Lecture 5: Compilation Information

Bart Iver van Blokland

(Rune Sætre)

Last lecture

- Animation
- Vectors
- Arrays
- Random number generation

And today we'll also need:

- Function declarations vs. Function definition

Declaration vs. Definition

Function declaration:

The characteristics of a function, but not its implementation

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

The return type, function name, and parameter list are the characteristics of a function.
If any of these is different in any way, the function is considered different.

Function definition:

A declaration of a function, along with its implementation

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

A function is defined when it has the {} braces following its declaration

You can declare functions as many times as you like.

However, there can only be one single implementation of a function.

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;
}
```

The compiler processes .cpp files from top to bottom. Therefore:

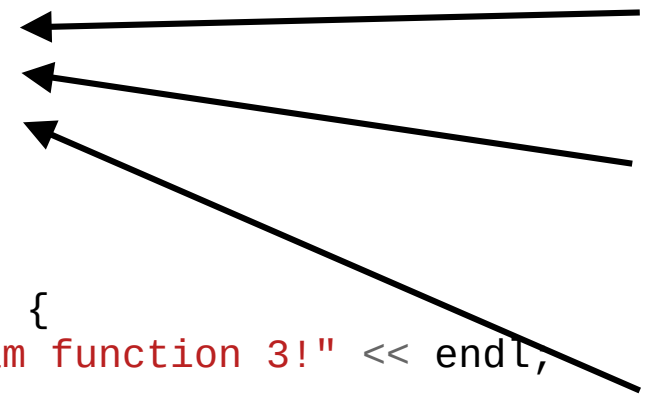
```
void function1() {
    cout << "I am function 1!" << endl;
}

void function3();

int main() {
    cout << "I am function 2!" << endl;
    function1();
    function2();
    function3();
    return 0;
}

void function2() {
    cout << "I am function 3!" << endl;
}

void function3() {
    cout << "I am function 4!" << endl;
}
```



Allowed: function1() has been declared (and defined) above.

Error: function2() has not yet been declared above and therefore does not yet exist.

Allowed: function3() has been declared above, and can therefore be used. It does not yet have to be defined.

What happens if a function is declared but not defined?

```
void function3();

int main() {
    function3();
}
```

```
Undefined symbols for architecture arm64:
  "function3()", referenced from:
      _main in main.cpp.o
ld: symbol(s) not found for architecture arm64
clang-15: error: linker command failed with exit code 1
(use -v to see invocation)
ninja: build stopped: subcommand failed.
```

Today

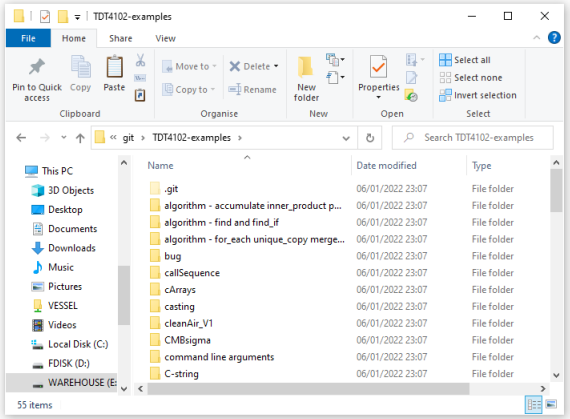
- **A hitchhiker's guide to the Terminal**
- Compiling a C++ program
- Intermezzo: namespaces
- Compiling many C++ files

The terminal

- A text-based interface to your computer, much like the graphical interface you use every day
- Terminals run one command at a time, then wait for you to put in a new one
- In contrast to the graphical interface, you can run programs with parameters that affect their behaviour

The terminal

- Terminals have a «current directory», like File Explorer on Windows or Finder in MacOS
- Any files you use in commands will be relative to the current directory



Relative file paths

A relative file path specifies the location of a file or directory relative to the current directory

Syntax:

- Directories are separated using a /
  Example: “`builddir/meson-logs/meson-log.txt`”
- A period (.) means “this directory”
  Example: “`./builddir/program.exe`”
- A double period (..) means “the directory above”
  Example: “`../../main.cpp`”

The terminal: current directory

- List the contents of the current directory:
  All: `ls`
  Windows: `dir`

- Change the current directory:
  `cd [directory]`

- Show the path of the current directory:
  All: `echo "${PWD}"`
  Windows: `pwd`

The terminal: files and directories

- Create a new directory:
  `mkdir [directory name]`

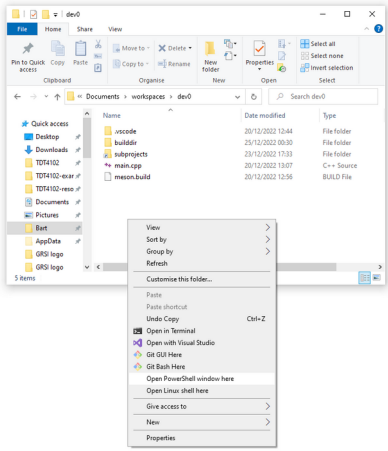
- Delete a directory (and its contents):
  Windows: `rmdir [directory name]`
  MacOS / Linux: `rm -rf [directory name]`

- Create a new file:
  Windows: `New-Item -ItemType file [filename]`
  MacOS / Linux: `touch [filename]`

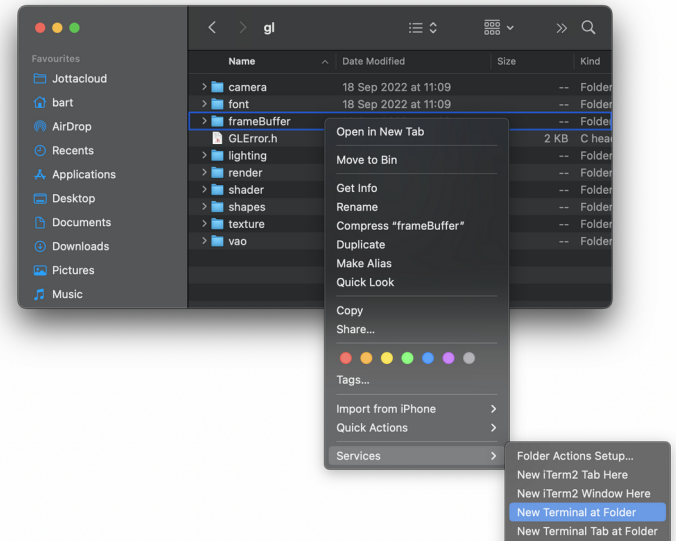
- Delete a file:
  `rm [filename]`

Open a terminal at a folder

- Windows:
  - Hold shift
  - Right click in an empty part of a folder
  - Select “open PowerShell window here”



- Mac:
  - Right click on a folder
  - Select services
  - Select “New Terminal at Folder”



Today

- A hitchhiker’s guide to the Terminal
- **Compiling a C++ program**
- Intermezzo: namespaces
- Compiling many C++ files

Compiling a C++ program

The first thing we need is a C++ source file.

We'll use the basic one for now:

```
#include "std_lib_facilities.h"

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Next we need a compiler!

There are many compilers..

- LLVM / Clang
  - Used in the course
  - LLVM is a set of tools to build compilers
- GCC (GNU Compiler Collection)
  - Developed by the Free Software Foundation
- MSVC (Microsoft Visual C++)
  - Microsoft's C++ compiler
  - Only available on Windows
- icc (Intel C++ compiler)

And a LOT more!

But how do they differ?

- Implement C++ (and its standard library) in a slightly different way
- Optimise your code in different ways
- Sometimes have specific language extensions or features
- Some tools only work with a specific compiler
  E.g. clang-tidy

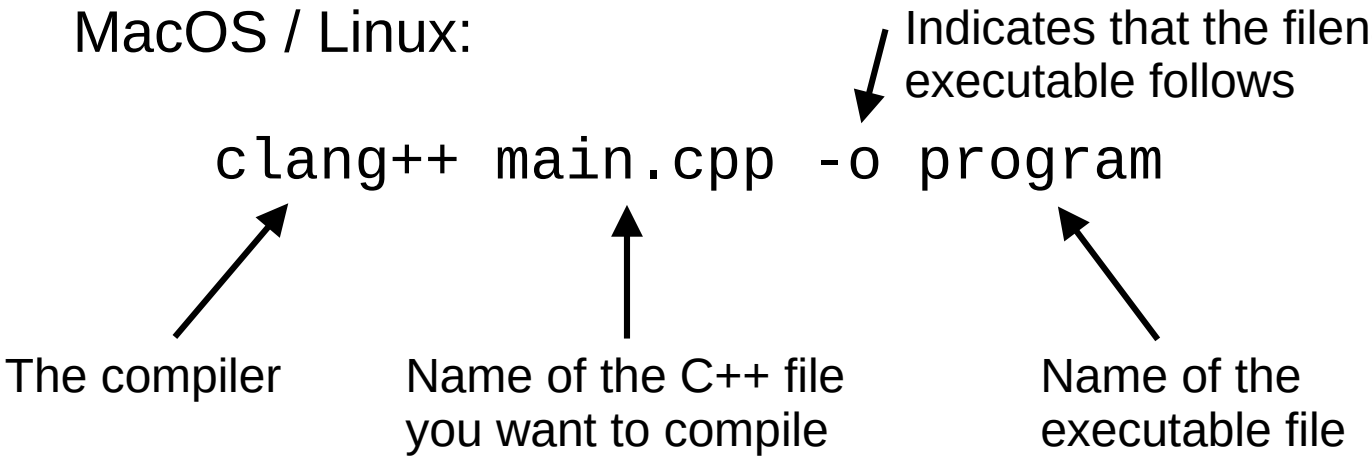
Compiling a program

Windows:

```
clang++ main.cpp -o program.exe
```

MacOS / Linux:

```
clang++ main.cpp -o program
```



We need to make some changes..

```
#include "std_lib_facilities.h"

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

```
main.cpp:1:10: fatal error:
'std_lib_facilities.h' file not found
#include "std_lib_facilities.h"
         ^~~~~~~~~~~~~~~~~~~~~~
1 error generated.
```

Now the program compiles!

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

```
bart@TURBONINJA:~/workspace$ clang++ main.cpp -o program
bart@TURBONINJA:~/workspace$
```

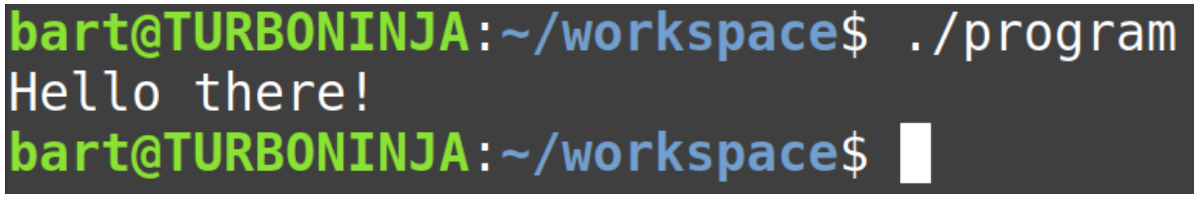
Running a program

Windows:

```
./program.exe
```

MacOS / Linux:

```
./program
```



Today

- A hitchhiker’s guide to the Terminal
- Compiling a C++ program
- **Intermezzo: namespaces**
- Compiling many C++ files

Namespaces

- Functions, variables, and data types all need names
- Sometimes different things have the same name
  e.g. vector

The C++ solution: Namespaces

Namespaces

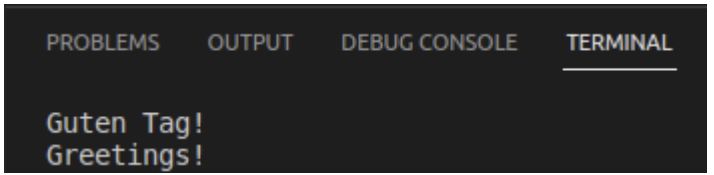
- Namespaces are a “directory” for data type and function names
- When using anything declared inside a namespace, you must specify explicitly in which namespace it resides
- It is possible to declare a namespace inside of another namespace
- The C++ standard library uses the `std` namespace
- Namespaces can’t be declared inside a function

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;
}
```



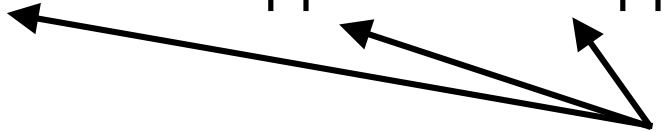
Today

- A hitchhiker’s guide to the Terminal
- Compiling a C++ program
- Intermezzo: namespaces
- **Compiling many C++ files**

Compiling multiple files

Windows:

```
clang++ main.cpp file1.cpp file2.cpp -o program.exe
```



Filenames of all the files you want to compile

MacOS / Linux:

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

.. But only if your project is small!

However, this does not always work..

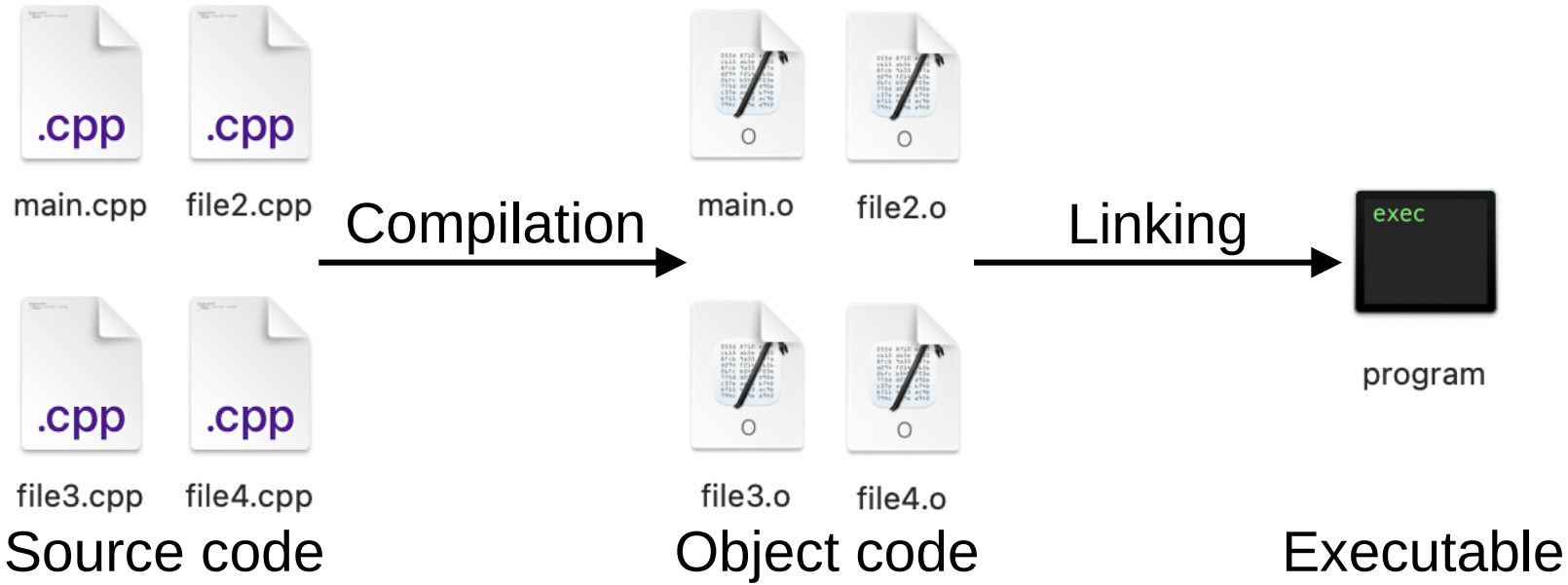
When your project becomes large:

- Compile times become very long (can be multiple hours!)
- The system may not have enough RAM available to compile all files at once
- Companies may not want to distribute source code (.cpp files) for proprietary libraries

The solution: we need partial compilation!

Partial compilation

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



Compilation

- Compile each file separately as much as possible
- Only leaves function calls
- Produces one “object code” file per .cpp file

```
int doWork(int a, int b) {
    if(isGreater(a, b)) {
        return add(a, b);
    } else {
        return b;
    }
}
```

Compilation

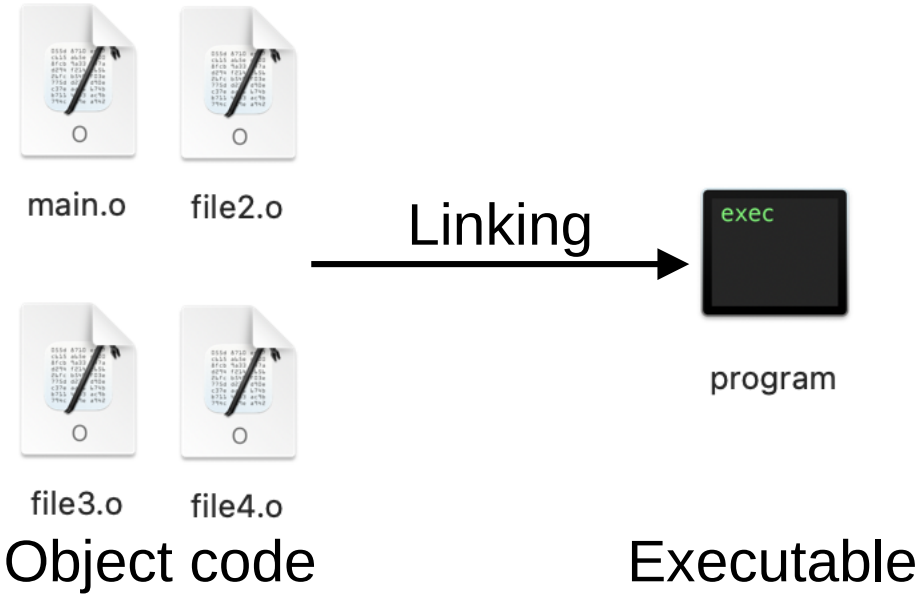
Function calls

```
doWork(int, int):
        stp     x29, x30, [sp, -32]!
        mov     x29, sp
        str     w0, [sp, 28]
        str     w1, [sp, 24]
        ldr     w1, [sp, 24]
        ldr     w0, [sp, 28]
        bl      isGreater(int, int)
        and     w0, w0, 255
        cmp     w0, 0
        beq     .L6
        ldr     w1, [sp, 24]
        ldr     w0, [sp, 28]
        bl      add(int, int)
        b       .L7
.L6:
        ldr     w0, [sp, 24]
.L7:
        ldp     x29, x30, [sp], 32
        ret
```

https://godbolt.org/z/1MoTb9Px1

Linking

- Combine all object code files into an executable file
- Generate the binary instructions needed to do the function calls



Partial compilation

Advantages:

- No need to compile the program from scratch every time
- No need to compile the program all at once

Disadvantages:

- Slow means of compilation on modern computers
- Linking issues can be difficult to resolve
- **All functions across all .cpp files must have a unique name**

```
/usr/bin/ld: /tmp/main-296c3f.o: in function `main':
main.cpp:(.text+0x77): undefined reference to `glfwSetErrorCallback'
/usr/bin/ld: main.cpp:(.text+0x7c): undefined reference to `glfwInit'
/usr/bin/ld: main.cpp:(.text+0xad): undefined reference to `glfwCreateWindow'
/usr/bin/ld: main.cpp:(.text+0xc1): undefined reference to `glfwTerminate'
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Partial compilation in the terminal

Compilation

We compile each file separately. The .o files are object code files.

```
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
```

The -c flag prevents the linking step from taking place

Linking

We then provide all object code files and run the compiler without the -c flag.

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

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;
}
```

A better way: Header files

- The `#include` directive copies and pastes the content of another file into the current one
  - **NEVER `#include` a .cpp file.**
  - **ALWAYS `#include` a .h file.**

- To prevent a number of headaches, always make the first line of a header file:

```
#pragma once
```

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)

Static and Dynamic linked libraries

C++ compilers have two types of libraries:

- Statically linked libraries: a set of object code (.o) files “taped together”. Usually a .a file, but MSVC uses .lib
- Dynamically linked libraries: a library of functions that is linked automatically each time you run the program.

Why does compilation work this way?

- When C and C++ were designed, computers were SLOW!
- Compiling one file at a time was a great way to make compilation faster
- Nowadays it degrades performance, but fixing the problem is difficult.

http://www.computersciencelab.com/ComputerHistory/HtmlHelp/Images2/TapeDrive.jpg

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!