

Lecture 10: Pointer Pointers

Bart Iver van Blokland

PSA 1 / 2: Kompendium is now available

- You can find it in the left hand sidebar on Blackboard

TDT4102 Prosedyre- og objektorientert programmering

Vår egen nettside finner du her: tdt4102.idi.ntnu.no

Uke 9:

Når	Hva	Hvor
Tirsdag kl. 08:15-10:00	Om øving 7	F1
Tirsdag kl. 14:15-16:00	Forelesning 10	R1
Torsdag kl. 12:15-14:00	Interaktiv: Ønskeforelesning	R1

My Announcements

TDT4102 Prosedyre- og objektorientert programmering (2025 VÅR)

> Kunngjøring uke 9

Lenker / Links

Checkmark, Email, Megaphone, Calendar, Group, Info icons

PSA 2 / 2: Project information

- Purpose: get experience with writing a program from scratch
- You can do what you like, however, you must:
 - Use the course's graphics library (AnimationWindow)
 - Use inheritance somewhere in your project
 - Use exception handling somewhere in your project
 - Read and write something from/to a file
 - Write some documentation for your work
 - Deliver a program that compiles
- Can count as 1-3 assignments
- Can work alone or in pairs
 - For 2 people we expect roughly 2x the work
- Deadline: 11th of March
- We will have awards for the best projects

Do you remember?

- What is a pointer?
- How is a pointer declared?
- What is the difference between stack and heap allocation?
- What is one risk with heap allocated values?

Do you remember?

- Where are the variables `i`, `vec`, and `element` allocated?
- When are they deallocated?
- The vector and string both contain heap-allocated data. Which mechanism is used to deallocate that?

```
void printVector(std::vector<int> vec) {  
    for(int i = 0; i < vec.size(); i++) {  
        std::string element = std::to_string(vec.at(i));  
        std::cout << element << ", ";  
    }  
}
```

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
- 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.

A few more things about last week

- Why pointers and heap memory?
- Difference between a pointer and a reference?
- One small bit of pointer syntax

Why pointers and heap memory?

- Stack is often too small (usually ~1 MB), heap can use whatever your computer has in RAM capacity (> 4GB)
- Stack allocated variables must all be declared at compile time. You don't always know how many you will need.
- Polymorphism (inheritance) requires that objects are being referenced
- **BUT:** use the stack as much as possible, because deallocation is automatic

Difference between pointers and references

	Pointers	References
References another value	Yes	Yes
Can reference a value that does not exist	Yes	Yes, but easier to do with a pointer
Can be set to nullptr	Yes	No (big advantage!)
Can modify the address being referenced	If not const	No (big advantage!)
Possible to create a vector or array containing these	Yes (big advantage!)	No
Need to dereference the reference explicitly	Yes	No

Best practice: use references when you have a choice!

Syntax: pointers to members of objects

- You can use the -> operator to access member variables or methods of an object

```
std::vector<int> integers(10);  
std::vector<int>* pointerToIntegers = &integers;
```

```
pointerToIntegers->at(3) = 8;
```

// These two lines are equivalent:

```
std::cout << (*pointerToIntegers).at(3) << std::endl;  
std::cout << pointerToIntegers->at(3) << std::endl;
```

Today..

What is an issue with this program?

```
Car* createCar(std::string model) {  
    Car* car = new Car(model);  
  
    if(name.empty()) {  
        return nullptr;  
    }  
  
    return car;  
}
```

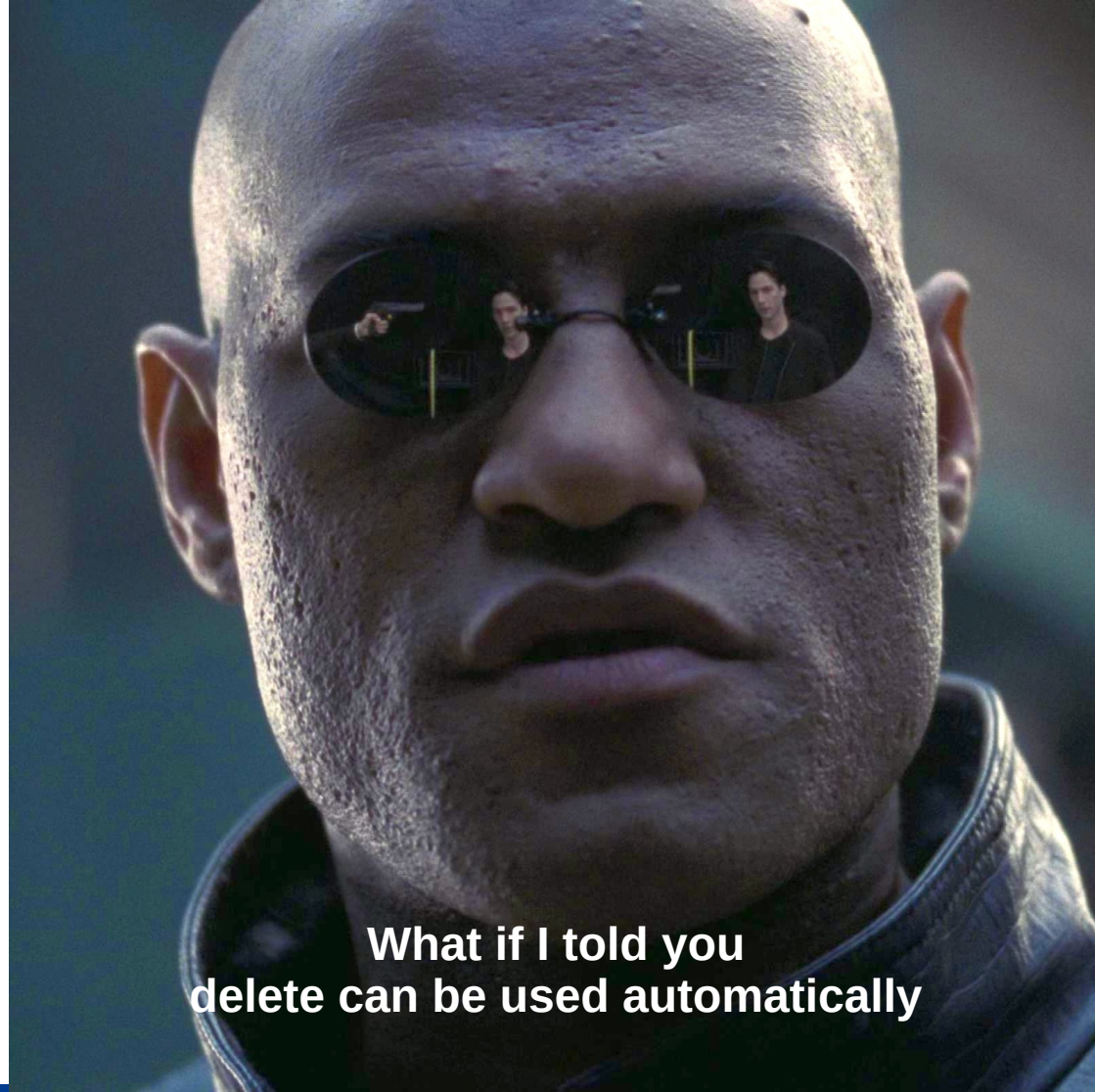
There are problems..

The delete statement may not always end up being run:

```
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.



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

Kaboom?



Yes, Rico, Kaboom



DESTRUCTOR



Today: cleaning up mess automatically



Today

- **std::unique_ptr**
- std::shared_ptr
- Graphical User Interface (GUI)

std::unique_ptr

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

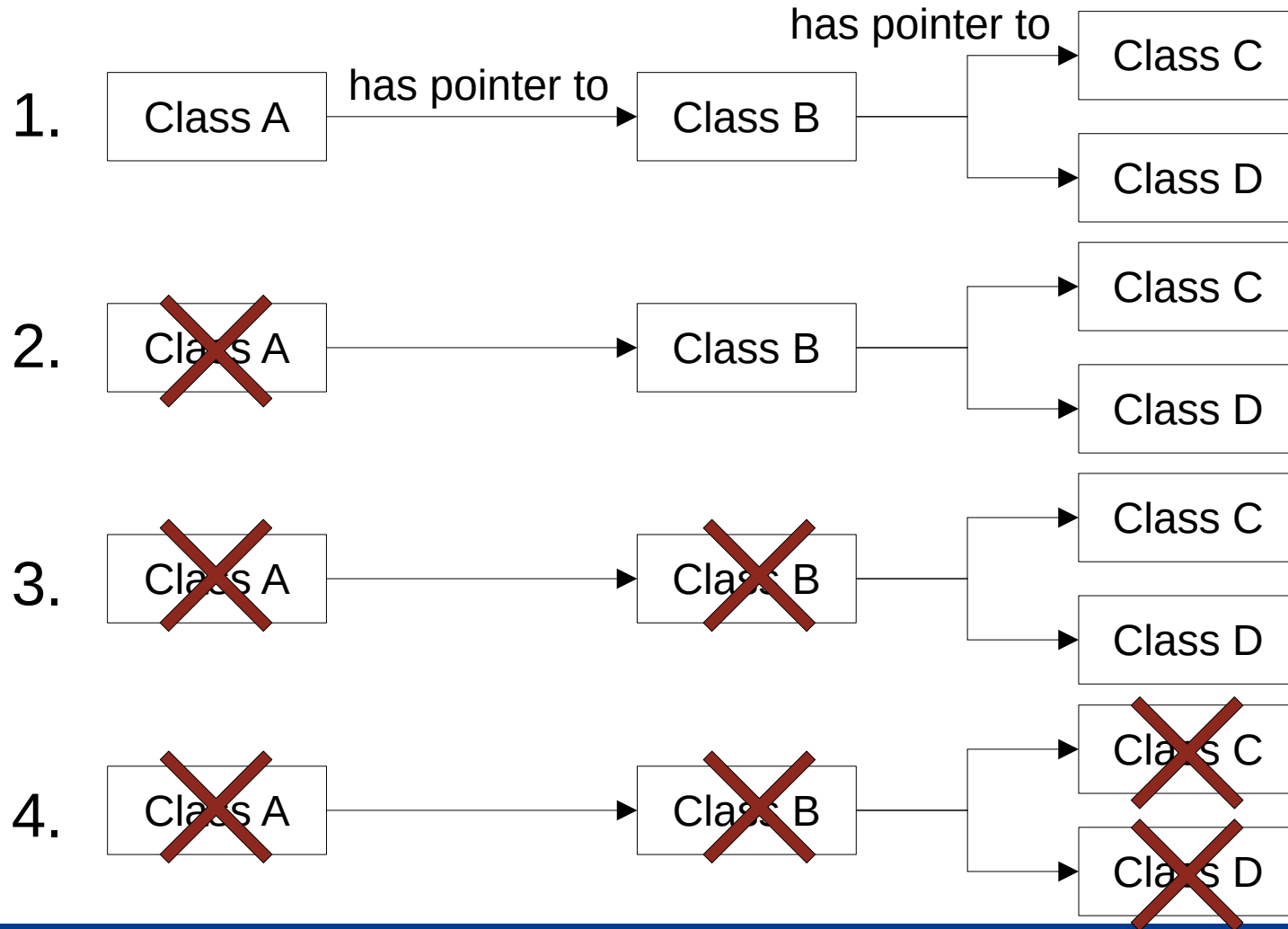
```
void doWork() {  
    std::unique_ptr<Printer> {new Printer()};  
}
```

Note: no * behind Printer

Memory is allocated here..

.. And deleted here automatically by the destructor of std::unique_ptr

- 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<Printer> printer = std::make_unique<Printer>();
```

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

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

```
printer->print();  
(*printer).print();
```

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<Printer> copyOfPrinter = printer; // error!
```

- 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<Printer> printer = make_unique<Printer>();  
std::unique_ptr<Printer> otherPrinter = std::move(printer);  
// printer now points to nullptr
```

Memory ownership

- A `unique_ptr` is responsible for deleting the heap memory it references.

This is usually called 'owning' that memory.

- `std::move()` transfers that responsibility along with the pointer

```
void doWork(std::unique_ptr<int> pointer) {  
    *pointer = 5;  
}  
void callDoWork() {  
    std::unique_ptr<int> number {new int{4}};  
    // Ownership of 'number' is transferred to doWork()  
    doWork(std::move(number));  
}
```


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

- `std::unique_ptr`
- **`std::shared_ptr`**
- Graphical User Interface (GUI)

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;
```

Task: what could go wrong?

- There exists a scenario in which a `std::shared_ptr` can cause a memory leak. How could this happen?
- Reminder: a memory leak occurs when heap memory is not deallocated, and the reference(s) to that memory are lost by the program

```
void leakMemory() {  
    int* array = new int[100];  
}
```

← only reference is deallocated
when function ends

- Information from last slide, for reference:
 - Used in the same way as `unique_ptr`, except it can be copied
 - `shared_ptr` counts how many copies of the pointer exist. When no more copies exist, the referenced memory is deleted.

`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

- `std::unique_ptr`
- `std::shared_ptr`
- **Graphical User Interface (GUI)**

Graphical user interfaces

- A much more familiar way to interact with a program!
- Motivation:
 - Easier to use (as a user) than the terminal
 - Good example of event-driven programming
- Events are handled through «callback» functions
 - We only do something when the user interacts with the interface. Our program is idle otherwise



User interfaces

- AnimationWindow has different widgets available:

TDT4102::Button



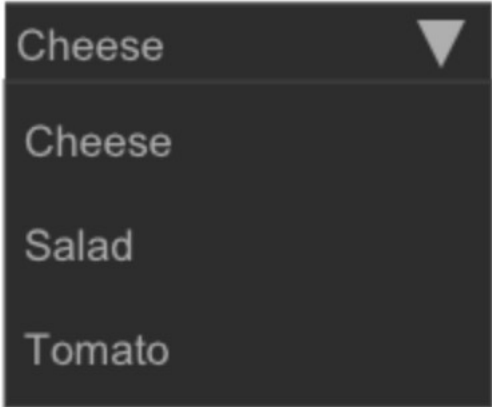
Click me!

TDT4102::TextInput



Text input box

TDT4102::DropDownList



Cheese

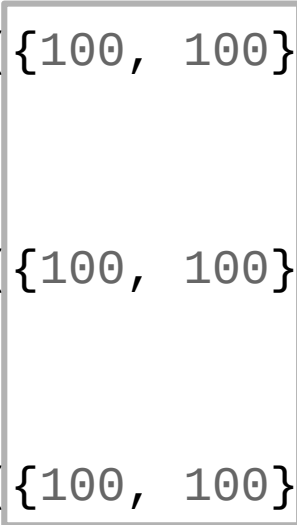


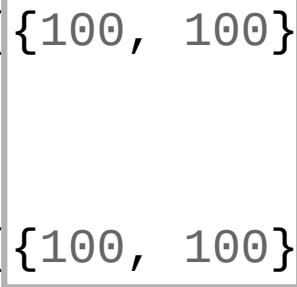





Cheese

Salad

Tomato

User interfaces

- Creating widgets

	Location	Width	Height	
<code>TDT4102::Button({100, 100},</code>				<code>"Click me!")</code> ← Button label text
<code>TDT4102::TextInput({100, 100},</code>				<code>"Change me!")</code> ← Initial text
<code>TDT4102::DropDownList({100, 100},</code>				<code>dropdownOptions)</code> ← String vector with possible options

Building an interface

```
void buttonClicked() {  
    std::cout << "Clicked!" << std::endl;  
}
```

A callback function must return void, and have no parameters

```
int main() {  
    TDT4102::AnimationWindow window;  
    TDT4102::Button button({100, 100}, 140, 50, "Continue");  
  
    window.add(button);  
    button.setCallback(&buttonClicked);  
    window.wait_for_close();  
}
```

Adding the widget makes it available in the interface

Use the & operator to get a pointer to the callback function



From here on out we just do stuff when the user does something

std::bind

- You will need to use a workaround in the assignment to use a member function as a callback function

```
class GUIWindow : TDT4102::AnimationWindow {
    TDT4102::TextInput textInput;

    void textChanged() {
        std::cout << textInput.getText() << std::endl;
    }
public:
    GUIWindow() : textInput({100, 150}, 350, 50, "Change me!") {
        add(textInput);
        textInput.setCallback(std::bind(&GUIWindow::textChanged, this));
    }
};
```

Callback pointer goes here  **this** goes here 

Today

- `std::unique_ptr`
- `std::shared_ptr`
- Graphics User Interface (GUI)

Next week

GOTTA GO FAST