

Lecture 5: Compilation Information

Bart Iver van Blokland

Do you remember?

- What are the main steps for generating a random number?

Do you remember?

- What are the main steps for generating a random number?
 - What does a random engine do?
 - Why do we (often) need a random seed?
- What is the difference between a `std::vector` and `std::array`?
- What is the issue with using `[]` to index a vector or array?
- What is an example of a situation in which you would choose a `std::vector` or `std::array`?
- Where is the `const` keyword used, and what does it mean?
- What is the difference between `const` and `constexpr`?

Last time

- Random number generation
- `std::vector` and `std::array`
- `const` and `constexpr`

Functions: declaration vs. definition

Functions: 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;  
}
```

```
../main.cpp:6:8: error: redefinition of 'add'  
6 | double add(double a, double b) {  
  | ^  
../main.cpp:3:8: note: previous definition is here  
3 | double add(double a, double b) {  
  | ^  
1 error generated.
```

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.

Today

- **The #include statement**
- A compilation of C++ compilation
- The terminal

This will be all you need to solve assignment 3

#include

- C++ equivalent of the **import** statement in Python
- Takes a file name as parameter, which can be specified in one of two ways:

```
#include <amazingLibrary.h>  
#include "amazingLibrary.h"
```

- Effect: copies and pastes the contents of the specified file into your current file.
 - Works recursively: #include statements in the file that was inserted are also processed

#include

- How to interpret <> and “ ” is up to the compiler
 - In practice they are the same
 - Typically <> are used to import portions of the standard library

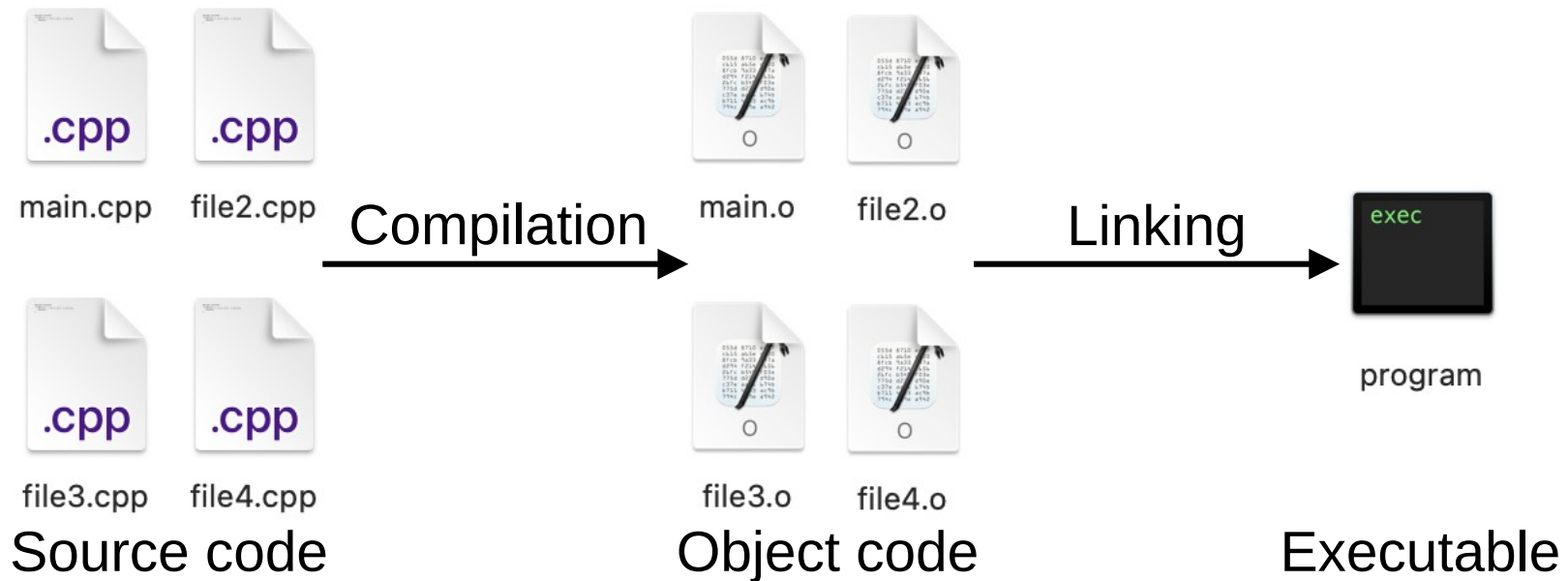
```
#include <amazingLibrary.h>  
#include "amazingLibrary.h"
```

Today

- The `#include` statement
- **A compilation of C++ compilation**
- The terminal

How is C++ code compiled?

Compilation is done into two stages; compilation and linking



Compilation

- Each file is converted into partially compiled code in isolation of other files

main.cpp

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



main.o

main:

```
    push    rax  
    call    sayHello()  
    xor     eax, eax  
    pop     rcx  
    ret
```

file2.cpp

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



file2.o

sayHello():

```
    push    rbx  
    mov     rbx, qword ptr [rip + std::cout@GOTPCREL]  
    lea     rsi, [rip + .L.str]  
    mov     edx, 6  
    mov     rdi, rbx  
    call    std::basic_ostream<char, std::char_traits<char>>& std::_
```

Compilation

- Each file is converted into partially compiled code in isolation of other files
- 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

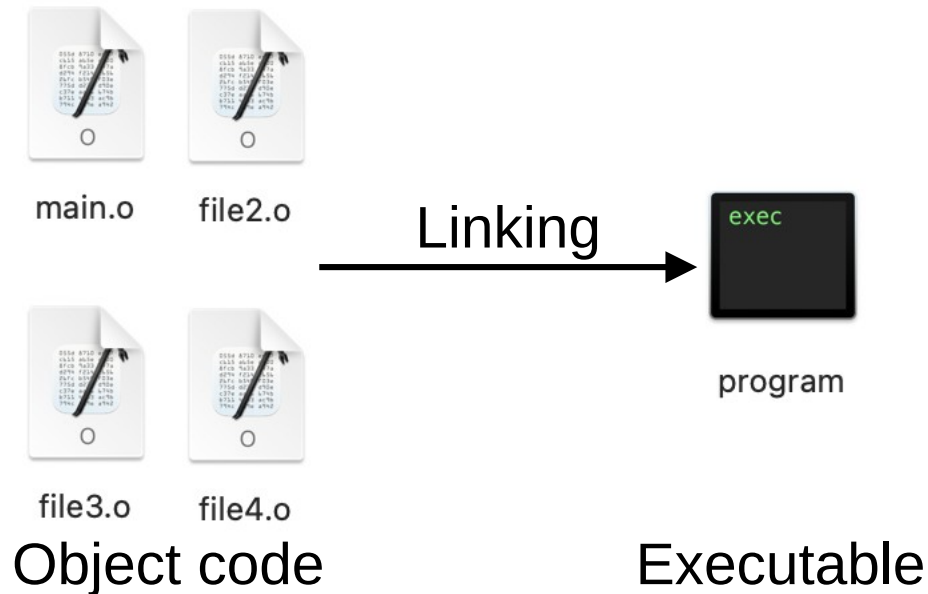
Function calls

```
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/1MoTb9Px>

Linking

- Combine all object code files into an executable file
 - Looks for function definitions across all files for each function call stub
 - Replaces each stub with a 'jump' instruction



Linking

- What this looks like for our example:

main.cpp

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



main.o

Defines:

- int main()

Calls:

- void sayHello()

file2.cpp

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



file2.o

Defines:

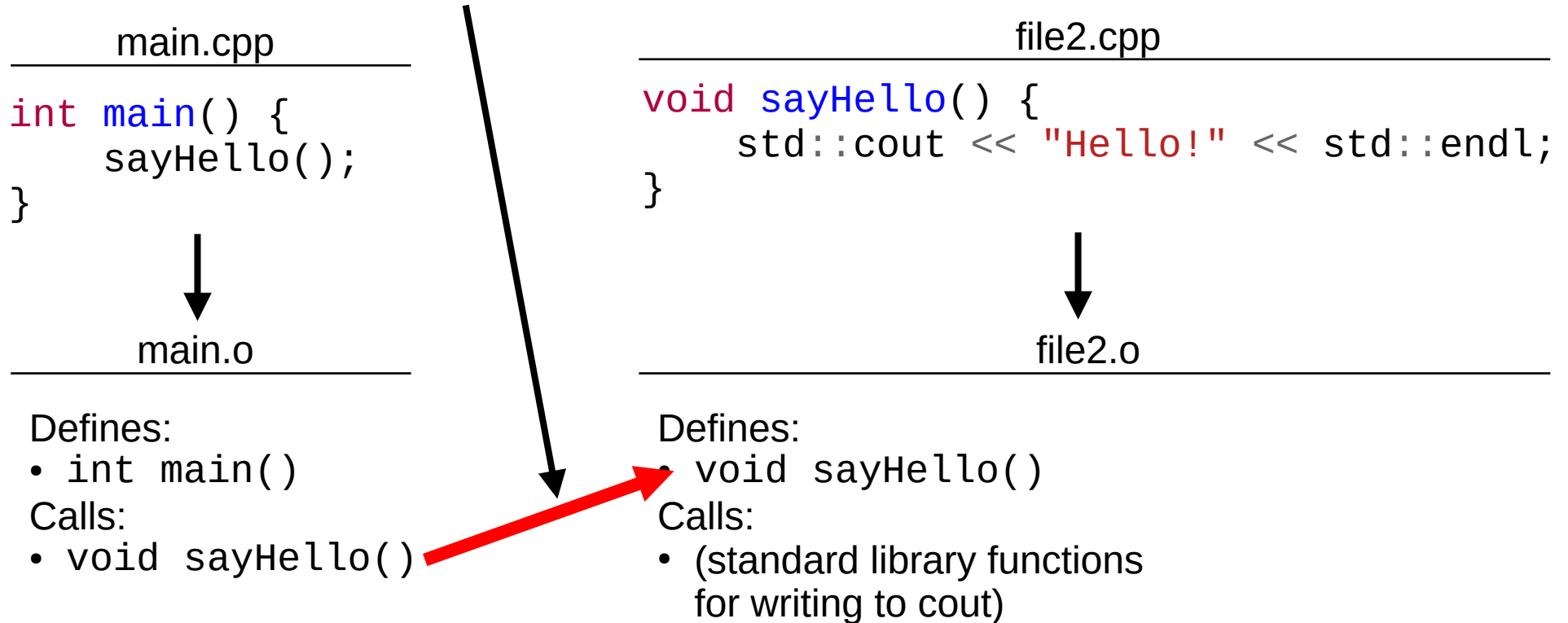
- void sayHello()

Calls:

- (standard library functions for writing to cout)

Linking

- Matching function calls to definitions is the primary job of the linker



Why compile files separately?

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 unique [name+parameters]

```
/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)
```

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.



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;  
}
```

#pragma once

- Avoids reading the same header file more than once
- Risks infinite loops if two files include each other

```
In file included from ../file2.h:1:  
In file included from ../file1.h:1:  
In file included from ../file2.h:1:  
In file included from ../file1.h:1:  
In file included from ../file2.h:1:  
In file included from ../file1.h:1:  
In file included from ../file2.h:1:  
In file included from ../file1.h:1:  
In file included from ../file2.h:1:  
In file included from ../file1.h:1:  
In file included from ../file2.h:1:  
In file included from ../file1.h:1:  
In file included from ../file2.h:1:  
In file included from ../file1.h:1:  
In file included from ../file2.h:1:  
..../file1.h:1:10: error: #include nested too deeply  
    1 | #include "file2.h"  
      ^
```

file1.h

```
#include "file2.h"
void sayHello();
```

file2.h

```
#include "file1.h"
void sayGoodbye();
```

#pragma once

- Avoids reading the same header file more than once
- Risks infinite loops if two files include each other
- Must be the first line in the header file!

file1.h

→ #pragma once
#include "file2.h"
void sayHello();

Now our program compiles:

[2/2] Linking target program

file2.h

→ #pragma once
#include "file1.h"
void sayGoodbye();

What caused our errors?

Let's take a look at
common errors!

main.cpp

```
#include "file2.h"

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

file2.h

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

file2.cpp

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

Error: use of undeclared identifier

- **Compilation:** **fails**
main.cpp lacks declaration of sayHello()
- **Linking:** **would succeed (but compilation fails before it gets there)**
Definitions for all used functions exist

main.cpp

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

file2.h

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

file2.cpp

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

Error: use of undeclared identifier

- Identifier: name of a variable, object, data type, or function
- Common causes:
 - Typo
 - Forgot to include header file in which it is declared
 - The thing exists, but is declared further down, and the compiler has not reached that point in the compilation process.

Error: undefined symbol / undefined reference to

- **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>
```

Error: undefined symbol / undefined reference to

- One or more functions were declared, but no .cpp file in the project contains a definition
- Common causes:
 - Definition does not match declaration
 - Source file that contains definition is not part of the project
 - Our extension does not add .cpp files in subfolders, only those in the «root» of the project
 - Forgot to create a main function

Error: duplicate symbol / redefinition of

- **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;  
}
```

Error: duplicate symbol / redefinition of

- **Compilation:** succeeds

Each file has the declarations it needs

- **Linking:** fails

Multiple definitions of `sayHello()` exist across all files

main.cpp

```
#include "file2.cpp"
```

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

file2.h

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

file2.cpp

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

Error: duplicate symbol / redefinition of

- One or more .cpp files contain a definition of the same function or object
- Common causes:
 - #include a .cpp file
 - A .h file contains a definition

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)

Task: Compiler complaints

- Setup:
 - Open a new folder in VS Code
 - Use Create new project > Lecture 05 > Task - include
 - Compile the project (it will fail)
- Objective:
 - What can you do to get the program to compile?
 - Why did that fix the problem?
 - How many unique error types can you get the compiler to print? (hint: try to cause the problems we discussed)
Example: **error: use of undeclared identifier**
- Rules:
 - You can only modify files by adding or removing `#include` statements.

Today

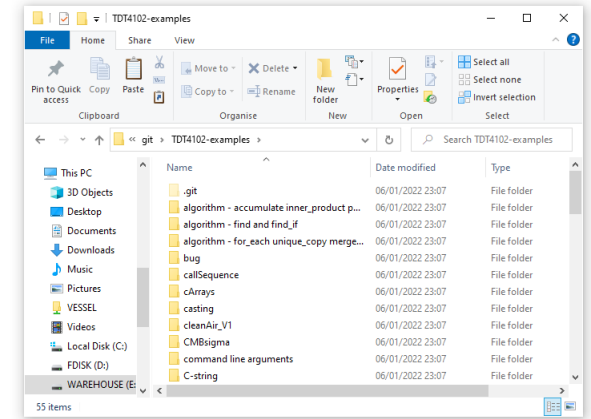
- The `#include` statement
- A compilation of C++ compilation
- **The terminal**

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

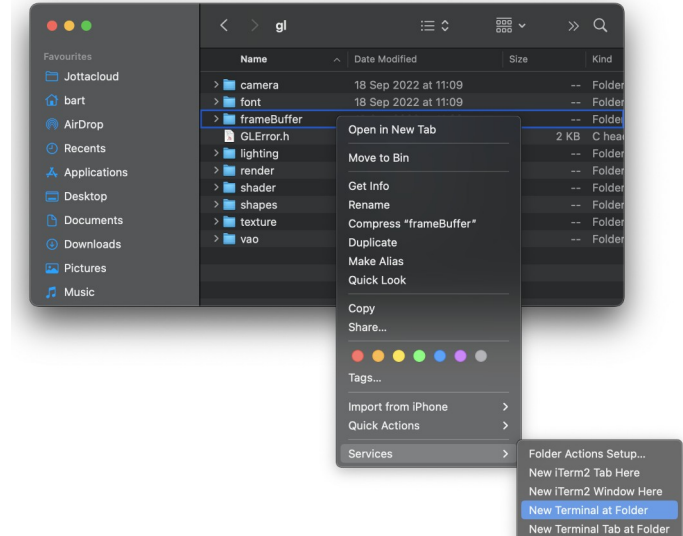
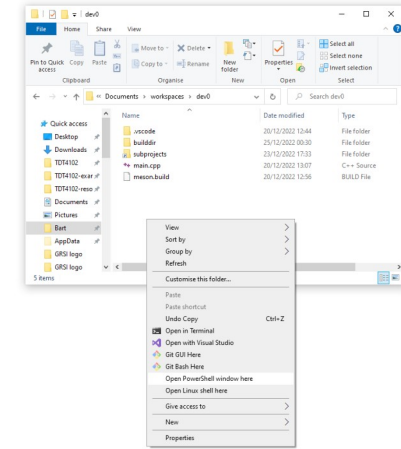
- List the contents of the current directory:
`ls`
- Change the current directory:
`cd [directory]`
- Show the path of the current directory:
`echo "${PWD}"`

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

- The `#include` statement
- A compilation of C++ compilation
- The terminal