap, **new[]** and **delete[]** (or **malloc/free**).

Warning: For your data it is better to use existing **vector** or **cv::Mat** something !

Just make a **vector** field in your class. Don't reinvent the wheel!

Example 2 below (**IntBox**) is just for better understanding!

Don't try to reproduce it in real life!

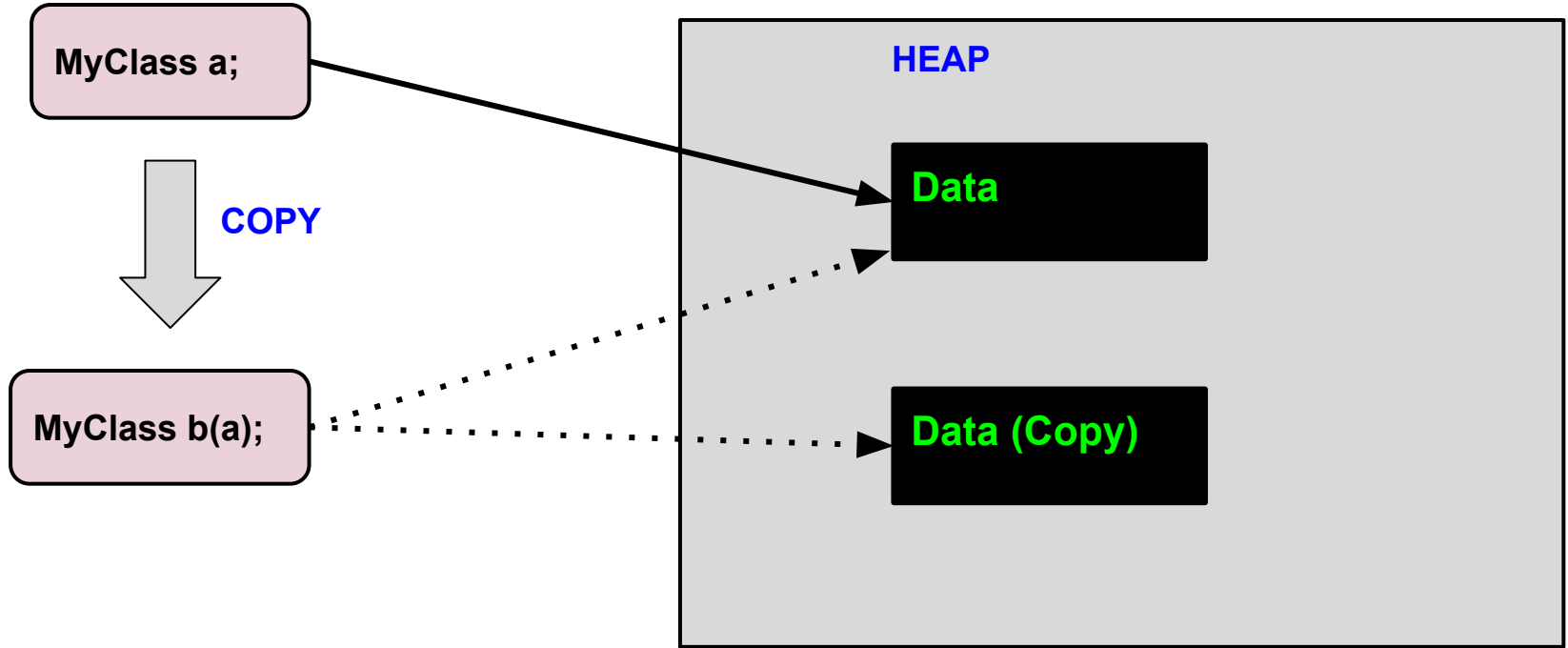
Classes which use the heap. How do they behave when copied?

Value behaviour : **vector** and other C++ containers. Copies data.

Pointer behavior: **cv::Mat** , **shared_ptr**. Copies reference, not data.

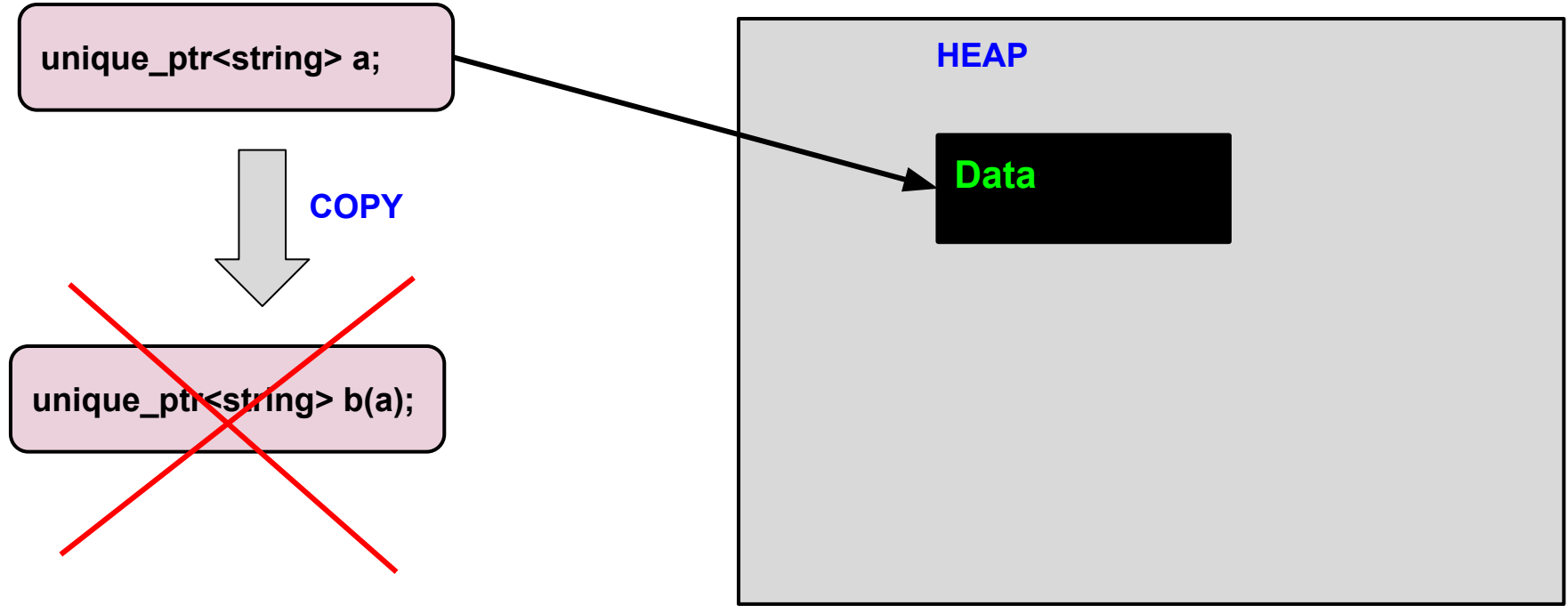
Unique behavior: **unique_ptr**, **istream**, **ostream**. Cannot be copied.

Class with heap resources:



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

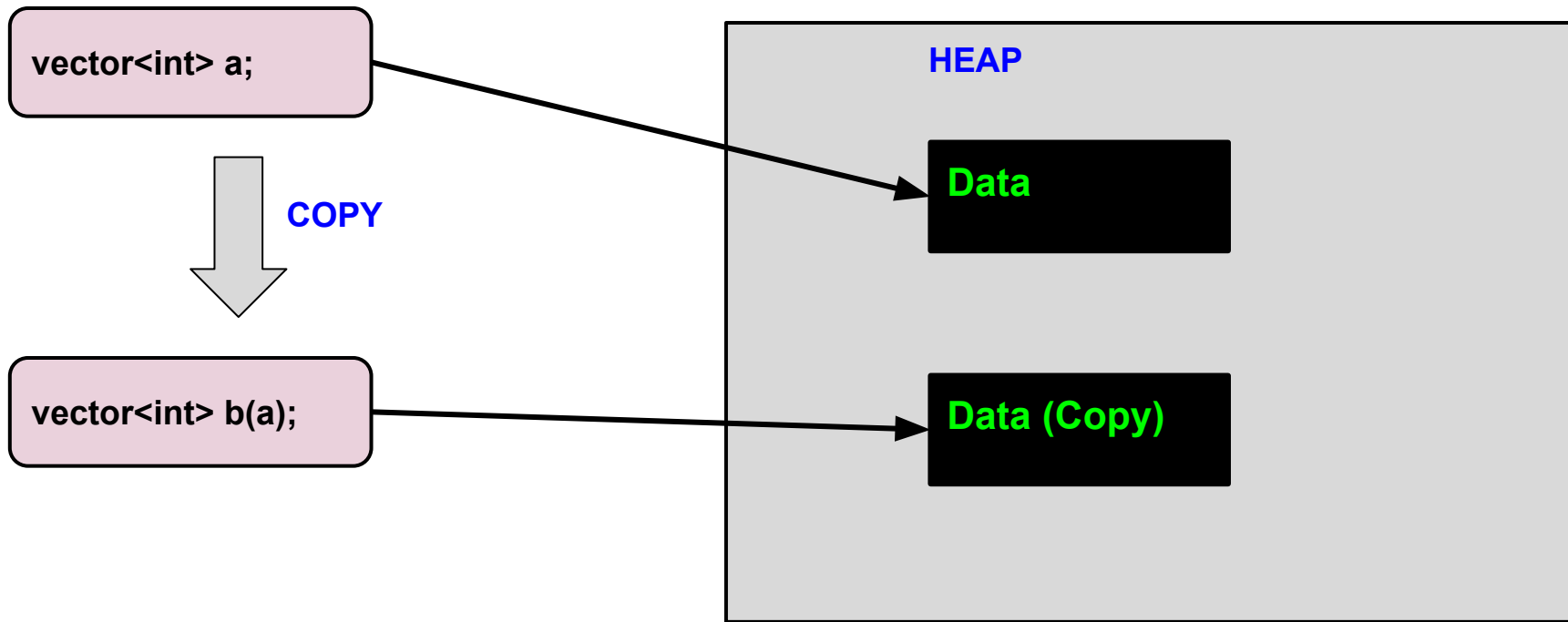
Unique object: Copying is forbidden



unique_ptr, istream, ostream

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

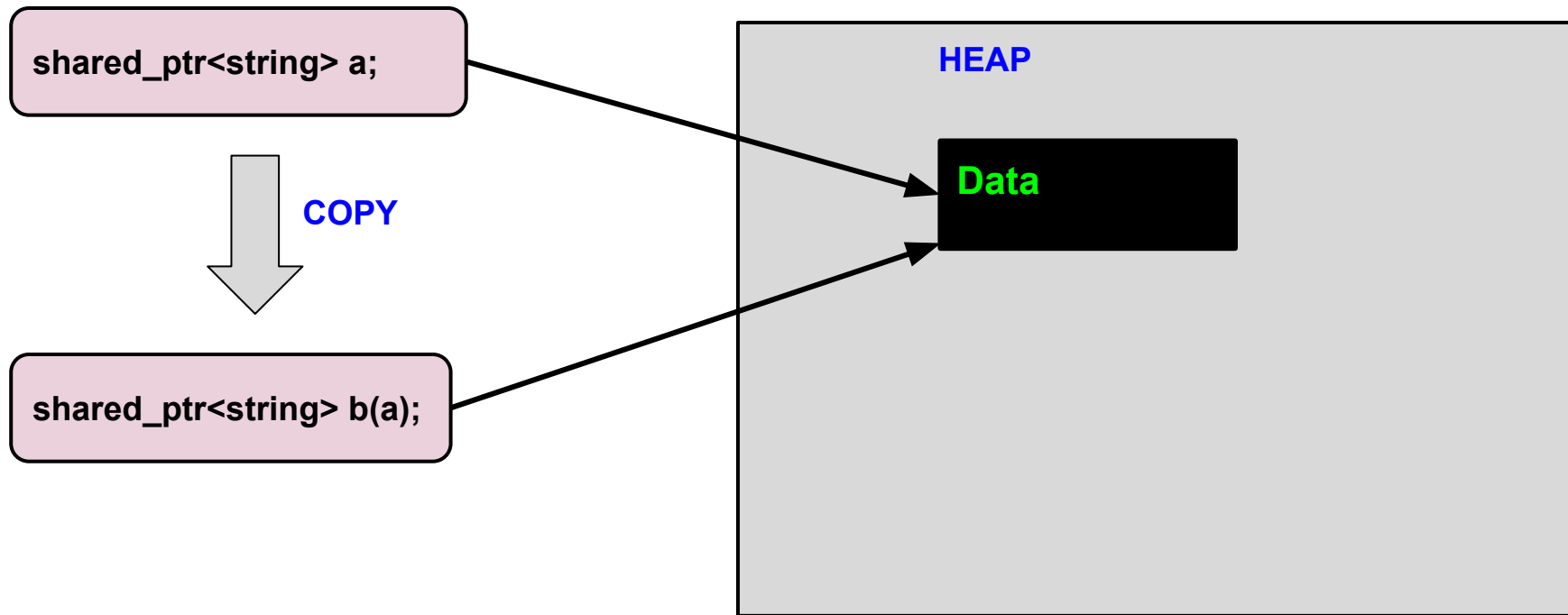
Value behavior: Copy resources



`std::vector` and most containers behave like this.

You can use container (e.g. **`std::vector`**) fields in your class for value behavior.

Pointer behavior: Do not copy resources



`shared_ptr` and **`cv::Mat`** (from OpenCV) behave like this. You have to count references !
Use **`shared_ptr`** fields in your class for pointer behavior !

Example 2: Implementing value behavior with raw pointers.

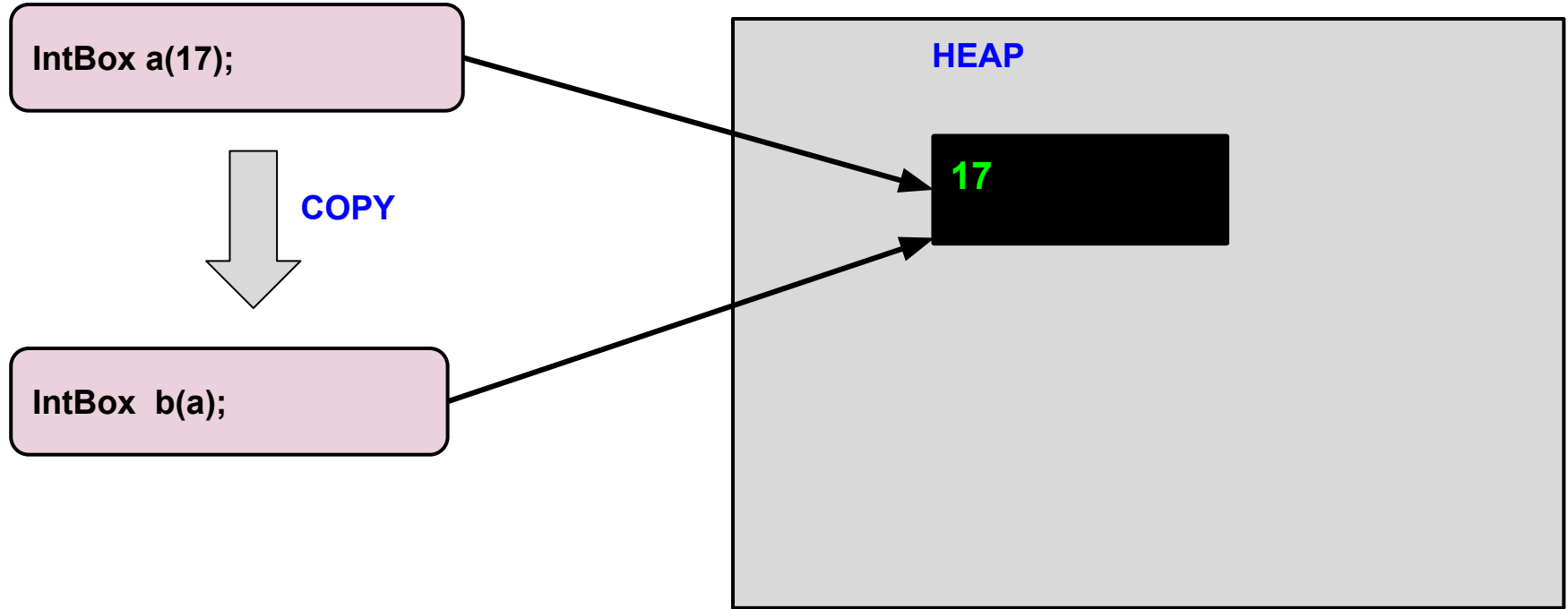
Let us design our own class: a trivial container for an **int** value.

But it uses heap memory. Can be empty (**nullptr**).

Let us manage heap with raw pointers (reinventing the wheel !).

```
class IntBox{
public:
    IntBox() { }    // Empty Ctor
    IntBox(int n) : data(new int(n)) { }    // Ctor, creates new heap object
    ~IntBox(){      // Dtor
        if (data)  // nullptr check
            delete data;
    }
    ...
private:
    int * data = nullptr;    // Pointer to data, nullptr if empty
}
```


Can we copy InBox using default copy ctor+assignment ?



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

clear() method makes IntBox empty

```
class IntBox{
```

```
public:
```

```
..
```

```
void clear(){
```

```
    if (data) { // nullptr check
```

```
        delete data; // Free the heap memory
```

```
        data = nullptr; // Don't forget to set data to 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
    } // Otherwise data stays nullptr
}
```

Move constructor, move heap block from rhs to *this :

```
IntBox(IntBox && rhs) {
    data = rhs.data;           // Copy the pointer from rhs to *this !
    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 ! Before any copy/move !  
        if (rhs.data) {           // If rhs is not empty  
            data = new int(*rhs.data); // Deep clone the heap object  
        } // Otherwise data stays nullptr  
    }  
    return *this; // As usual, return self by ref  
}
```

Self - assignment: **a = a**; We must always check for this stupid possibility!
Otherwise something will always go wrong.
For example, **clear()** will screw up things!

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;  
}
```

Extra slides: Implementing efficient swap()

C++ numerics : special algorithms

Sum all numbers in a container:

```
vector<double> v{1., 2., 4., 7., 10.};  
double res = accumulate(cbegin(v), cend(v), 0.);
```

Product of all numbers in a container: use lambda, start with 1:

```
accumulate(cbegin(v), cend(v), 1., [](double x, double y) {return x*y;});
```

Product of all numbers in a container: use `std::multiplies` (wrapper of *):

```
accumulate(cbegin(v), cend(v), 1., multiplies<int>());
```

Range : create vector {2., 3., .. , 11.}:

```
iota(v.begin(), v.end(), 2.);
```

Scalar product of two vectors:

```
inner_product(cbegin(v1), cend(v1), cbegin(v2), 0.)
```

Read yourself: more numerics, complex, cmath. But use Eigen for linear algebra !

Technology of the day : Profiling

Profiling = Run your code and examine CPU time usage.

!!! Profiling takes skills !!! It is easy to misunderstand the results !!!

Flat profile = CPU time used by each function + number of calls.

Call tree = All function calls (number+time) in each function.

Self time = Time spent by the function itself (excluding other function).

Inclusive (cumulative) time = Time spent by the function, including function it calls.

Profiling is typically done in Release:

cmake -DCMAKE_BUILD_TYPE=Release ..

But with debug symbols !

set(CMAKE_CXX_FLAGS_RELEASE "\${CMAKE_CXX_FLAGS_RELEASE} -g")

Pitfalls:

1. C++ name mangling (Encode classes, labdas, templates, namespaces etc.)
2. Threads
3. Calling functions from external .so libraries (e.g. OpenCV), gprof has problems with that

Using gprof (Built-in profiler for gcc, probably not the best)

1. Set compiler flags "-pg" in CMakeLists.txt:

```
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -pg")
```

2. Run the program (generates file gmon.out):

```
./demo
```

3. Run the profiler:

```
gprof ./demo gmon.out > analysis.txt
```

Result: flat profile and call tree (Note : C++ names hurt !):

Flat profile:

Each sample counts as 0.01 seconds.

| % | cumulative | self | | self | total | |
|-------|------------|---------|-------|---------|---------|---|
| time | seconds | seconds | calls | ms/call | ms/call | name |
| 55.56 | 1.30 | 1.30 | 4002 | 0.32 | 0.32 | std::_Function_handler<std::unique_ptr<std::__future_base::_Result_base, std::__future_base::_Result_base::_Deleter> (), std::__future_base::_Task_setter<std::unique_ptr<std::__future_base::_Result<unsigned long>, std::__future_base::_Result_base::_Deleter>, std::__future_base::_Task_state<linelib::mcmlsd::findMax(double*, unsigned long*, unsigned long, ctpl::thread_pool&)::{lambda(int)#1}, std::allocator<int>, unsigned long (int)>::_M_run(int&&)::{lambda()}#1}, unsigned long> >::_M_invoke(std::_Any_data const&) |

Using google profiler (gperftools)

1. Set compiler flags "-g" in CMakeLists.txt (no -p !):

```
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -g")
```

2. Run the program with the profiler, generate file main.prof :

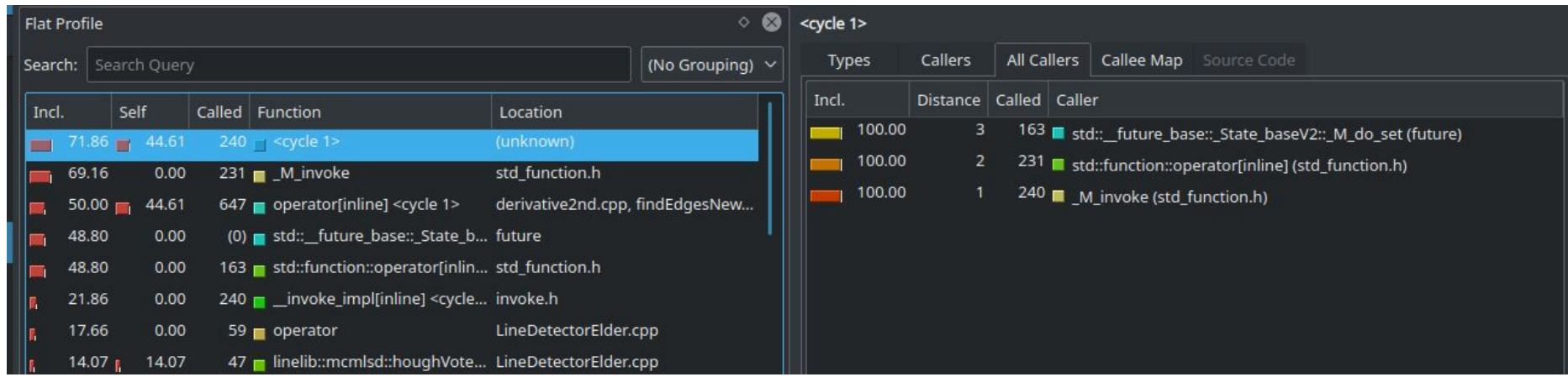
```
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libprofiler.so.0 CPUPROFILE=main.prof demo
```

3. Output the profile info in callgrind format (also available : text, PS etc.):

```
google-perfprof --callgrind ./demo main.prof > demo.callgrind
```

4. View the callgrind profile using KCacheGrind (nice interactive GUI):

```
kcachegrind demo.callgrind &
```



Using valgrind

Valgrind runs programs in a x64 emulator (20-30 times slower).

It has many tools:

Look for memory leaks (memcheck, the default tool):

```
valgrind demo 2>result.txt
```

CPU usage profiling (callgrind):

```
valgrind --tool=callgrind demo  
kcachegrind callgrind.out.12653
```

CPU cache profiling (cachegrind):

```
valgrind --tool=cachegrind demo  
kcachegrind cachegrind.out.13200
```

Other GUI tools : Valkyrie, Alleyoop for memcheck.

Thank you for your attention !

title

text

Implementing efficient swap()

```
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.