

Lecture 10.5: Pointyer Pointers

Bart Iver van Blokland

(Rune Sætre)

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

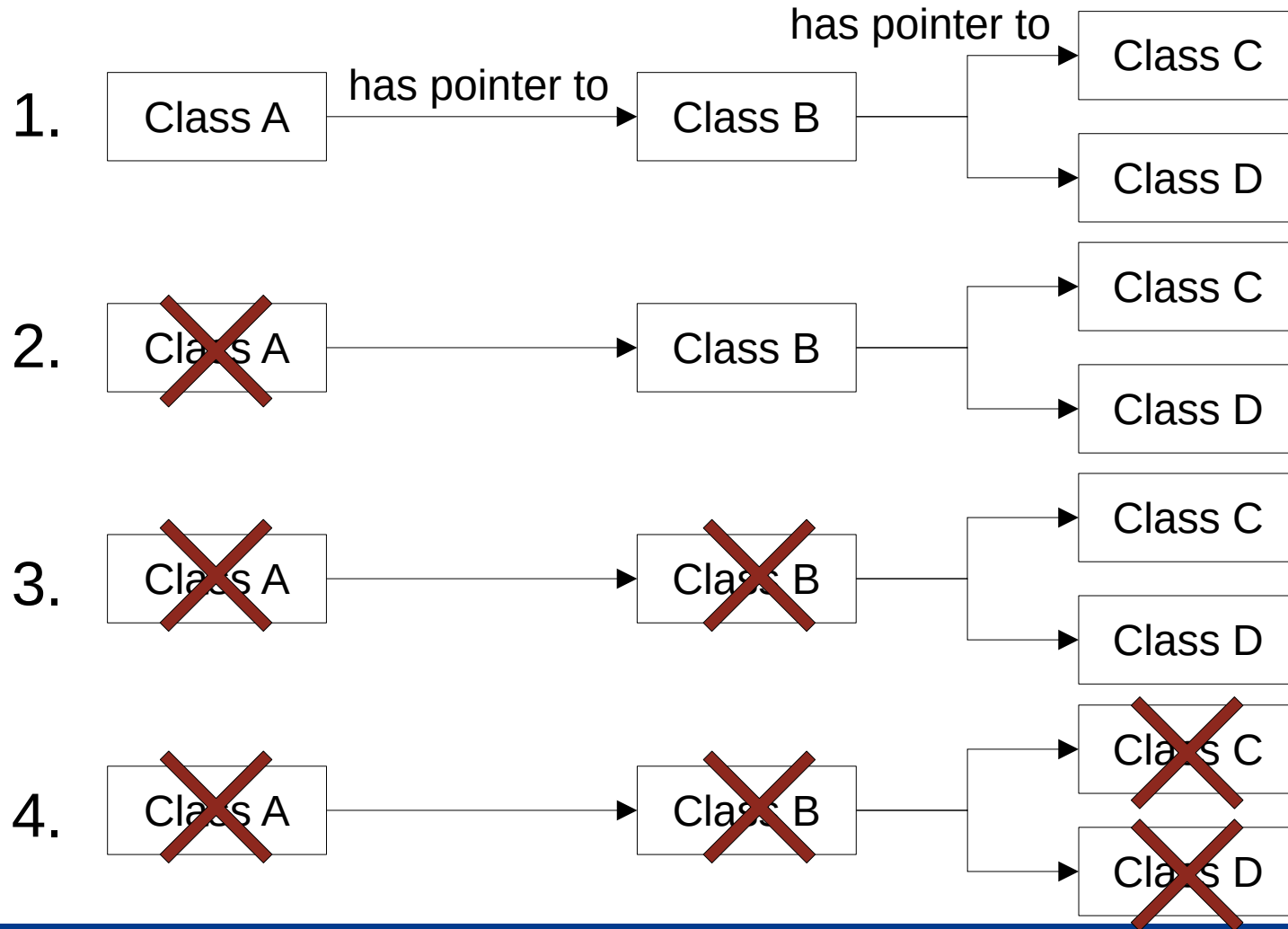


How does `std::unique_ptr` work?

- `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!
```

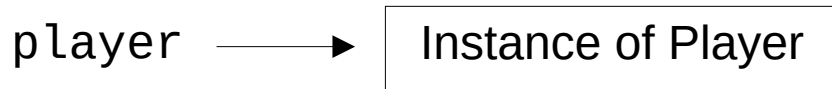
```
../main.cpp:82:33: error: call to implicitly-deleted copy constructor of 'std::unique_ptr<GamePlayer>'
  82 |     std::unique_ptr<GamePlayer> copyOfPlayer = _player;
      |                                     ^~~~~~
```


std::unique_ptr

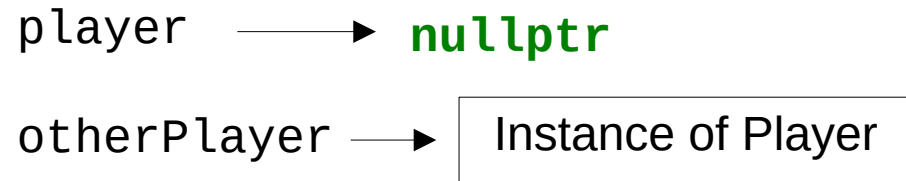
- It is, however, possible to *move* the pointer from one variable to another. The original unique_ptr is automatically set to nullptr:

```
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

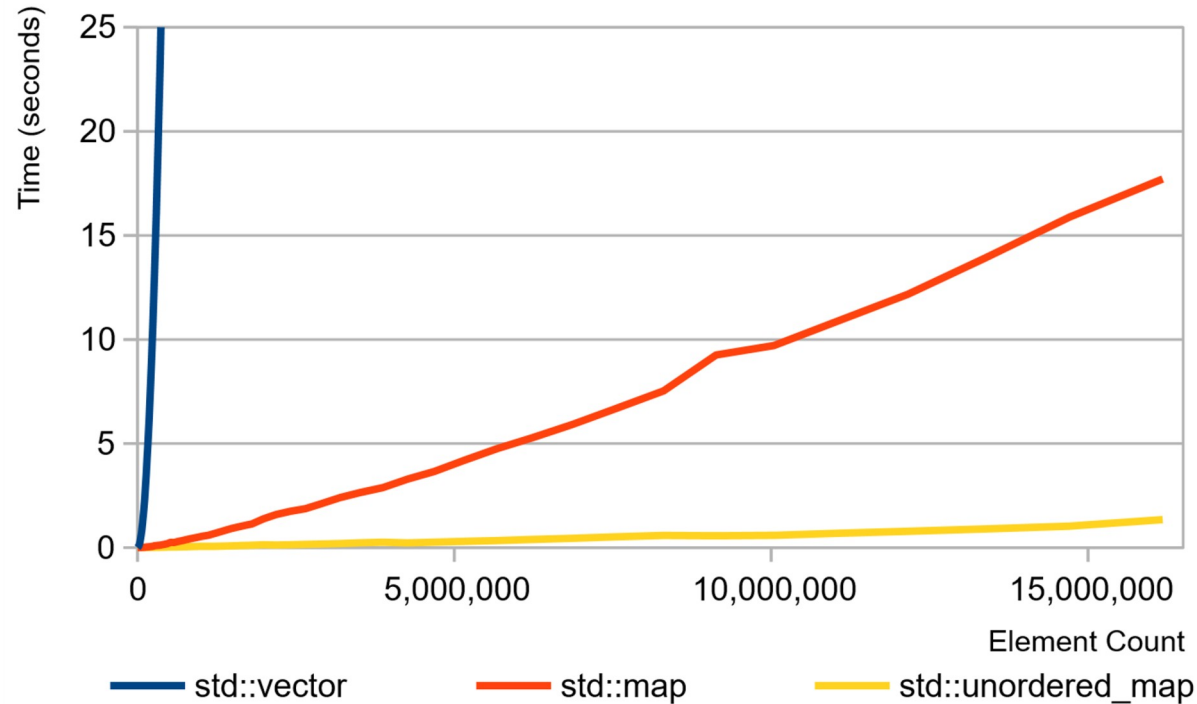
Initialisation and iterating over contents:

```
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;
}
```

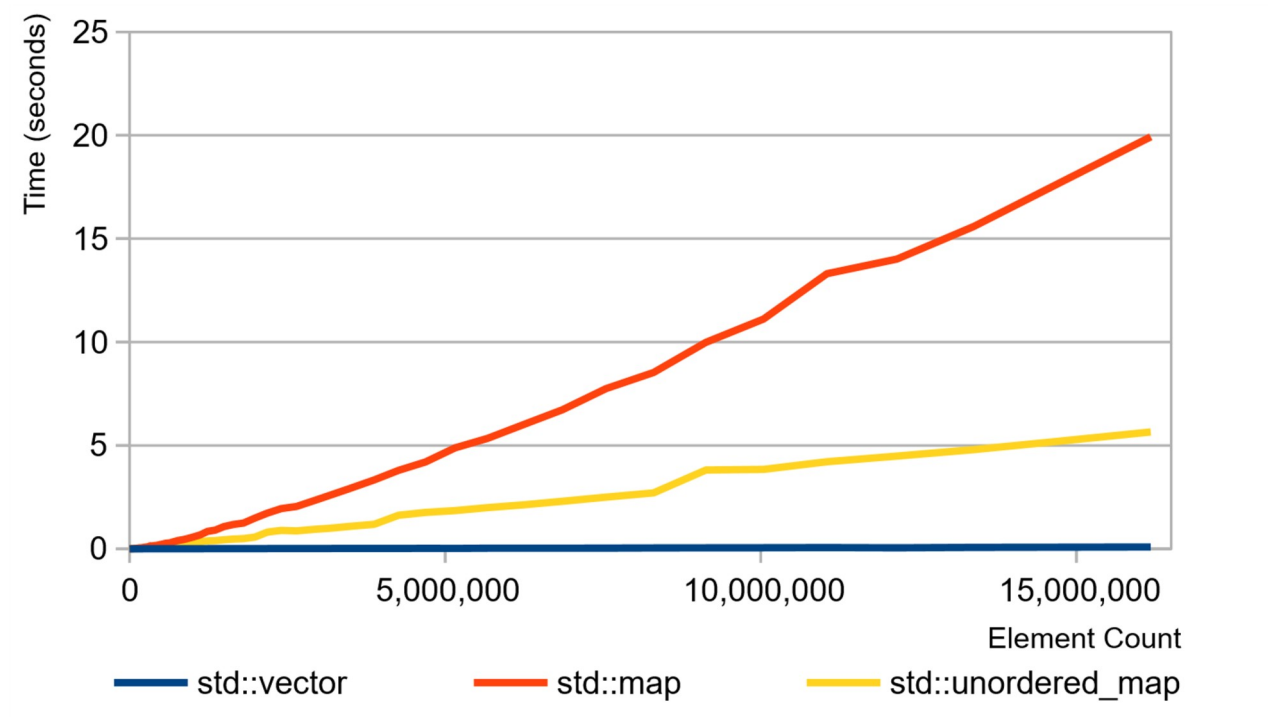
`std::unordered_map` is usually faster than `std::map`

Searching / retrieving each element



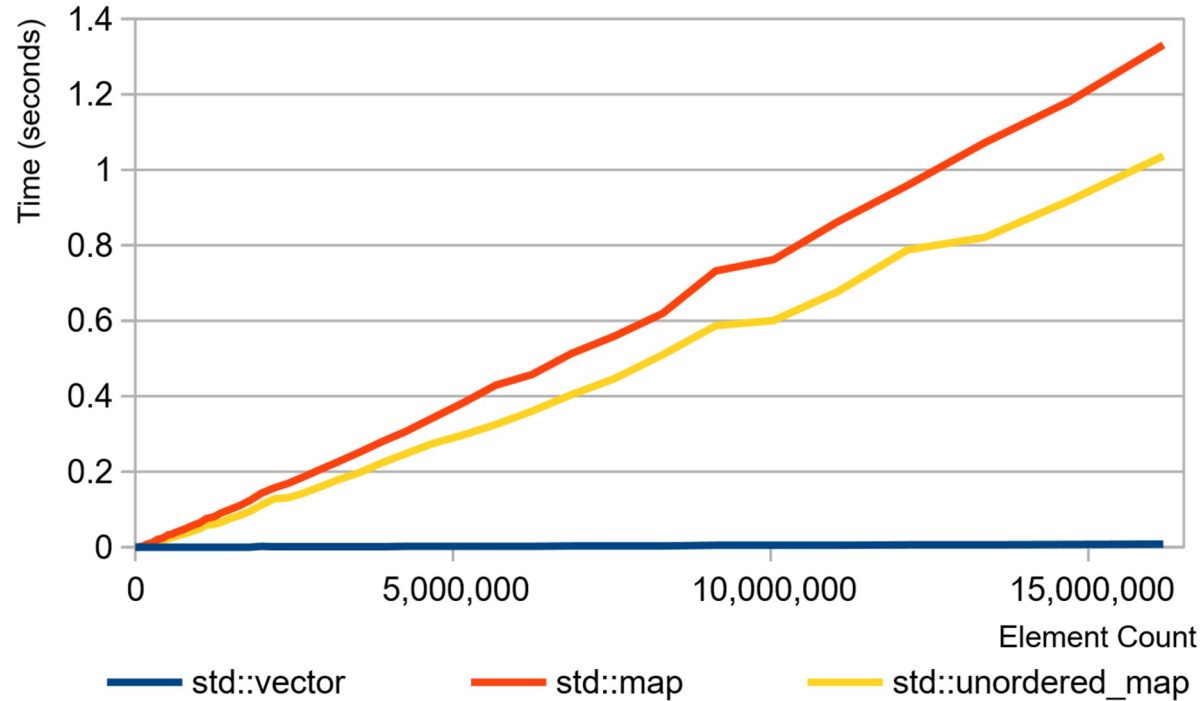
`std::unordered_map` is usually faster than `std::map`

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