

# Lecture 10: Pointer Pointers

Bart Iver van Blokland

# PSA 1 / 4: Kompendium is now available

- You can find it under «Files» on Canvas

2026 VÅR

Home

Syllabus

Pages

Announcements

Panopto

Nettside

Piazza

**Files**

Inginious (øvingssystem)

Modules

Assignments





## Files

Search files

Search files...

Enter at least 2 characters to search

TDT4102-26V :: Prosedyre- og objektorientert programmering

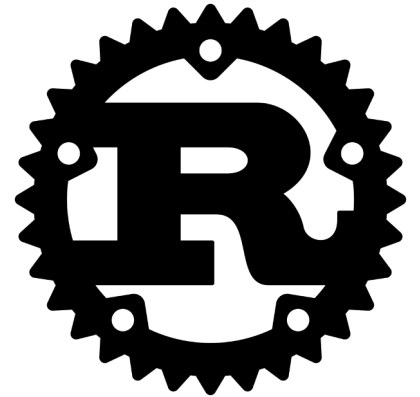
<input type="checkbox"/>	Name ▲	Created ▼	Last Modified ▼
<input type="checkbox"/>	 Forelesninger	Dec 28, 2025	Dec 28, 2025
<input type="checkbox"/>	 Prosjekt	Feb 13, 2026	Feb 13, 2026
<input type="checkbox"/>	 Forelesning_0_TDT4102.pdf	Jan 9, 2026	Jan 9, 2026
<input type="checkbox"/>	 Kompendium (2023)	Dec 28, 2025	Feb 13, 2026

# PSA 2 / 4: Project information

- Purpose: get experience with writing a program from scratch
- You can do what you like, however, there are some requirements. Refer to the project description text that has been published on Canvas (under Files → Prosjekt).
- Can count as 0, 1, 2, or 3 assignments depending on quality and size
- Can work alone or in pairs
  - For 2 people we expect roughly 2x the work
- Check the video on Canvas for inspiration on what to do =)
- Deadline: 10<sup>th</sup> of April
- **We will have awards for the best projects!**

# PSA 3 / 4: Guest lecture next week

- Isak Kyrre Lichtwarck Bjugn will come to talk about the programming language Rust
  - Rust attempts to avoid some problems in C++, while maintaining the high performance characteristics
- Isak works as a developer at Sparebank1 / Intern-alliansen
- Date: 6<sup>th</sup> of March (Friday next week)



# PSA 4 / 4: Reminder – halfway evaluation

- Please tell us how we're doing overall!  
(sorry to ask again, final time I promise!)
- We do this instead of the reference group



# Do you remember?

- What is a pointer?
- What does it mean to “dereference” a pointer?
- What is a difference between a pointer and a reference?
- How can you tell that a variable is allocated on the stack?
- What is the difference between stack and heap allocation?

# Today

- **Pointers (continued)**

# Pointers (continued)

- **Memory addresses**



# Memory Addresses

- Each byte in memory has its own number, known as its *address*
  - Each consecutive byte has a consecutive address
  - Pointers are variables that store such an address (= an integer)
  - Modern systems tend to use 64-bit addresses, because 32-bit ones can only address up to 4GB

Address	Value
...	...
20,285,638	00000000
20,285,639	00000000
20,285,640	00000000
20,285,641	00000101
20,285,642	00000000
20,285,643	00000000
20,285,644	00000000
20,285,645	00000000
20,285,646	00000001
20,285,647	00110101
20,285,648	10001000
20,285,649	11000110
...	...

# Memory Addresses

- Pointers always refer to the first byte of a value \*

- For example:

```
int main() {
    int value = 5;
    int* ptr = &value;
}
```

ptr will have the value  
20,285,638

Address	Value	Contents
...	...	...
20,285,638	00000000	int value
20,285,639	00000000	
20,285,640	00000000	
20,285,641	00000101	
20,285,642	00000000	int* ptr
20,285,643	00000000	
20,285,644	00000000	
20,285,645	00000000	
20,285,646	00000001	
20,285,647	00110101	
20,285,648	10001000	
20,285,649	11000110	
...	...	...

\* what is the «first» byte depends on the architecture of the processor

# Memory: Arrays

- C-style arrays use a pointer to the start of the array:

```
float* array = new float[3];
```

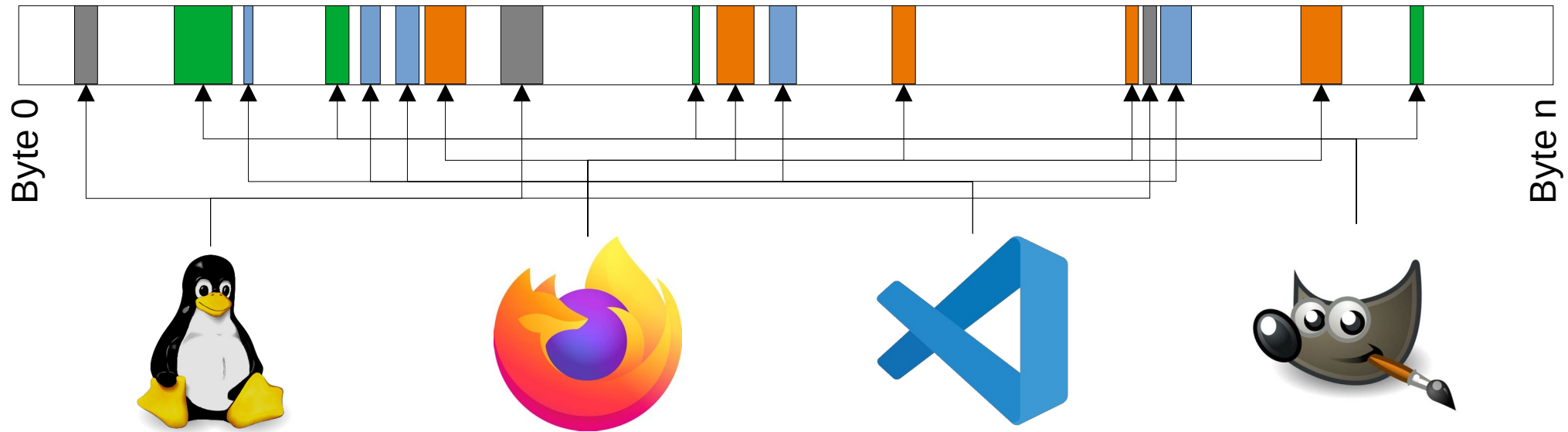
Here the address stored in array would be 77,285,638

- Array elements are always stored next to each other.
- The [ ] operator calculates which address to read from based on the value's size and index in the array

Address	Value	Contents
...	...	...
77,285,638	01100010	array[0]
77,285,639	01110010	
77,285,640	01100001	
77,285,641	00100000	
77,285,642	01101010	array[1]
77,285,643	01101111	
77,285,644	01100010	
77,285,645	01100010	
77,285,646	01100001	array[2]
77,285,647	00100000	
77,285,648	00111010	
77,285,649	00101001	
...	...	...

# Memory regions

- The addresses available on your computer are reserved for specific applications
- For safety, programs can only read memory allocated to them



# Pointers (continued)

- Memory addresses
- **Modifying pointers**

# Modifying pointers

- Using arithmetic operators directly changes the address, not the value it references:

```
int number = 5;
int* pointerToNumber = &number;

pointerToNumber++;
// no idea what this prints, but it's not the value of number
// because pointerToNumber now references another address
std::cout << *pointerToNumber << std::endl;
```

# Modifying pointers

- The `[]` operator is effectively a shorthand for dereferencing a computed address:

```
float* arrayOfNumbers = new float[10];
```

```
// These lines are equivalent
```

```
arrayOfNumbers[3] = 4.2;
```

```
*(arrayOfNumbers + 3) = 4.2;
```

# Pointers (continued)

- Memory addresses
- Modifying pointers
- **C-strings**



# C strings

- C strings are arrays of **char**acters, called as such because it's the only way possible to store strings in C.

```
char* usefulText = "The more you drink, the more WC!";
```

Stack variable

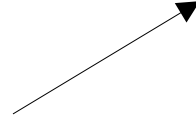
```
char* usefulText;
```

Somewhere in memory

'T'	'h'	'e'	' '	'm'	'o'	'r'	'e'	' '	'y'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Read characters using the [] operator, just like all arrays:

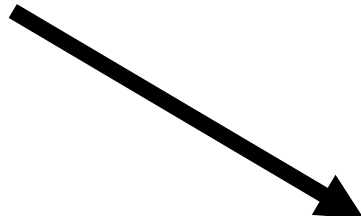
`usefulText[4]`



# C strings

- They were designed in a time when memory was expensive, so rather than store a length (like `std::string`'s `size()`), a byte set to 0 is used instead to denote the end of the string

Somewhere in memory



'e'	' '	'm'	'o'	'r'	'e'	' '	'W'	'C'	'!'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------

```
char* usefulText = "The more you drink, the more WC!";
```

# Pointers (continued)

- Memory addresses
- Modifying pointers
- C-strings
- **Pointers and const**

# Pointers and const

- There exist two possible placements of const when declaring a pointer, and both have a different meaning

- Right of data type: address is constant

```
int* const constAddress = &number;  
constAddress++; // error! You're modifying the address  
*constAddress++; // all good, increments referenced number
```

- Left of data type: value is constant

```
const int* constValue = &number;  
constValue++; // All good, changes the referenced address  
*constValue++; // error! Modifying the value
```

- Both sides: address and value are constant

```
const int* const bothConst = &number;
```

# Demonstration – Continuation from last week

# Today

- Pointers (continued)
- **Pointers: risks**

# Memory Management Risks

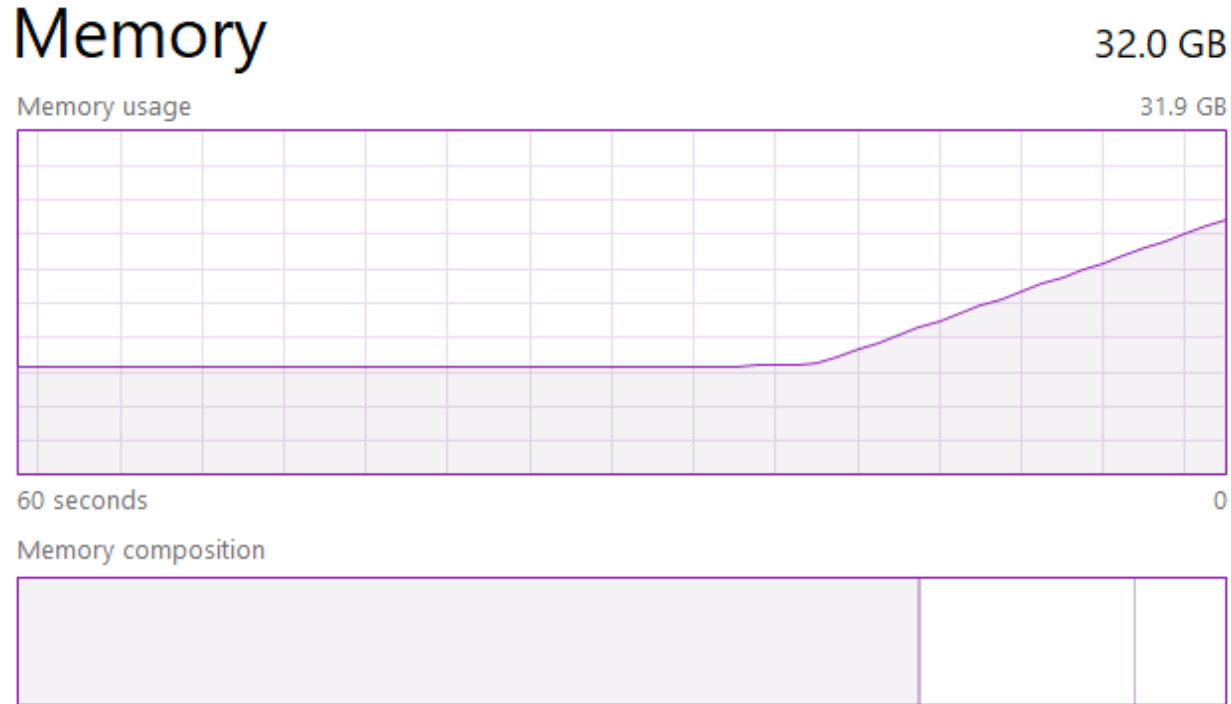
- Memory leak:  
The pointer to heap allocated memory is lost before the heap memory is deallocated.

Delete can only be used with a pointer to allocated heap memory, so when this pointer is lost, calling delete becomes impossible.

```
for(int i = 0; i < 10; i++) {  
    std::string* helpMessage = new std::string("Oh no!");  
    std::cout << *helpMessage << std::endl;  
}
```

# Memory Management Risks

- Memory leak:  
The worst ones are quite easy to detect





# Memory Management Risks

- Dangling reference:  
A pointer that references memory that has already been deleted

```
int* heapArray = new int[5] {1, 2, 3, 4, 5};  
delete[] heapArray;
```

```
// heapArray is now a dangling reference  
// it could still be used, but no longer can be used  
// heapArray[3] = 32; <- this is for example not allowed
```

# Memory Management Risks

- Double free:

Attempting to delete heap memory that has already been deleted

```
int* heapArray = new int[5] {1, 2, 3, 4, 5};  
  
for(int i = 0; i < 10; i++) {  
    delete[] heapArray;  
}
```

# Today

- Pointers (continued)
- Pointers: risks
- **Pointer problems: solutions?**

# How do we solve these?

- Heap memory (especially in objects) can give several issues:
  - Memory leaks
  - Double free
  - Dangling pointers?

It is extremely easy to cause memory leaks!

```
for(int i = number; i < number + 10; i++) {  
    std::string* text = new std::string("Leaking memory!");  
    if(number == 1) {  
        return;  
    } else if(number == 2) {  
        continue;  
    } else if(number == 3) {  
        break;  
    } else if(number == 4) {  
        throw std::runtime_error("Oh no!");  
    }  
    delete text;  
}
```

Each of these lines are cases where text is not deleted, because the line deleting it will not be run

← We will look at throw statements in a later lecture.

It is extremely easy to cause memory leaks!

```
struct GameWorld {  
    Terrain* terrainPtr;  
  
    GameWorld() {  
        terrainPtr = new Terrain();  
    }  
};
```

It is extremely easy to cause memory leaks!

```
struct GameWorld {
    Terrain* terrainPtr;

    GameWorld() {
        terrainPtr = new Terrain();
    }
};

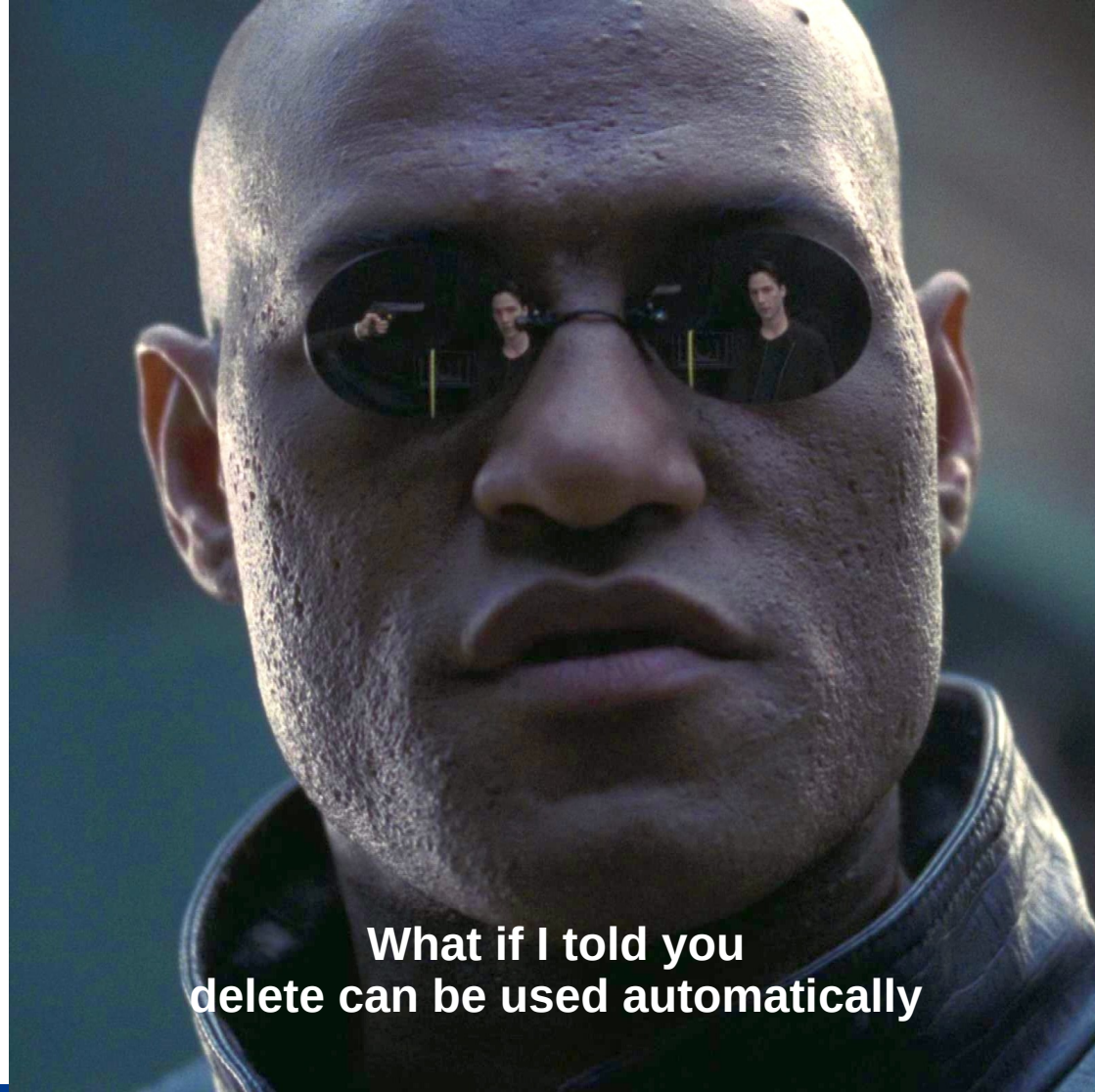
void saveGame(GameWorld _world) {}

int main() {
    GameWorld world;
    saveGame(world);
    delete world->terrain;
    return 0;
}
```

# How do we solve these?

- Heap memory (especially in objects) can give several issues:
  - Memory leaks
  - Double free
  - Dangling pointers?
- We want to be able to clean up after ourselves automatically
  - In every situation!





**What if I told you  
delete can be used automatically**

# DESTRUCTOR



# Destructors

- Destructors are called when an object is deleted  
For stack allocated objects: when its scope ends

```
void writeMessage(int n) {  
    std::string message = "n is: ";  
    message += std::to_string(n);  
    std::cout << message << std::endl;  
} ← message is deallocated here, which  
    will call its destructor
```

- For heap allocated objects: when **delete** is used

```
void heapMessage() {  
    std::string* message = new std::string("n is: ");  
    delete message; ← message is deallocated here  
}                    (with delete), which will call  
                    its destructor
```



# Destructors

- If you always delete memory allocated using `new` / `new[]`, **all destructors will always be called**
  - Including all elements in an array, but need to use `delete[]`
- Destructors can for example be used for:
  - Deleting heap allocated fields (with `new`) in the constructor
  - Automatically closing a file (`std::ofstream` does this)
  - Automatically exiting a network connection
- A class only needs to delete its own memory. When using inheritance, a child class does not need to clean up its parent's memory.

But we are not done yet!

```
struct GameWorld {
    Terrain* terrainPtr;

    GameWorld() {
        terrainPtr = new Terrain();
    }
    ~GameWorld() {
        delete terrainPtr;
    }
};

void saveGame(GameWorld _world) {}

int main() {
    GameWorld world;
    saveGame(world);

    return 0;
}
```

# Copy Constructor

- A special constructor that is called when copying an object.
  - Happens when assigning a variable to another, or when using pass-by-value for a function parameter
- If your object allocates heap memory in its constructor, your copy constructor should ensure that memory is duplicated in a separate region of memory

# Copy constructor: syntax

The copy constructor is another constructor which takes a **const** reference to an instance of the same object as its only parameter

```
struct GameWorld {  
    Terrain* terrainPtr;  
  
    GameWorld() {  
        terrainPtr = new Terrain();  
    }  
    ~GameWorld() {  
        delete terrainPtr;  
    }  
    GameWorld(const GameWorld& _w) {  
        terrainPtr = new Terrain();  
        *terrainPtr = _w->terrainPtr;  
    }  
};
```

# Copy constructor

```
void saveGame(GameWorld _world) {}
```

```
int main() {  
    GameWorld world;  
    saveGame(world);  
    GameWorld copyOfWorld = world;  
  
    return 0;  
}
```

```
struct GameWorld {  
    Terrain* terrainPtr;  
  
    GameWorld() {  
        terrainPtr = new Terrain();  
    }  
    ~GameWorld() {  
        delete terrainPtr;  
    }  
    GameWorld(const GameWorld& _w) {  
        terrainPtr = new Terrain();  
        *terrainPtr = _w->terrainPtr;  
    }  
};
```

Both of these create a copy,  
which calls the copy constructor



# Deep vs Shallow copy

- What the copy constructor should do is context dependent
- Deep copy: duplicate the contents of anything that is referenced by heap pointers
  - Default for C++ standard library (e.g. `std::vector`)
- Shallow copy: create a copy of the reference/pointer, but not the content being referenced
  - Default for when no copy constructor is defined

# Today

- Pointers (continued)
- Pointers: risks
- **std::unique\_ptr**
- std::shared\_ptr

# std::unique\_ptr

- A class which manages a pointer, and whose destructor automatically deletes the memory it references

```
void doWork() {  
    std::unique_ptr<Player> {new Player()};  
}
```

Note: no \* behind Player

Memory is allocated here..

.. And deleted here automatically by the destructor of std::unique\_ptr

# How does `std::unique_ptr` work?

- `std::unique_ptr` is a class
- How can a class make sure that `delete` is used on the contents being referenced when the class itself is deallocated?

Kaboom?



Yes, Rico, Kaboom



# DESTRUCTOR



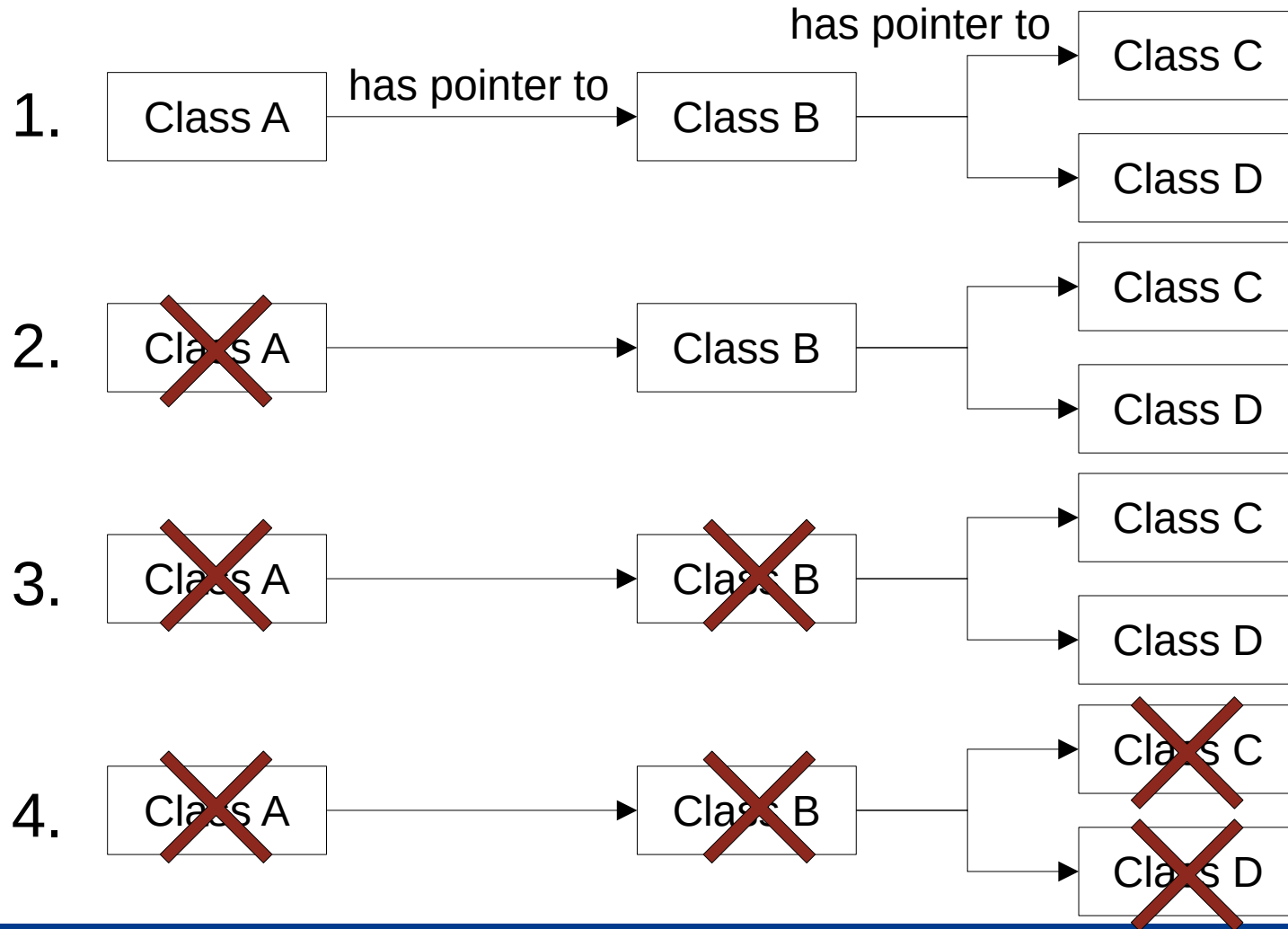
# How does `std::unique_ptr` work?

- `std::unique_ptr` is a class
- Its destructor is basically:

```
~unique_ptr() {  
    delete pointer_to_contents;  
}
```



- We want to create a «domino» of destructors calling destructors





# std::unique\_ptr

- A slightly more efficient way to create a unique\_ptr is using the make\_unique function:

```
std::unique_ptr<Player> player = std::make_unique<Player>();
```



Constructor parameters (if any) go here!

# std::unique\_ptr

- Using the magic of operator overloading, you can use it as a normal pointer

```
std::unique_ptr<Player> player {new Player()};
```

```
player->attack();  
(*player).attack();
```

# std::unique\_ptr

- Unique\_ptr guarantees there only exists one single copy of the pointer. It is therefore not possible to create a copy:

```
std::unique_ptr<Player> copyOfPlayer = player; // error!
```

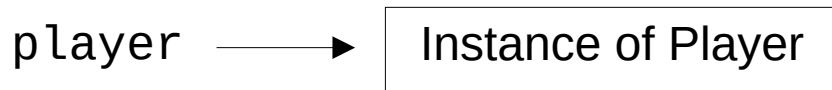
```
../main.cpp:82:33: error: call to implicitly-deleted copy constructor of 'std::unique_ptr<GamePlayer>'
  82 |     std::unique_ptr<GamePlayer> copyOfPlayer = _player;
      |                                     ^~~~~~
```

# std::unique\_ptr

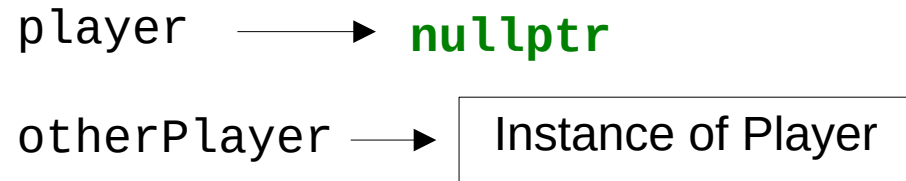
- It is, however, possible to *move* the pointer from one variable to another. The original unique\_ptr is automatically set to nullptr:

```
std::unique_ptr<Player> player = std::make_unique<Player>();  
std::unique_ptr<Player> otherPlayer = std::move(player);
```

Before:



After:



# std::unique\_ptr

- unique\_ptr is easiest to pass into a function by reference, as you avoid having to use std::move():

```
void alsoUsePrinter(std::unique_ptr<Printer>& ref) {  
    ref->print();  
}
```

- Returning a unique\_ptr from a function does not require you to use std::move()

```
std::unique_ptr<Printer> createPrinter() {  
    std::unique_ptr<Printer> printer {new Printer()};  
    return printer;  
}
```

# std::unique\_ptr

- It is possible to create a std::vector containing unique\_ptr, but you need to use either std::move() to move an existing pointer, or emplace\_back() to create a new element

```
std::vector<std::unique_ptr<std::string>> strings;
```

```
// Move an existing
```

```
std::unique_ptr<std::string> text {new std::string("Hello")};  
strings.push_back(std::move(text));
```

```
// Create a new element
```

```
strings.emplace_back(new std::string("Hello"));
```

# Today

- Pointers (continued)
- Pointers: risks
- `std::unique_ptr`
- **`std::shared_ptr`**

# std::shared\_ptr

- Used in the same way as unique\_ptr, except it can be copied

```
std::shared_ptr<Printer> shared = std::make_shared<Printer>();  
std::shared_ptr<Printer> copyOfShared = shared; // no problem!
```

- shared\_ptr counts how many copies of the pointer exist. When no more copies exist, the referenced memory is deleted.
- use\_count ( ) returns the number of copies that exist:

```
std::cout << shared.use_count() << std::endl;
```



# `std::unique_ptr` or `std::shared_ptr`?

- Use `std::unique_ptr` as much as possible
- Otherwise, use `std::shared_ptr`
  - Motivation: creating a `std::shared_ptr` allocates some memory, which when done often is costly
- Only use «raw» pointers (e.g. `int*`) when a library demands it



# Today

- Pointers (continued)
- Pointers: risks
- `std::unique_ptr`
- `std::shared_ptr`
- **`std::unordered_map`**

# std::unordered\_map and std::map

- Maps connect a unique “key” value to an associated and not necessarily unique “value”
- Include the <map> or <unordered\_map> header
- The C++ equivalent of a dictionary in Python

Jalapeno	→	5,000
Serrano	→	15,000
Cayenne	→	40,000
Ghost pepper	→	900,000
Carolina reaper	→	2,200,000

# std::unordered\_map and std::map

- Declaring a map:

```
std::unordered_map<std::string, std::string> opposites;
```

Data type of key values

Data type of mapped values

- Insert a value into the map:

```
opposites.insert({"true", "false"});
```

```
opposites.insert({"false", "true"});
```

- Read a value:

```
std::string value = opposites.at("false");
```

```
std::cout << value << std::endl; // prints true
```

# std::unordered\_map and std::map

- **Do NOT use the [ ] operator!**

```
std::unordered_map<std::string, double> strengths;  
if(strengths["Cayenne"] < 1000) {  
    std::cout << "Cayenne is under 1000" << std::endl;  
}
```

When using the [ ] operator, if the value being requested is not in the map, *it is created implicitly* even when you are not explicitly assigning a value.

Always use insert() and at() instead.

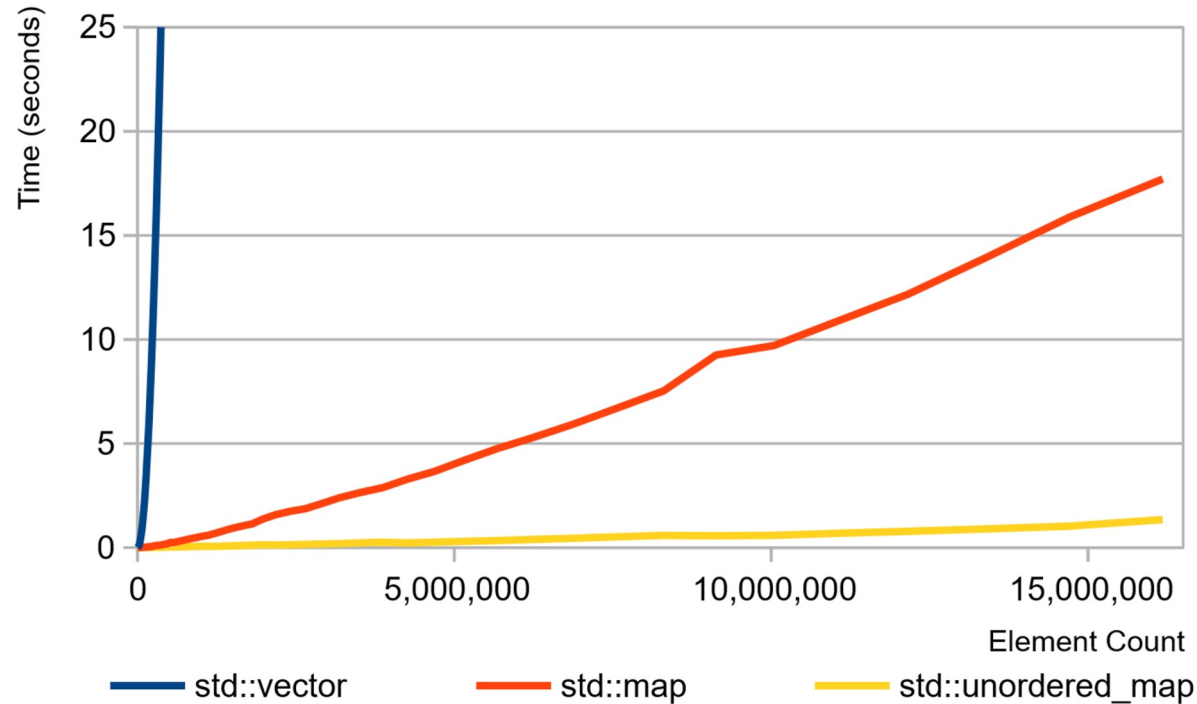
# std::unordered\_map and std::map

Initialisation and iterating over contents:

```
std::unordered_map<std::string, double> strengths {  
    {"Jalapeno", 5000.0},  
    {"Serrano", 15000.0},  
    {"Cayenne", 40000.0}  
};  
  
for(const std::pair<std::string, double> entry : strengths)  
{  
    std::cout << entry.first << " -> "  
               << entry.second << std::endl;  
}
```

# `std::unordered_map` is usually faster than `std::map`

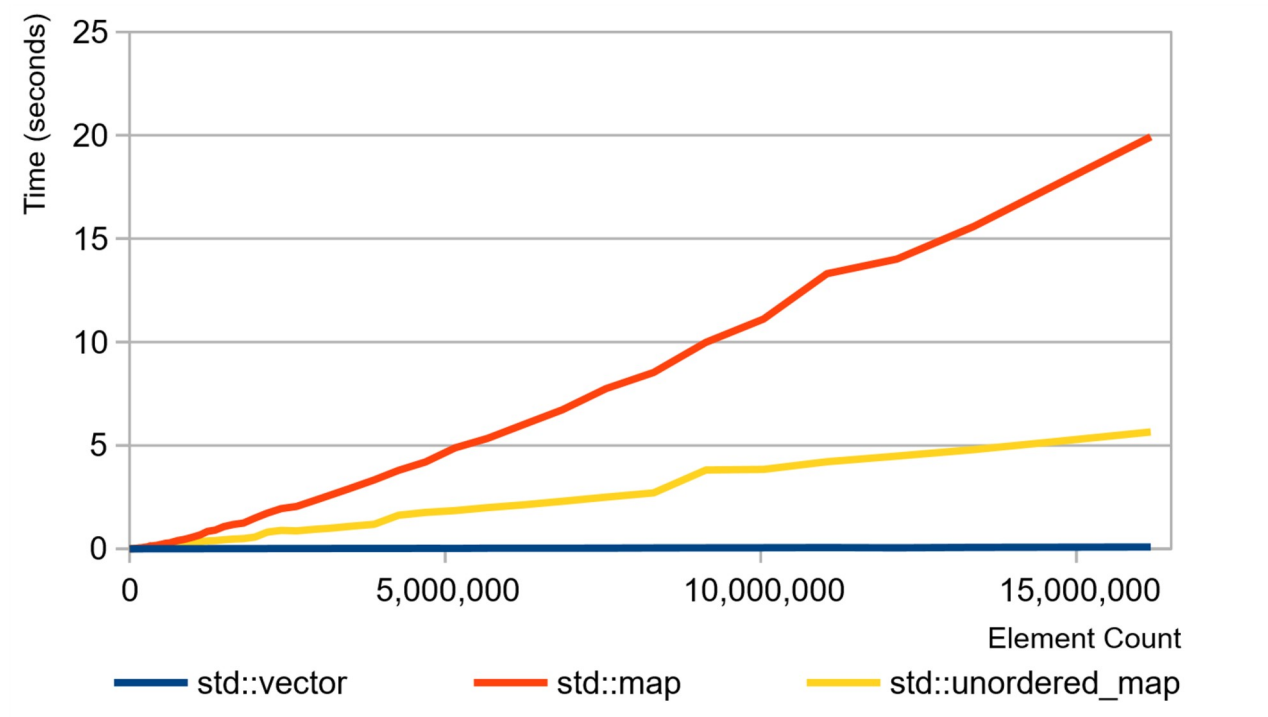
Searching / retrieving each element





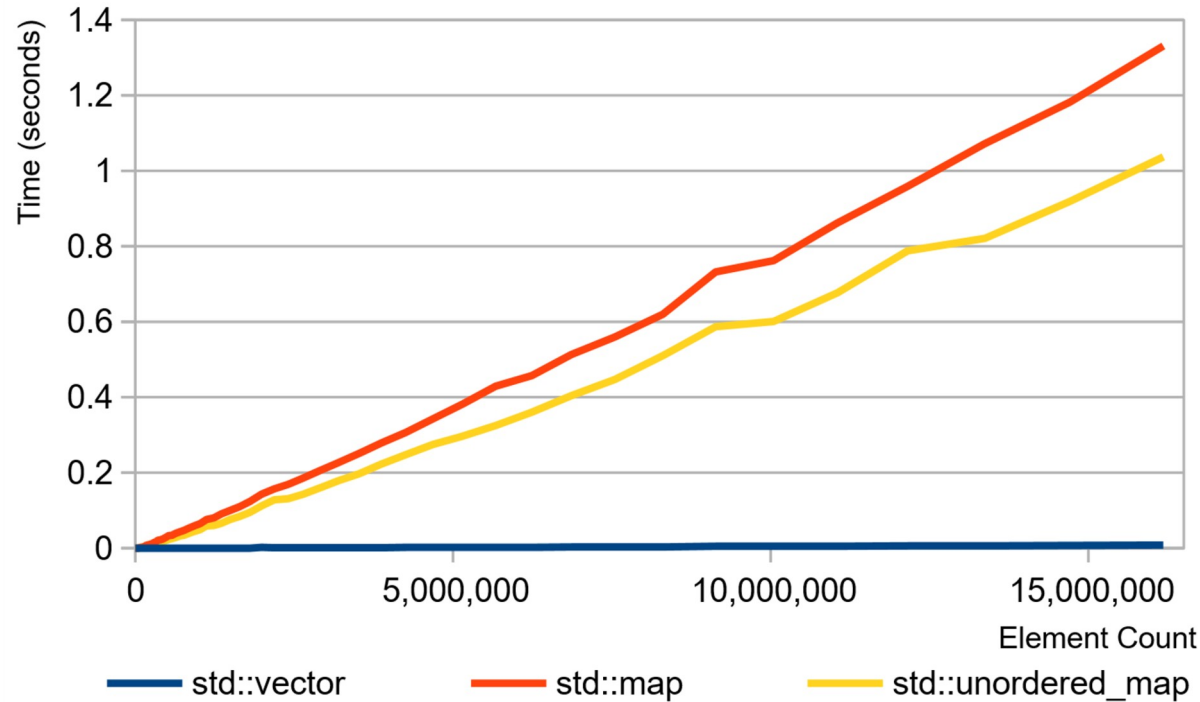
# `std::unordered_map` is usually faster than `std::map`

## Inserting elements



# `std::unordered_map` is usually faster than `std::map`

Iterating over all elements



# Today

- More on pointers
- `std::unique_ptr`
- `std::shared_ptr`
- `std::unordered_map`

Next week

***GOTTA GO FAST***