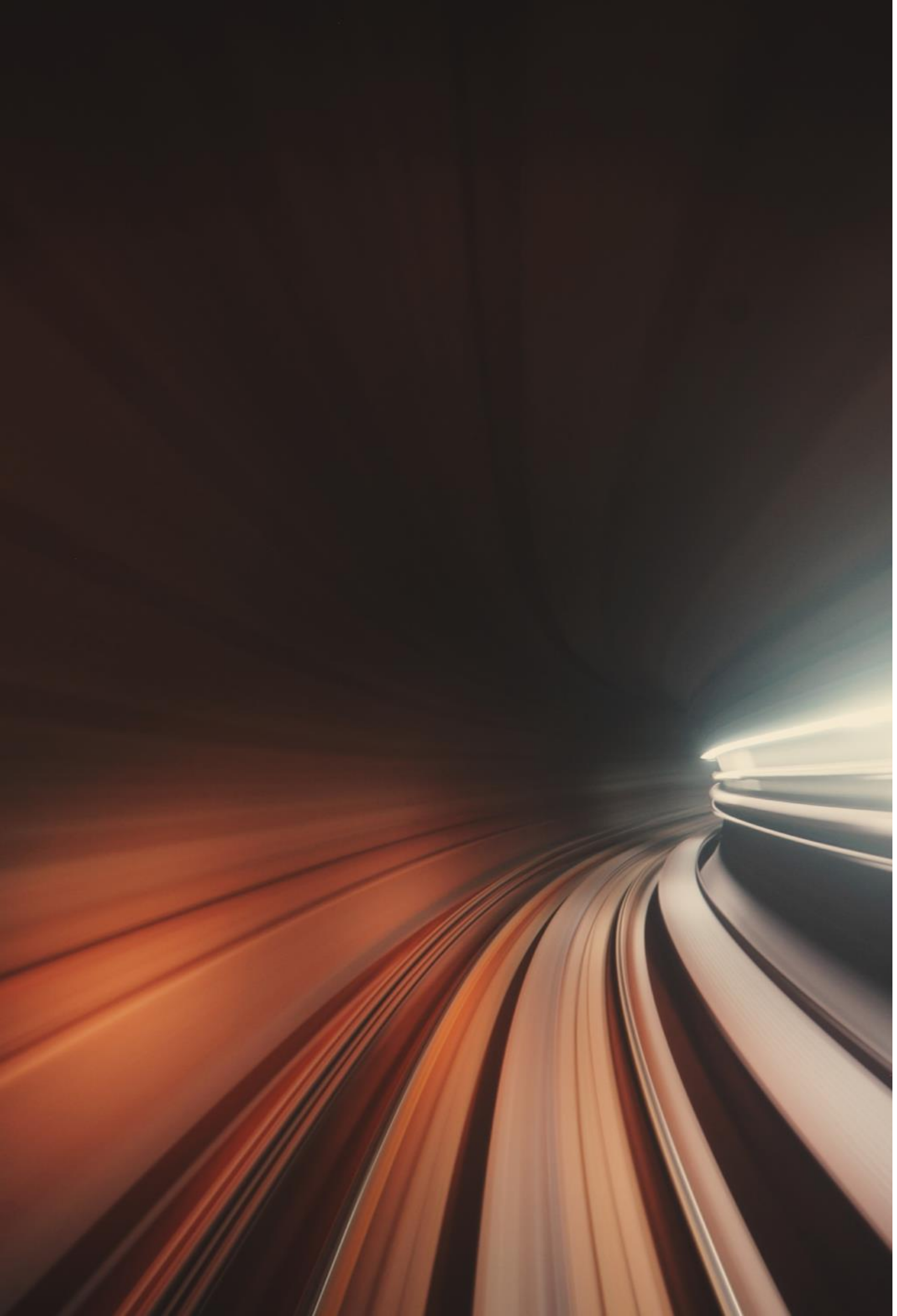
How to Think Like a Modern C++ Programmer

TIM STRAUBINGER – CPSC 427 – SPRING 2021

Poll Time

How you are doing today?

- I am alive
- Doing well
- Doing great
- I am not ready to be in classes again
- I wish I was in bed
- I am in bed



Talk Outline

- BRIEF HISTORY OF C++
- TEMPLATES
- THE DARK SIDE OF C++
- LIFETIMES

Who is Tim (a.k.a `timstr`)?

- MSc. Student studying under Helge Rhodin and Robert Xiao
- timstr@cs.ubc.ca
- https://timstr.github.io
- Began learning C++ in early 2012
- “understood” C++ circa mid-2018
- Two years of professional experience with C++
- Around 3000-5000 total hours spent with C++
- Still learning new things about C++ 9 years later

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)

Poll Time

What does C++ make you think of?

- 👏 I don't remember CPSC 221
- 👍 `new/delete` instead of `malloc/free`
- ❤️ I love C++!
- 😂 Memory leaks and dangling pointers
- 😮 Templates!
- 🎉 Please, I just want to graduate

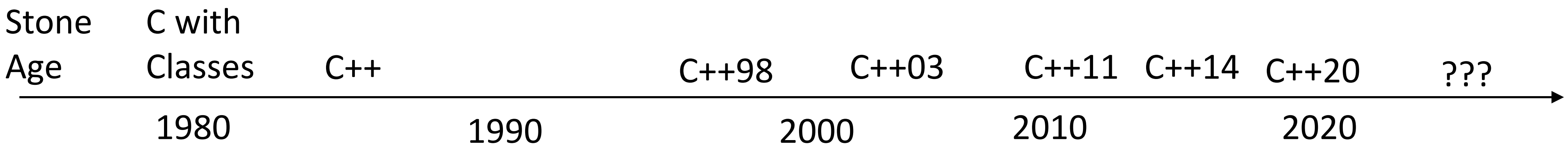
A Brief Tour of C++

C++ **began** being invented in 1979 by Danish computer scientist

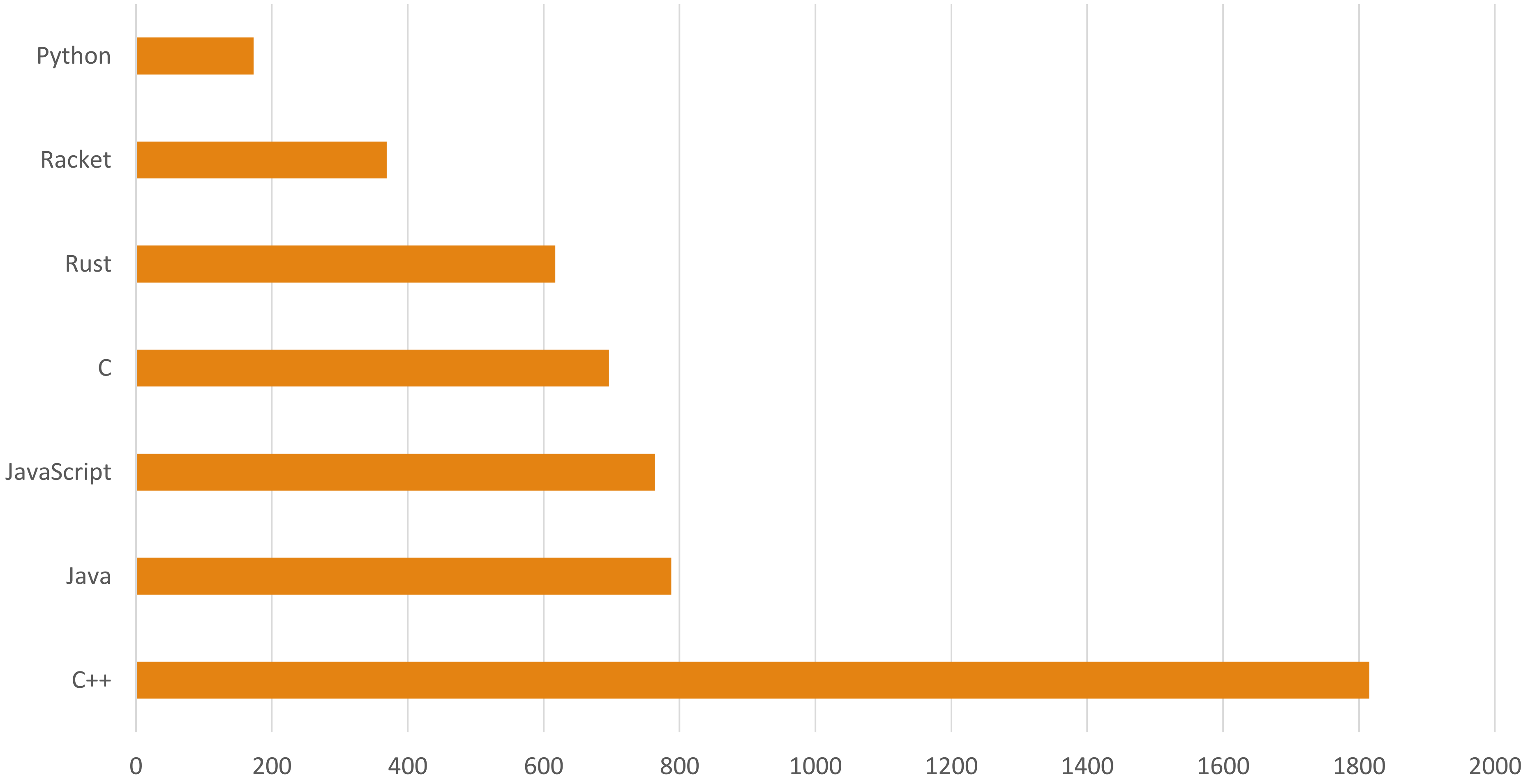
Bjarne Stroustrup

(pictured right)

C++ is Not Done Being Invented



Length of Language Specification (Number of Pages)



Why do C++ programmers like C++?

- Runtime performance
  - Zero-cost abstractions
  - Compiler optimization
  - Easy and efficient resource management
  - Compile-time programming (for advanced users)
- Type Safety
  - Many * potential bugs are eliminated at compile time
- Expressiveness
  - Many diverse tools are provided by C++
  - Many styles of programming are possible
    - generic, object-oriented, functional, imperative
      compile-time, template meta-programming, etc

* but not all

Why **don't** C++ programmers like C++?

- Undefined Behaviour
  - C++ gives you the freedom to hurt yourself
- Complexity
  - The C++ language is **huge**
  - C++ programmers readily over-engineer
  - Reasoning about C++ can cause headaches
- Compilation speed
  - Being a C++ compiler is not easy

this has been

A Brief Tour of C++

thank you for watching

C++ Templates

Avoiding Manual Code Duplication

DOES THIS CODE LOOK FAMILIAR TO YOU?

```
int add_int(int x, int y) {
    int result = x + y;
    return result;
}

double add_double(double x, double y) {
    double result = x + y;
    return result;
}

std::string add_string(std::string x, std::s
    std::string result = x + y;
    return result;
}

float add_float(float x, float y) {
    float result = x + y;
    return result;
}
```

Templates to the rescue!

AUTOMATED CODE DUPLICATION!

```cpp
template<typename T>
T add(T x, T y) {
    T result = x + y;
    return result;
}
```

Templates in C++

C++ is **statically typed**, and **all types must be known at compile time**

So how do templates work in C++?

- **Automated code duplication**! (*technically called monomorphisation)*

  Each time you provide a template function/class with a different type, **a different function/class is generated by the compiler!**

  - This enables **type checking**
    - The compiler can inspect types and perform all the normal safety checks
  - This enables **optimization**
    - The compiler can generate faster code that is specific to each type
  - This enable **expressive tools**
    - Templates are *extremely* powerful at doing many different things

member access in Python

Nearly everything is checked at **runtime!**

Lots of testing required ☹

```python
def foo(x):
    print(x.bar)

class A:
    def __init__(self):
        self.bar = "Blab"

a = A()
b = 99
foo(a)
foo(b)
```

```
Blab
Traceback (most recent call last):
  File "blah.py", line 12, in <module>
    foo(b)
  File "blah.py", line 3, in foo
    print(x.bar)
AttributeError: 'int' object has no attribute 'bar'
```

member access in C++ templates

Templates are checked **at compile time!**

```
template<typename T>
void foo(T t){
    std::cout << t.bar << std::endl;
}

struct A {
    std::string bar = "Blab";
};

int main(){
    auto a = A{};
    auto b = 99;
    foo(a);
    // foo(b); ERROR: request for member 'bar' in 't',
    //                which is of non-class type 'int'

    return 0;
}
```

Generic functions in Java

Only one function is generated!

Types are erased ☹

Simple things are impossible ☹

```java
public static <T> T create() {
    T t = new T();
    return t;
}
```

```
Main.java:11: error: unexpected type
        T t = new T();
                  ^
  required: class
  found:    type parameter T
  where T is a type-variable:
    T extends Object declared in method <T>create()
```

Template functions in C++

Types can be provided explicitly for **great good**

The ECS system uses this extensively.
Take a look ☺

```
template<typename T>
T create(){
	auto t = T{};
	return t;
}


int main(){
	auto i = create<int>();
	auto d = create<double>();
	auto s = create<std::string>();

	return 0;
}
```

In conclusion:

- C++ templates allow code reuse with multiple types
- C++ templates are type-checked at compile time
- C++ templates are efficient and powerful

The Dark Side of C++

Undefined Behaviour

C++ is **not safe**

- C++ lets you break the rules of language
- When you break the rules, ***anything*** can happen

- A good C++ programmer knows **how not to break the rules**

But what are these "rules?"

- As you read your C++ book or documentation, look out for the term "Undefined Behaviour"
  - The are many, **many** ways to invoke Undefined Behaviour
  - Any situation causing Undefined Behaviour is a situation that **you need to prevent!**

Definition of Undefined Behaviour

- "**Renders the entire program meaningless** if certain rules of the language are violated." [1]
- "There are **no restrictions on the behavior of the program**" [1]
- "**Compilers are not required to diagnose undefined behavior** […], and the compiled program is **not required to do anything meaningful**." [1]
- "Because **correct C++ programs are free of undefined behavior**, compilers may produce unexpected results when a program that actually has UB is compiled with optimization enabled" [1]
- If a program encounters UB when given a set of inputs, there are no requirements on its behavior "**not even with regard to operations preceding the first undefined operation**" [2]

[1] https://en.cppreference.com/w/cpp/language/ub

[2] C++20 Working Draft, Section 4.1.1.5

Undefined Behaviour in Simpler Terms

If you do something wrong, **literally anything** can happen when your code runs.

This includes:

- Your code runs and does nothing 🤔
- Your code runs as you expect it to 🙂
- Your code crashes with a helpful error message 😢
- Your code crashes for no explainable reason 😖
- Your code runs and does something just … *weird* 😵
- Your code runs as you expect it to, but fails later at the worst possible moment 🤮
- Your code passes all tests, but hackers can steal your passwords 😵
- Demons come flying out of your nose

```cpp
#include <iostream>

int main() {
    std::cout << "Start ---" << std::endl;
    char ch; // Oops! Forgot to initialize :-)
    std::cout << ch << std::endl;
    std::cout << "Finish ---" << std::endl;
    return 0;
}
```

```
Start ---
Finish ---
```

Undefined Behaviour means:

your code *may* do nothing

```cpp
#include <iostream>

int main(){
    int i;
    double d;
    bool b;
    uint8_t u;
    std::cout << i << '\n';
    std::cout << d << '\n';
    std::cout << b << '\n';
    std::cout << u << '\n';
}
```

```
0
0
0
```

Undefined Behaviour means:

your code *may* do what you believe it should

```
#include <iostream>

int main(){
    int i;
    double d;
    bool b;
    uint8_t u;
    std::cout << i << '\n';
    std::cout << d << '\n';
    std::cout << b << '\n';
    std::cout << u << '\n';
}
```

```
0
6.95255e-310
0
```

Undefined Behaviour means:

your code *may* do what you believe it should

…until you change your compiler settings

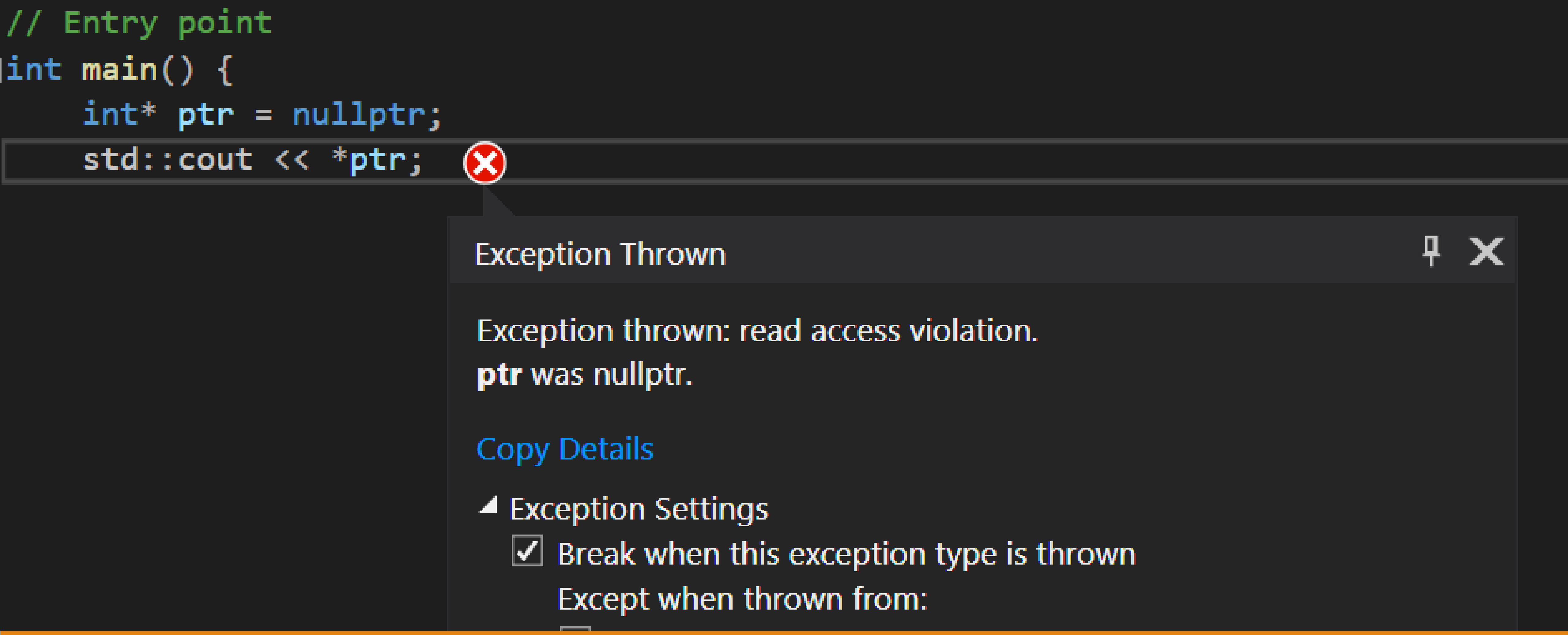
```cpp
#include <iostream>

int main(){
    int i;
    double d;
    bool b;
    uint8_t u;
    std::cout << i << '\n';
    std::cout << d << '\n';
    std::cout << b << '\n';
    std::cout << u << '\n';
}
```

```
718172376
0
0
```

Undefined Behaviour means:

your code *may* do what you believe it should

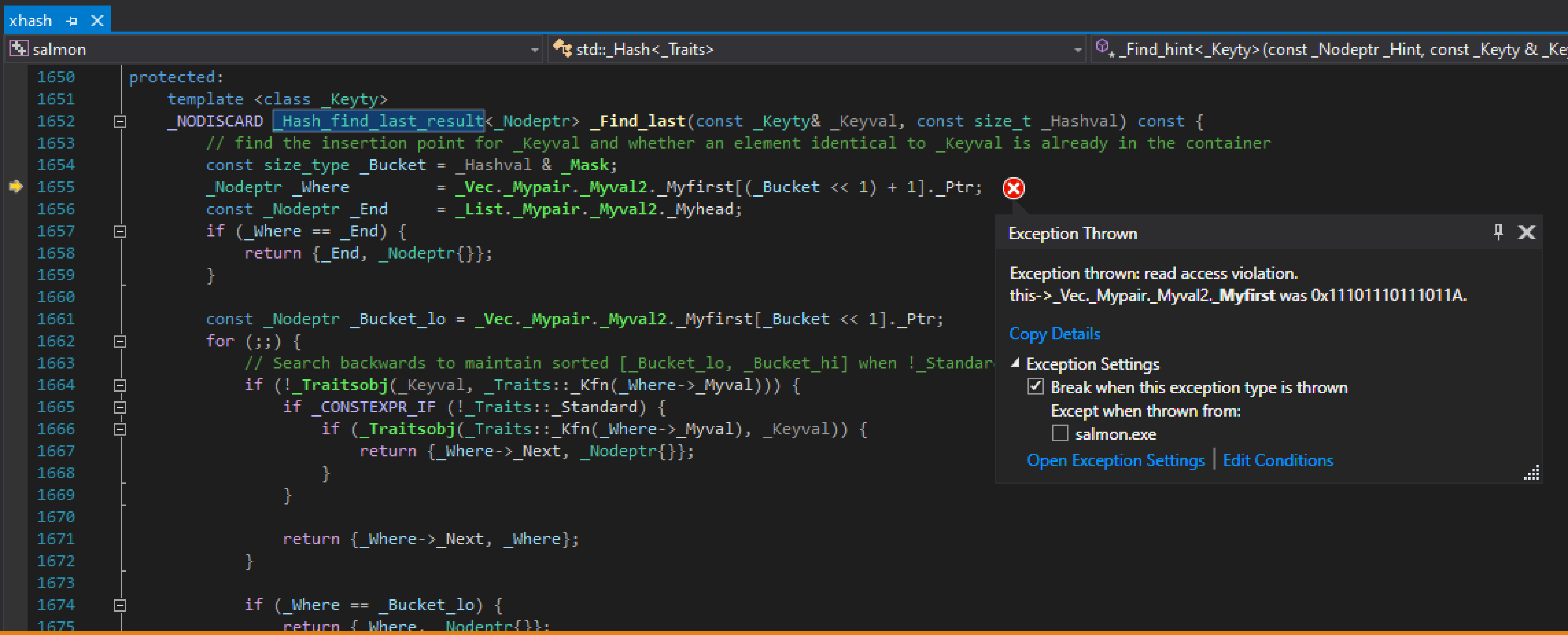
…until you change your compiler settings
…or try a different compiler



Undefined Behaviour means:
your code may crash with a helpful error message

```
// Entry point
int main() {
    return 0;
}
```

Undefined Behaviour means:
your code may **crash for no explainable reason**



Undefined Behaviour means:

your code may **crash for no explainable reason**

```cpp
#include <iostream>

bool fn() {
    // Oops! Forgot to return :-)
}

int main() {
    std::cout << "Start ---" << std::endl;
    if (fn()) {
        std::cout << "fn() returned true\n";
    } else {
        std::cout << "fn() returned false\n";
    }
    std::cout << "Finish ---" << std::endl;
    return 0;
}
```

```
Start ---
Start ---
bash: line 7:  1737 Segmentation fault      (core dumped) ./a.out
```

Undefined Behaviour means:

your code may run and do something **unexplainable**

```c
#ifndef OPENSSL_NO_HEARTBEATS
int
tls1_process_heartbeat(SSL *s)
	{
	unsigned char *p = &s->s3->rrec.data[0], *pl;
	unsigned short hbtype;
	unsigned int payload;
	unsigned int padding = 16; /* Use minimum padding */

	/* Read type and payload length first */
	hbtype = *p++;
	n2s(p, payload);
	pl = p;

	if (s->msg_callback)
		s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
			&s->s3->rrec.data[0], s->s3->rrec.length,
			s, s->msg_callback_arg);
```

This is a pointer to an array

This *should* be the length of that array

Undefined Behaviour means:

your code might run fine, but hackers can steal your passwords

The code will crash if I read an array out of bounds, right?

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

Undefined Behaviour means:

your code might run fine, but hackers can steal your passwords

Reading past the end of an array does not guarantee a crash.

Common Causes of **Undefined Behaviour**

- Reading from an **uninitialized** variable
- Reading an array **out of bounds**
- **Dereferencing a null pointer**
- **Dereferencing a pointer** that **does not point to a valid object**
- **`delete`-ing** dynamically allocated memory **twice**

Many famous software bugs and vulnerabilities are due to Undefined Behaviour!

Why does C++ have Undefined Behaviour? This sounds **terrible**!

- Undefined Behaviour *simplifies* compilation (and language design)
  - Compilers can (and do!) assume that Undefined Behaviour never happens
  - Compiler's don't need to do extra work to ensure safety
  - The concept of Undefined Behaviour was inherited from C
  - Detecting all types of Undefined Behaviour in C++ is **impossible.**

What Undefined Behaviour means for **you**

- **The C++ language cannot be learned by trial-and-error.**
- Read **good C++ books** and **reliable documentation** to learn to avoid Undefined Behaviour
  - see https://isocpp.org/get-started and https://en.cppreference.com/w/
- If you **write safe code** to begin with, you will **spend less time debugging** 🙂 🙂 🙂
- **Read compiler warnings** and **increase your compiler's warning level**
  - We've already turned on extra warnings in the starter code for you 😉
- **Write your own safety checks** when you're unsure

Avoiding Undefined Behaviour with Safety Checks

Enter the `assert(condition)` macro!

- In **debug mode**, halts the program *immediately* with a helpful error message if `condition` is false
  - **Use your debugger!** it will take you right to the problem!
- In **release mode**, does **nothing.**
  - Useful for optimization (fast code)
  - **Not useful for input validation!**
- Use assertions to **test your assumptions** and to find **unrecoverable errors**
- Ordinary exceptions may be `throw`-n for **recoverable errors** (which you can `catch`)

```cpp
#include <iostream>


int main() {
    int x = 88;
    int* ptr = nullptr;
    for (int i = 0; i < 100 && !ptr; ++i) {
        for (int j = 0; j < 100 && !ptr; ++j) {
            if (i*i + j*j == x) {
                ptr = &x;
            }
        }
    }
    std::cout << *ptr << std::endl;
    return 0;
}
```

```
bash: line 7: 29756 Segmentation fault      (core dumped) ./a.out
```

```cpp
#include <iostream>
#include <cassert>

int main() {
    int x = 88;
    int* ptr = nullptr;
    for (int i = 0; i < 100 && !ptr; ++i) {
        for (int j = 0; j < 100 && !ptr; ++j) {
            if (i*i + j*j == x) {
                ptr = &x;
            }
        }
    }
    assert(ptr != nullptr);
    std::cout << *ptr << std::endl;
    return 0;
}
```

```
a.out: main.cpp:14: int main(): Assertion `ptr != nullptr' failed.
bash: line 7: 30544 Aborted                 (core dumped) ./a.out
```

```cpp
#include <iostream>
#include <vector>


class A {
public:
    A() : m_items{1, 2, 3, 5, 7, 11, 13, 17, 19} {}
    int getItem(int index){

        return m_items[index];
    }
private:
    std::vector<int> m_items;
};

int main() {
    auto a = A{};
    std::cout << a.getItem(0) << std::endl;
    std::cout << a.getItem(13) << std::endl;
    return 0;
}
```

```
1
0
```

```cpp
#include <iostream>
#include <vector>
#include <cassert>

class A {
public:
    A() : m_items{1, 2, 3, 5, 7, 11, 13, 17, 19} {}
    int getItem(int index){
        assert(index >= 0 && index < m_items.size());
        return m_items[index];
    }
private:
    std::vector<int> m_items;
};

int main() {
    auto a = A{};
    std::cout << a.getItem(0) << std::endl;
    std::cout << a.getItem(13) << std::endl;
    return 0;
}
```

```
1
a.out: main.cpp:9: int A::getItem(int): Assertion `index >= 0 && index < m_items.size()' failed.
bash: line 7: 32109 Aborted                 (core dumped) ./a.out
```

```
#include <iostream>
#include <cmath>
#include <exception>

int main() {
    auto x = 0.0;
    std::cin >> x;
    std::cout << "x is " << x << std::endl;



    std::cout << "sqrt(x) is " << std::sqrt(x) << std::endl;
    return 0;
}

```

```
x is -22
sqrt(x) is -nan
```

```cpp
#include <iostream>
#include <cmath>
#include <exception>

int main() {
    auto x = 0.0;
    std::cin >> x;
    std::cout << "x is " << x << std::endl;
    if (x < 0.0) {
        throw std::runtime_error("Oops! Please enter a non-negative number, thanks! :-)");
    }
    std::cout << "sqrt(x) is " << std::sqrt(x) << std::endl;
    return 0;
}
```

```
x is -22
terminate called after throwing an instance of 'std::runtime_error'
  what():  Oops! Please enter a non-negative number, thanks! :-)
bash: line 7:  3873 Done                    echo "-22"
      3874 Aborted                 (core dumped) | ./a.out
```

```cpp
int main() {
    showLoginPrompt();
    if (getUserCommand() == DatabaseAction::Drop){
        auto uc = getUserCredentials();
        std::cout << "LOG: " << uc << " wants to delete the database" << std::endl;
        assert(uc == User::Admin);
        deleteTheEntireDatabase();
    }
    return 0;
}
```

```
Welcome to Database Management System (Development Version 9.04.12)
LOG: Guest wants to delete the database
a.out: main.cpp:29: int main(): Assertion `uc == User::Admin' failed.
bash: line 7: 14871 Done
     14872 Aborted                 (core dumped) | ./a.out
```

```cpp
int main() {
    showLoginPrompt();
    if (getUserCommand() == DatabaseAction::Drop){
        auto uc = getUserCredentials();
        std::cout << "LOG: " << uc << " wants to delete the database" << std::endl;
        assert(uc == User::Admin);
        deleteTheEntireDatabase();
    }
    return 0;
}
```

```
Welcome to Database Management System (Release 10.05.71)
LOG: Guest wants to delete the database
LOG: The database was successfully deleted. Everything is gone.
```

😭

```cpp
int main() {
    try {
        showLoginPrompt();
        if (getUserCommand() == DatabaseAction::Drop){
            auto uc = getUserCredentials();
            std::cout << "LOG: " << uc << " wants to delete the database" << std::endl;
            if (uc != User::Admin) {
                throw AuthenticationError{};
            }
            deleteTheEntireDatabase();
        }
    } catch (const std::exception& e){
        std::cout << "ERROR: " << e.what() << std::endl;
    }
    return 0;
}
```

```
Welcome to Database Management System (Release 10.05.71)
LOG: Guest wants to delete the database
ERROR: Authentication failed
```

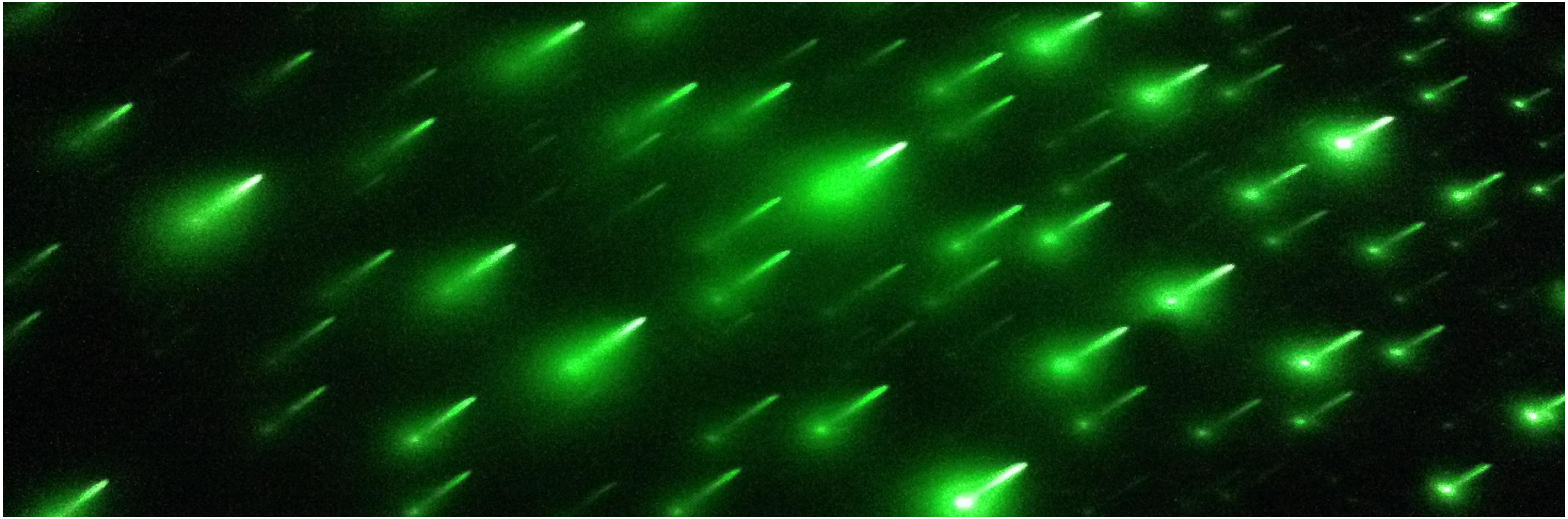
In conclusion:

- C++ is not safe
- *Undefined Behaviour* is weird
- *Undefined Behaviour* must be avoided
- Safety checks make life better

Lifetimes and Resource Management in C++

Lifetimes and Value Semantics

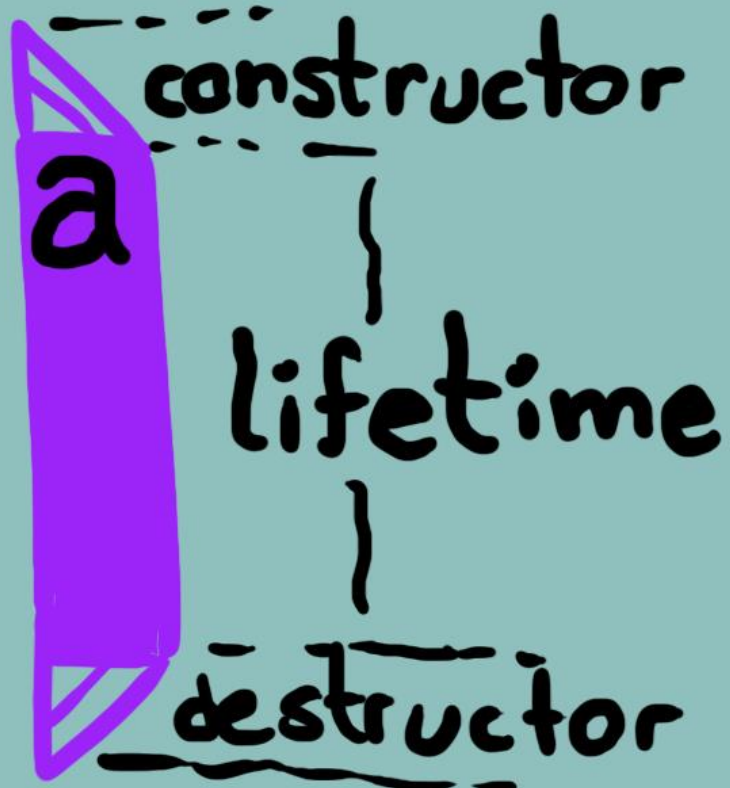
- One of C++'s **most important features**
- C++ **lets you decide** what happens when objects are **created**, **destroyed**, **copied**, and **moved**
- If used correctly, the C++ language will do the extra work for you
  - This results in **automatic**, **efficient**, and **deterministic** resource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



Lifetimes Visualized

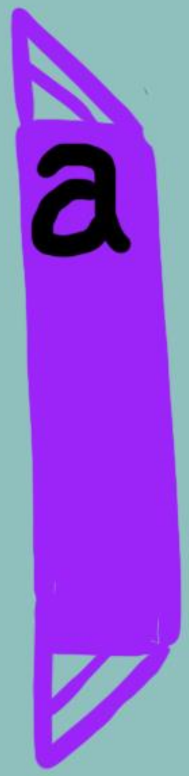
Source Code	Automatic Storage	Dynamic Storage

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

  std::cout << a;

  return 0;
}
```



Source Code | Automatic Storage | Dynamic Storage

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



  std::cout << a;



  return 0;
}
```



Source Code	Automatic Storage	Dynamic Storage

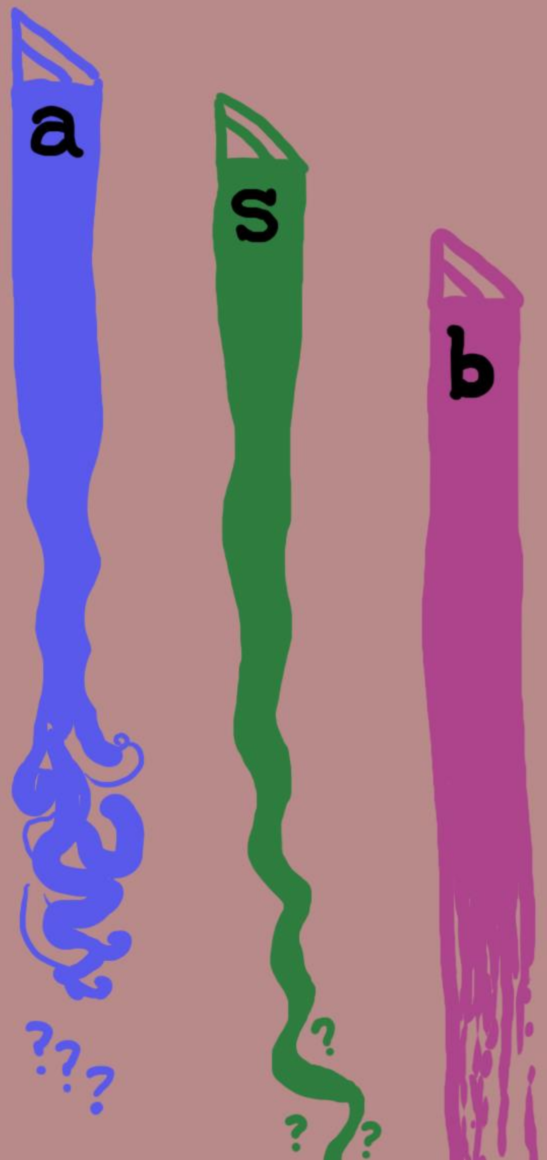
```
int main(){
  int a = 0;
  const char* s = “Foo”;


  std::cout << a << s;



  return 0;
}
```

Lifetimes in Python
(garbage collection)

Source Code

```
def main():
  a = 0
  s = "Hello"
  if 99 < 100:
    b = False
  print(b)
  return

// ...
```

Storage



Types of Lifetimes

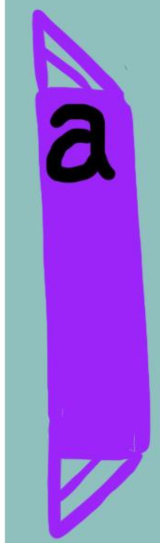
Any object in a running C++ program has one of three kinds of lifetimes, a.k.a. *storage durations*:

- 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

Automatic Storage Duration

Source Code	Automatic Storage	Dynamic Storage

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



  std::cout << a;



  return 0;
}
```

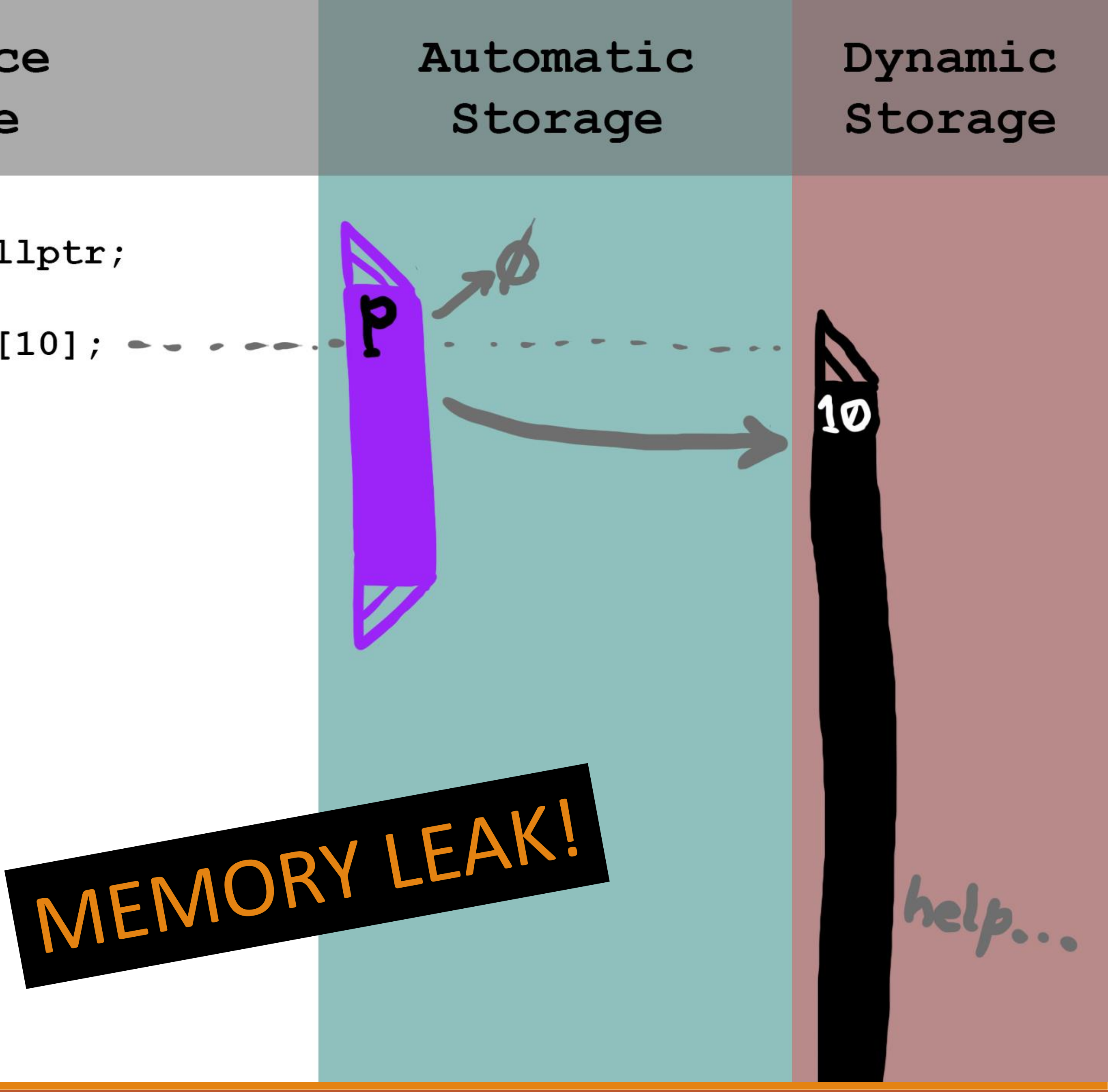


Dynamic Storage Duration

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

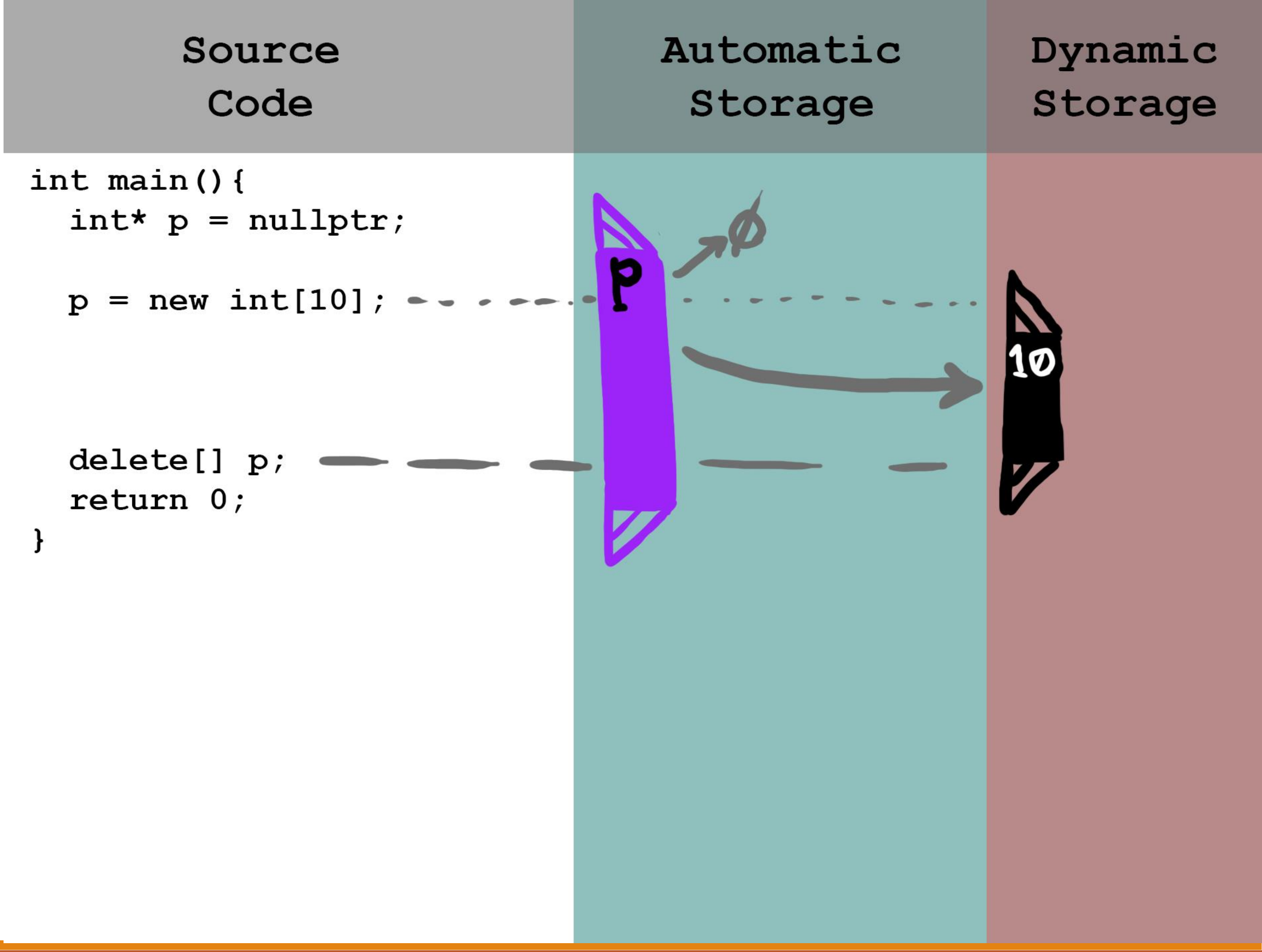
Source Code

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

  p = new int[10];




  return 0;
}
```



Dynamic Storage Duration

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



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

Managing Resources with Lifetimes

`std::ofstream` is a handle to an output file

```cpp
#include <fstream>
#include <iostream>

int main() {
    auto f = std::ofstream{"out.txt"};

    if (!f) {
        std::cout << "Error :(" << std::endl;
        return 1;
    }

    f << 'A';

    return 0;
}
```

`f` ***owns*** a file handle!

When the lifetime of `f` ends, the file is released!

cleanup is automatic!

Managing Resources with Lifetimes

`std::ofstream` is a handle to an output file

```
#include <fstream>
#include <iostream>

int main() {
    auto f = std::ofstream{"out.txt"};

    if (!f) {
        std::cout << "Error :(" << std::endl;
        return 1;
    }

    f << 'A';

    return 0;
}
```

The file gets closed here!

…or here!

Compare C++ to C

```cpp
#include <fstream>
#include <iostream>

int main() {
    auto f = std::ofstream{"out.txt"};

    if (!f) {
        std::cout << "Error :(" << std::endl;
        return 1;
    }

    f << 'A';

    return 0;
}
```

```c
#include <stdio.h>

int main() {
    FILE* f = fopen("out.txt", "w");

    if (f == NULL) {
        printf("Error :(");
        return 1;
    }

    fprintf(f, "A");

    fclose(f);
    return 0;
```

Gotta close it manually

Compare C++ to Python

Extra code is needed to close it properly

```cpp
#include <fstream>
#include <iostream>

int main() {
    auto f = std::ofstream{"out.txt"};

    if (!f) {
        std::cout << "Error :(" << std::endl;
        return 1;
    }

    f << 'A';

    return 0;
}
```

```python
try:
    with open("out.txt", "w") as f:
        f.write("A")
except:
    print("Error :(")
```

Compare C++ to Java

```cpp
#include <fstream>
#include <iostream>

int main() {
    auto f = std::ofstream{"out.txt"};

    if (!f) {
        std::cout << "Error :(" << std::endl;
        return 1;
    }

    f << 'A';

    return 0;
}
```

Wow, just wow!
So much code just
to close a file!

```java
package blah;

import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamDemo {
    public static void main(String[] args) {
        FileOutputStream f = null;

        try {
            f = new FileOutputStream("out.txt");

            f.write(65);
            f.flush();
            f.close();
        } catch (Exception e) {
            System.out.print("Error :(");
        } finally {
            if (f != null) {
                f.close();
            }
        }
    }
}
```

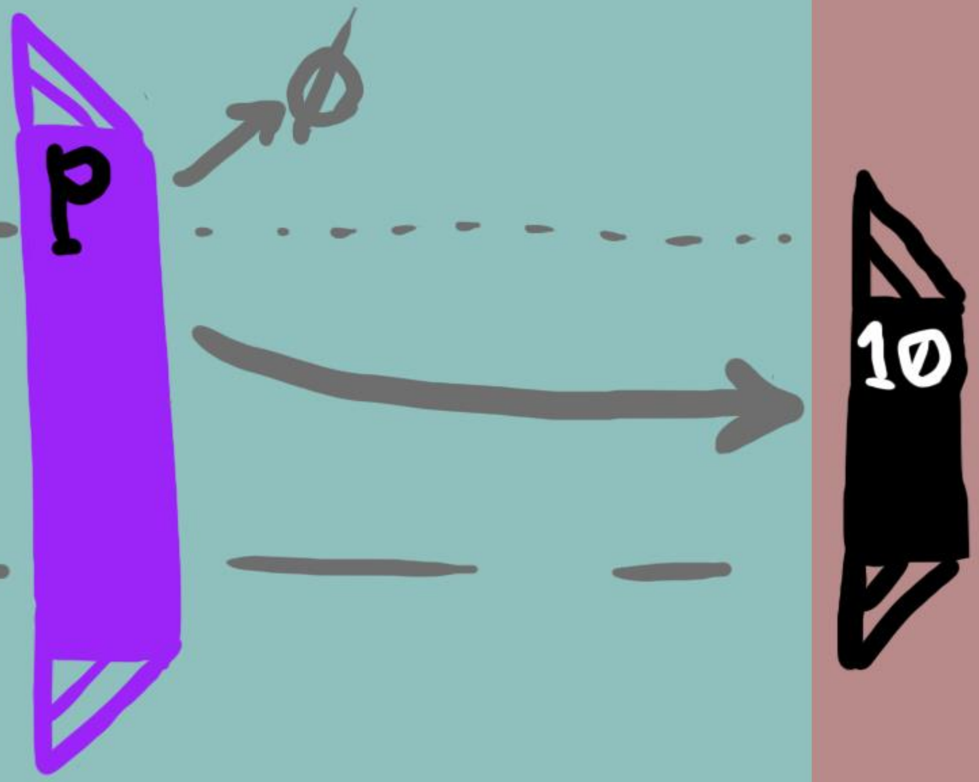
Resource Management in Modern C++

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

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

Source Code	Automatic Storage	Dynamic Storage

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

  p = new int[10];



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



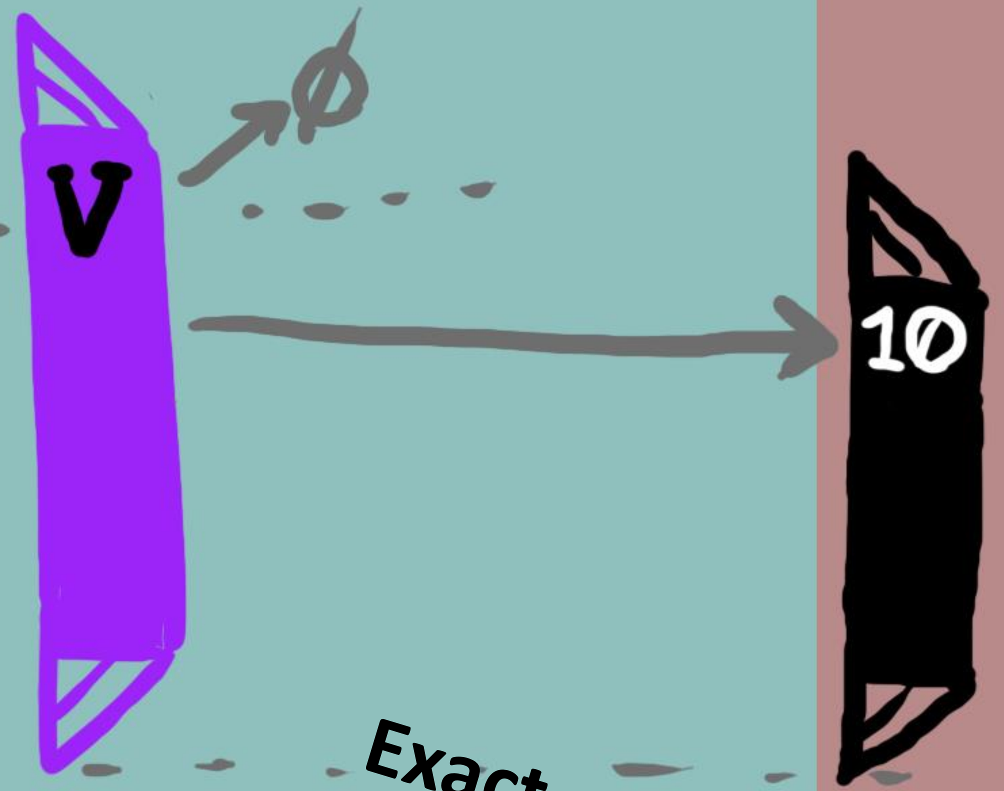
Source Code	Automatic Storage	Dynamic Storage

```
int main(){
  auto v = vector<int>{};

  v.resize(10);



  return 0;
}
```

No manual cleanup!

No more leaks!

No more screw-ups!



Exact same benefits!

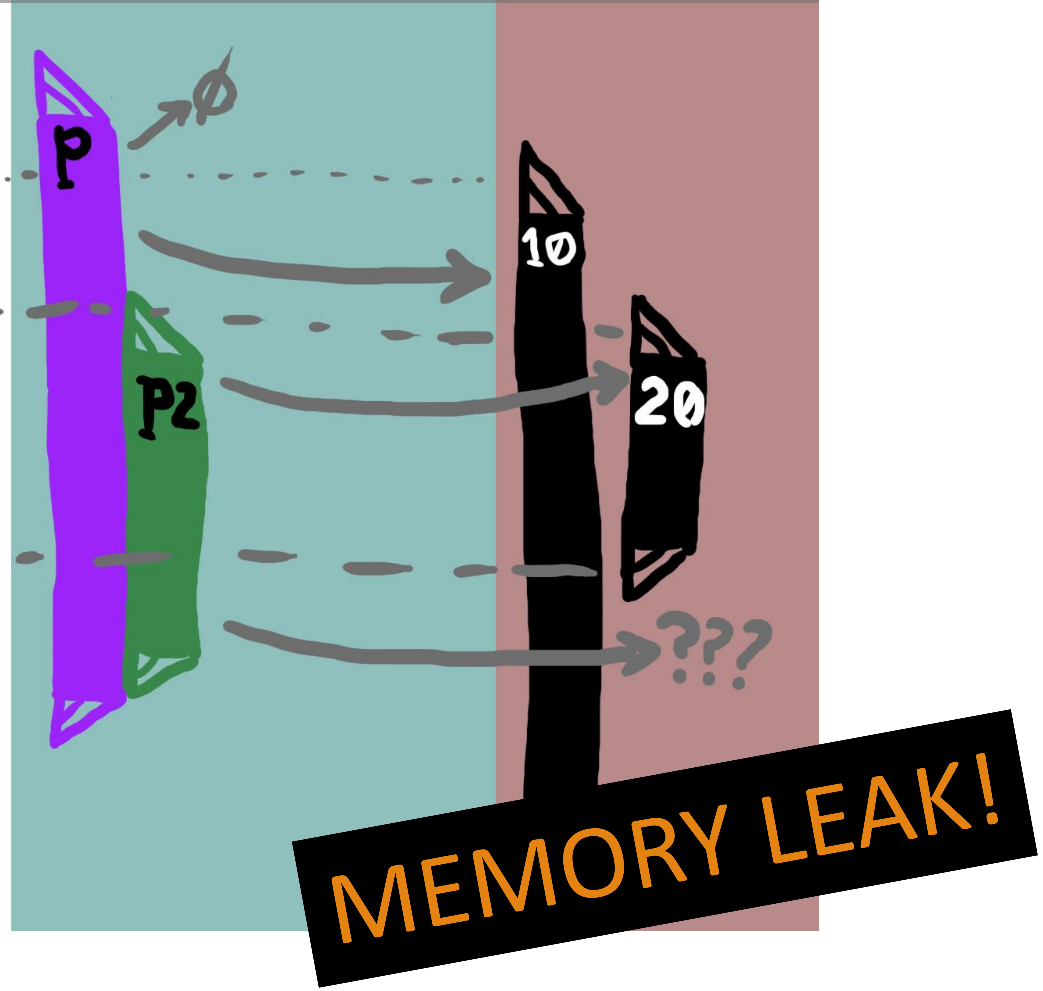
How to Resize a Dynamic Array

Using

Manual Memory Management

Source Code	Automatic Storage	Dynamic Storage

```
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];
  }
  p = p2;
  delete[] p2;



  return 0;
}
```



How to Resize a Dynamic Array

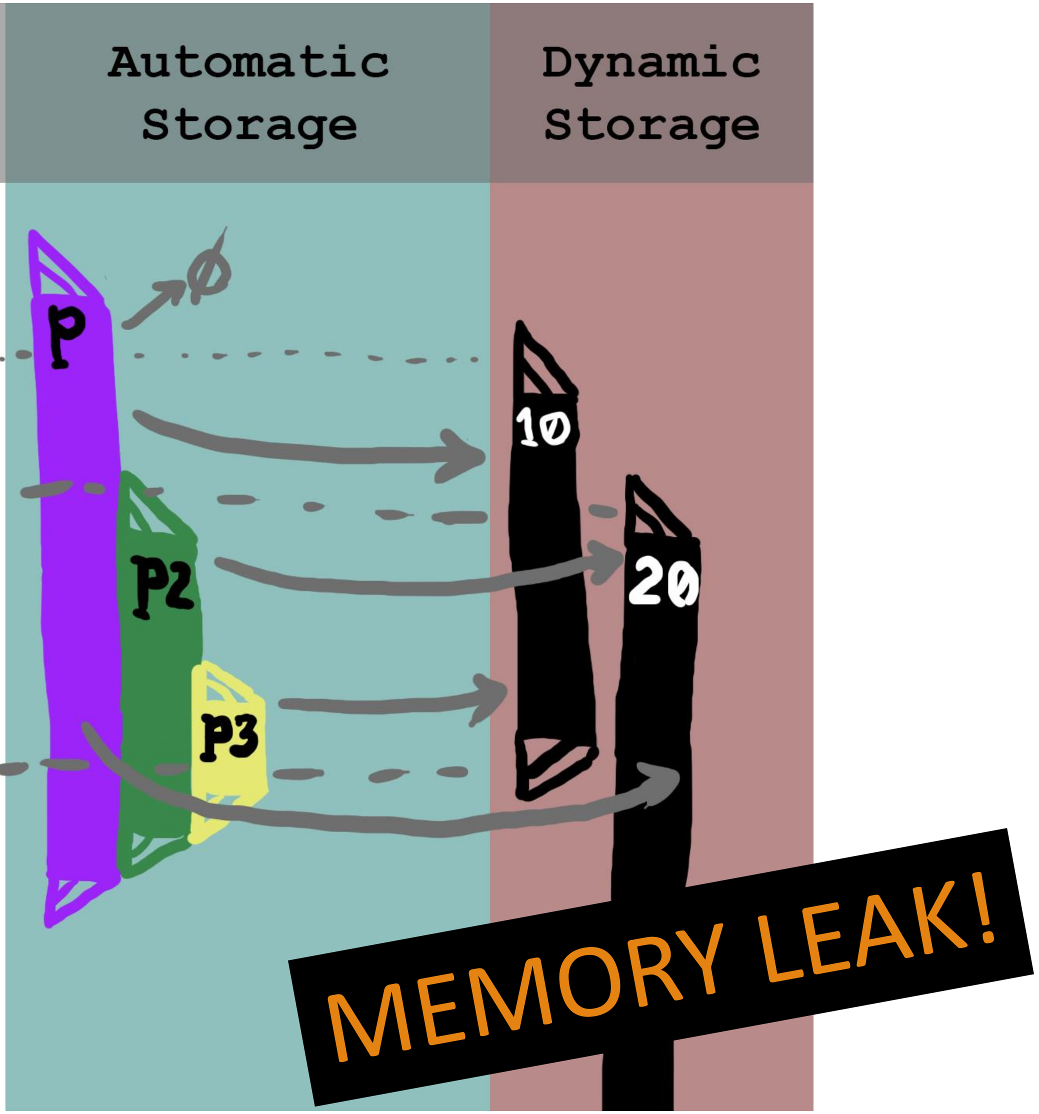
Using

Manual Memory Management

Attempt 2/N

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;



  return 0;
}
```



How to Resize a Dynamic Array

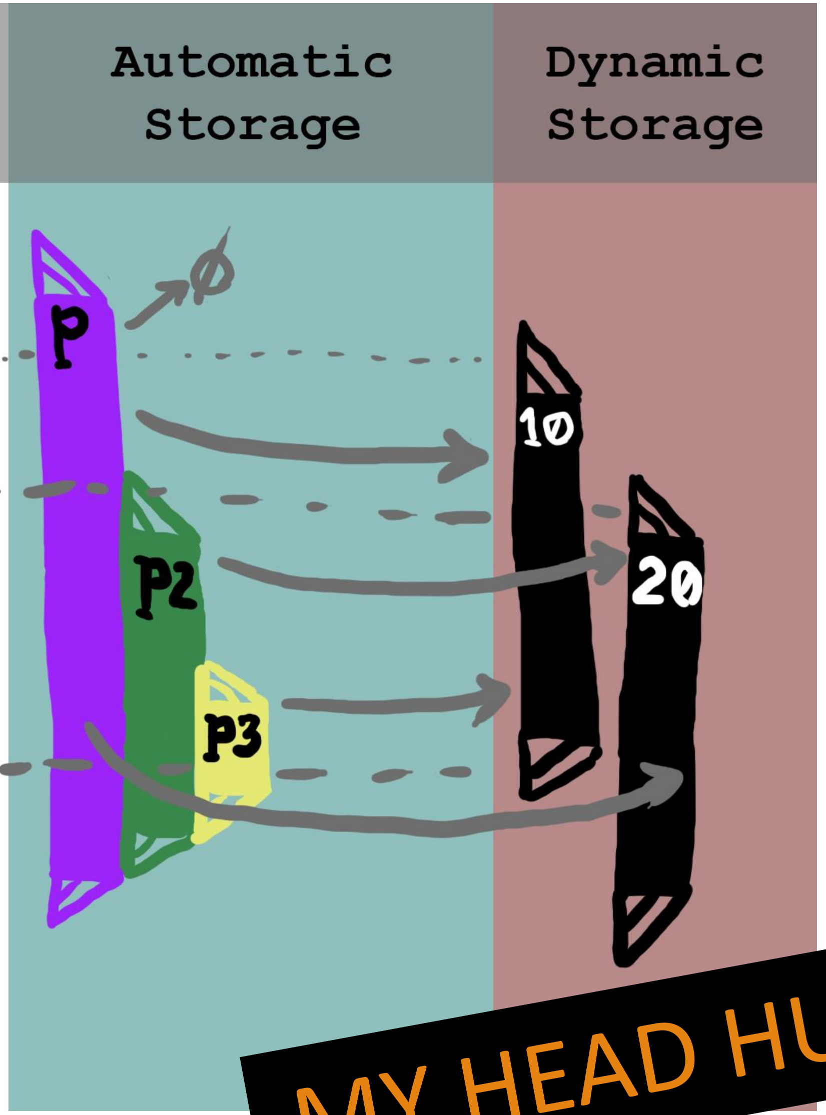
Using

Manual Memory Management

Attempt 3/N

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;
}
```



MY HEAD HURTS

How to Resize a Dynamic Array

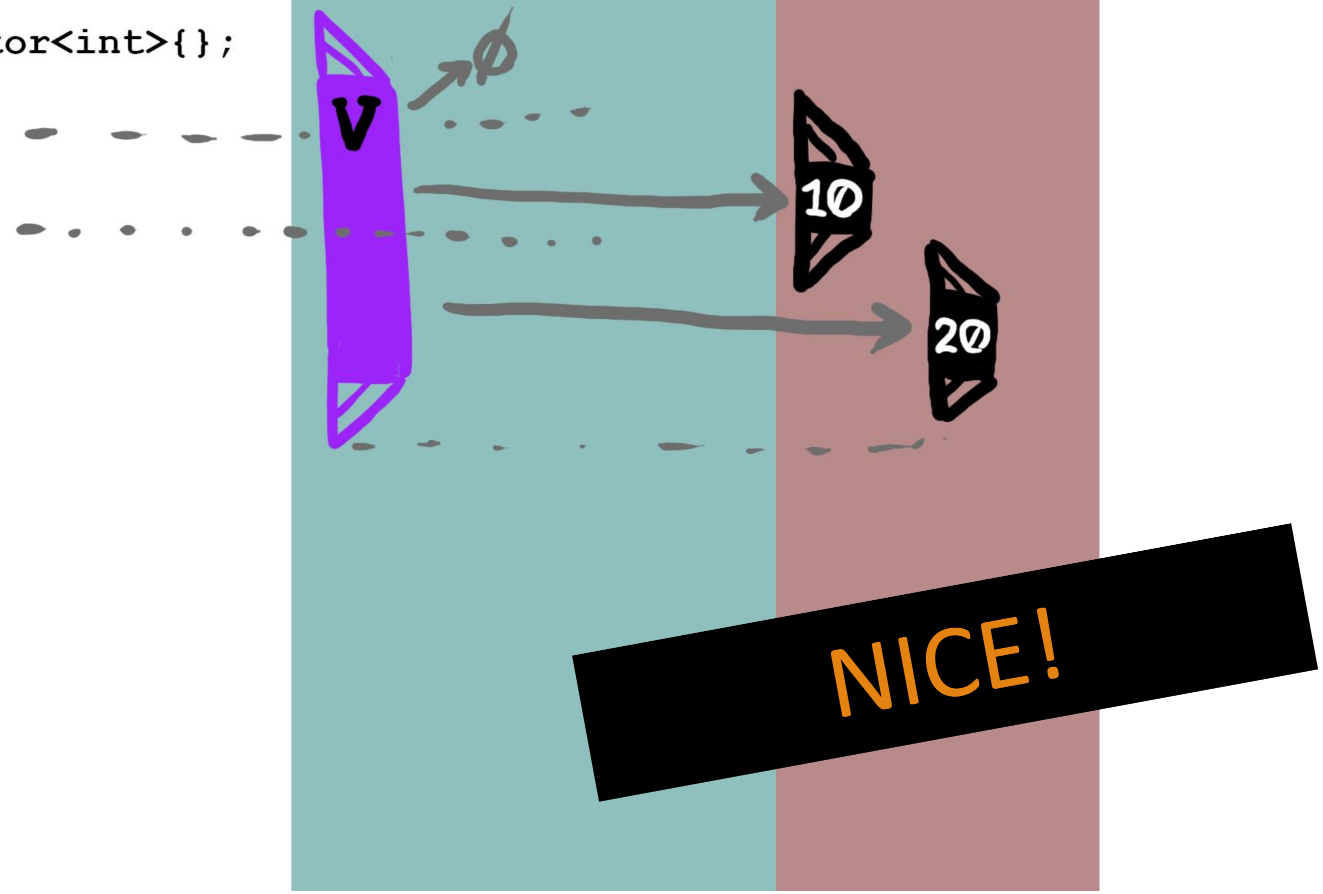
Using

vector

Attempt 1/1

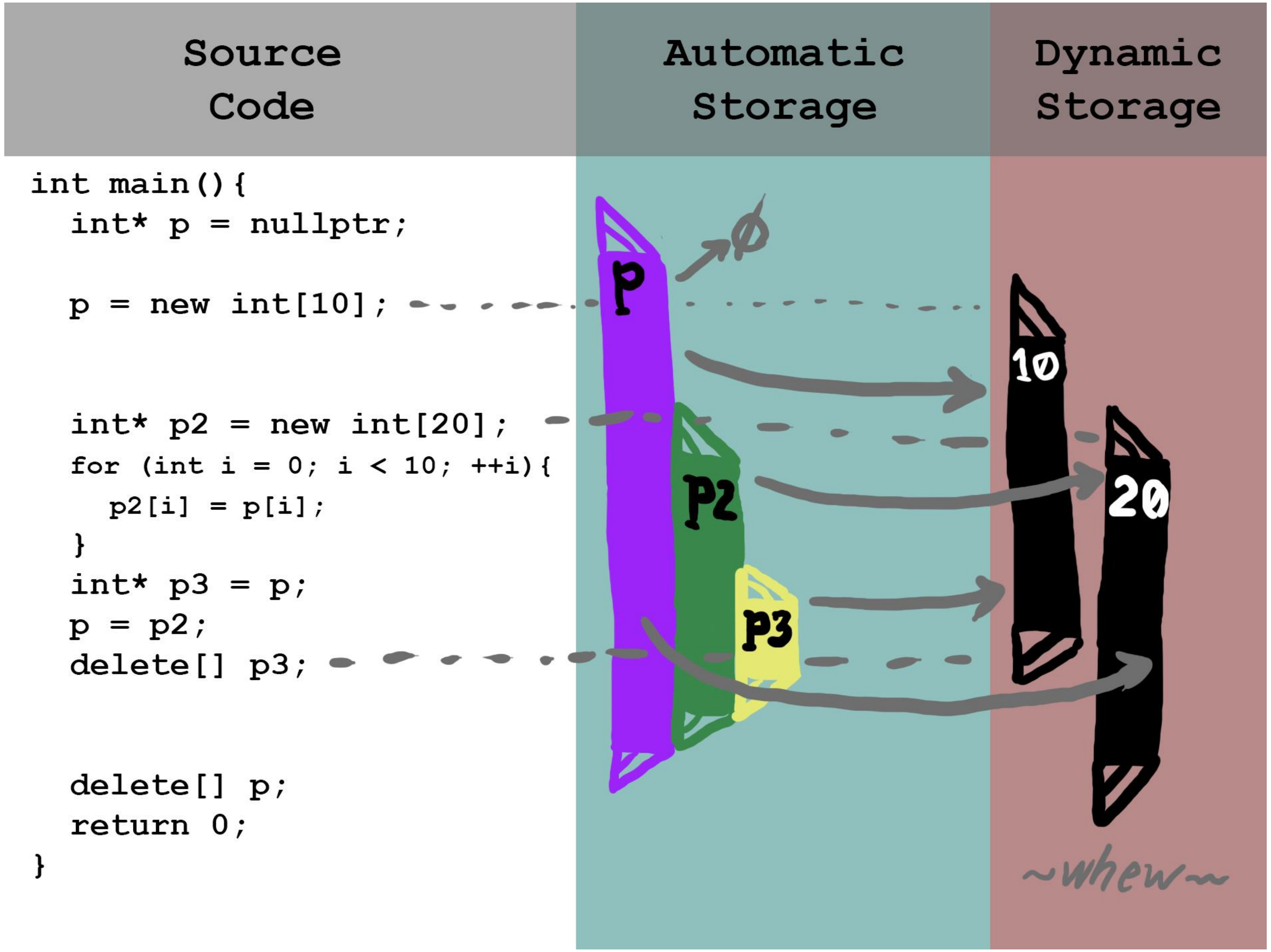
Source Code	Automatic Storage	Dynamic Storage

```
int main(){
  auto v = vector<int>{};

  v.resize(10);

  v.resize(20);


  return 0;
}
```



Option A

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

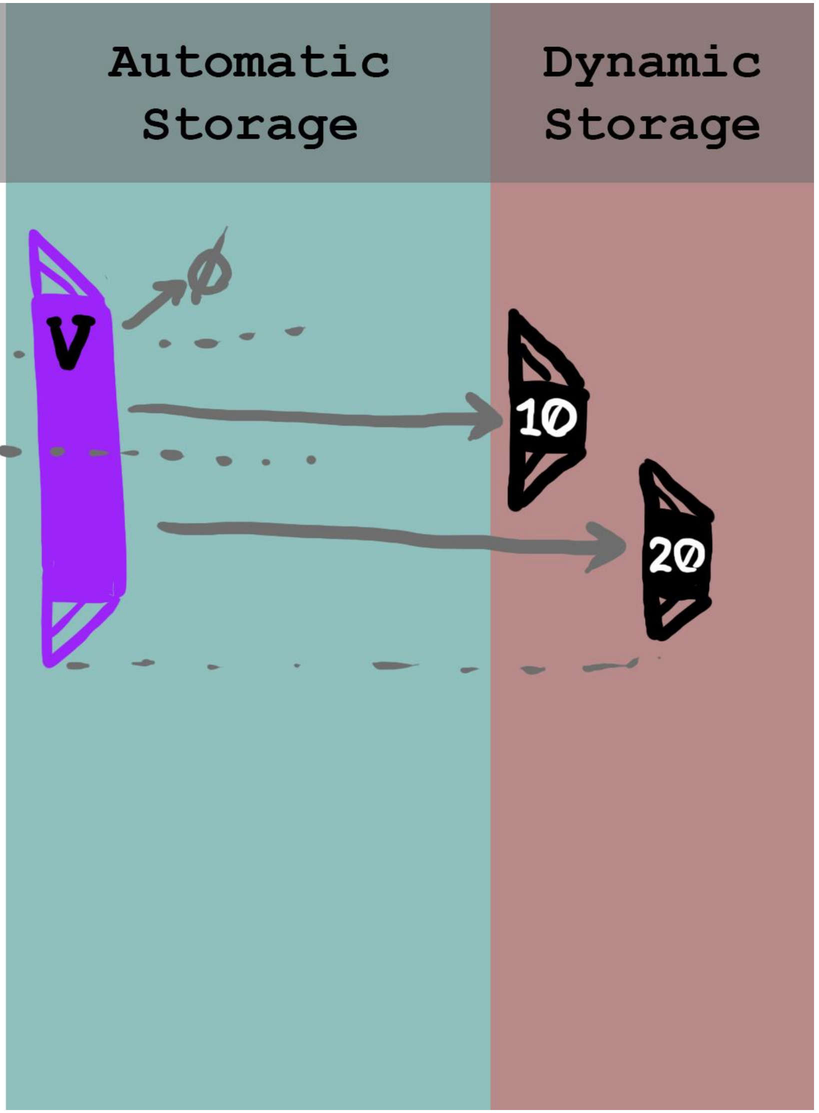


Option B

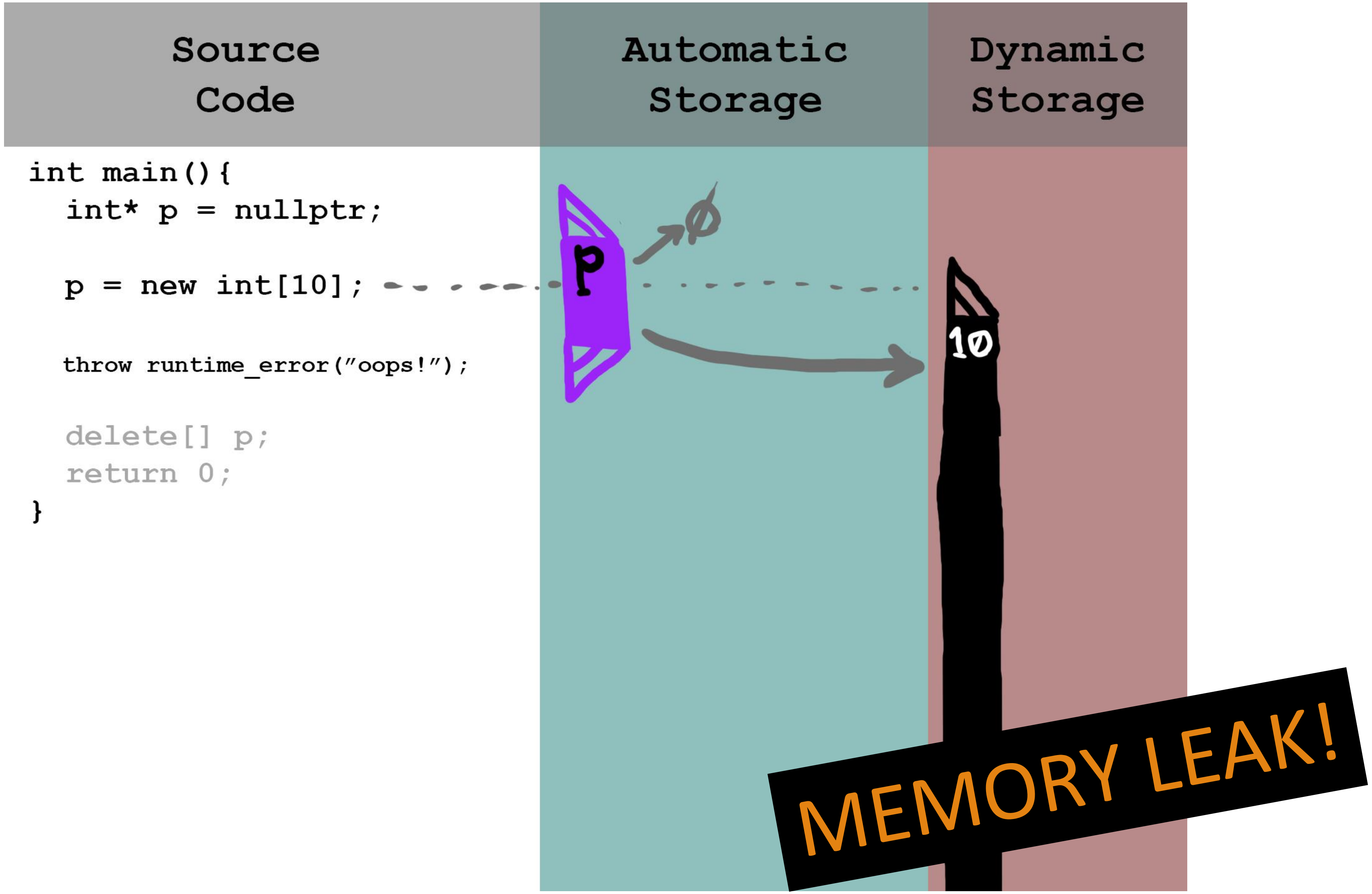
"Modern C++ programmer"

Source Code

```
int main(){
  auto v = vector<int>{};

  v.resize(10);

  v.resize(20);


  return 0;
}
```



Exception Safety
Using
Manual Memory Management



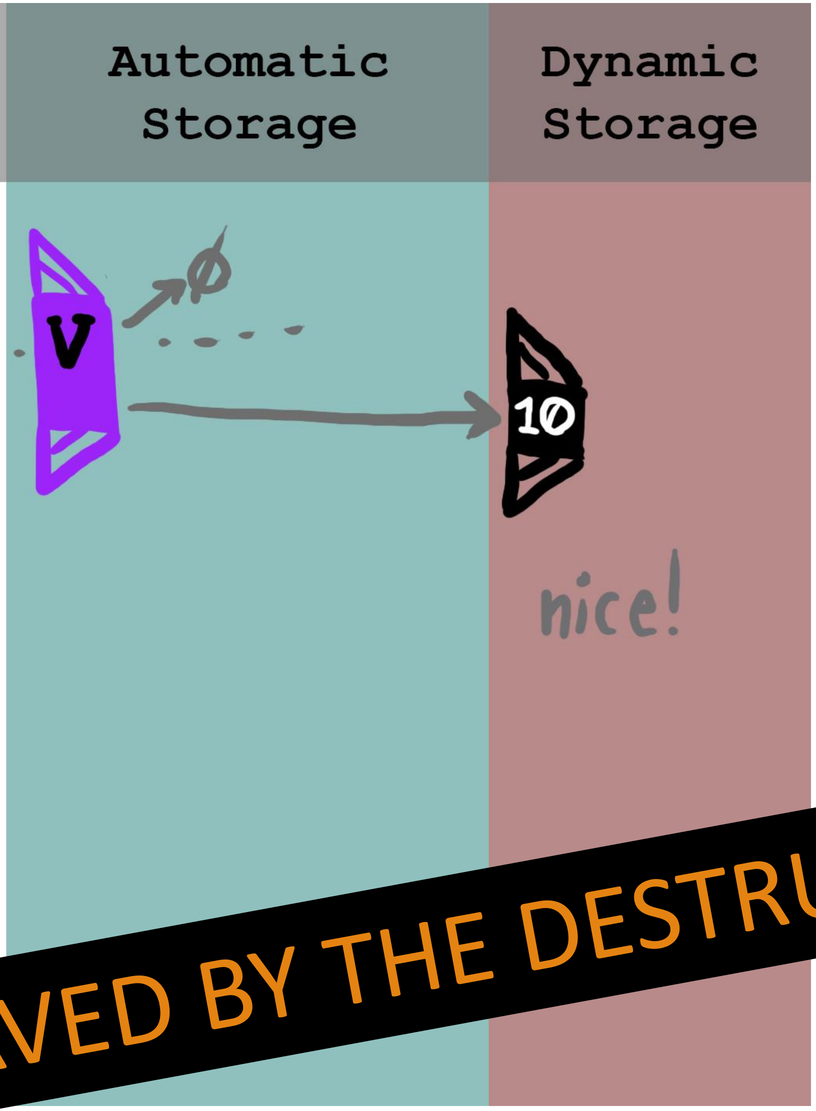
Exception Safety

Using

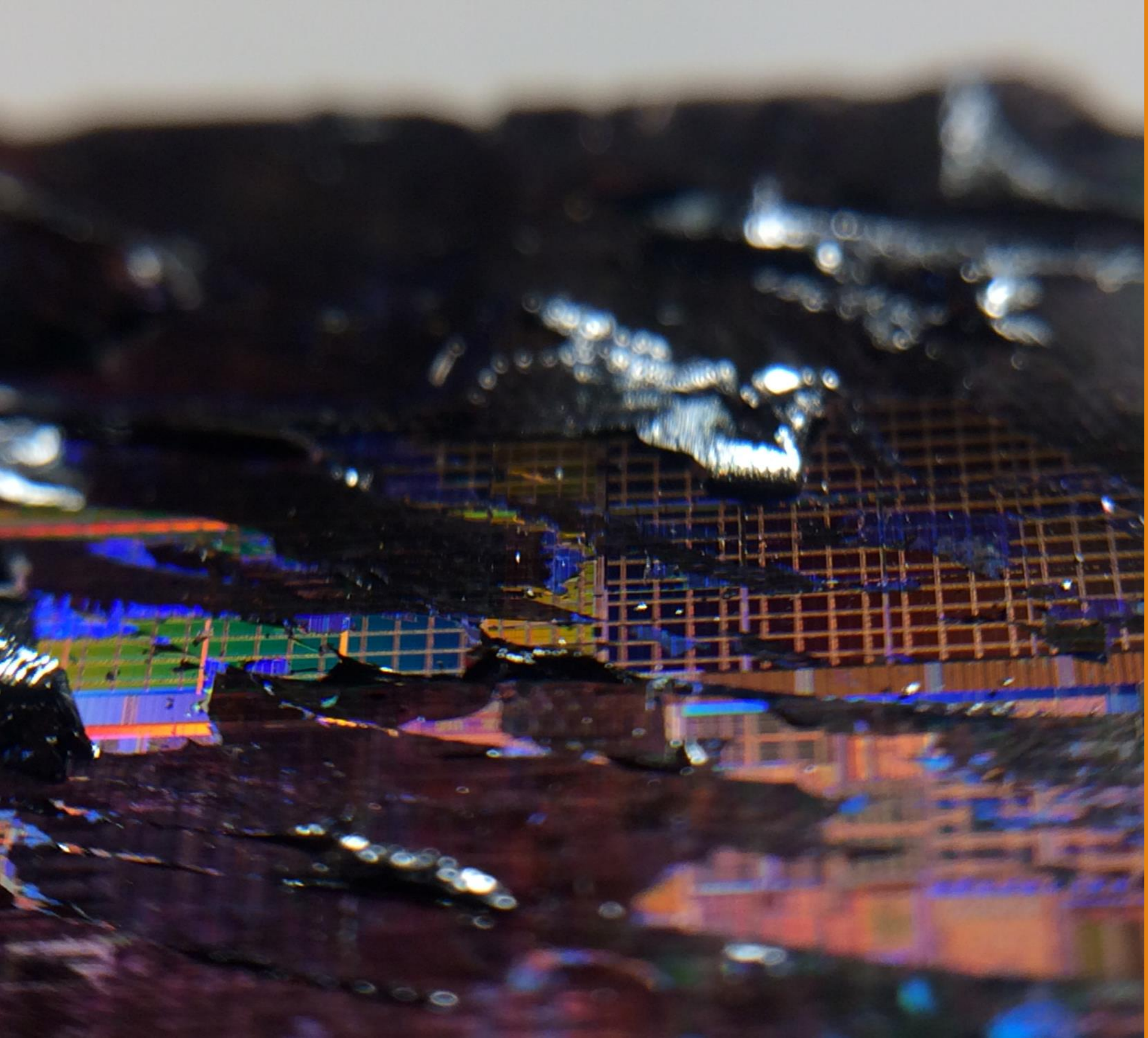
vector

Source Code

```
int main(){
  auto v = vector<int>{};

  v.resize(10);

  throw runtime_error("oops!");


  return 0;
}
```



SAVED BY THE DESTRUCTOR!



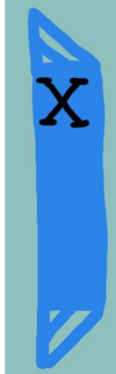
Special Member Functions

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!**

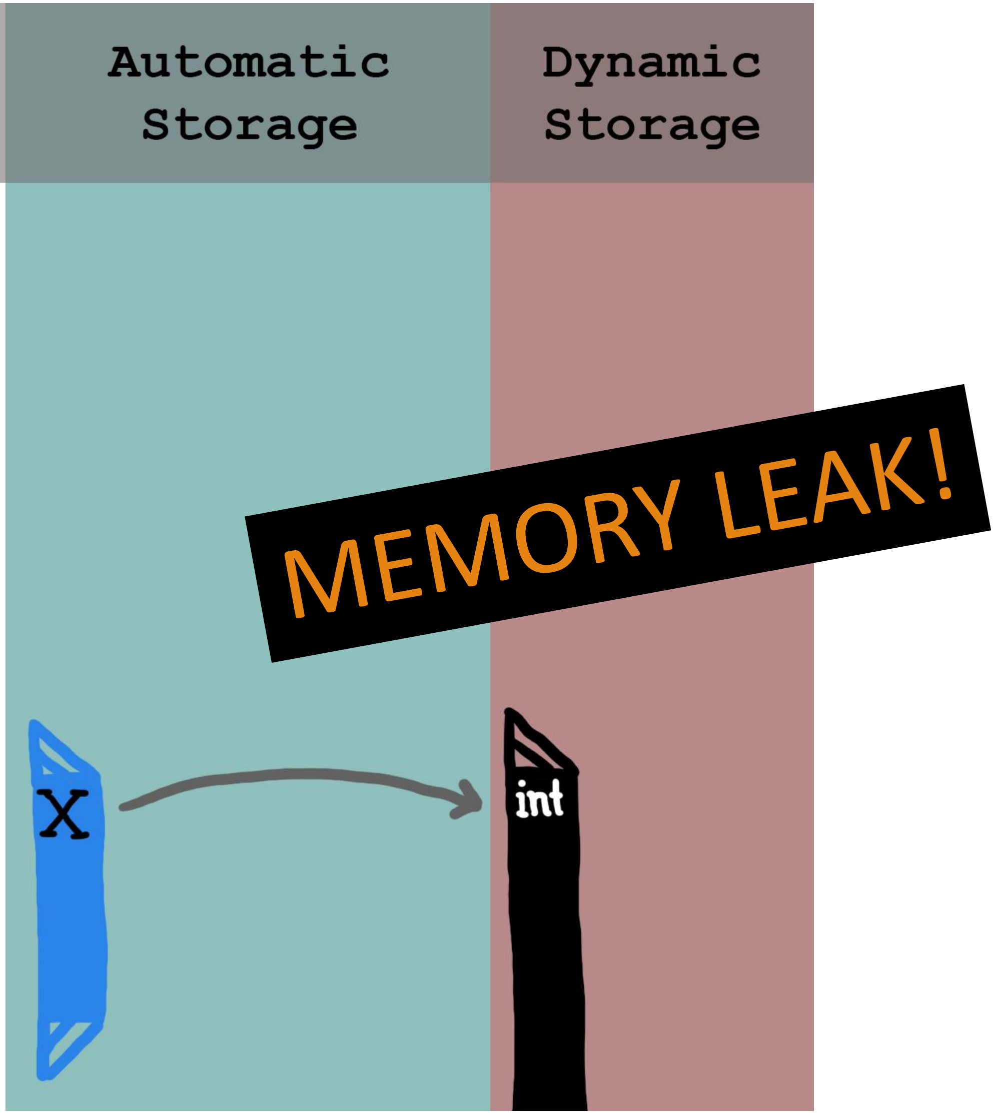
Source Code	Automatic Storage	Dynamic Storage

```
struct A {

};

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

  return 0;
}
```



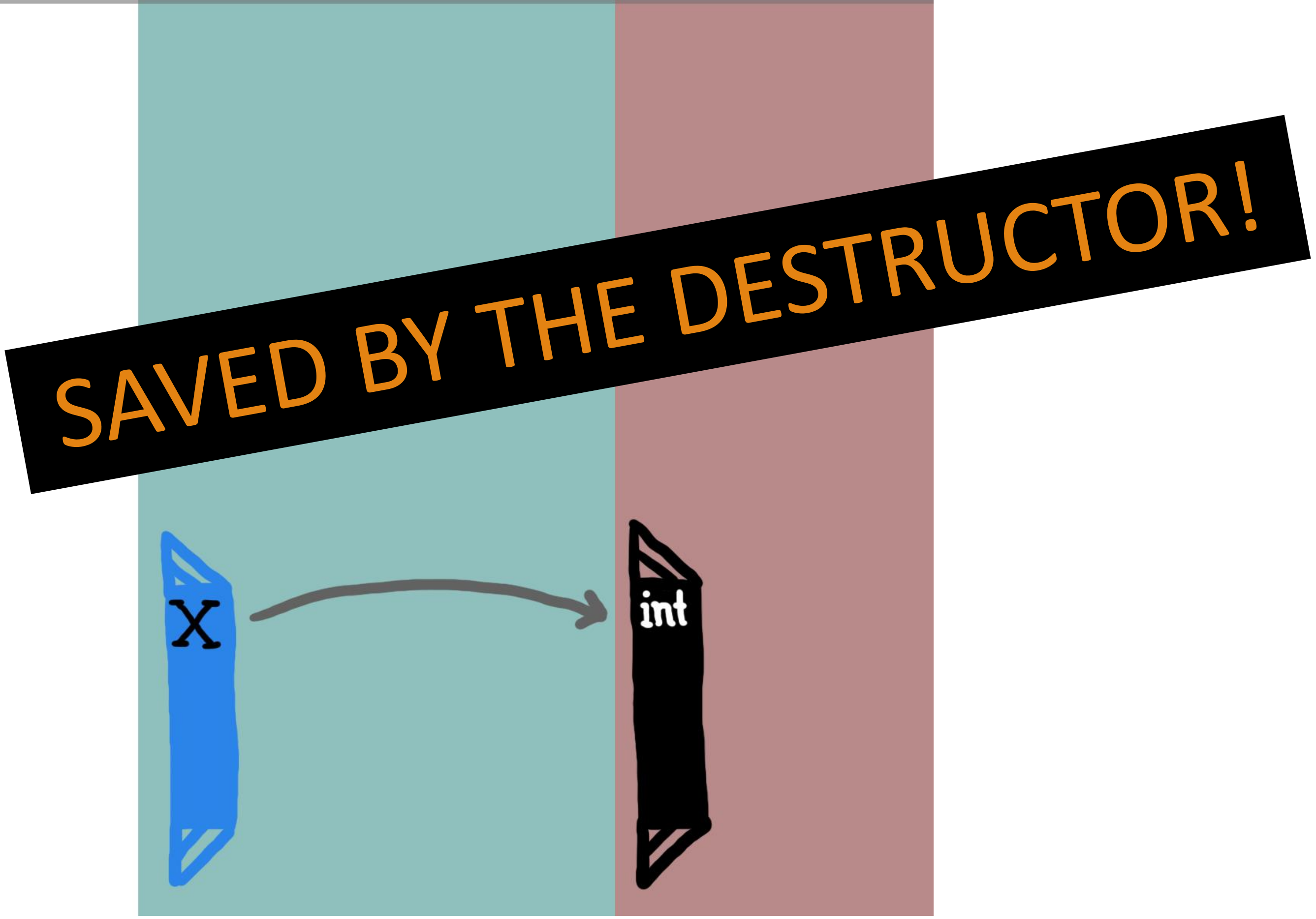
Source Code

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




private:
  int* p;
};

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




  return 0;
}
```



Source Code	Automatic Storage	Dynamic Storage

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

private:
  int* p;
};

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

  return 0;
}
```

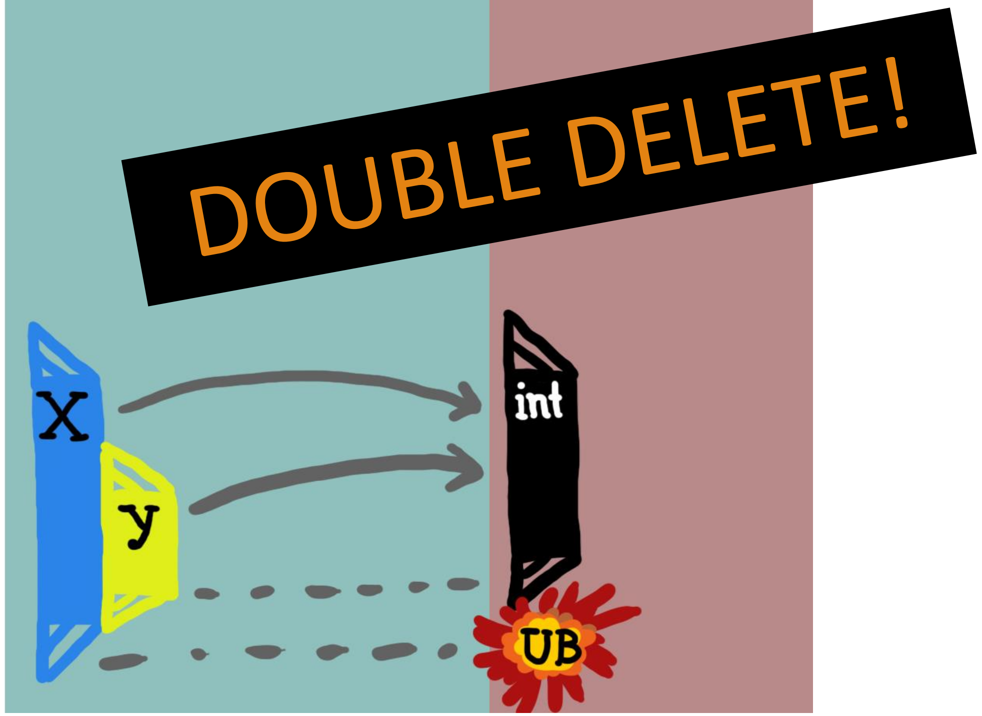


Copy Semantics – Construction and Assignment

- 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 ne copied
- Called **implicitly** as part of language
  - Use this to your advantage!

Source Code	Automatic Storage	Dynamic Storage

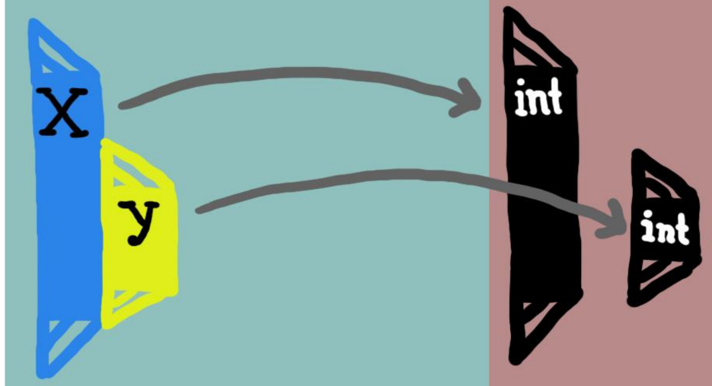
```
struct A {
  A():p{new int(32)} { }
  ~A(){ delete p; }

private:
  int* p;
};

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

  auto y = x;

  return 0;
}
```



Source Code	Automatic Storage	Dynamic Storage

```
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;
}
```

SAVED BY THE COPY CONSTRUCTOR!



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;`

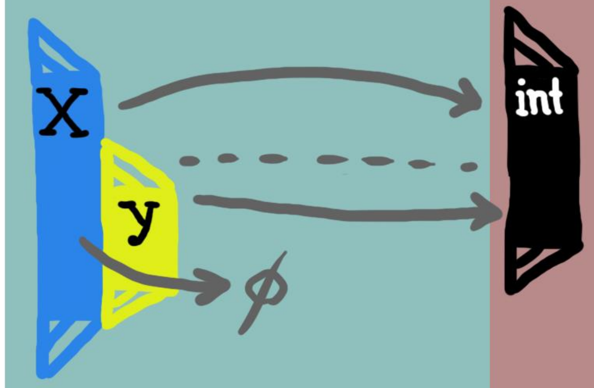
Move Semantics – Construction and Assignment

- 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

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;
}
```



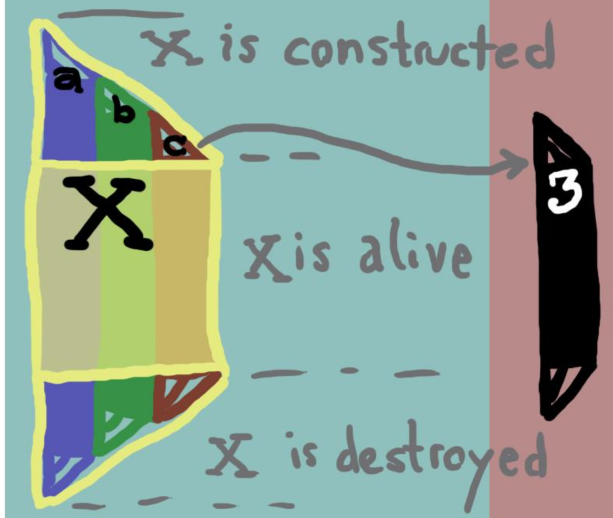
Destructor, Copy Constructor, Move Constructor, Copy Assignment Operator, Move Assignment Operator, Oh My!

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 member variables
- *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)

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

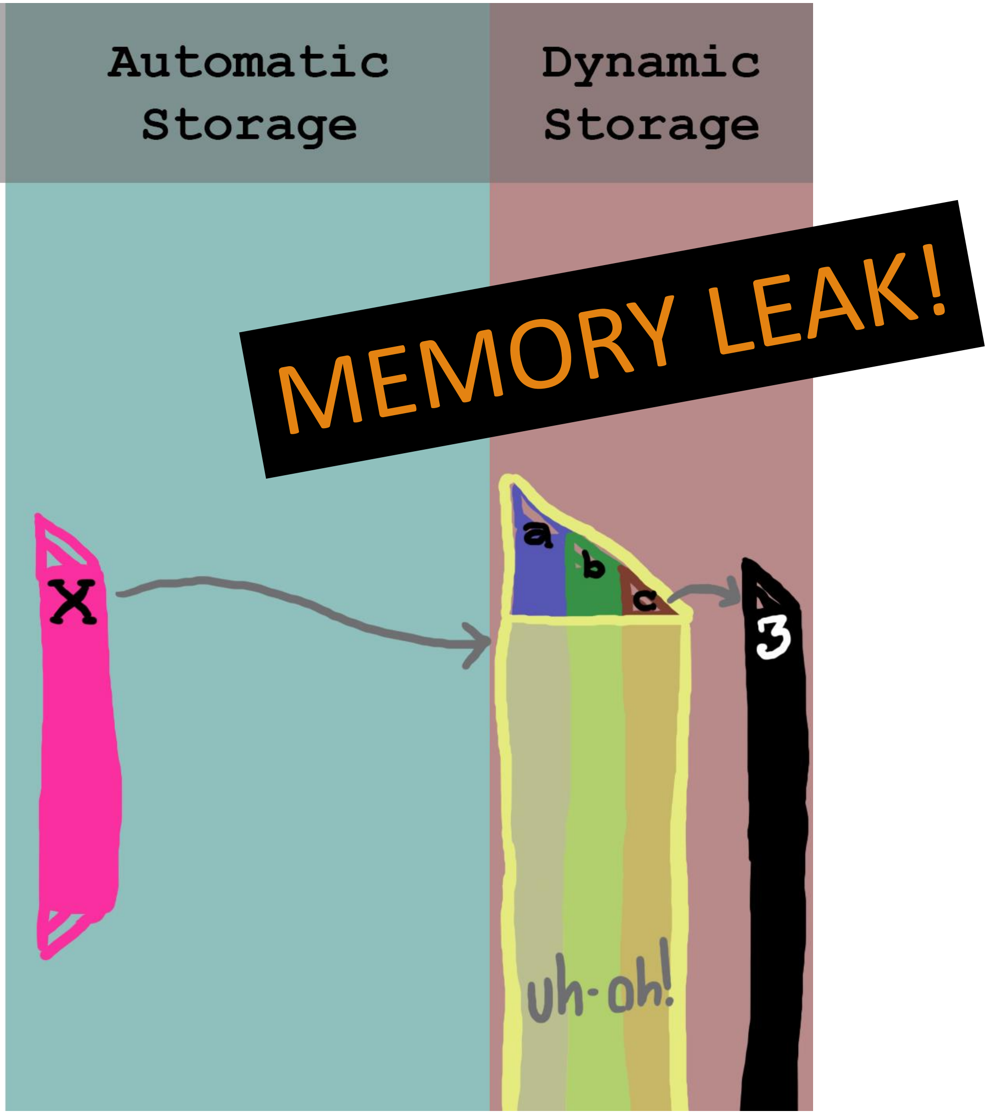
"I need heap allocation for some reason"

Using

Manual Memory Management

Source Code

```
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"

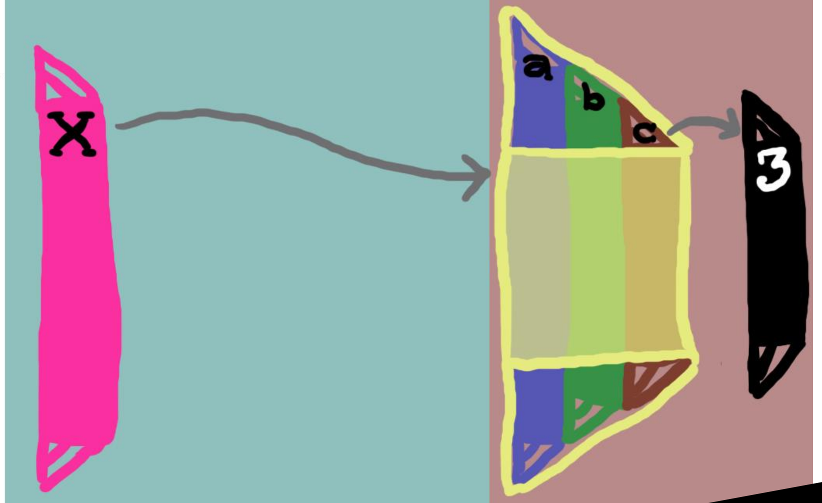
Using

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