

# Lecture 7: Object Orientation

Bart Iver van Blokland

# PSA: Student assistants

- Remember that student assistants are available between 8 – 18 from monday to friday!
  - Undass are also present between 10 – 14
- Assistants are mostly in A3-138
- We have a good amount of spare capacity. Please come on by and get help with the assignments!

# Do you remember?

```
void doStuff()  
{  
    std::string greeting = "Well hello there!";  
    std::string message = greeting;  
    greeting = "Good afternoon!";  
  
    std::string& otherMessage = message;  
    otherMessage = "And now for something completely different.";   
  
    std::string& oneMoreMessage = greeting;  
    oneMoreMessage = otherMessage;  
}
```

# Do you remember?

```
void functionA(std::vector<int> list) {  
    list.push_back(10);  
}
```

```
void functionB(std::vector<std::string>& list) {  
    list.push_back("Test");  
}
```

```
void functionC(const std::vector<std::string>& list) {  
    list.push_back("Burp");  
}
```

```
void doStuff() {  
    std::vector<int> numbers = {1, 2, 3};  
    functionA(numbers);  
  
    std::vector<std::string> strings = {"word1", "word2", "word3"};  
    functionB(strings);  
    functionC(strings);  
}
```

# Do you remember?

- What is a struct, and what it is used for?
- Where should a struct be defined?
- The compiler gives you this error. What is the problem?

Windows:

```
/usr/bin/ld: program.p/main.cpp.o: in function main':  
/home/bart/shenanigans/main.cpp:6:(.text+0x9):  
    undefined reference to doStuff()'   
collect2: error: ld returned 1 exit status  
ninja: build stopped: subcommand failed.
```

Mac:

```
Undefined symbols for architecture arm64:  
  "doStuff()", referenced from:  
      _main in main.cpp.o  
ld: symbol(s) not found for architecture arm64
```

# Today

- **Namespaces**
- Object-Oriented Programming
- Objects



# Crewmate

There is 1 Impostor among us



Your role is  
**Engineer**



Can use the vents



# Today

- **Namespaces**
- Object-Oriented Programming
- Objects

```
#include "std_lib_facilities.h"
```

```
struct vector {  
    float x = 0;  
    float y = 0;  
    float z = 0;  
};
```

```
int main() {  
    vector vec {5.7, 5.8, 5.9};  
    return 0;  
}
```

```
#include "std_lib_facilities.h"
```

```
struct vector {  
    float x = 0;  
    float y = 0;  
    float z = 0;  
};
```

```
../main.cpp:10:2: error: reference to 'vector' is ambiguous  
10 |         vector vec;  
    |         ^  
in ../main.cpp:3:8: note: candidate found by name lookup is 'vector'  
3 |     struct vector {  
    |     ^  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/__vector/vector.h:86:28: note: candidate found by name lookup is 'std::vector'  
86 |     class _LIBCPP_TEMPLATE_VIS vector {  
    |     ^  
1 error generated.
```

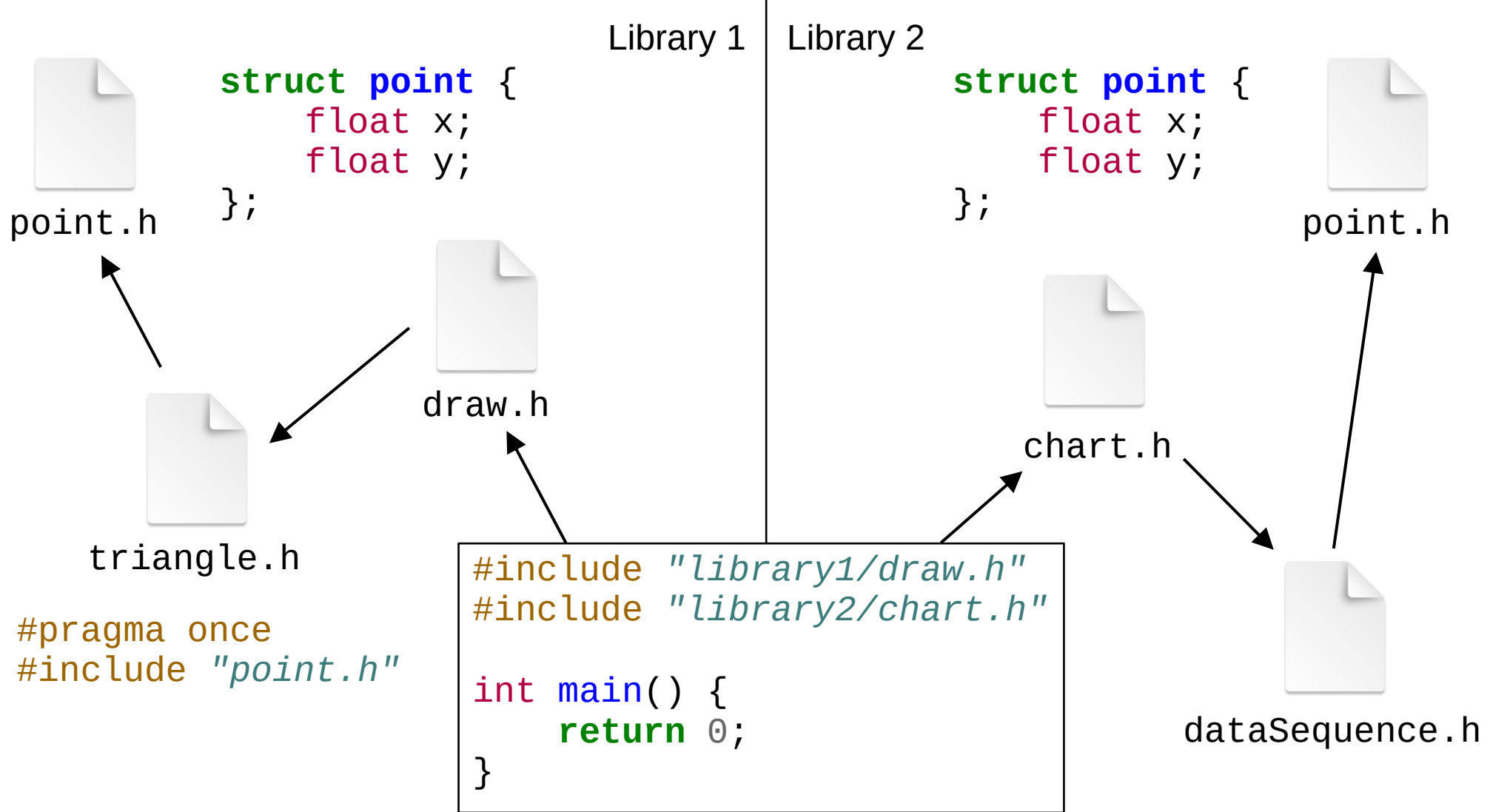
```
#include "std_lib_facilities.h"
```

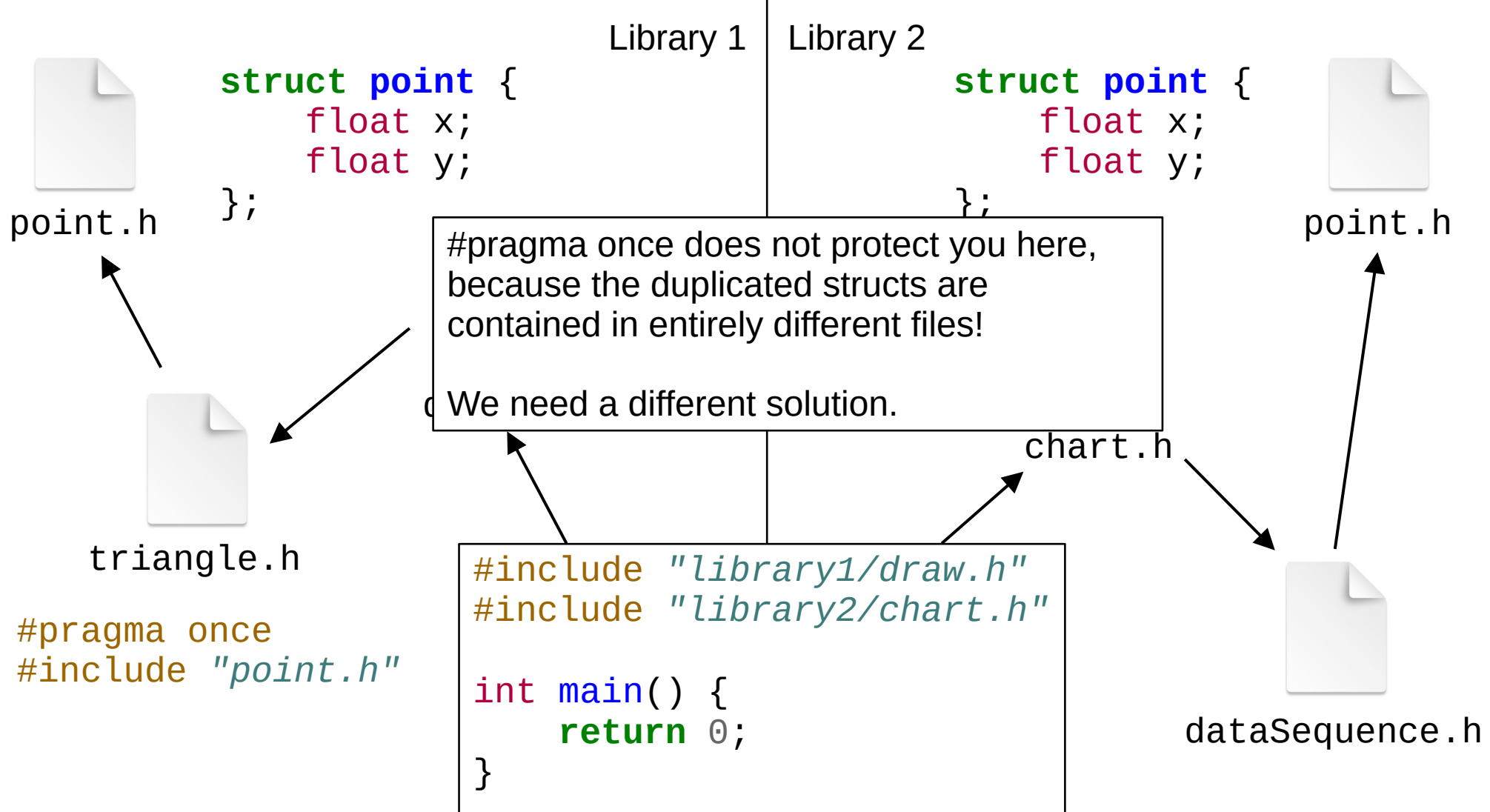
```
struct point {  
    float x;  
    float y;  
};
```

```
struct point {  
    float x;  
    float y;  
};
```

```
int main() {  
    return 0;  
}
```

```
../main.cpp:8:8: error: redefinition of 'point'  
      8 | struct point {  
        | ^  
../main.cpp:3:8: note: previous definition is here  
      3 | struct point {  
        | ^  
1 error generated.
```





# Namespaces

- Namespaces are a “directory” for data type and function names
  - By default, variables are added into the *global* namespace
- When using anything declared inside a namespace, you must specify explicitly in which namespace it resides

```
int main() {  
    // Function sayHello inside the German namespace  
    German::sayHello();  
    // Function sayHello inside the global namespace  
    sayHello();  
    return 0;  
}
```

# Namespaces

```
namespace German {  
    void sayHello() {  
        std::cout << "Guten Tag!" << std::endl;  
    }  
}  
  
void sayHello() {  
    std::cout << "Greetings!" << std::endl;  
}  
  
int main() {  
    German::sayHello();  
    sayHello();  
    return 0;  
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
  
Guten Tag!  
Greetings!
```



# Namespaces

- How to declare functions in a header file:

```
// In the header file:  
namespace German {  
    void sayHello();  
}
```

```
// In the .cpp file:  
void German::sayHello() {  
    std::cout << "Guten Tag!" << std::endl;  
}
```

- Data types (such as structs) can also be declared inside a namespace
  - That's why you write `std::` in front of `std::vector` and `std::string`

# Namespaces

- It is possible to declare a namespace inside of another namespace
  - Causes pretty long names, so not really recommended

```
namespace Amogus {  
    namespace Roles {  
        struct Engineer {  
            bool canVent = true;  
        };  
    }  
}  
  
// Data type: Amogus::Roles::Engineer
```

- The C++ standard library uses the std namespace
- Namespaces can't be declared inside a function

# Namespaces

- The using keyword can copy types in a namespace into the global namespace

```
namespace Amogus {  
    namespace Roles {  
        struct Engineer {  
            bool canVent = true;  
        };  
    }  
}  
  
using namespace Amogus::Roles;  
// Can now create an instance of Engineer  
// No need to specify the entire namespace  
  
using namespace std;  
// Often used in C++ examples and std_lib_facilities.h  
// Not recommended because it pollutes the global namespace
```

# Task: namespaces

- Using what you've seen about namespaces, the `using` keyword, and `structs`, find a way to get the compiler to produce the following error:

**error:** reference to 'Engineer' is ambiguous

- Managed to get the error? Fix the above error (get your program to compile again), but you may not remove any code you have already written in the first task.

# Today

- Namespaces
- **Object-Oriented Programming**
- Objects



# TDT4102 - Prosedyre- og objektorientert programmering



We have done this



Now it's time for this

# What's an object?

```
void count(Counter& counter) {  
    counter.count++;  
}
```

```
struct Counter {  
    int count = 0;  
};
```



# What's an object?

We have already seen a number of them previously!

```
std::vector  
std::array  
std::string  
std::random_device  
std::default_random_engine  
std::uniform_int_distribution
```

```
TDT4102::AnimationWindow
```

More concretely: **objects are a combination of data (variables often hidden inside the object), and functions that modify that data in specific ways.**

# Example: Procedure-Oriented Programming

- Procedure-Oriented Programming:
  - Data is stored inside functions
  - Functions apply modifications on data provided to them as a parameter

```
void computeAverage(const std::vector<double>& numbers) {  
    double sum = 0;  
    for(int i = 0; i < numbers.size(); i++) {  
        sum += numbers.at(i);  
    }  
    return sum / numbers.size();  
}  
  
int main() {  
    std::vector<double> list = {5.7, 5.8, 5.9, 5.7, 5.9};  
    computeAverage(list);  
    return 0;  
}
```

# Example: Object-Oriented Programming

- Object-Oriented Programming:
  - Data is hidden within objects
  - Objects have functions that modify the data within, and thus define what is allowed to be done with that data

```
int main() {  
    AnimationWindow window;  
    window.draw_circle({100, 100}, 50);  
    window.wait_for_close();  
    return 0;  
}
```


# Comparison

## Object-Oriented Programming

```
int main() {  
    AnimationWindow window;  
    window.draw_circle({100, 100}, 50);  
    window.wait_for_close();  
    return 0;  
}
```

The **object** controls what happens to the data, and can define if and how it is allowed to be modified (example: cannot change width)

## Procedure-Oriented Programming

```
int main() {  
    AnimationWindow window = open_wimdown();  
    draw_circle(window, {100, 100}, 50);  
    window.width = 50;   
    wait_for_close(window);  
    return 0;  
}
```

The **function** controls what happens to the data, and is free to modify it however it wants

# Object-Oriented Programming - Advantages

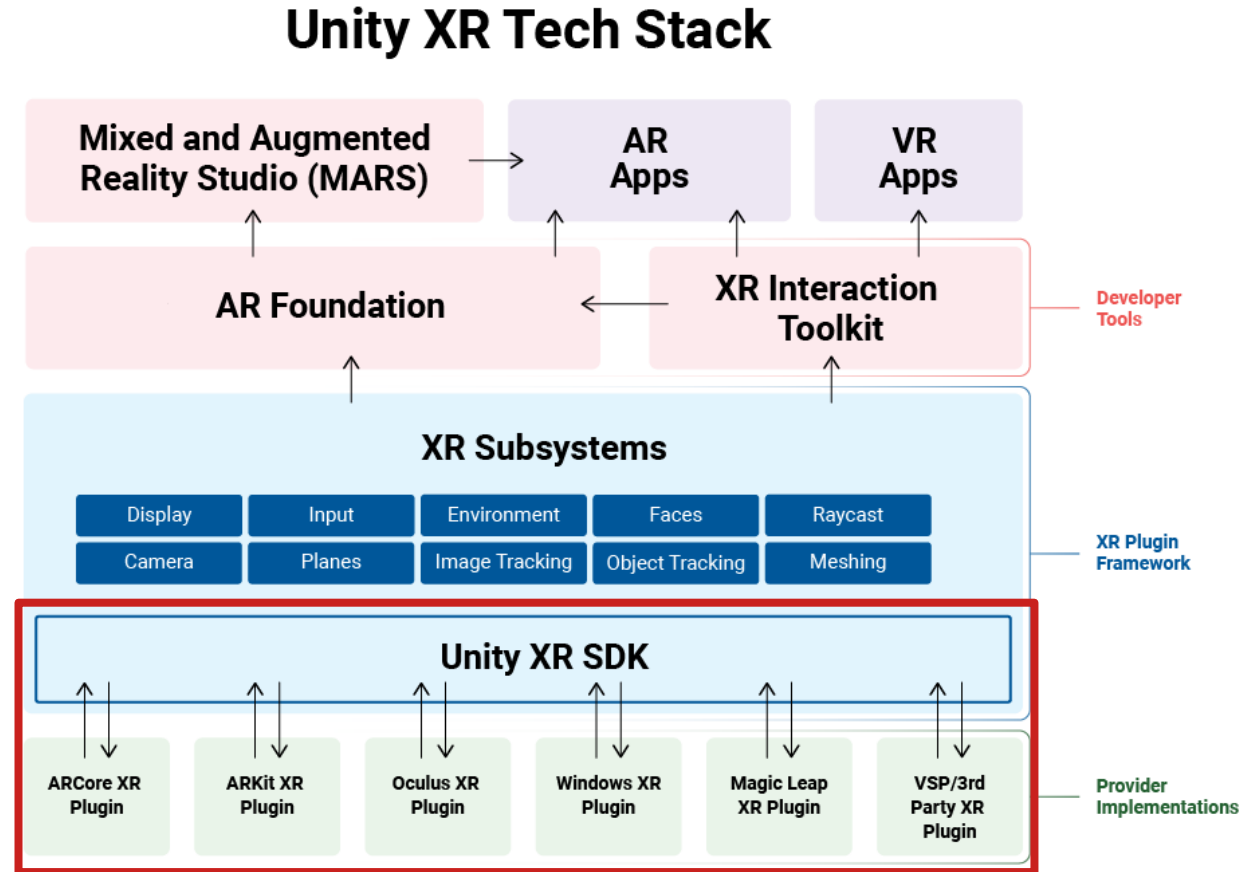
- Data (variables) are stored near all the operations that can be done with it, which aids readability
- You don't need to consider how an object is implemented, you only care about its functionality
  - We thus abstract away its contents!
- We will later see more object-oriented features that allow a lot of flexibility in implementing functionality

# Object-Oriented Programming - Disadvantages

- Some of the features commonly used for object-oriented programming can lead to a reduction in performance
- The methods (functions) of an object are somewhat «tied» to that object. For more general functionality, a regular function is easier to use than an object.
  - Most programs end up mixing procedure- and object-oriented programming for this reason

# When does Object-Oriented work well?

- Modularity
  - As all data and functionality is abstracted away, it can easily be swapped where necessary.
- Complex programs with many subsystems that all have to cooperate together



# When does Object-Oriented work well?

Complex programs with many subsystems that all have to cooperate together

(shown: source folders of the Unreal engine)

[illegible]



# Today

- Namespaces
- Object-Oriented Programming
- **Objects**

# C++ has two object variants

```
struct AnObject {  
  
};
```

```
class AlsoAnObject {  
  
};
```



# EMERGENCY MEETING



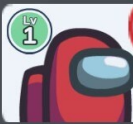
TOTAL TASKS COMPLETED



# Who Is The Impostor?



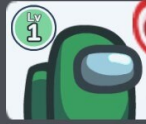
Compostor



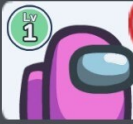
Vect0r



De-struct-0r



Boolguy



Enum class



Floaty



Stringent



SKIP VOTE

Voting Ends In: 119s

De-struct-0r was not An Impostor  
(1 Impostor remains)

# C++ has two object variants

```
struct AnObject {  
  
};
```

```
class AlsoAnObject {  
  
};
```

So both are objects, but why do we have two separate ones?

# Variable visibility

```
struct Example {  
    private:
```

```
        int variable1 = 5;  
        double variable2 = 10;
```

```
    public:
```

```
        string variable3;  
        int variable4 = 15;
```

```
};
```

```
int main() {
```

```
    Example instance;
```

```
    instance.variable3 = "Hey";
```

```
    instance.variable1 = 2;
```

```
}
```

Private variables can only be used by functions in the same object

Public variables can be used by all functions

Allowed: variable3 is public

Error: variable1 is private

# Variable visibility

- **Visibility is what allows objects to control what can or cannot be done with the member variables contained within them**
  - Primary pillar of object-oriented programming



# Struct vs. class: the **only** difference

Struct: visibility is public by default

```
struct AnObject {  
    int exampleVariable = 10;  
private:  
    int secondExample = 8;  
};
```

← The visibility of this variable is public

← The visibility of this variable is private

Class: visibility is private by default

```
class AnotherObject {  
    int exampleVariable = 10;  
};
```

← The visibility of this variable is private

# Methods

File: counter.h

```
#pragma once
class Counter {
    int count = 0;
public:
    void increment();
};
```

File: counter.cpp

```
#include "counter.h"

void Counter::increment() {
    count++;
}
```

The functions of an object are called «methods»

Like other functions, method declarations should be in the header file, and method definitions in the .cpp file

Methods follow the same visibility rules as variables

A class creates its own namespace, and you need to specify that namespace when defining a method.

# Methods and const

File: counter.h

```
#pragma once
class Counter {
    int count = 0;
public:
    int get() const;
    void increment();
};
```

File: counter.cpp

```
#include "counter.h"
int Counter::get() const {
    // count++ is not allowed here
    return count;
}
```

A method declared with const at the end is not allowed to modify member variables

- Communicates to another programmer that calling that method has no side effects
- Allows the compiler to apply better optimisations in certain conditions

# Const correctness

Term used to denote that something is declared as const that can be declared as const. Look out for:

- Reference parameters
  - Regular parameters create copies and therefore usually do not need to be const
- Methods
  - Declare a method const when it only reads from member variables

# Constructors



**YOU MUST CONSTRUCT  
ADDITIONAL PYLONS**



# Constructors

File: pylon.h

---

```
#pragma once
class Pylon {
    int usedMinerals = 0;
public:
    Pylon(int minerals);
};
```

A special method that is called automatically when an instance (variable) of the object is created

The method must have the exact same name as the class, and has no return type

File: pylon.cpp

---

```
#include "pylon.h"
```

```
Pylon::Pylon(int minerals)
: usedMinerals{minerals} {
    usedMinerals = minerals;
}
```

Fields can be initialised through normal assignment, or using the : syntax

# Constructors

File: pylon.h

---

```
#pragma once
class Pylon {
    int usedMinerals = 0;
public:
    Pylon(int minerals);
};
```

File: main.cpp

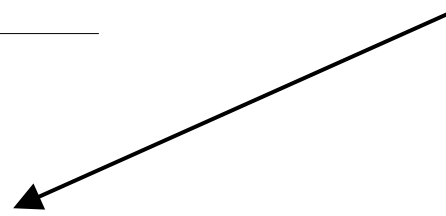
---

```
#include "pylon.h"

int main() {
    Pylon pylon(100);
}
```

A constructor is called when creating a variable of that type.

Parameters are specified using regular braces.





# Constructors

File: counter.h

---

```
#pragma once
class Counter {
    int count = 0;
public:
    Counter(int start);
};
```

File: counter.cpp

---

```
#include "counter.h"

Counter::Counter(int start)
    : count{start} {
    count = start;
}
```

# Default constructor

File: counter.h

---

```
#pragma once
class Counter {
    int count = 0;
public:
    Counter(int start);
    Counter() = default;
};
```

File: main.cpp

---

```
#include "counter.h"

int main() {
    Counter counter(30);
}
```

All objects have a default constructor that is removed if a different one that requires at least one parameter is defined.

The = default syntax is used to explicitly recreate the default constructor with no parameters



# Constructors

- An object can have multiple different constructors
- Constructors follow visibility like all other functions (e.g. you can create a private constructor)
- By default, the constructors of member variable objects are called automatically
  - Unless they require a parameter, in which case you must hardcode it, or use the : syntax

File: counter.h

---

```
#pragma once
class Counter {
    int count = 0;
public:
    Counter(int start);
};
```

File: countdown.h

---

```
#include "counter.h"

class Countdown {
    Counter counter{10};
};
```

# Constructors and const

File: counter.h

---

```
#pragma once
class Counter {
    int count = 0;
    const int limit;
public:
    Counter(int start, int max);
};
```

File: counter.cpp

---

```
#include "counter.h"

Counter::Counter(int start, int max)
    : count{start}, limit{max} {
}
```

Const members *must* be initialised, either through assigning a default value or a constructor.

A non-default value can *only* be assigned through the : syntax

# Task: classes

- For each member variable, determine whether the visibility should be public or private
- In VS Code, create a project using the task – Constructors template, and implement the three listed constructors.

```
struct RaceResult {  
    const std::string nameOfAthlete;  
    std::vector<double> lapTimesInSeconds;  
    double fastestLapTime = 0;  
    int numberOfLaps = 0;  
    double totalDistanceCoveredMeters();  
};
```



**DEAD BODY REPORTED**



TOTAL TASKS COMPLETED



DEAD



Floaty



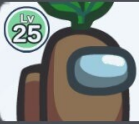
Compostor



TOTAL TASKS COMPLETED



# Who Is The Impostor?



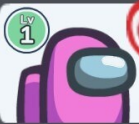
Compostor



Vect0r



Boolguy



Enum class



Stringent



De-struct-0r



Floaty

SKIP VOTE

Voting Ends In: 119s



Enum class was An Impostor  
(0 Impostors remain)

# Victory



# Today

- Namespaces
- Object-Oriented Programming
- Classes
- **Enumerations (enum)**

# Enumerations

File: color.h

---

```
#pragma once
enum class Colour {
    blue, red, green, grey
};
```

File: main.cpp

---

```
#include "color.h"
```

```
int main() {
    Colour colour = Colour::blue;
    return 0;
}
```

A data type that can only hold one of a specific set of values

- Despite containing the keyword `class`, enums are not objects

Should be defined in its own header file

← Specifying a value requires explicitly naming the enum type it belongs to

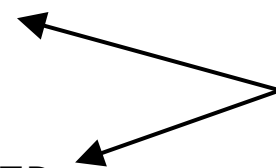
# Why enum class, not just enum?

The **enum** is a C-style enum, and they have two problems:

```
enum CarColour {  
    BLUE, BLACK, RED  
};  
enum SweaterColour {  
    GREEN, YELLOW, RED  
};
```

Problem 1: each value can only exist once.

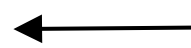
These enums will not compile because both contain the value RED.



```
void useColour(CarColour colour) {}
```

```
int main() {  
    printColour(GREEN);  
    return 0;  
}
```

Problem 2: when using an enum value it is not clear where that value came from



# Enumerations and int

```
enum class Colour {  
    blue = 6, red, green, grey  
};  
  
int main() {  
    Colour colour = Colour::green;  
    std::cout << static_cast<int>(colour)  
               << std::endl; // prints 8  
    Colour grey = static_cast<Colour>(9);  
    return 0;  
}
```

Under the hood, each enum value becomes an integer

Can assign a number to each enum value.

Values without a number are assigned one higher than the one defined prior to them.

Use `static_cast<>()` to convert between enum and int values

# Today

- Namespaces
- Object-Oriented Programming
- Classes
- Enumerations (enum)

# Next week

- Files
- Streams
- `std::filesystem`