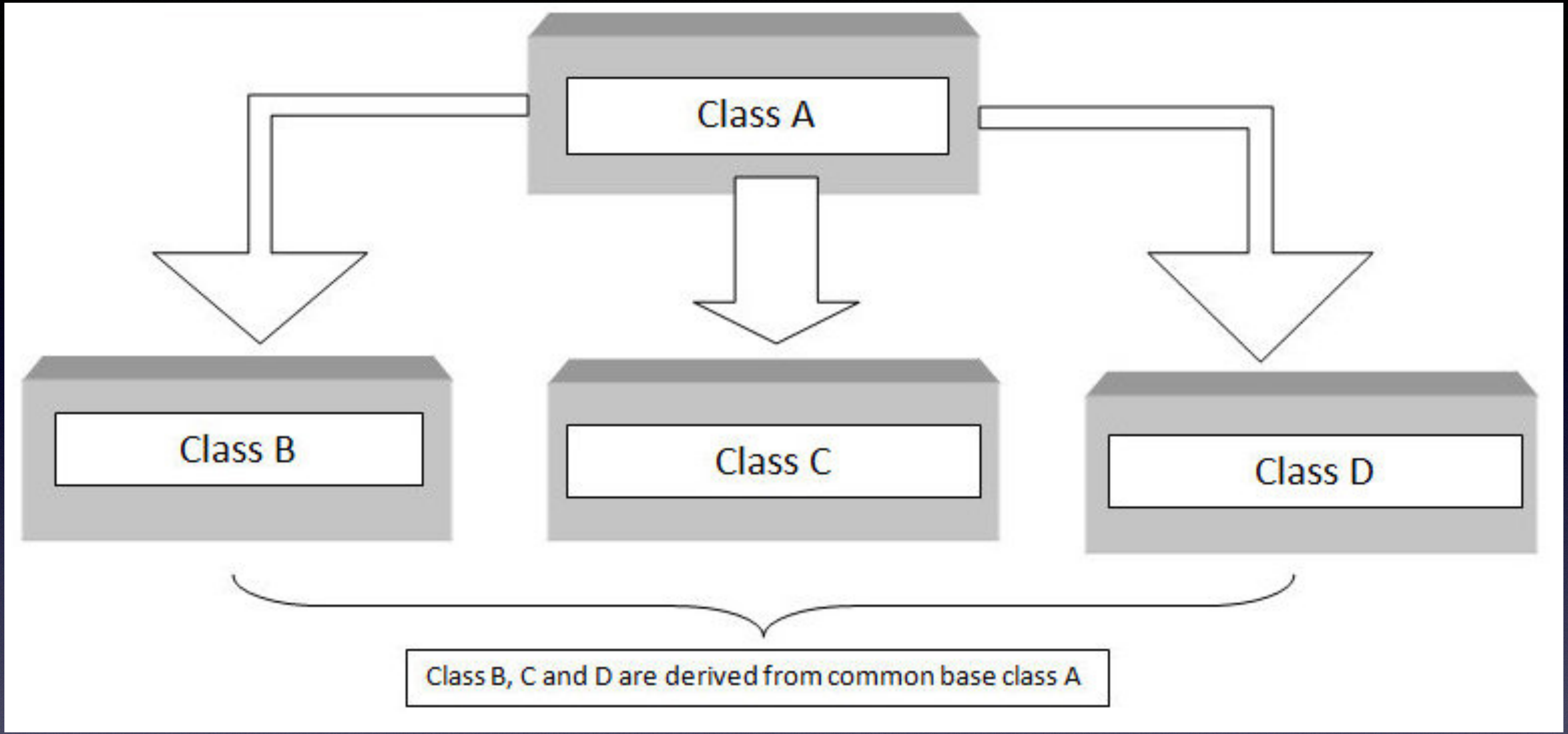
CS4414 Recitation 4

Continuing on with classes. And a bit on compiling.

02/17/2023

Ricky Takkar



http://www.trytoprogram.com/cplusplus-programming/hierarchical-inheritance/

Part 1/2

Continuing on with classes

Recap: Constructors

- A constructor has the same name as the class and no return type. It can have as many arguments as needed (just like a regular function)
- You can write as many constructors as you need
- E.g.,
  - myClass();
  - myClass(int x, std::string str);
  - myClass(someOtherClass otherClassObject) *and so on*

Recap: Constructors

- Special constructors:
  - Default constructor – takes no arguments
  - Copy constructor (careful with this!) – `myClass(const myClass& other);`
  - Move constructor – `myClass(myClass&& other);` **see Ed post #71 for more info between "&" and "&&"*
- The compiler provides a default constructor (public) when no constructors are defined
- It also provides a default copy and a default move constructor unless the user defines them

Recap: Constructors

- Using the keywords **default** and **delete**, you can enable or disable a constructor
- What if you want to disable the copy constructor? For e.g., you want unique ownership of a resource and don't want it duplicated.
  - myClass(const myClass& other) = delete;
- What if you write a custom constructor that takes some arguments, but still want to keep a default constructor?
  - myClass() = default;

Constructors you may be familiar with

- Think of different ways to construct a **vector** object

```
    std::vector<int> numbers; // default constructor
    std::vector<int> numbers(5); // notice the parentheses, creates a vector of size 5, all 0s
    std::vector<int> numbers(5, 100); // all 5 elements initialized to 100
    std::vector<int> one_to_ten = {1, 2, 3, 4, 5, 6, 7,. 8, 9, 10}; // uses initializer list
    std::vector<int> numbers(one_to_ten); // one_to_ten is of type std::vector<int>, invokes the copy
constructor
```

- Vectors use dynamically allocated arrays to store elements. What happens when this array needs to grow to accommodate new elements?
    - A new array is allocated to which all elements are moved. This is **expensive**
- Tip: **reserve()** allocates sufficient memory to store specified number of elements in vector

Find the error

```cpp
#include <iostream>
#include <vector>

class myClass {
    public:
        myClass(int x) {}
    private:
        int myInt;
};

int main() {
    std::vector<myClass> myObjects(4);
    return 0;
}
```

```
⊗ rt398@en-ci-cisugcl18:~/recitation/4/examples$ g++ -std=c++2a -Wall constituent.cpp -o c
In file included from /usr/include/c++/9/bits/stl_tempbuf.h:60,
                 from /usr/include/c++/9/bits/stl_algo.h:62,
                 from /usr/include/c++/9/string:52,
                 from /usr/include/c++/9/bits/locale_classes.h:40,
                 from /usr/include/c++/9/bits/ios_base.h:41,
                 from /usr/include/c++/9/ios:42,
                 from /usr/include/c++/9/ostream:38,
                 from /usr/include/c++/9/iostream:39,
                 from constituent.cpp:1:
/usr/include/c++/9/bits/stl_construct.h: In instantiation of 'void std::_Construct(_T1*, _Args&& ...) [
with _T1 = myClass; _Args = {}]':
/usr/include/c++/9/bits/stl_uninitialized.h:545:18:   required from 'static _ForwardIterator std::__uni
nitialized_default_n_1<_TrivialValueType>::__uninit_default_n(_ForwardIterator, _Size) [with _ForwardIt
erator = myClass*; _Size = long unsigned int; bool _TrivialValueType = false]'
/usr/include/c++/9/bits/stl_uninitialized.h:601:20:   required from '_ForwardIterator std::__uninitiali
zed_default_n(_ForwardIterator, _Size) [with _ForwardIterator = myClass*; _Size = long unsigned int]'
/usr/include/c++/9/bits/stl_uninitialized.h:663:44:   required from '_ForwardIterator std::__uninitiali
zed_default_n_a(_ForwardIterator, _Size, std::allocator<_Tp>&) [with _ForwardIterator = myClass*; _Size
 = long unsigned int; _Tp = myClass]'
/usr/include/c++/9/bits/stl_vector.h:1603:36:   required from 'void std::vector<_Tp, _Alloc>::_M_defaul
t_initialize(std::vector<_Tp, _Alloc>::size_type) [with _Tp = myClass; _Alloc = std::allocator<myClass>
; std::vector<_Tp, _Alloc>::size_type = long unsigned int]'
/usr/include/c++/9/bits/stl_vector.h:509:9:   required from 'std::vector<_Tp, _Alloc>::vector(std::vect
or<_Tp, _Alloc>::size_type, const allocator_type&) [with _Tp = myClass; _Alloc = std::allocator<myClass
>; std::vector<_Tp, _Alloc>::size_type = long unsigned int; std::vector<_Tp, _Alloc>::allocator_type =
std::allocator<myClass>]'
constituent.cpp:12:37:   required from here
/usr/include/c++/9/bits/stl_construct.h:75:7: error: no matching function for call to 'myClass::myClass
()'
   75 |     { ::new(static_cast<void*>(__p)) _T1(std::forward<_Args>(__args)...); }
      |       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
constituent.cpp:6:9: note: candidate: 'myClass::myClass(int)'
    6 |         myClass(int x) {}
      |         ^~~~~~~
constituent.cpp:6:9: note:   candidate expects 1 argument, 0 provided
constituent.cpp:4:7: note: candidate: 'constexpr myClass::myClass(const myClass&)'
    4 | class myClass {
      |       ^~~~~~~
constituent.cpp:4:7: note:   candidate expects 1 argument, 0 provided
constituent.cpp:4:7: note: candidate: 'constexpr myClass::myClass(myClass&&)'
constituent.cpp:4:7: note:   candidate expects 1 argument, 0 provided
```

Solution:

Vector cannot default construct constituent objects

How to rectify

```cpp
#include <iostream>
#include <vector>

class myClass {
    public:
        myClass(int x) {}
    private:
        int myInt;
};

int main() {
    // std::vector<myClass> myObjects(4)
    std::vector<myClass> myObjects;
    myClass obj1(5);
    myClass obj2(7);
    myObjects.push_back(obj1);
    myObjects.push_back(obj2);

    return 0;
}
```

```
rt398@en-ci-cisugcl18:~/recitation/4/examples$ g++ -std=c++2a -Wall constituent.cpp -o c
```

Solution:
push_back() constructed elements. push_back() will invoke the copy constructor to copy objects into the vector

Bonus

```cpp
int main() {
    // std::vector<myClass> myObjects(4)
    std::vector<myClass> myObjects;
    // myClass obj1(5);
    // myClass obj2(7);
    // myObjects.push_back(obj1);
    // myObjects.push_back(obj2);
    myObjects.emplace_back(5);
    myObjects.emplace_back(7);

    return 0;
}
```

- To reduce copy operations (i.e., improve performance), one can use emplace_back() instead of push_back().
- Note the argument passed to emplace_back(): it matches that of myClass constructor.

Constructor initializer list

- Problem: How to construct constituent objects of a class in the constructor?
  - e.g.,
    - Suppose we have **Person(std::string name);**, constructor for Person
    - Next, we have Group constructor that contains three Person objects A, B, and C
    - How can we construct the Person objects, part of a group, in the constructor of Group?

Constructor initializer list

- Unlike Java, you cannot construct data members in the body of the constructor. In Java, you would do something like,

```
Group::Group() {
    this->A("Ken");
    this->B("Ricky");
    this->C("Alicia");
}
```

- But in C++, objects cannot be null. Member objects must be constructed when the enclosing class object is constructed.

Constructor initializer list

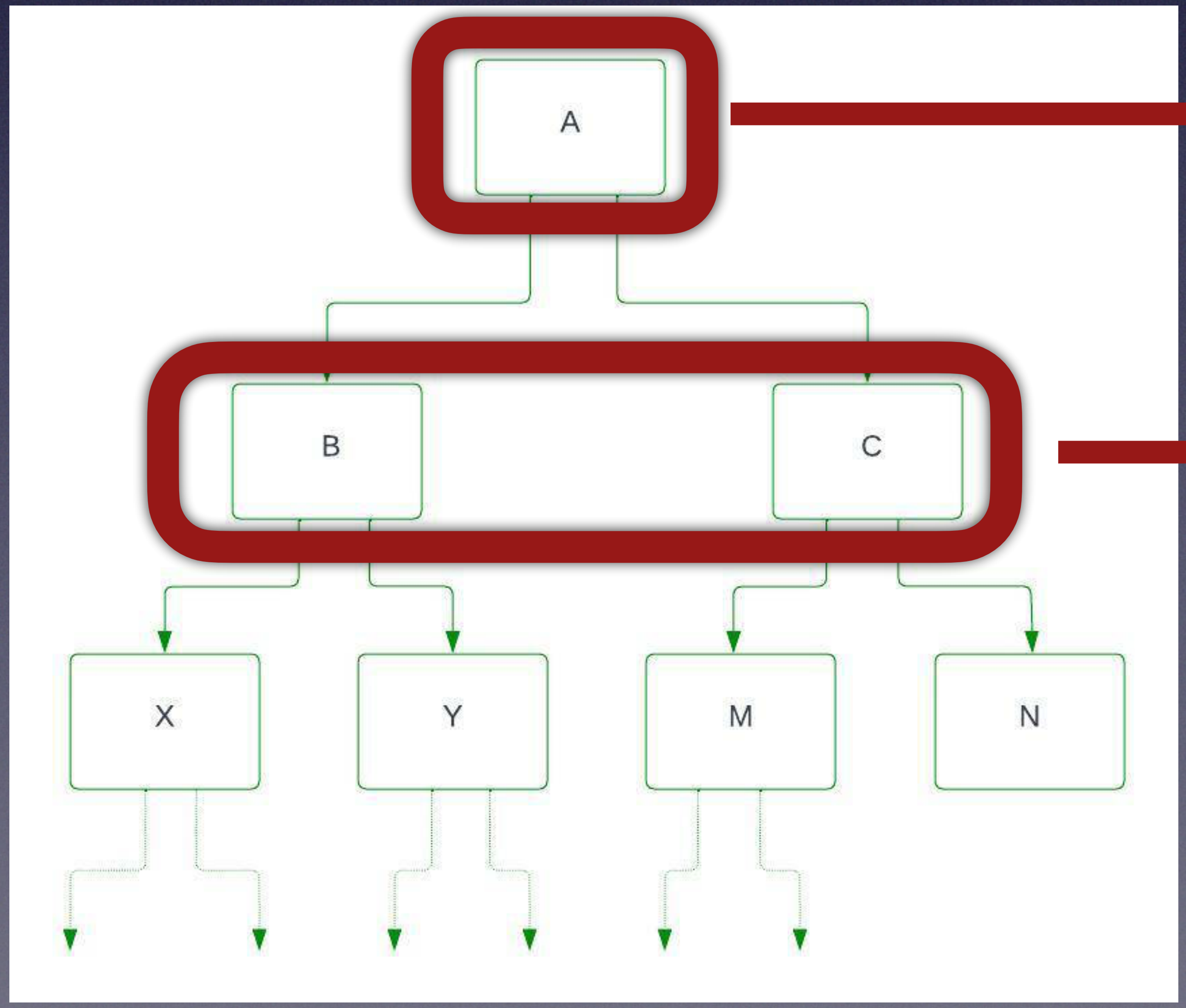
- So, the signature of the constructor and before the body, include a constructor initializer list

```
Group::Group(std::string name1, std::string name2,
std::string name3) : A(name1),
                     B(name2),
                     C(name3) {
      // body of constructor
}
```

- comma-separated list of the type class_member(args...)

Hierarchical Inheritance

- Sometimes, it's important to create a new (sub) class derived from some base class so objects of the derived class have both: access to inherited traits of the base class, but liberty to extend beyond…
  - e.g., child inherits traits of parents but also develops unique features



Base/Parent class

Derived/
Child class

https://www.geeksforgeeks.org/cpp-hierarchical-inheritance/

Hierarchical Inheritance

```
class BaseClass
{
// data members
// member functions
}
```

```
class DerivedClass1 : visibility_mode BaseClass
{
    // data members
    // member functions
}
```

```
class DerivedClass2 : visibility_mode BaseClass
{
    // data members
    // member functions
}
```

Recap: Access Specifiers

- 3 access specifiers for class variables and methods in C++:
  - **public** - accessible outside the class
  - **private** (default) - inaccessible outside the class
  - **protected** - only accessible to inherited classes outside the class itself. More on Inheritance later…

Hierarchical Inheritance: Visibility Mode

- Determines how base class features will be inherited by child class

class DerivedClass1 : **public** BaseClass
{ // body }

→ Access specifiers of base class maintained as is (private remains private, public remains pub…)

class DerivedClass2 : **private** BaseClass
{ // body }

→ Public and protected access specifiers from base become private (i.e., inaccessible by derived class objects)

class DerivedClass3 : **protected** BaseClass
{ // body }

→ Public members from base class become protected (while protected and private members remain as is).

Exercise: Fill in the blanks

Base Class	Derived Class	Derived Class	Derived Class
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not inherited	Not inherited	Not inherited

Function Overloading

- What happens if functions share the same name in the same scope?
  - No problem! As long as…
    - At compile time, the compiler can can choose which overload to use based on types and number of arguments passed in by caller

Function Overloading

- Both, free and member functions, can be overloaded

Overloading Considerations

Function declaration element	Used for overloading?
Function return type	No
Number of arguments	Yes
Type of arguments	Yes
Presence or absence of ellipsis	Yes
Use of `typedef` names	No
Unspecified array bounds	No
`const` or `volatile`	Yes, when applied to entire function
Reference qualifiers (`&` and `&&`)	Yes

https://learn.microsoft.com/en-us/cpp/cpp/function-overloading?view=msvc-170

What about function overloading with hierarchical inheritance?

```cpp
class BaseClass
{
public:
    int foo(int i)
    {
        std::cout << "foo(int): ";
        return i+1;
    }
};
```

```cpp
class DerivedClass : public BaseClass
{
public:
    double foo(double d)
    {
        std::cout << "foo(double): ";
        return d+1.1;
    }
};
```

```cpp
int main()
{
    DerivedClass dObject = DerivedClass();

    std::cout << dObject.foo(4) << std::endl;
    std::cout << dObject.foo(4.3) << std::endl;

    return 0;
}
```

Question: What will the program output?

A. foo(double): 5.1
foo(double): 5.4

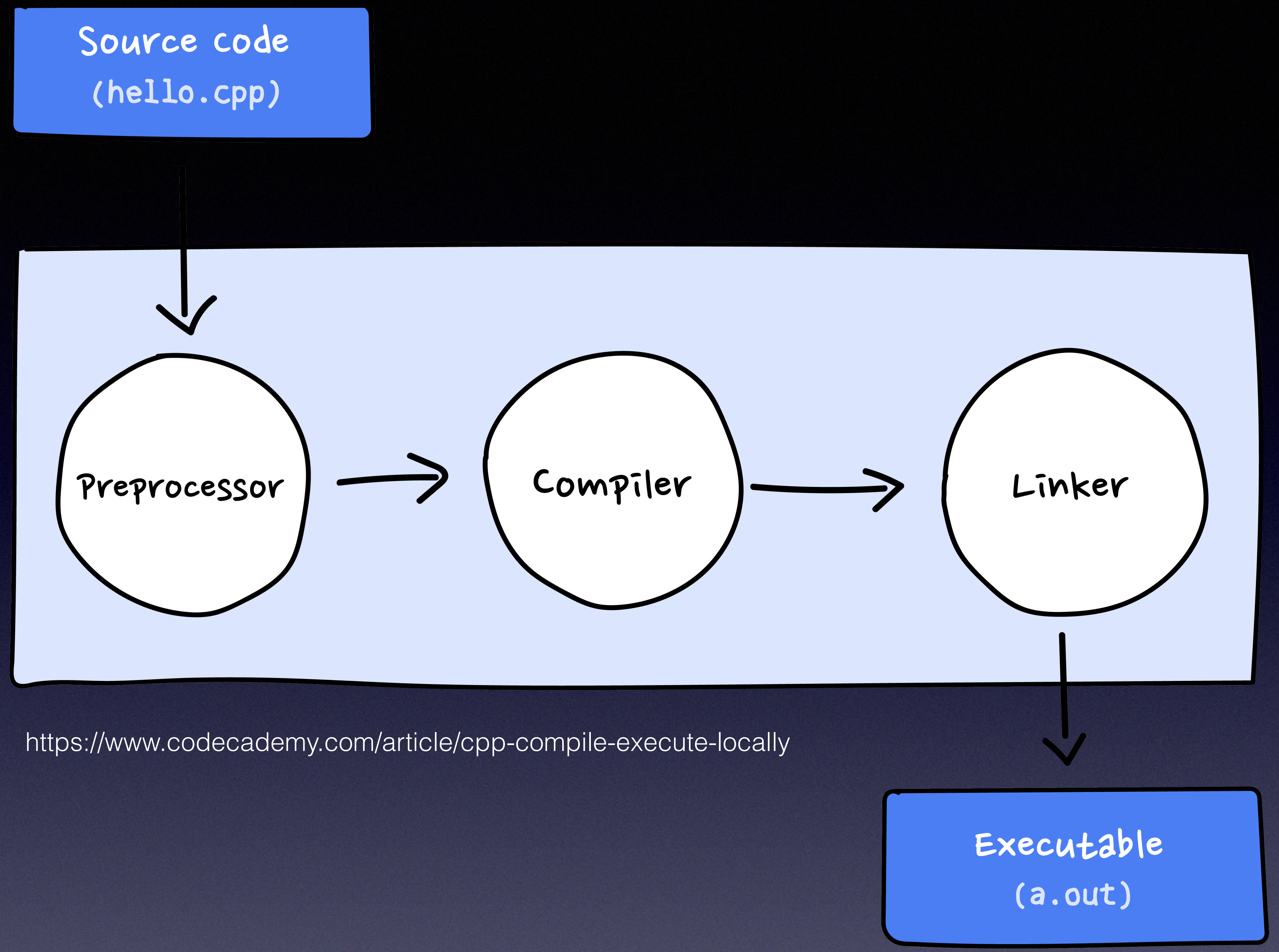
B. foo(int): 5
foo(double): 5.4

C. Error

No overload resolution between class hierarchy in C++

Exercise

- Using demo code from Recitation 3, implement
    - Hierarchical inheritance (hint: create some new classes that inherit from Student - feel free to modify Student)
    - Constructor initializer list
    - Function overloading



https://www.codecademy.com/article/cpp-compile-execute-locally

Part 2/2

A bit about compiling

Recap: Compiling Classes

- Run “**g++ -o exec_name main.cpp rest.cpp …**”
- Include all the cpp files in the g++ command
- Ignore header files in compilation command as they should be included in the cpp files
- Only one program should contain the main function (in the above example, main.cpp)

Journey of C++ Compilation

- Step 1: The preprocessor
  - Before the C++ compiler compiles, the source code file is **processed** by a **preprocessor**
  - The compiler automatically invokes the preprocessor
  - Preprocessor commands start with “#”, e.g., #include <iostream>

Journey of C++ Compilation

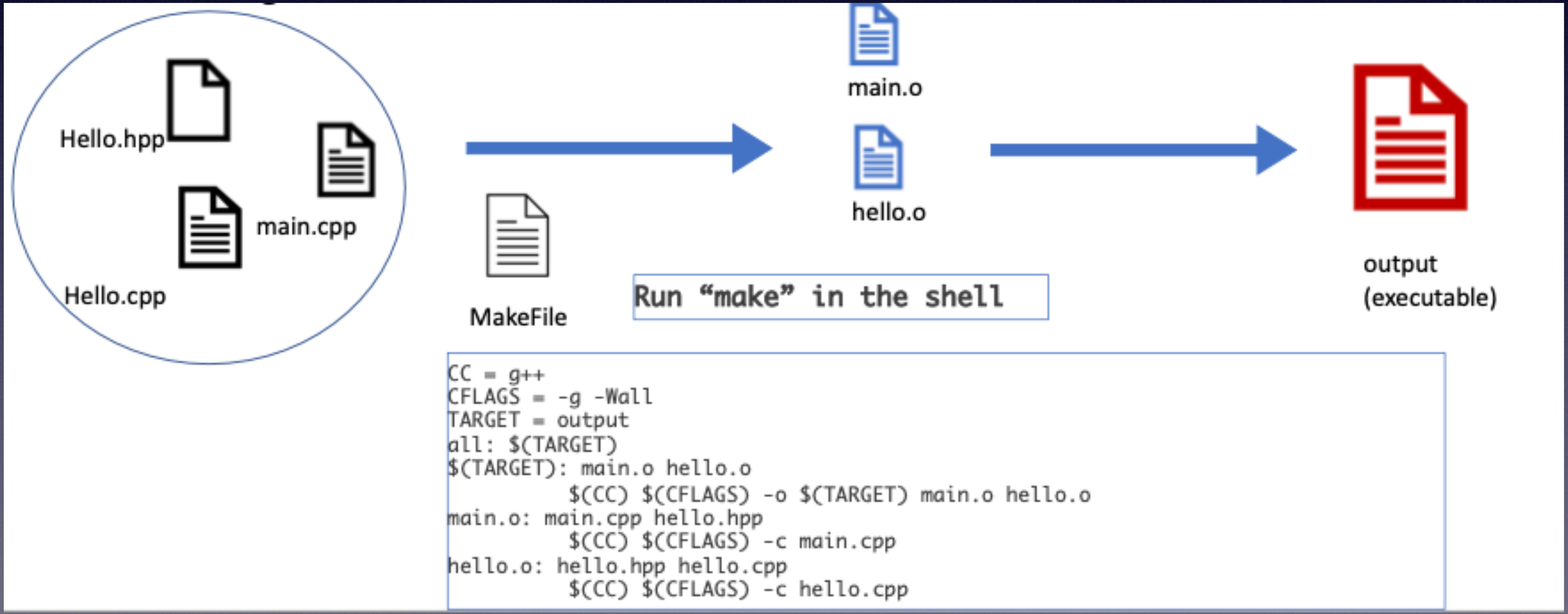
- Step 2: The compiler
  - By now, the compiler has included all header files and expanded #include statements
  - Compiler transforms C++ source code into **object code** file (*.o) containing binary version of source code
  - Object code file is not directly executable

Journey of C++ Compilation

- Step 3: The linker
  - Separate program called **ld** akin to preprocessor (also invoked automatically by compiler like preprocessor program)
  - Links together object files (including object files created from source code and pre-compiled object files collected into **library files** with *.a or *.so extension) into a single binary executable

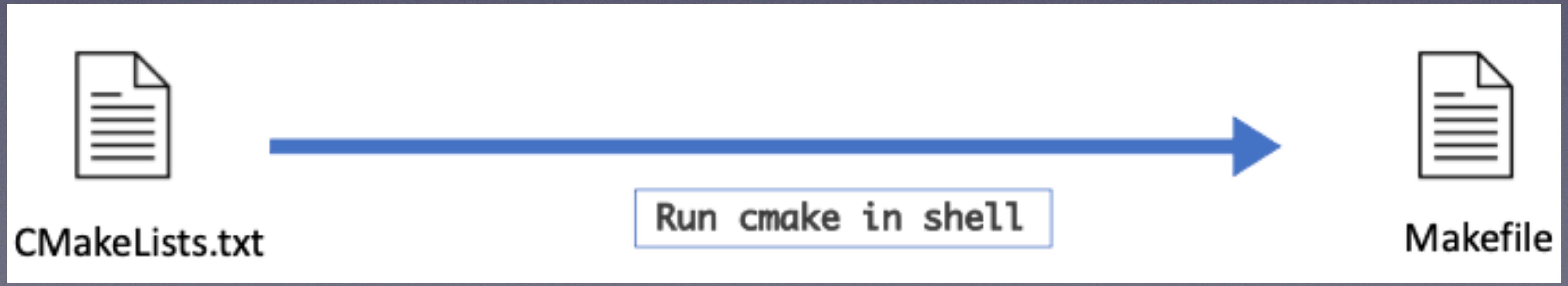
Build Files and Generate Executables Makefile

- Makefile is a special file containing shell commands executed by running the ‘make’ command inside the Makefile directory



Build Files and Generate Executables CMake

- Why CMake?
  - Makefiles are low-level, clunky creatures
  - CMake is a higher level language to automatically generate Makefiles
  - CMake contains more features, such as finding library, files, header files; it makes the linking process easier, and gives readable errors
- What is CMake?
  - CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.
  - CMakeLists.txt files in each source directory are used to generate Makefiles



Build Files and Generate Executables CMake

Example

cmakelists.txt

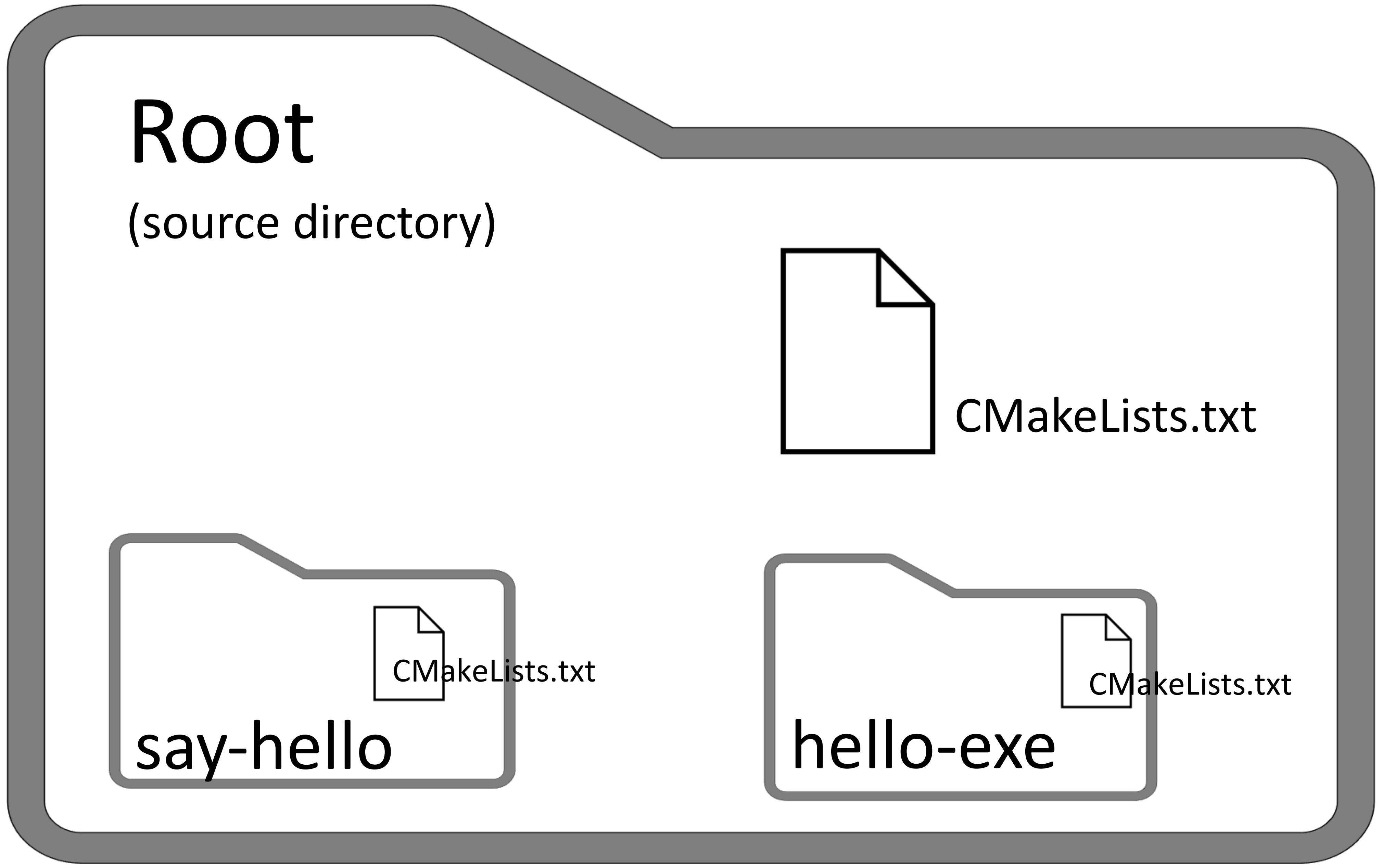
```
cmake_minimum_required(VERSION 3.10) # set the project
name project(MyProject) # add the executable
add_executable(output main.cpp)
```

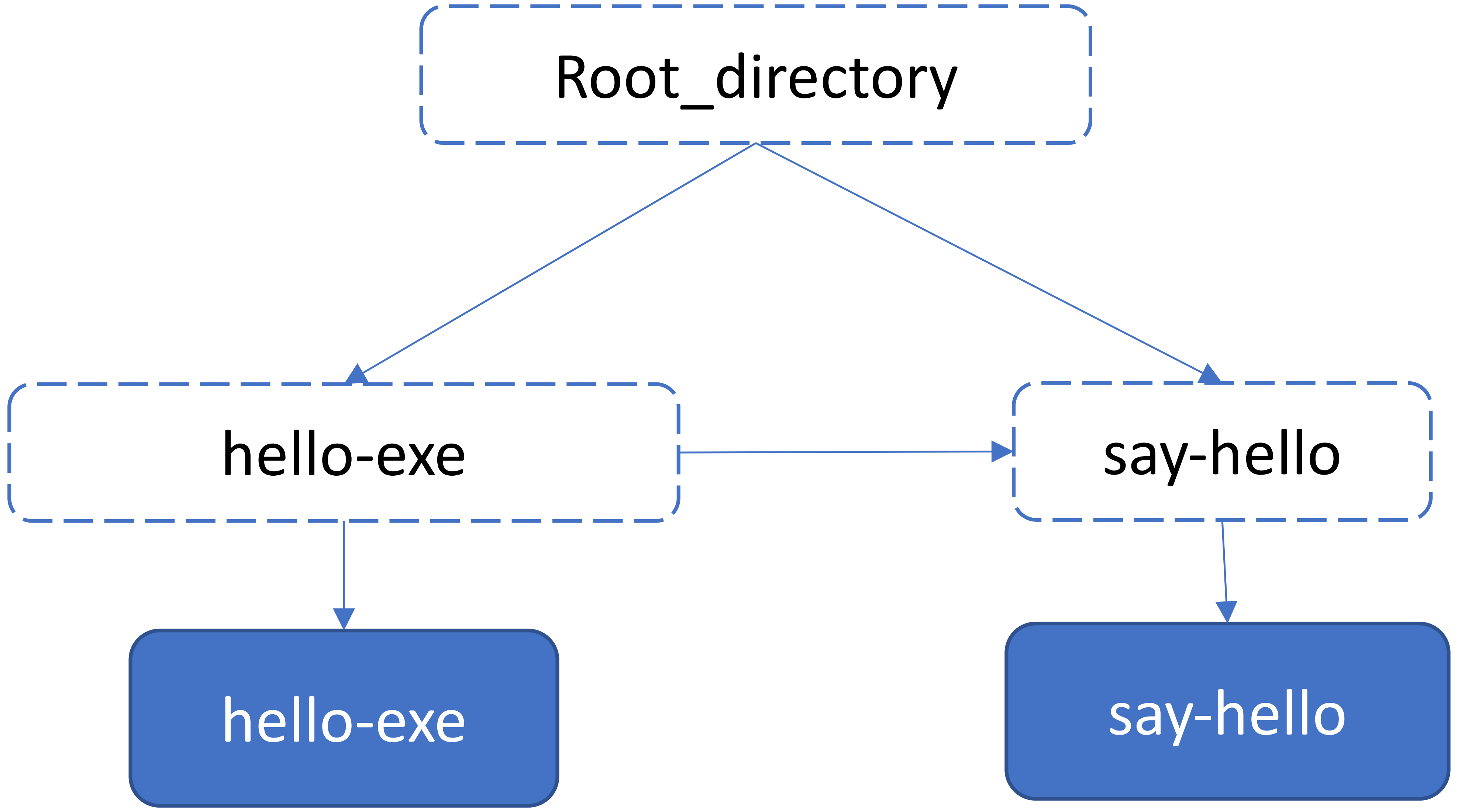
- Build and Run
  - Navigate to the source directory, and create a build directory
    - $ cd ./myproject & $ mkdir build
  - Navigate to the build directory, and run CMake to configure the project and generate a build system
    - $ cd build & $ cmake
  - Call build system to compile/link the project
    - Either run $ make
    - Or run $ cmake-build .

Cmake

3. Cmake with subdirectory

- CMakeLists.txt files placed in each source directory are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC).
- CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree.





Demo

Cmake

3. Cmake with subdirectory

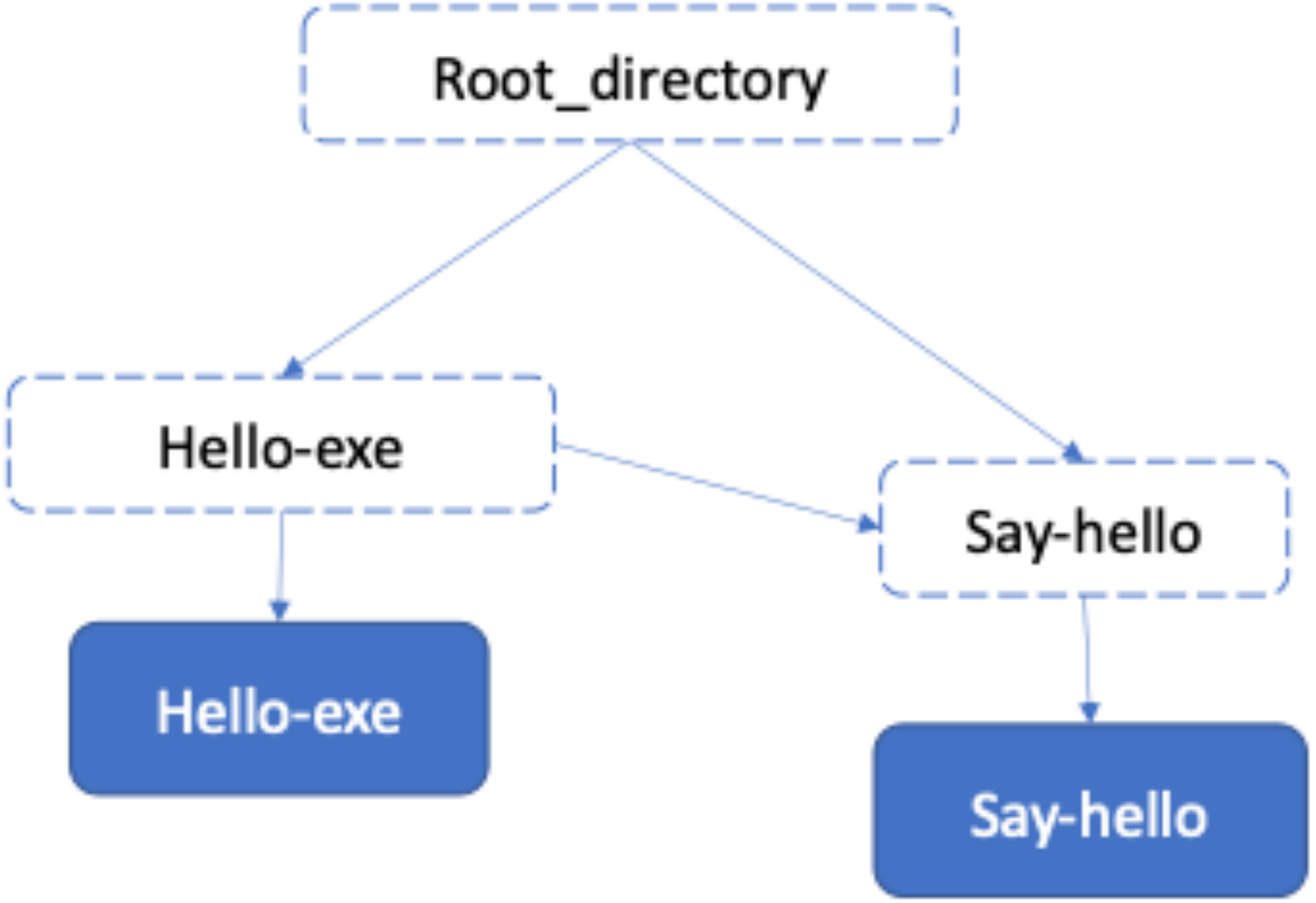
- add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
- Adds a subdirectory to the build. The source_dir specifies the directory in which the source CMakeLists.txt and code files are located.

```
cmake_minimum_required(VERSION 3.12)
project(MyProject VERSION 1.0.0)

add_subdirectory(say-hello)
add_subdirectory(hello-exe)
```

```
add_executable(hell
o_exe main.cpp)

target_link_librari
es(hello_exe
PRIVATE say-hello)
```




```
add_library(
    say-hello
    hello.hpp
    hello.cpp
)

target_include_directories
(say-hello PUBLIC
"${CMAKE_CURRENT_SOURCE_DI
R}")

target_compile_definitions
(say-hello PUBLIC
SAY_HELLO_NUM=5)
```

Cmake

- target_include_directories(<target> [SYSTEM] [AFTER|BEFORE]

  <INTERFACE|PUBLIC|PRIVATE> [items1...])

- Set include directory properly

- The PUBLIC, PRIVATE and INTERFACE keywords can be used to specify both the link dependencies and the link interface in one command.
  - PUBLIC(default): All the directories following PUBLIC will be used for the current target and the other targets that have dependencies on the current target
  - PRIVATE: All the include directories following PRIVATE will be used for the current target only
  - INTERFACE: All the include directories following INTERFACE will NOT be used for the current target but will be accessible for the other targets that have dependencies on the current target

References

1. https://www.youtube.com/watch?v=HcESuwmlHEY

2. https://www.simplilearn.com/tutorials/cpp-tutorial/hierarchical-inheritance-in-cpp

3. https://www.geeksforgeeks.org/does-overloading-work-with-inheritance/

4. http://courses.cms.caltech.edu/cs11/material/cpp/mike/misc/compiling_c++.html