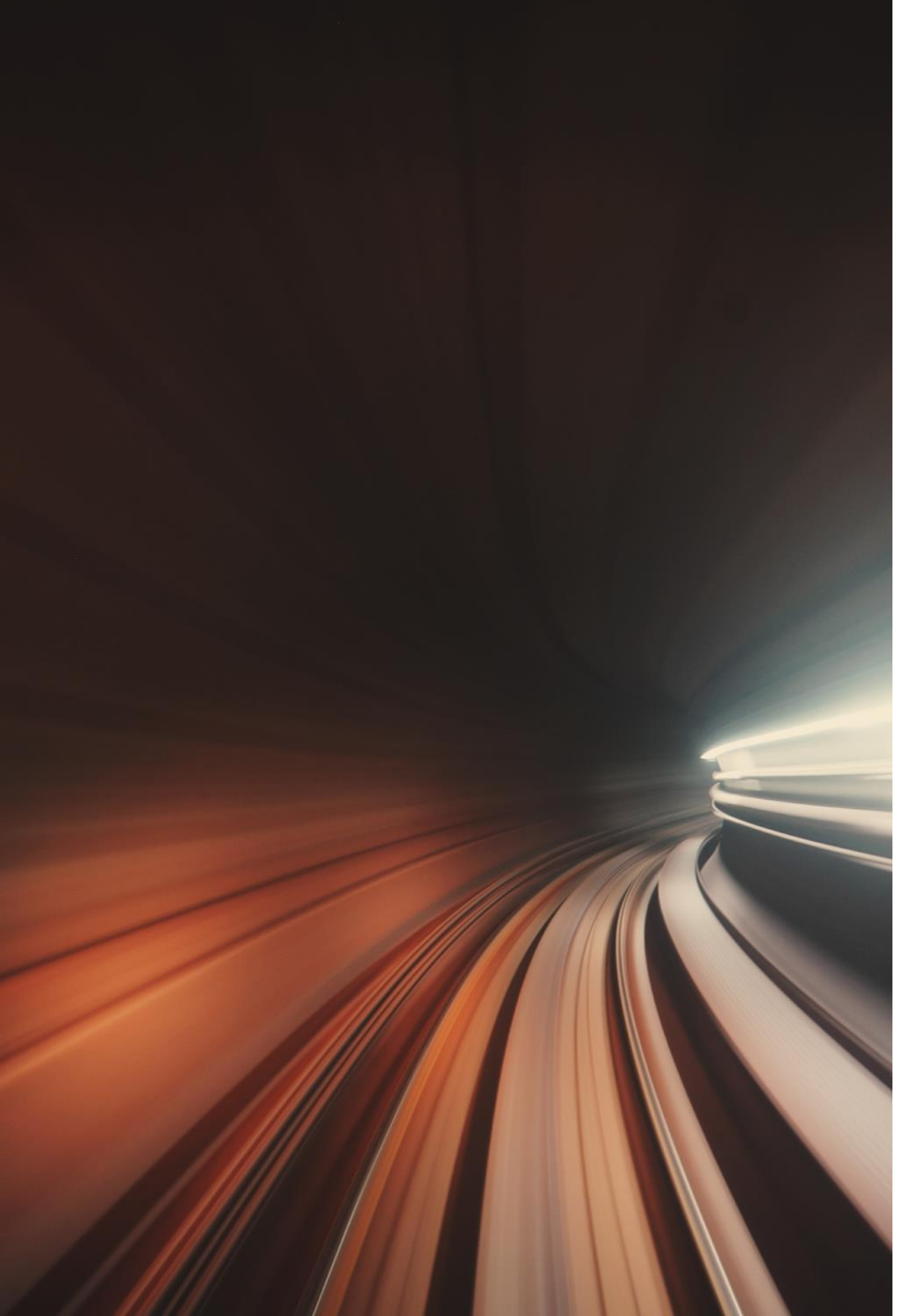
How to Think Like a

Modern C++ Programmer

Episode 2

TIM STRAUBINGER – CPSC 427 – SPRING 2021



Talk Outline

LIFETIMES

USING THE RIGHT TOOLS

MOVE SEMANTICS

Additional Resources

isocpp.org/get-started

- Recommended book list
- high-level explanations, tutorials, and design guidance

cppreference.com/w/

- Language and standard library documentation

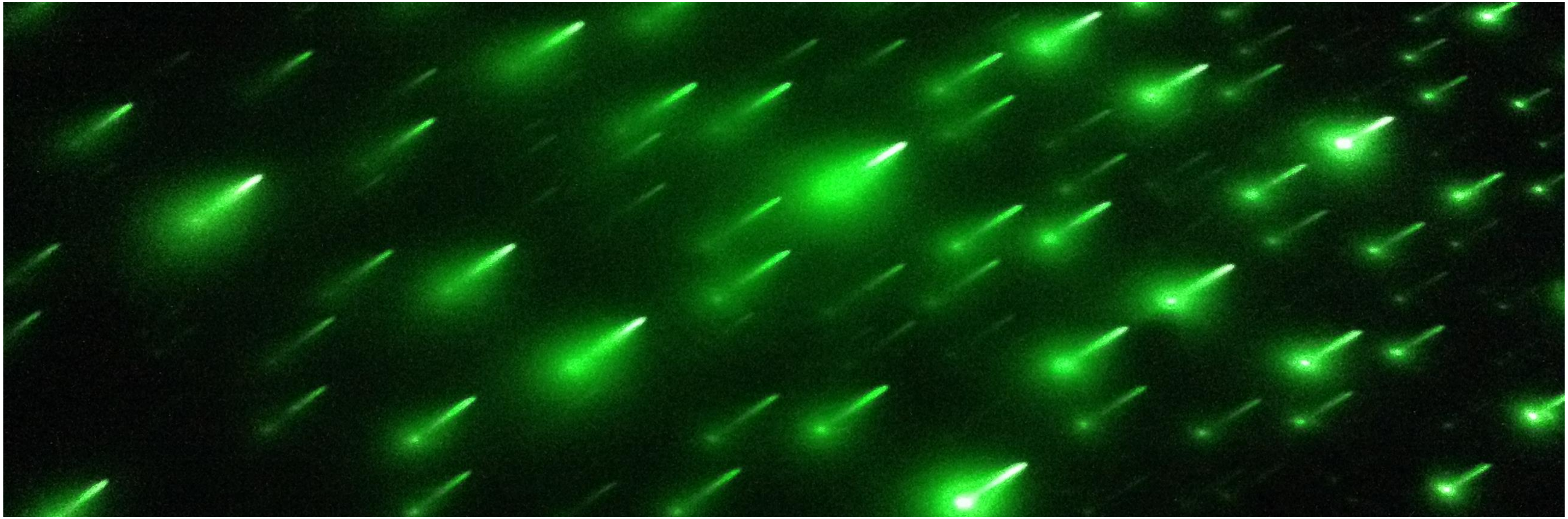
coliru.stacked-crooked.com

- Free online compiler (great for small exercises)

Lifetimes and Resource Management in C++

Lifetimes and Value Semantics

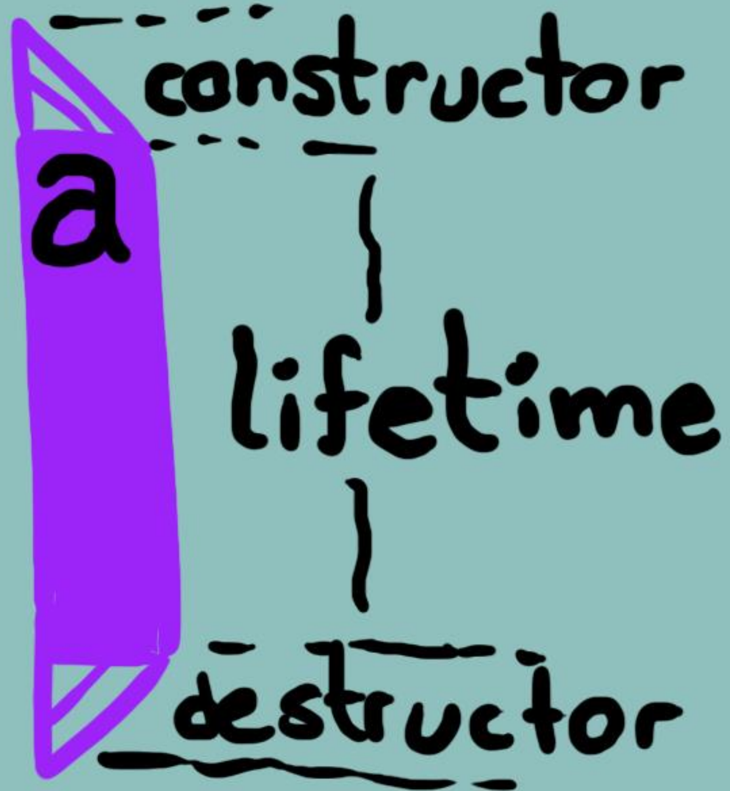
- One of C++'s **most important features**
- C++ **lets you decide** what happens when objects are **created**, **destroyed**, **copied**, and **moved**
- If used correctly, the C++ language will do the extra work for you
  - This results in **automatic**, **efficient**, and **deterministic** resource management
  - Far more powerful than garbage collection
  - Way easier than manual memory management
- Related concept: *RAII* (Resource Acquisition is Initialization)



Lifetimes Visualized

Source Code	Automatic Storage	Dynamic Storage

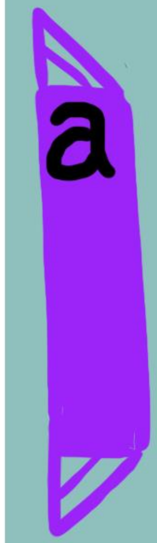
```
int main(){
  int a = 0;

  std::cout << a;

  return 0;
}
```



Automatic Storage Duration

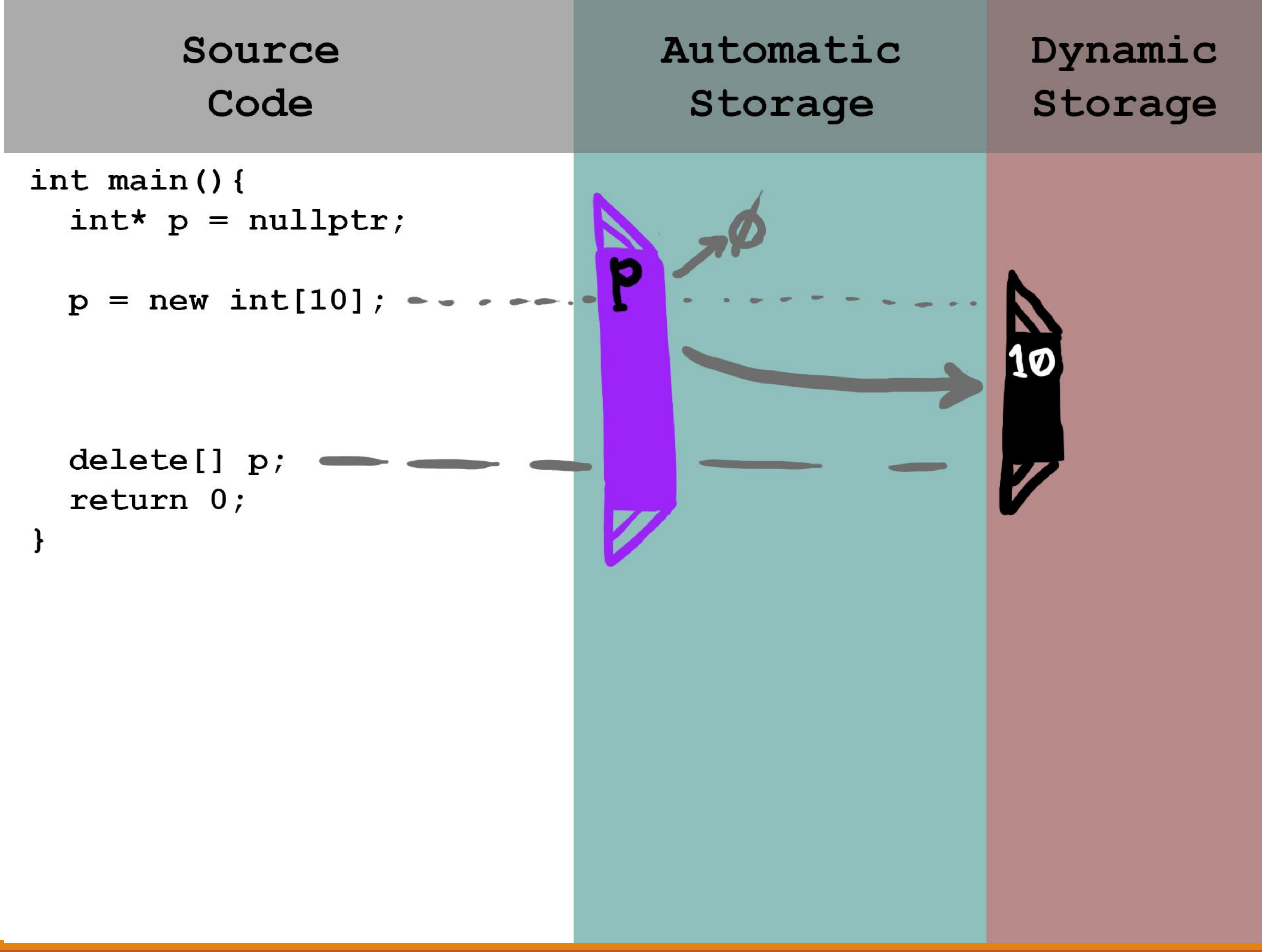
Source Code	Automatic Storage	Dynamic Storage

```
int main(){
  int a = 0;


  std::cout << a;


  return 0;
}
```



Dynamic Storage Duration

A heap-allocated object is a **resource** that needs **cleanup**



Types of Lifetimes

Any object in a running C++ program has one of three kinds of lifetimes, a.k.a. *storage durations*:

- Static storage duration
  - the object lives until the program exits
  - **Global variables** have static storage duration
- Dynamic storage duration
  - The start and end of life are not known until runtime
  - **Heap-allocated objects** have dynamic storage duration (think of `new` or `malloc` and garbage collection)
- **Automatic storage**
  - The **most underrated** type of lifetime!
  - The object lives until it goes out of scope
  - **Local variables**, **function arguments**, and class **member variables** have automatic storage duration

Thinking about resource management

A **resource** is something that **needs additional work to clean up** when you're done using it

Examples of resources:

- Data structures that grow over time (dynamic arrays, trees, linked lists, etc)
- Opened files (operating systems want these back eventually)
- Most hardware devices (things like "connections" and "contexts" and "handles")

The part of code that is **responsible for cleaning up** a resource is called the ***owner***

- This part of code **has *ownership*** of that resource

Resource Management in Modern C++

In modern C++, *Lifetimes* and *Ownership* are **combined**

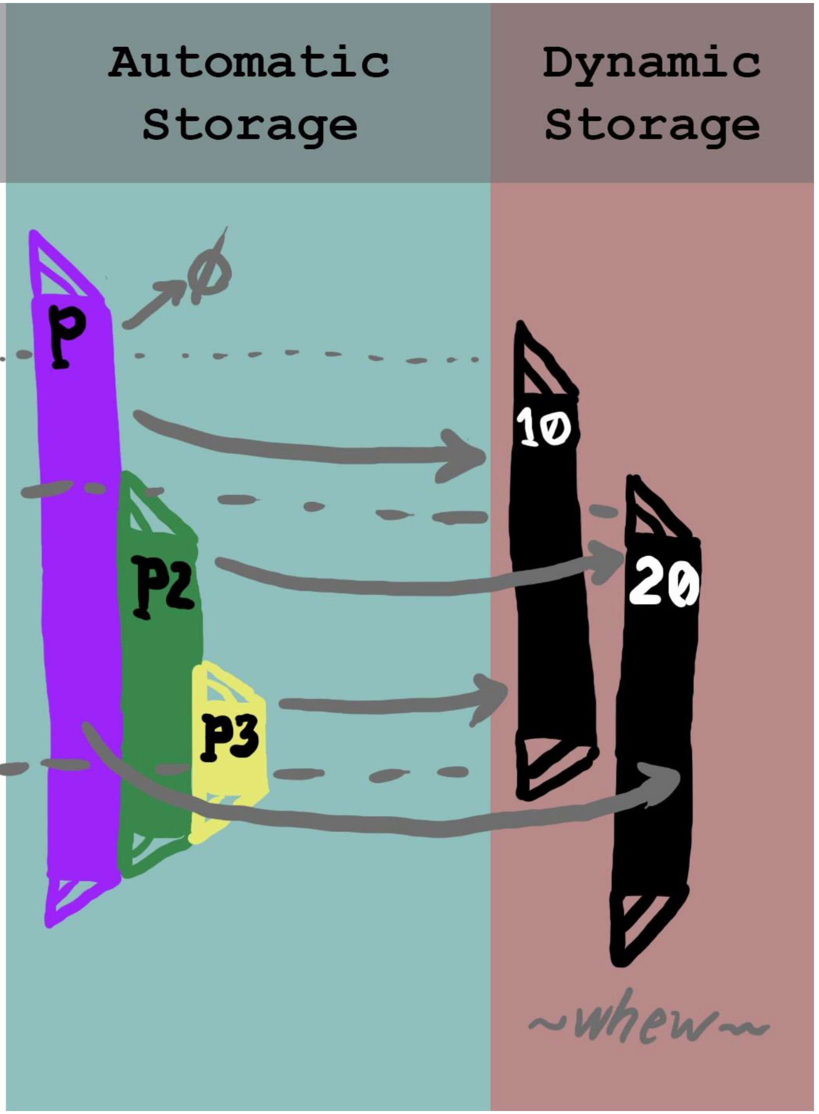
This allows **automatic, implicit, and efficient resource management**

Option A

"C++ beginner following a 20-year-old tutorial"

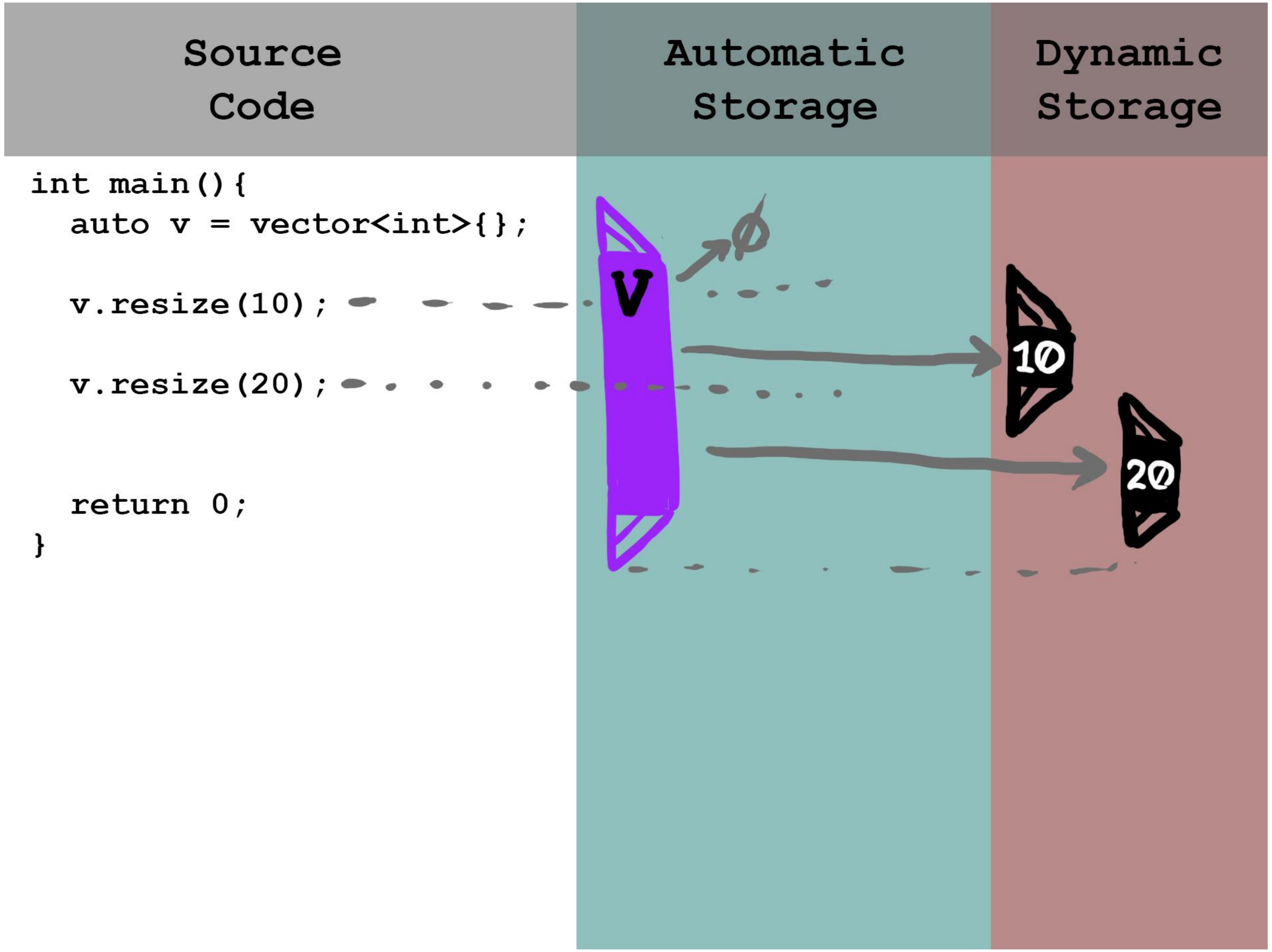
Source Code

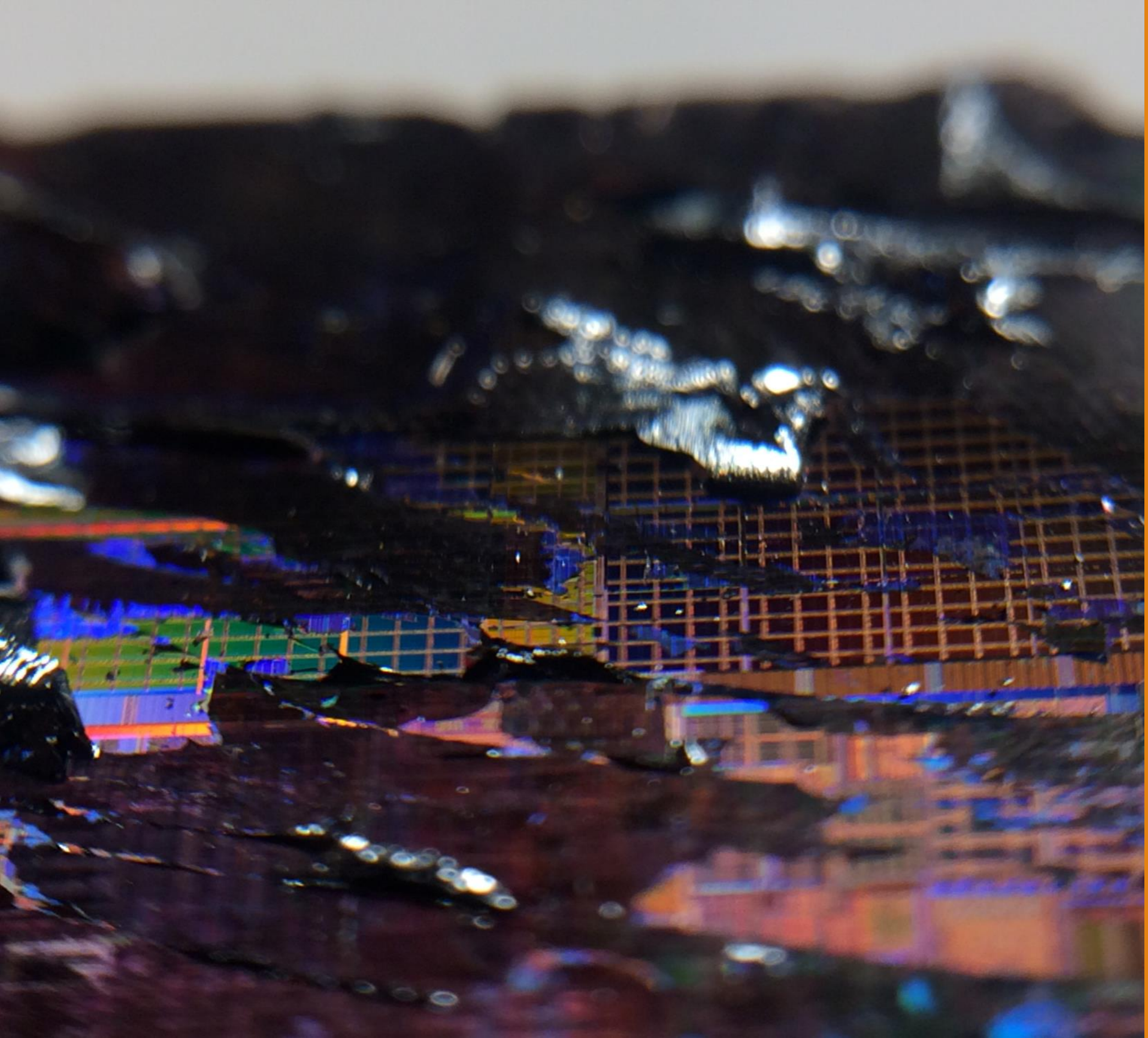
```
int main(){
  int* p = nullptr;

  p = new int[10];


  int* p2 = new int[20];
  for (int i = 0; i < 10; ++i){
    p2[i] = p[i];
  }
  int* p3 = p;
  p = p2;
  delete[] p3;


  delete[] p;
  return 0;
}
```



Option B

“Modern C++ programmer”





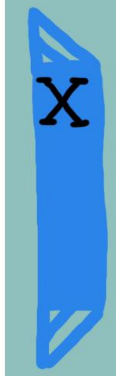
Special Member Functions

Constructor and Destructor

- An object's **lifetime begins with a constructor**
- An object's **lifetime ends with the destructor**
- A constructor should guarantee that an object is **always** in a valid state
  - Constructors often **acquire a resource**
- A destructor should **clean up everything that the object is responsible for**
  - Destructors often **release a resource**
- Constructors and destructors are called **implicitly** as part of the language
  - **Use this to your advantage!**

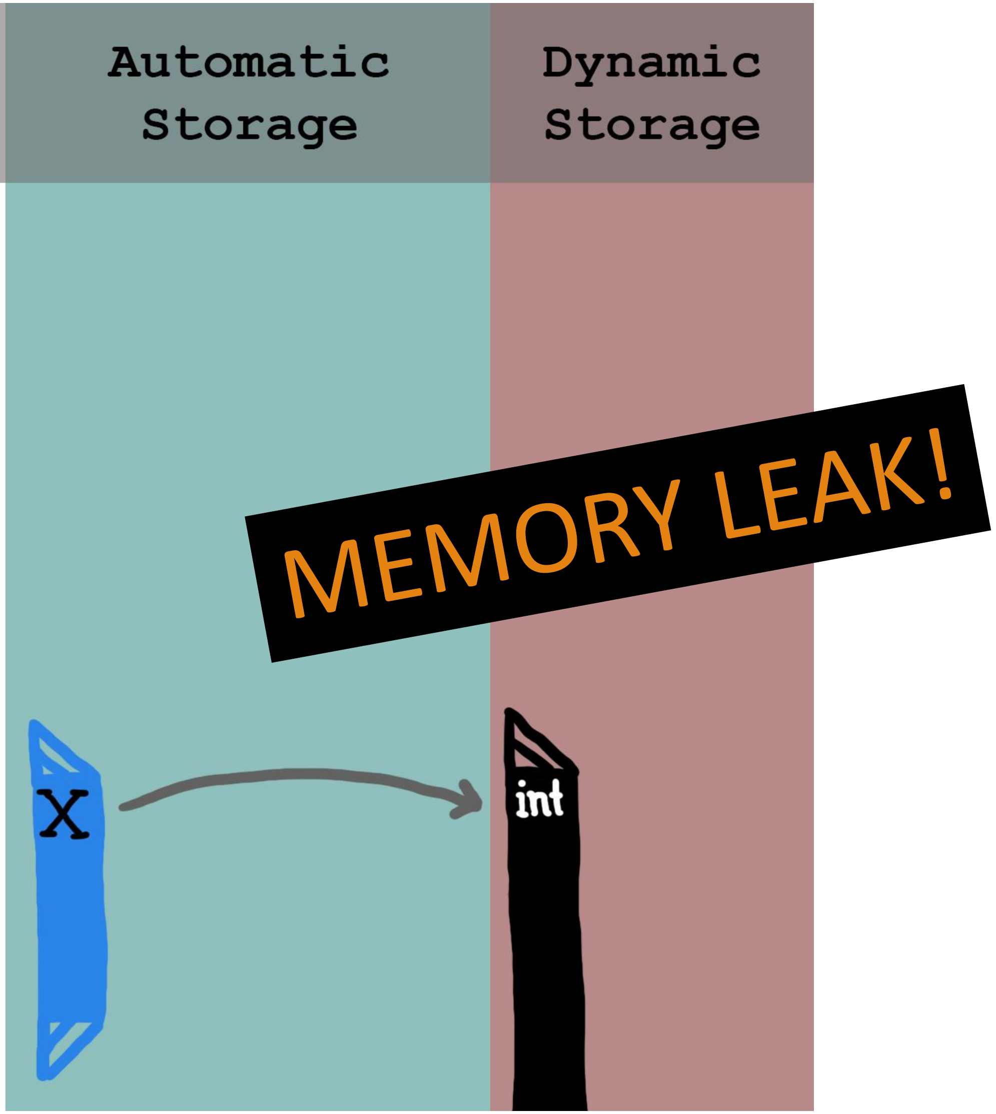
Source Code	Automatic Storage	Dynamic Storage

```
struct A {


};

int main(){
  auto x = A{};



  return 0;
}
```



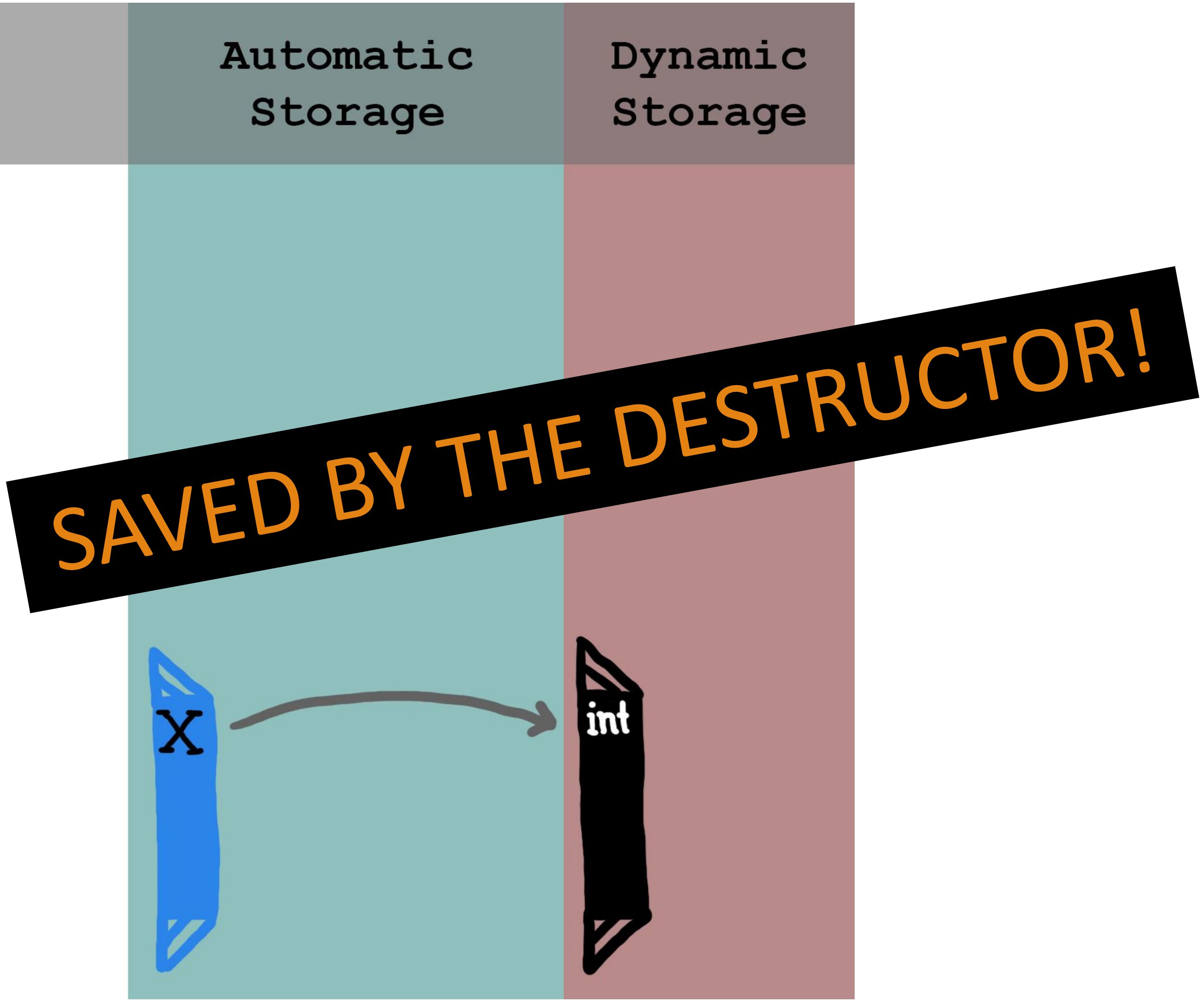
Source Code

```
struct A {
  A():p{new int(32)} { }

private:
  int* p;
};

int main(){
  auto x = A{};

  return 0;
}
```



Source Code

```
struct A {
  A():p{new int(32)} { }
  ~A(){ delete p; }



private:
  int* p;
};

int main(){
  auto x = A{};



  return 0;
}
```

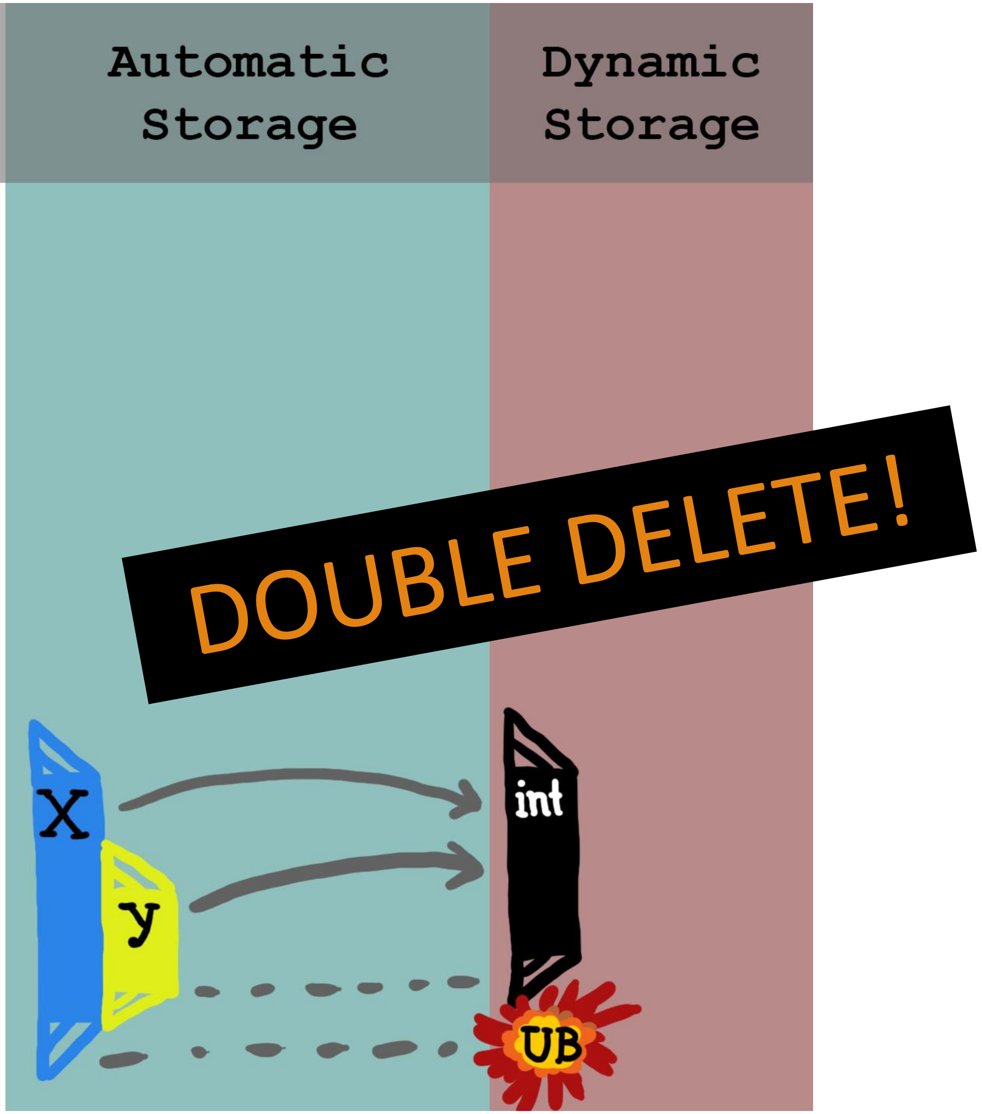


Copy Semantics – Construction and Assignment

- Let you define what it means to duplicate object (without modifying the original)
- **Copy constructor** is called when a **new object is cloned** from another object
- **Copy assignment operator** is called when an object's value is **overwritten** from another object
- Can be enabled or disabled (sometimes it doesn't make sense to create a copy)
  - Example: copying a `std::vector` copies all elements
  - Example: `std::fstream` (file handle) can't be copied
- Called **implicitly** as part of language
  - Use this to your advantage!

Source Code

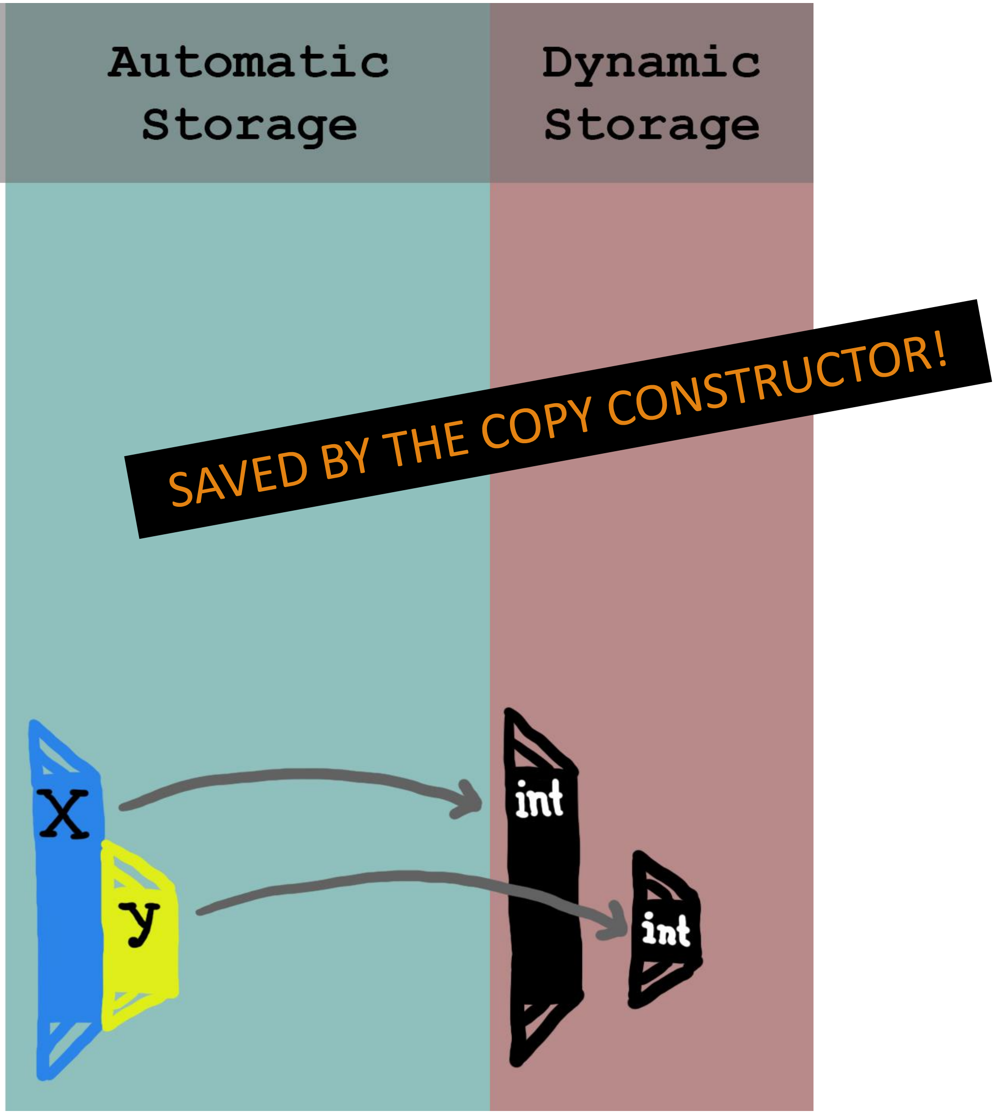
```
struct A {
  A():p{new int(32)} { }
  ~A(){ delete p; }




private:
  int* p;
};

int main(){
  auto x = A{};

  auto y = x;



  return 0;
}
```



Source Code

```
struct A {
  A():p{new int(32)} { }
  ~A(){ delete p; }
  A(const A& a):p{new int(32)} { }
  A& operator=(const A&){
    *p = *a.p;
    return *this;
  }


private:
  int* p;
};

int main(){
  auto x = A{};

  auto y = x;



  return 0;
}
```

`const A&` will bind to any kind of `A` value



Source Code	Automatic Storage	Dynamic Storage

```
struct A {
  A():p{new int(32)} { }
  ~A(){ delete p; }
  A(const A&) = delete;
  A& operator=(const A&) = delete;

private:
  int* p;
};

int main(){
  auto x = A{};

  auto y = x; // ERROR

  return 0;
}
```

Special member functions can be disabled using `= delete;`

Move Semantics – Construction and Assignment

- Used for transferring **ownership of a resource** (by modifying the previous owner)
- **Move constructor** creates a new object that **takes ownership** from another object
- **Move assignment operator** lets an existing object **take ownership** from another object
- Useful **only when making a copy is expensive or impossible**
- **Not needed when there is no cleanup work to be done**
  - In this case, copying is the same thing
- Can also be enabled or disabled

`const A&` will bind to any kind of `A` value

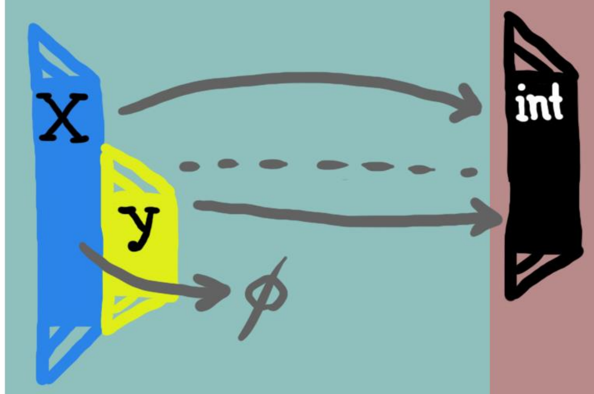
`A&&` binds to **temporary values**!

NOTE: In everyday code, this is the **only place** that T&& is needed

Source Code	Automatic Storage	Dynamic Storage

```
struct A {
  A():p{new int(32)} { }
  ~A(){ delete p; }
  A(const A& a) = delete;
  A& operator=(const A&) = delete;
  A(A&& a):p{exchange(a.p, nullptr)}{ }
  A& operator=(A&& a){
    delete p;
    p = exchange(a.p, nullptr);
    return *this;
  }
private:
  int* p;
};
```

```
int main(){
  auto x = A{};

  auto y = std::move(x);

  return 0;
}
```



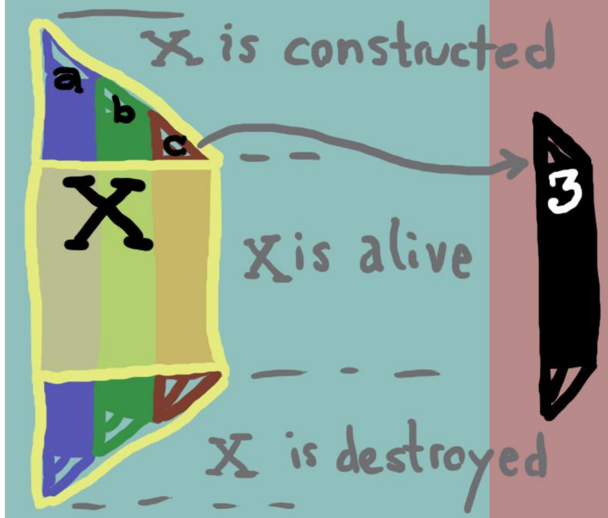
Constructor, Destructor, Copy Constructor, Move Constructor, Copy Assignment Operator, Move Assignment Operator, Oh My!

That’s a lot of functions to think about!
How can I wrap my head around writing these?

- *Most* of the time, **you don’t have to write these**
- Why? **Your C++ compiler generates them for you** if you don’t
- The implicitly generated special member functions will do the “obvious” thing
  - The generated default constructor will default-construct all member variables
  - The generated copy functions copy all member variables
  - The generated move functions move all members (but are disabled if you write copy functions)
- *Most* of the time, you only need to write constructors
- **But:** you **need** to write these when you are **directly managing a resource**

Source Code	Automatic Storage	Dynamic Storage

```
struct X {
  int a = 3;
  string b = "Hello";
  vector<int> c = {1, 2, 3};
};

int main(){
  auto x = X{};




  return 0;
}
```



Rule of 3/5/0

- If your class **explicitly defines a destructor**, then you're **probably managing a resource** (otherwise, you would have no cleanup work to do)
- …because you're probably managing a resource, you should **also define copy semantics**
  - …to prevent the default copy functions from doing something you don't intend (Rule of Three)
- …and if it makes sense for your resource, you should **also define move semantics**
  - …to allow relocating objects and transferring ownership (Rule of Five)
- If your special member functions do nothing special, get rid of them (they can be generated)
  - (Rule of Zero)

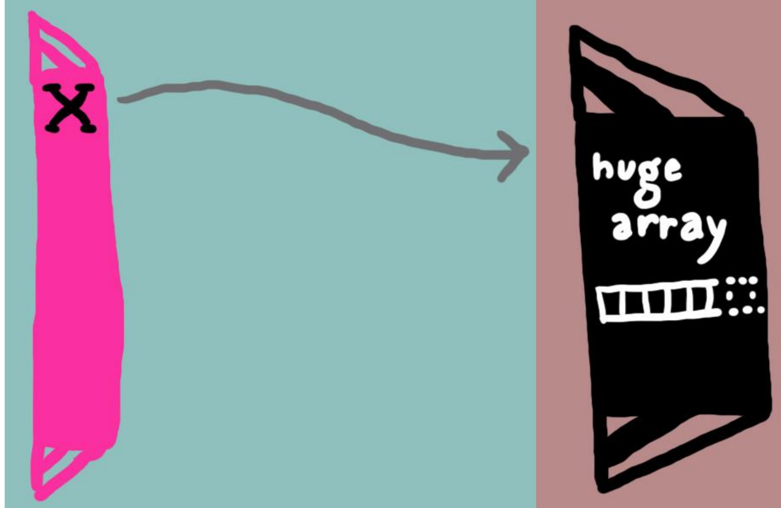
https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)

Automatic Storage with **Standard Library Containers**

std::vector<T>
is a resizable heap-allocated array

Source Code	Automatic Storage	Dynamic Storage

```
int main(){
  vector<int> x = makeHugeVector();


  return 0;
}
```

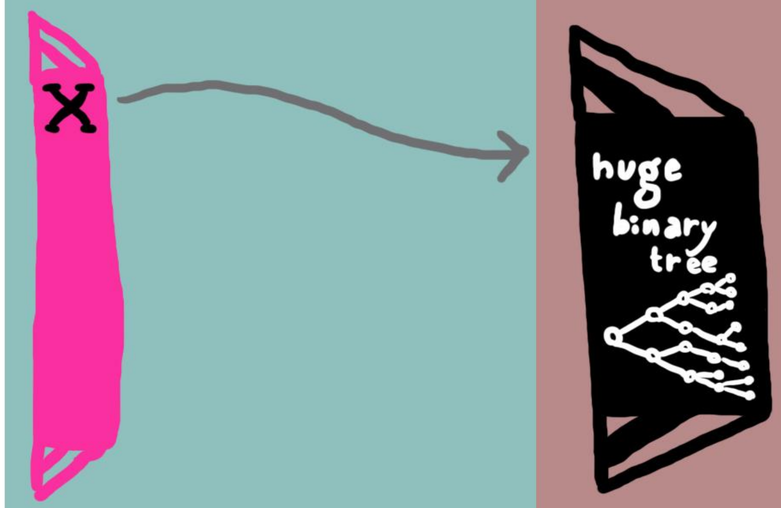


Automatic Storage with **Standard Library Containers**

`std::set<T>`
is a binary tree

Source Code	Automatic Storage	Dynamic Storage

```
int main(){
   set<int> x = makeHugeSet();




   return 0;
}
```

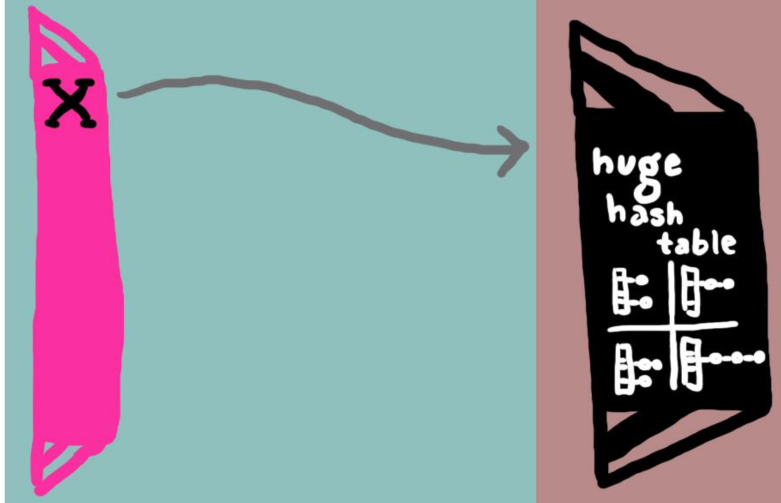


Automatic Storage with **Standard Library Containers**

`std::unordered_map<T>` is a hash table

Source Code	Automatic Storage	Dynamic Storage

```
int main(){
    unordered_map<int> x = makeHugeUnorderedMap();



    return 0;
}
```



Get to know your tools!

- Using the **Rule of 0** and **compiler-generated special member functions**, you can write **highly efficient, *correct*** code by reusing the following **standard library tools**:
  - `vector<T>` for dynamic arrays
  - `set<T>` and `map<T>` for binary trees
  - `unordered_set<T>` and `unordered_map<T>` for hash tables and hash maps
  - `optional<T>` for values that might not exist
  - `variant<T1, T2, ...>` for values from one of several different types
  - `unique_ptr<T>` for safely managing a heap object
  - `shared_ptr<T>` for safely managing a heap object with multiple owners
  - And **many, many more!** Consult your C++ book and documentation for ideas and guidance

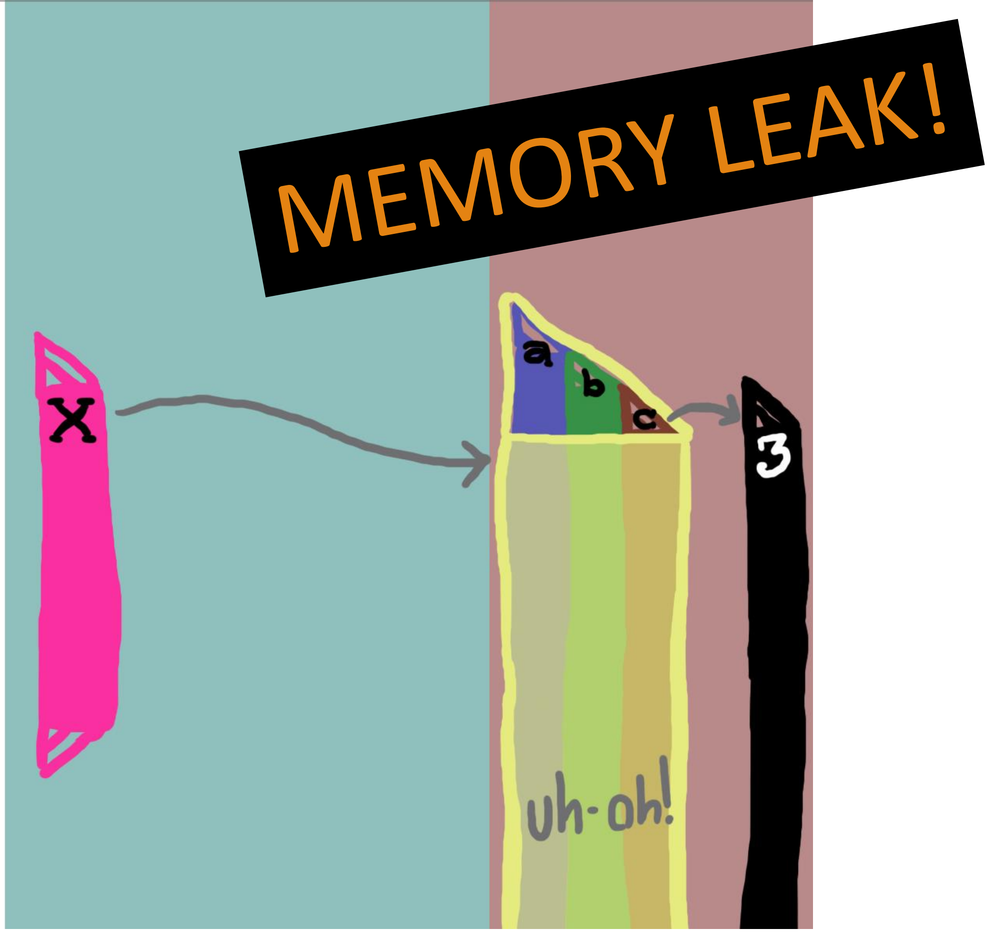
“I need heap allocation for some reason”

Using

Manual Memory Management

Source Code	Automatic Storage	Dynamic Storage

```
struct X {
  int a = 3;
  string b = “Hello”;
  vector<int> c = {1, 2, 3};
};

int main(){
  X* x = new X{};




  return 0;
}
```



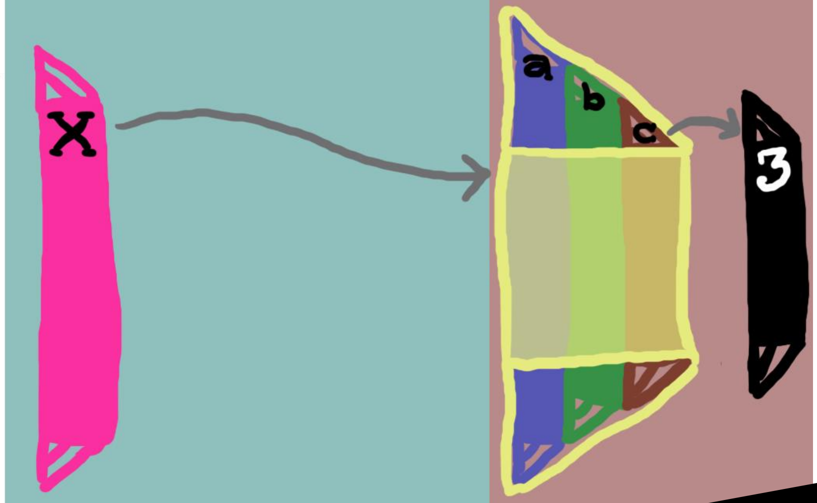
“I need heap allocation for some reason”

Using

unique_ptr

Source Code	Automatic Storage	Dynamic Storage

```
struct X {
  int a = 3;
  string b = “Hello”;
  vector<int> c = {1, 2, 3};
};

int main(){
 unique_ptr<X> x = make_unique<X>();



  return 0;
}
```



No manual cleanup!

NICE!

In conclusion:

- Understand special member functions
- Use copy and move semantics to your advantage
- Use automatic storage to do your cleanup for you

Hang on, what does `std::move` do?

BONUS TECHNICAL DETAILS

```cpp
void observe(const std::vector<BlahBlah>& v) {
    std::cout << v.size();
}

void modify(std::vector<BlahBlah>& v) {
    v.pop_back();
}

std::vector<BlahBlah> consume(std::vector<BlahBlah> v) {
    v.pop_back();
    return v;
}
```

```cpp
int main() {
    // local variable
    auto v = std::vector<BlahBlah>(99);

    observe(v); // no copy (pass by reference-to-const)
    modify(v); // no copy (pass by reference)
    consume(v); // copy
    consume(v); // copy (totally safe)
}
```

```cpp
int main() {
    observe(std::vector<BlahBlah>(99)); // temporary safely binds to reference-to-const

    // modify(std::vector<BlahBlah>(99)); ERROR!
    //     Temporary can't bind to non-const reference

    consume(std::vector<BlahBlah>(99)); // direct construction (efficient)
}
```

```cpp
std::vector<BlahBlah> makeMeAVector() {
    return std::vector<BlahBlah>(99);
}

int main() {
    observe(makeMeAVector()); // temporary safely binds to reference-to-const

    // modify(makeMeAVector()); ERROR!
    //     Temporary can't bind to non-const reference

    consume(makeMeAVector()); // direct construction (efficient)
}
```

```cpp
int main() {
    // local variable
    auto v = std::vector<BlahBlah>(99);

    observe(v); // no copy (pass by reference-to-const)
    modify(v); // no copy (pass by reference)
    consume(v); // copy (but what if I don't want v anymore???)
}
```

```cpp
int main() {
    // local variable
    auto v = std::vector<BlahBlah>(99);

    observe(v); // no copy (pass by reference-to-const)
    modify(v); // no copy (pass by reference)

    consume(std::move(v)); // MOVE
    // std::move converts `vector<BlahBlah>&` to `vector<BlahBlah>&&`
    // This results in vector<BlahBlah>'s move constructor being
    // chosen, which does the rest of the work

    // NOTE: `v` is now in a "valid but unspecified state"
    // For best safety, `v` should no longer be used for anything
}
```