Lecture 7: Object Orientation

Bart Iver van Blokland

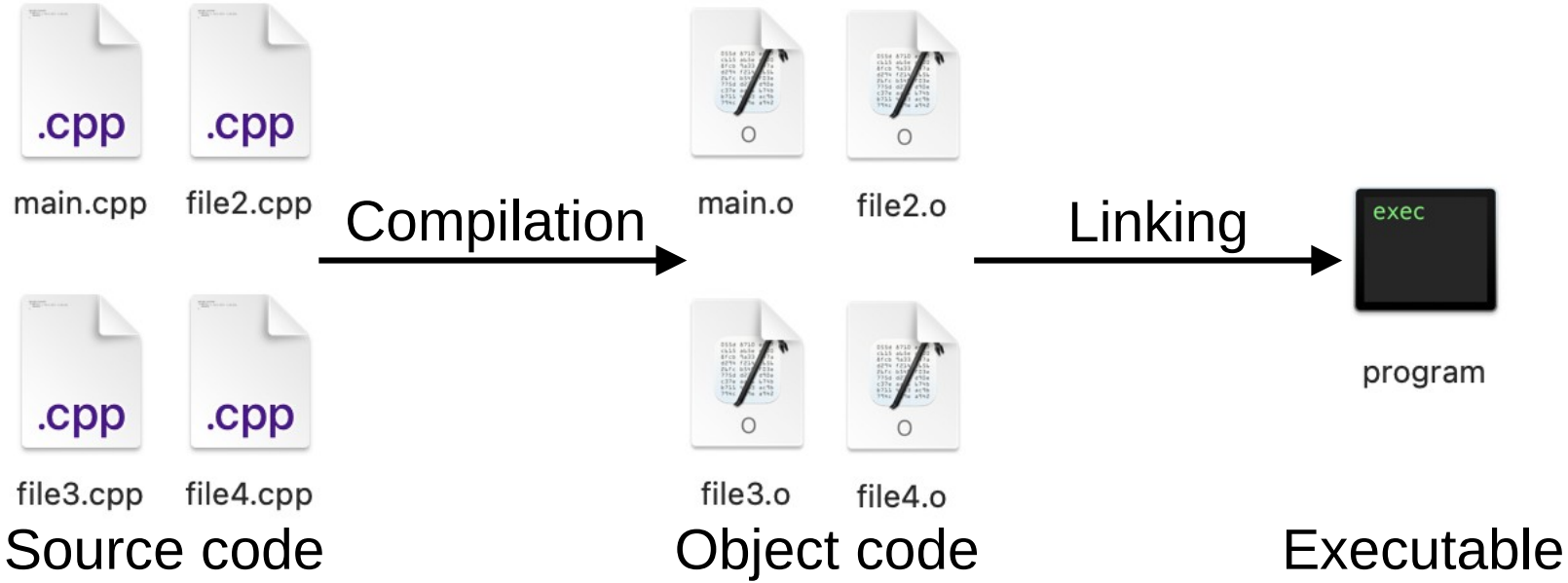
(Rune Sætre)

Last time

- **Compilation and build systems**
- Data structures
- Const
- References

Compilation in C++

1. All .cpp files are partially compiled, resulting in object code
2. A step called «linking» combines all object code into a program



Common mistake 1 / 2

If you use `#include` with a .cpp file:

Error: multiple definitions of `sayHello()`

file2.h

```
#pragma once
void sayHello();
```

main.cpp

```
#include "file2.cpp"

int main() {
    sayHello();
}
```

file2.cpp

```
#include "file2.h"
#include <iostream>
void sayHello() {
    std::cout << "Hello!" << std::endl;
}
```

Common mistake 2 / 2

If you put a definition in a header file:

Error: multiple definitions of `sayHello()`

main.cpp

```
#include "file2.h"

int main() {
    sayHello();
}
```

file2.h

```
#pragma once
void sayHello() {
    std::cout << "Hello!" << std::endl;
}
```

file2.cpp

```
#include "file2.h"
#include <iostream>
```

Build systems

A typical project has:

- A bunch of C++ source files
- A number of libraries used by those source files
  - Some libraries need to be compiled from source
  - Some libraries are configurable where unnecessary features can be enabled or disabled
  - Some libraries are installed on the computer but their development files need to be located
  - Libraries may use other libraries themselves

Setting this all up yourself is complicated and tedious

Solution: use a build system!

Build system

A build system figures out how to compile your code and any libraries used by it

- Examples:
  - CMake (look for the CMakeLists.txt file)
  - Meson (used in the course)
  - Gradle

Usually requires writing a configuration file

Usually generates a configuration file a build tool (e.g. GNU Make or Ninja) can use to do the actual compilation

Last week

- Compilation and build systems
- **Data structures**
- Const
- References

Struct

Effectively a group of variables (*)

```
struct Point {
    double x = 0;
    double y = 0;
};

int main() {
    Point point;
    point.x = 5;
}
```

Can be named anything, but by convention first name is capitalised

Contents are a list of variables called «members».

Use like any other datatype, but need to use . to reference members

Last week

- Compilation and build systems
- Data structures
- **Const**
- References

Const

- Used for constants
- The **`const`** keyword is added in front of the data type in any place a data type is used
- Can only be assigned a value once, and the variable becomes read-only

```
const int count = 10;
count++; // error
std::cout << count << endl; // ok!
```

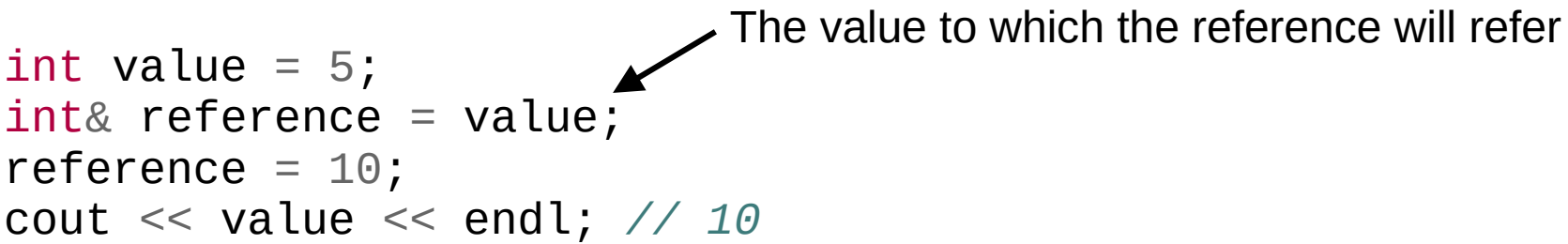
- If you know a variable is not modified, you should always declare it as const

Last week

- Compilation and build systems
- Data structures
- Const
- **References**

References

- Reference variable: a variable that does not contain a value itself but instead modifies the value of another
- Declared by appending an & to the data type
- Which variable is referenced cannot be changed after the reference has been created



Why references?

- Create functions with multiple return values

```
void readSensors(double &temperature, double &humidity);
```

- Create functions which apply modifications on a variable

```
void fillPetrolTank(Car &car);
```

- Avoids creating unnecessary copies

```
void applyChanges(std::vector<int>& bigVectorGoesHere);
```

Const reference

- Create a read-only version of a variable

```
int value = 5;
const int& reference = value;
value = 10; // allowed: value is not const
reference = 10; // not allowed: reference is const
```

- Useful in function parameters: signals that the function does not modify the value you pass in

```
int computeSum(const std::vector<int>& values);
```

Today

- **Object-Oriented Programing**
- Classes
- Enumerations (enum)

TDT4102 - Prosedyre- og objektorientert programmering

We have done this

Now it’s time for this

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) hidden inside the object, and functions that modify the data in specific ways.

Procedure vs. Object-Oriented Programming

The two styles differ in where program data is stored, and how that program data can be modified.

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

- Object-Oriented Programming:
  - Data is hidden within objects
  - Objects have functions that modify the data within

Example: Procedure-Oriented Programming

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

```
void increment(int& number) {
    number++;
}

int main() {
    int number = 0;
    increment(number);
    return 0;
}
```

Example: Object-Oriented Programming

- Object-Oriented Programming:
  - Data is hidden within objects
  - Objects have functions that modify the data within

```
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_wimdow();
    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

Which should you use?

- Using these styles is not “either or”. C++ programs can contain a mix of both
- There are good and bad ways to use each style in terms of code readability and performance
- Nothing mandates the use of either style. All programs can be implemented using either one.

In short: use the style that makes most sense for the task at hand.

Advantages – Object-Oriented Programming

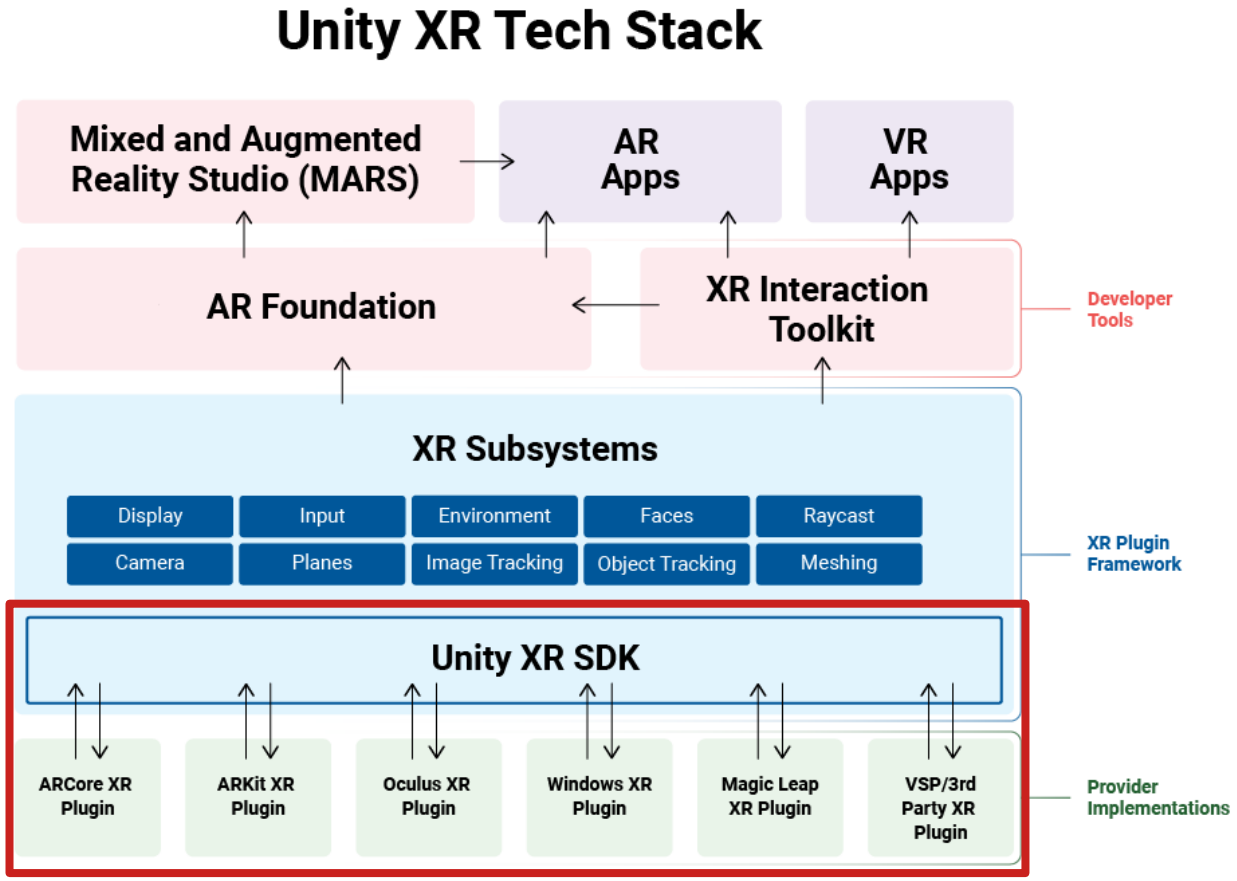
- Control:
  The object defines what can and cannot be done with the data that is hidden inside

- Grouping:
  Data and the functions modifying that data are located together

- Modularity:
  As all data and functionality is abstracted away, it can easily be swapped where necessary.

When does Object-Oriented work well?

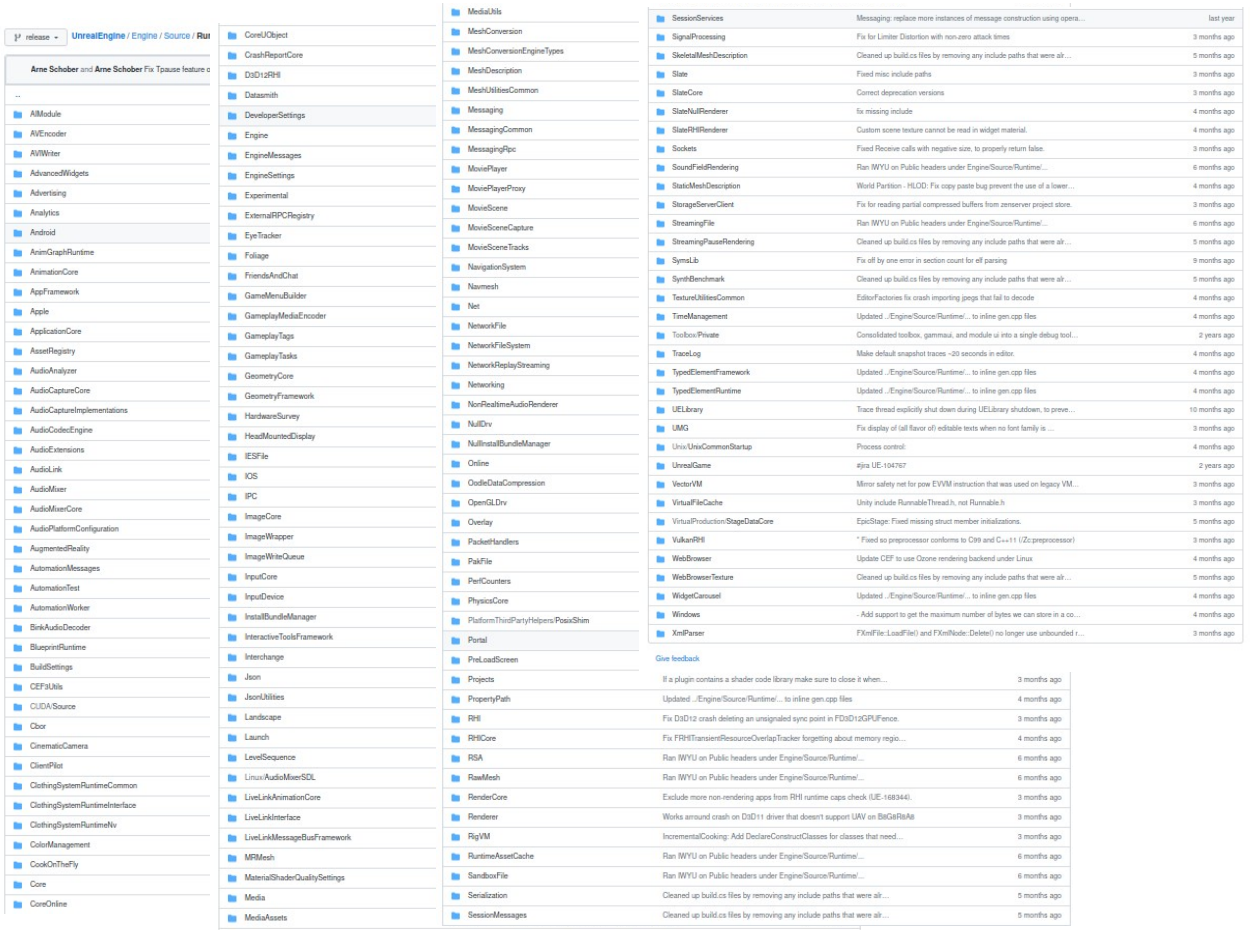
- Modularity
  - As all data and functionality is abstracted away, it can easily be swapped where necessary.
- Complex programs with many subsystems that all have to cooperate together



When does Object-Oriented work well?

Complex programs with many subsystems that all have to cooperate together

(shown: source folders of the Unreal engine)



Advantages – Procedure-Oriented Programming

- Freedom:
  A function can do whatever it needs to with any data it processes

- Granularity:
  Each function can be independently reused across the entire program

- Flexibility:
  Very few problems can be abstracted neatly, and functions can ensure each edge case is handled

When does Procedure-Oriented work well?

It is difficult to give a specific case, but here are some general areas:

- Programs that read some data, do something with it, then write out the results
- [controversial] Programs that must run fast (and OOP has some performance «gotchas»)

Both can be valid choices

Object-Oriented Programming	**Procedure-Oriented Programming**
Control: The object defines what can and cannot be done with the data that is hidden inside	Freedom: A function can do whatever it needs to with any data it processes
Grouping: Data and the functions modifying that data are located together	Granularity: Each function can be independently reused across the entire program
Modularity: As all data and functionality is abstracted away, it can easily be swapped where necessary.	Flexibility: Very few problems can be abstracted neatly, and functions can ensure each edge case is handled

Today

- Object-Oriented Programing
- **Classes**
- Enumerations (enum)

C++ has two object variants

```
struct AnObject {

};
```

```
class AlsoAnObject {

};
```

cppstruct86 was not An Impostor

(2 Impostors remain)

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

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.

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;
}
```

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

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

Constructors

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);
}
```

Once a constructor with a parameter is created, you must supply that parameter when creating an instance of the object

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

You can use either { } or ( ) for specifying constructor parameters

An object can have multiple different constructors

Constructors and const

File: counter.h

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

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

File: counter.cpp

```
#include "counter.h"

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

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

Today

- Object-Oriented Programing
- 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

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

- Object-Oriented Programing
- Classes
- Enumerations (enum)

Next week

- Files
- Streams
- std::filesystem