Lecture 11: Pointer Pointers

Bart Iver van Blokland

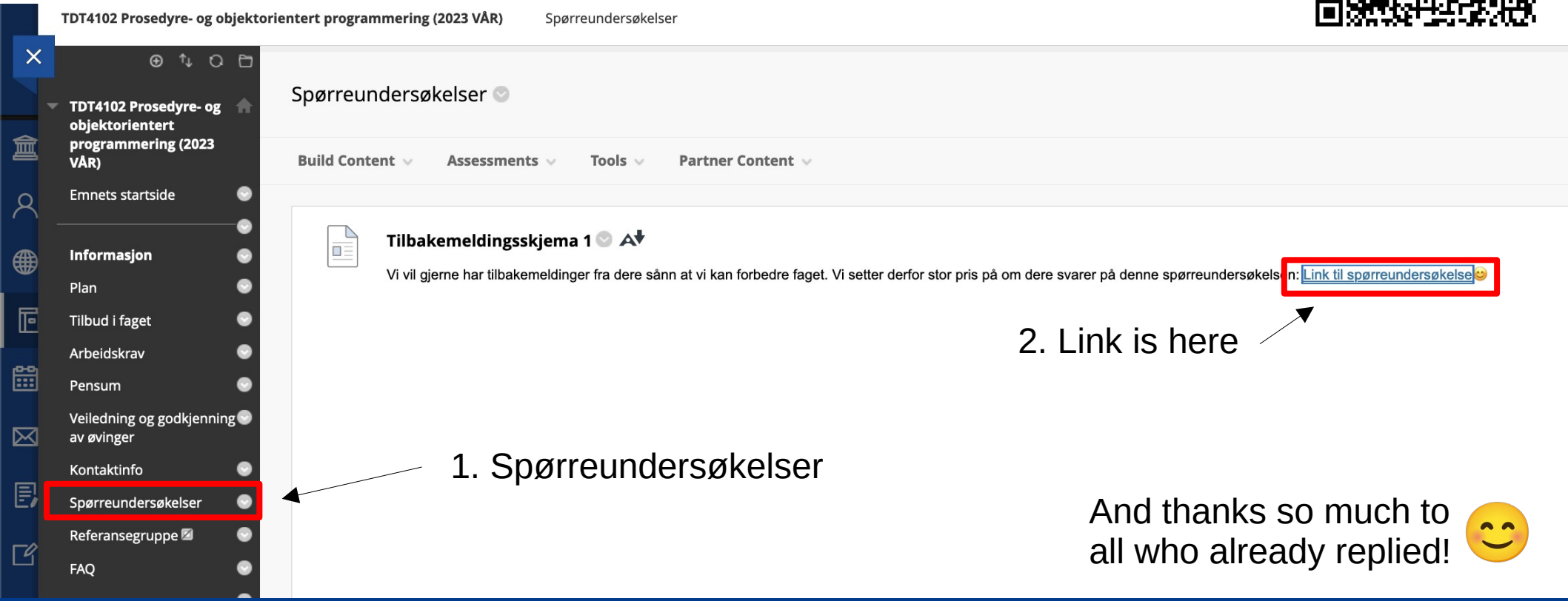
(Dragana Laketic)

PSA 1 / 3: Next week is a guest lecture

- Einar Johan Trøan Sømåen
- Staff Engineer at ARM Trondheim
  - ARM designs processors which are used in practically all phones and tablets
- Will be talking about software testing and exception handling

PSA 2 / 3: Please tell us what you think!

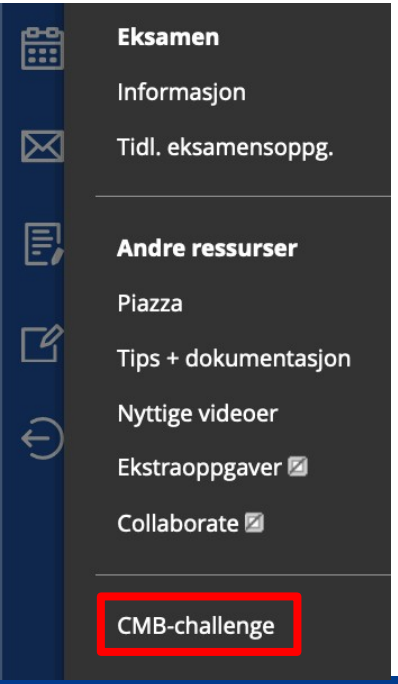
- We use this instead of a reference group
- Link is on Blackboard, or use the QR code



And thanks so much to all who already replied! 😊

PSA 3 / 3: Climbing Mont Blanc

- Remember to participate in Climbing Mont Blanc!
- Prizes for best submissions, as well as one drawn randomly between everyone who solves at least the two easiest tasks
- Ends on the 24th of March



Information is on the CMB-challenge page on Blackboard

Last week

- Memory Management
- Pointers
- Scope
- Memory Allocation / Deallocation
- DESTRUCTORS
- Copy Constructor

Today..

Today

- **More on destructors**
- `unique_ptr`
- `shared_ptr`
- Graphical User Interface (GUI)
- static


DESTRUCTOR

Destructors

- Destructors are functions that are automatically run when an object is deleted

- Syntax is the same as constructors, except the name has ~ in front, and it cannot have parameters.

```
class Houston {
    Satellite* satellite;
public:
    Houston() {
        satellite = new Satellite("sputnik");
    }
    ~Houston() {
        delete satellite;
    }
};
```

The desctructor is executed automatically when an instance of Houston is deleted

Example 1

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` ceases to exist 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;
}
```

Destructor of `message` is called when **`delete`** is used

Example 1

Destructors

- If you always delete memory allocated using **new** / **new**`[]`, **destructors will always be called**

- 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.

Example 1

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 next week

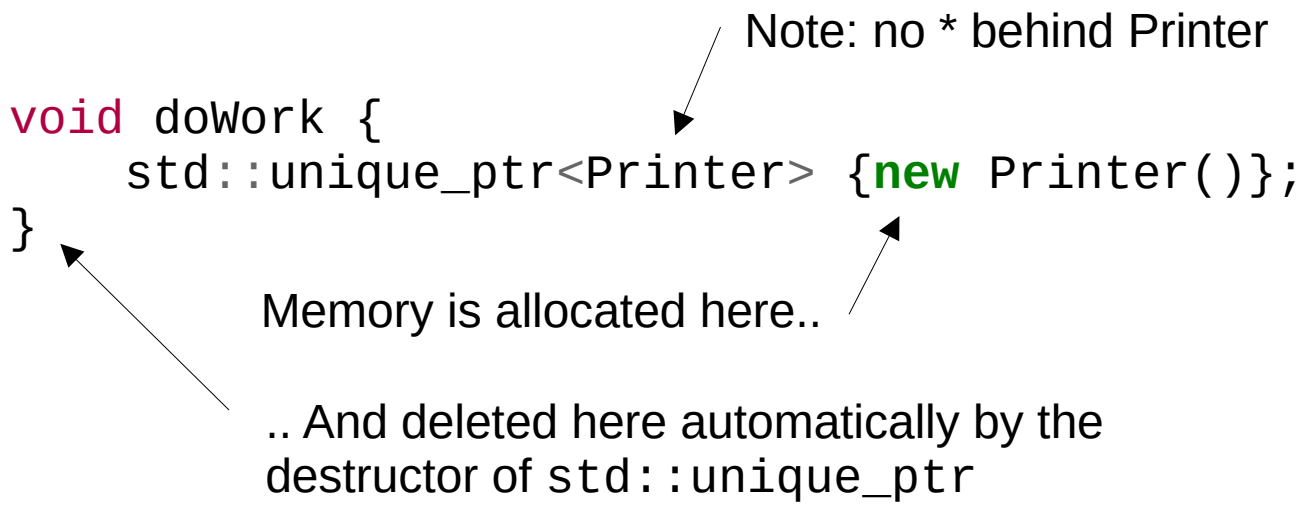

What if I told you
delete can be used automatically

Today

- More on destructors
- **`unique_ptr`**
- `shared_ptr`
- Graphical User Interface (GUI)
- static

unique_ptr

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



Example 2

unique_ptr

- A slightly more efficient way to create a unique_ptr is using the make_unique function:

```
std::unique_ptr<Printer> printer = 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();
```

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. Afterwards it is no longer possible to use the original variable:

```
std::unique_ptr<Printer> otherPrinter = std::move(printer);
```

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;
}
```

Today

- More on destructors
- `unique_ptr`
- **`shared_ptr`**
- Graphical User Interface (GUI)
- static

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:

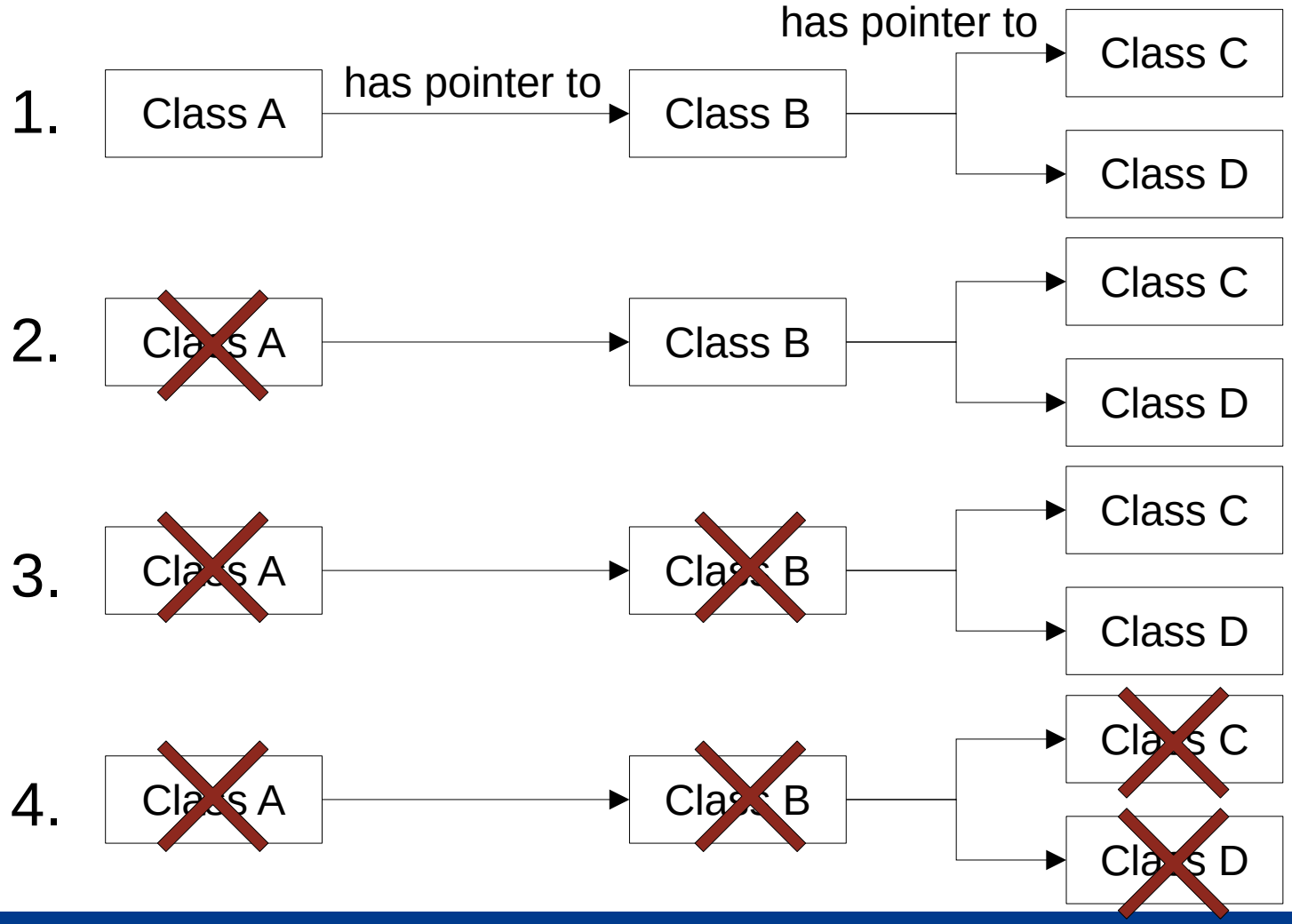
```
std::cout << shared.use_count() << std::endl;
```

Example 2

`unique_ptr` or `shared_ptr`?

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

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



Example 3

Today

- More on destructors
- `unique_ptr`
- `shared_ptr`
- **Graphical User Interface (GUI)**
- static

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

Example 4

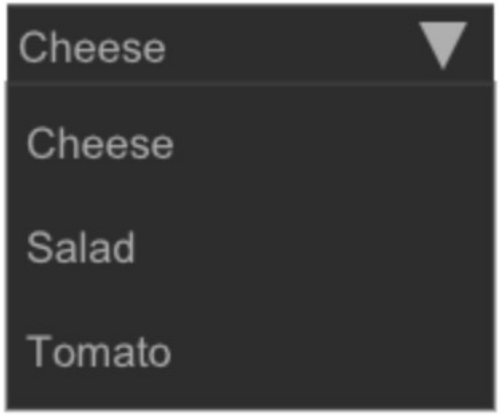
User interfaces

- AnimationWindow has different widgets available:

`TDT4102::Button`

`TDT4102::TextInput`

`TDT4102::DropdownList`



User interfaces

- Creating widgets

Location Width Height

```
TDT4102::Button({100, 100}, 100, 30, "Click me!")
```
Button label text

```
TDT4102::TextInput({100, 100}, 350, 50, "Change me!")
```
Initial text

```
TDT4102::DropdownList({100, 100}, 200, 50, dropdownOptions)
```
String vector with possible options

Building an interface

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

int main() {
    TDT4102::AnimationWindow window;
    TDT4102::Button button({100, 100}, 140, 50, "Continue");

    window.add(button);

    button.setCallback(&buttonClicked);

    window.wait_for_close();
}
```

A callback function must return void, and have no parameters

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

- Creating a class which inherits from AnimationWindow requires using a workaround to set a callback function

```
class GUIWindow : public 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

- More on destructors
- `unique_ptr`
- `shared_ptr`
- Graphical User Interface (GUI)
- **static**

Static keyword

- The static keyword roughly means «there is only one»
- A static variable stays around between function calls, and the next time you call that function the value will be what you left it the last time around
- The value of a static field is the same in all instances of that class. Useful for defining constants used within the class.

```
int counter() {
    static int next = 0;
    next++;
    return next;
}
```

Example 5

Today

- More on destructors
- `unique_ptr`
- `shared_ptr`
- Graphics User Interface (GUI)
- static

Next week

- **Guest lecture!**
- Error handling
- Testing