Lecture 6: Referential Treatment

Bart Iver van Blokland

(Rune Sætre)

Last lecture

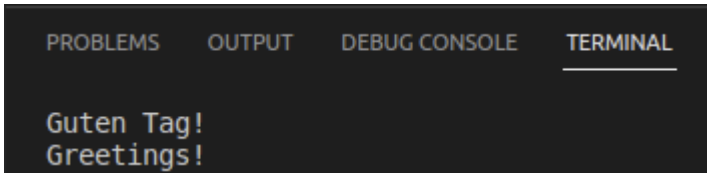
- **Namespaces**
- Function declarations and definitions
- Compiling C++ files

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;
}
```



Last lecture

- Namespaces
- **Function declarations and definitions**
- Compiling C++ files

You can declare functions as many times as you like.

However, there can only be one single implementation of a function across every single .cpp file in your project.

As such, this is allowed:

```
double add(double a, double b);
double add(double a, double b);
double add(double a, double b);
double add(double a, double b);
```

But this is not:

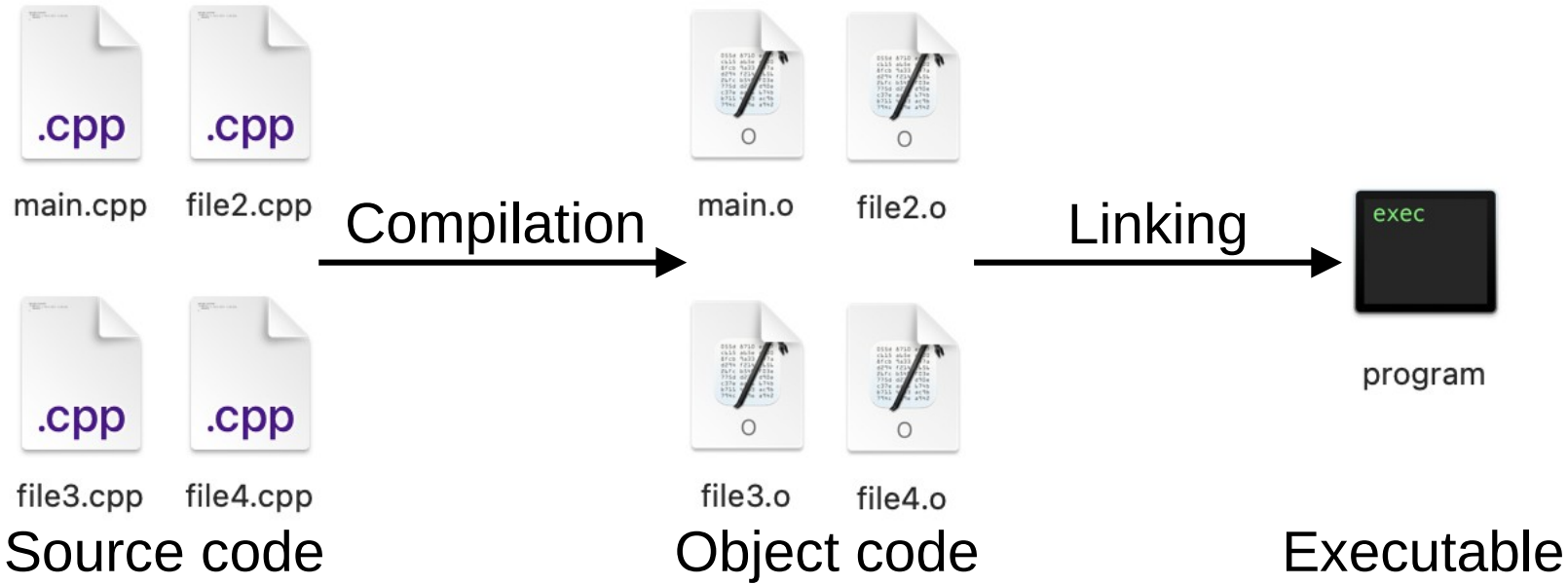
```
double add(double a, double b) {
    return a + b;
}
double add(double a, double b) {
    return a + b;
}
```

Last lecture

- Namespaces
- Function declarations and definitions
- **Compiling C++ files**

Partial compilation

We divide the compilation process into two stages; compilation and linking



How to use a function in another file

- The compilation step needs declarations
- The linking step needs definitions

As long as you have a definition in *any* .cpp file, you only need to declare it to use it in another file

main.cpp

```
void sayHello();

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

file2.cpp

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

A better way: Header files

- Put all declarations in another file that allow them to be reused

file2.h

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

main.cpp

```
#include "file2.h"

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

file2.cpp

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

How to add a new .cpp file:

1. Create an empty .cpp file
2. Create a corresponding .h file
3. Write `#pragma once` on the first line of the .h file
4. Write `#include "filename.h"` on the first line of the .cpp file (replace filename with your file name)
5. Update your build configuration file

- **Compilation:** fails
  main.cpp lacks declaration of `sayHello()`

- **Linking:** would succeed (but compilation fails before it gets there)
  Definitions for all used functions exist

file2.h

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

main.cpp

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

file2.cpp

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

- **Compilation:** succeeds
  Each file has the declarations it needs

- **Linking:** fails
  Definition of `sayHello()` is missing

file2.h

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

main.cpp

```
#include "file2.h"

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

file2.cpp

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

- **Compilation:** succeeds
  Each file has the declarations it needs

- **Linking:** fails
  Multiple definitions of `sayHello()` exist across all files

main.cpp

```
void sayHello() {

}

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

file2.h

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

file2.cpp

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

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>
```

In summary

- How to add a new .cpp file:
  1. Create an empty .cpp file
  2. Create a corresponding .h file
  3. Write `#pragma once` on the first line of the .h file
  4. Write `#include "filename.h"` on the first line of the .cpp file (replace filename with your file name)

- **NEVER** `#include` a .cpp file!

- **ONLY** use `#include` with .h files!

Today

- **Build systems**
- A sample project from scratch
- Data structures
- Const
- References

Our project

Let’s try to compile a project with several files:

Writing all compile commands every time by hand is tedious.

Can we automate that?

Example 1

Curriculum note

- You will not need to write a build configuration file on the exam
- You will need to understand how C++ projects are compiled (e.g. what we have discussed up to now)
- You will need build configuration files if you want to use C++ after the course is finished

Shell script

- A list of terminal commands that should be run one after the other
- Extremely difficult to do incremental compilation
- Difficult to update over time

```
rm -rf build
mkdir build

clang++ -c main.cpp -o main.o
clang++ -c file1.cpp -o file1.o
clang++ -c file2.cpp -o file2.o
clang++ -c file3.cpp -o file3.o

clang++ main.o file1.o file2.o file3.o -o program
```

Makefile

- A text file that is called «Makefile»
- Tries to be a smart shell script
- Can do partial compilation
- Syntax is often unintuitive

```
SRCDIR = src
BUILDDIR = build
EXECUTABLE = $(BUILDDIR)/program

CXX = clang++
CXXFLAGS = -std=c++17
LINK = $(CXX)
LDFLAGS = -o $(EXECUTABLE)

SOURCES := $(wildcard $(SRCDIR)/*.cpp)
OBJECTS := $(patsubst $(SRCDIR)/%.cpp,
           $(BUILDDIR)/%.o,$(wildcard $(SRCDIR)/*.cpp))

all: ${OBJECTS}
    ${CXX} ${CFLAGS} ${OBJECTS} ${LDFLAGS}

$(BUILDDIR)/%.o: $(SRCDIR)/%.cpp
    mkdir -p ${dir $@}
    ${CXX} -o $@ $< -c ${CXXFLAGS}

clean:
    rm -rf $(BUILDDIR)
```

CMake

- Makefiles and similar are difficult to write, so let's generate them automatically
- Arguably the industry standard tool
- Uses a tool such as make to do the compilation, and therefore requires a configuration step
- Configuration file must be called "CMakeLists.txt"

```
cmake_minimum_required (VERSION 3.6)
project(project)
add_executable(program main.cpp file1.cpp file2.cpp file3.cpp)
```

Meson

- Does in principle the same as Cmake
- Uses a more Python-like syntax
- Is able to use CMake projects
- Better documentation than CMake
- Used in the course

```
project('sampleproject', 'cpp', version : '0.1',
        default_options : ['cpp_std=c++17', 'default_library=static'])

src = ['file1.cpp', 'file2.cpp', 'file3.cpp', 'main.cpp']
exe = executable('program', src)
```

Which to pick?

- Most projects nowadays use a build system, and there are a LOT of them available
- CMake is popular, but by no means universal
- While not perfect, using a build system simplifies many things

Today

- Build systems
- A sample project from scratch

How to add a library

Easy mode: wrapdb

Meson WrapDB packages

This is a list of projects that have either an upstream Meson build system, or a port maintained by the Meson team. They can be used by your project to provide its dependencies.

Use the command line `meson wrap install <project>` to install the wrap file of any of those projects into your project's `subprojects/` directory. See Meson command line documentation.

If you wish to add your own project into this list, please submit your wrap file in a Pull Request. See Meson documentation for more details.

Project	Versions	Provided dependencies	Provided programs
abseil-cpp	**20220623.0-2** 20211102.0-3, 20210324.2-4, 20210324.1-4, 20200923.2-1, 20200225.2-3	absl_base, absl_container, absl_debugging, absl_flags, absl_hash, absl_numeric, absl_random, absl_status, absl_strings, absl_synchronization, absl_time, absl_types	
arduinocore-avr	**1.8.2-1** 1.6.20-1		
argparse	**2.9-1** 2.6-1, 2.5-1, 2.4-1, 2.2-1	argparse	
backward-cpp	**1.6-1**	backward-cpp	
bdwgc	**8.2.2-1** 8.2.0-1, 7.6.8-1	gc	
box2d	**2.4.1-3** 2.3.1-7	box2d	
catch	**2.2.2-1** 2.2.1-2		
catch2	**3.2.0-1** 3.1.0-1, 2.13.8-1, 2.13.7-1, 2.13.3-2, 2.11.3-1, 2.11.1-1, 2.9.0-1, 2.8.0-1, 2.7.2-1, 2.5.0-1, 2.4.1-1	catch2, catch2-with-main	

https://mesonbuild.com/Wrapdb-projects.html

How to add a library

The hard way:

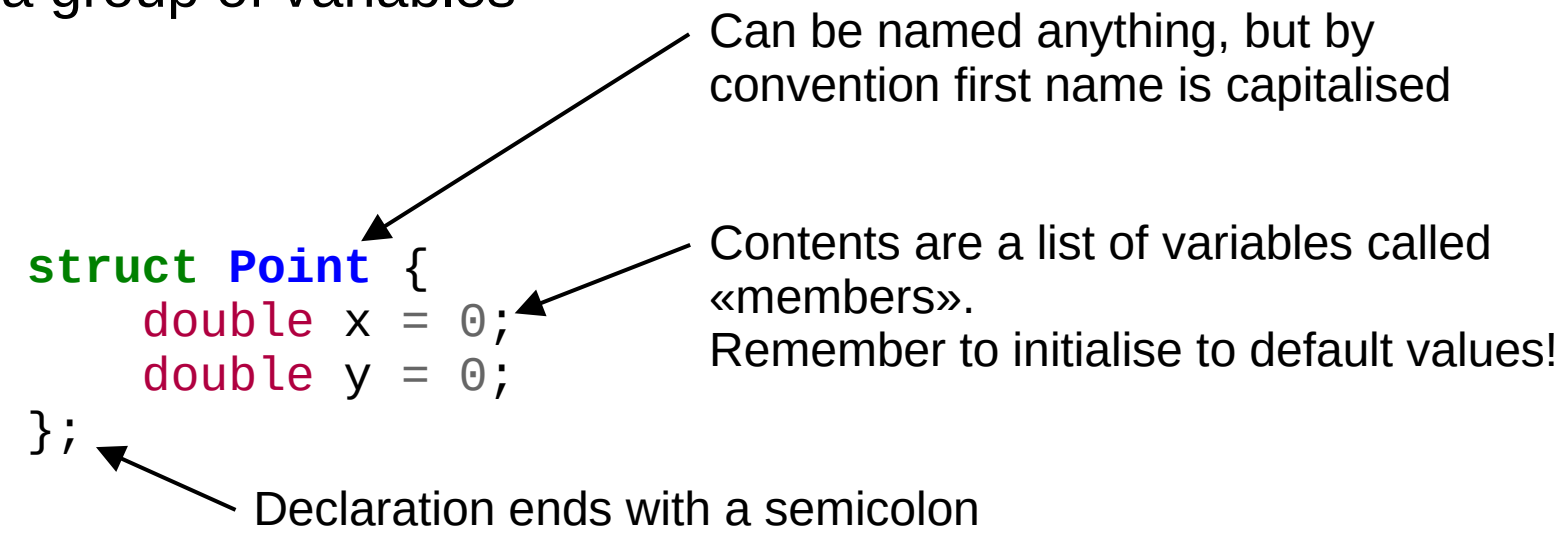
- If a library uses cmake (look for the CMakeLists.txt file):
  - Process is somewhat similar to using WrapDB
- If a library does not use cmake or meson:
  - If possible, write a meson file for compiling the library

Today

- Build systems
- A sample project from scratch
- **Data structures**
- Const
- References

Struct

- Effectively a group of variables



Struct

- Structs become their own data type, which can be used like any other

```
Point point;
```

- Read and write variables inside of it using a .

```
point.x = 5;
std::cout << point.y << std::endl;
```

- Values can be initialised using { }:

```
Point anotherPoint {5, 10};
```

Struct

- Best practice: define each struct in its own header file
- Not possible to print a struct using cout directly

```
std::cout << point << std::endl; // error!
```

- Assignment copies all values within the struct

```
Point point {1, 2};
Point anotherPoint {3, 6};
point = anotherPoint;
// point is now (3, 6)
```

Today

- Build systems
- A sample project from scratch
- 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!
```

- Constant members in a struct are possible, but not recommended due to the hassle they cause

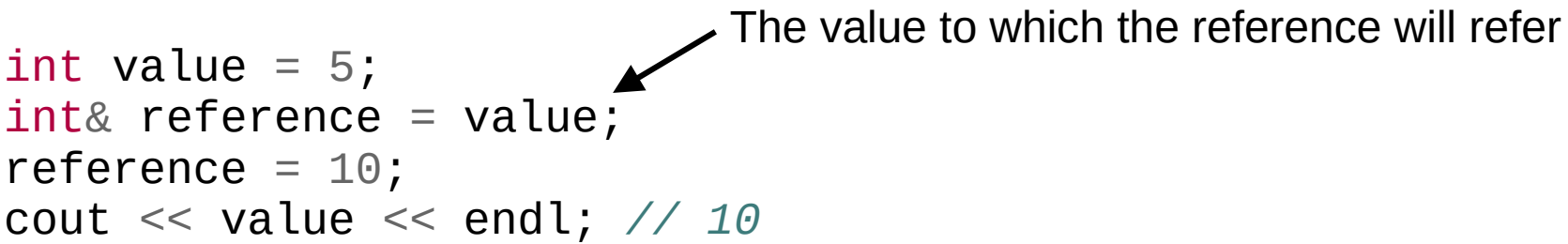
Today

- Build systems
- A sample project from scratch
- Data structures
- Const
- **References**

Upper D. (1974). The unsuccessful self-treatment of a case of "writer's block". Journal of applied behavior analysis, 7(3), 497. https://doi.org/10.1901/jaba.1974.7-497a

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

- Build systems
- A sample project from scratch
- Data structures
- Const
- References