C/C++ Program Design

LAB 10

CONTENTS

- Learn cmake
- Learn the concept of storage duration, scope and linkage
- Learn to use namespaces

2 Knowledge Points

2.1 Cmake

2.2 Storage duration, Scope and Linkage

2.3 Namespaces

2.1 CMake

What is CMake?

Cmake is an open-source, cross-platform family of tools designed to build, test and package software. Cmake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice.

CMake needs **CMakeLists.txt** to run properly.

A CMakeLists.txt consists of **commands** , **comments** and **spaces**.

- The **commands** include command name, brackets and parameters , the parameters are separated by spaces. Commands are not case sensitive.
- **Comments** begins with ‘#’.

Steps for generating a makefile and compiling on Linux using Cmake:

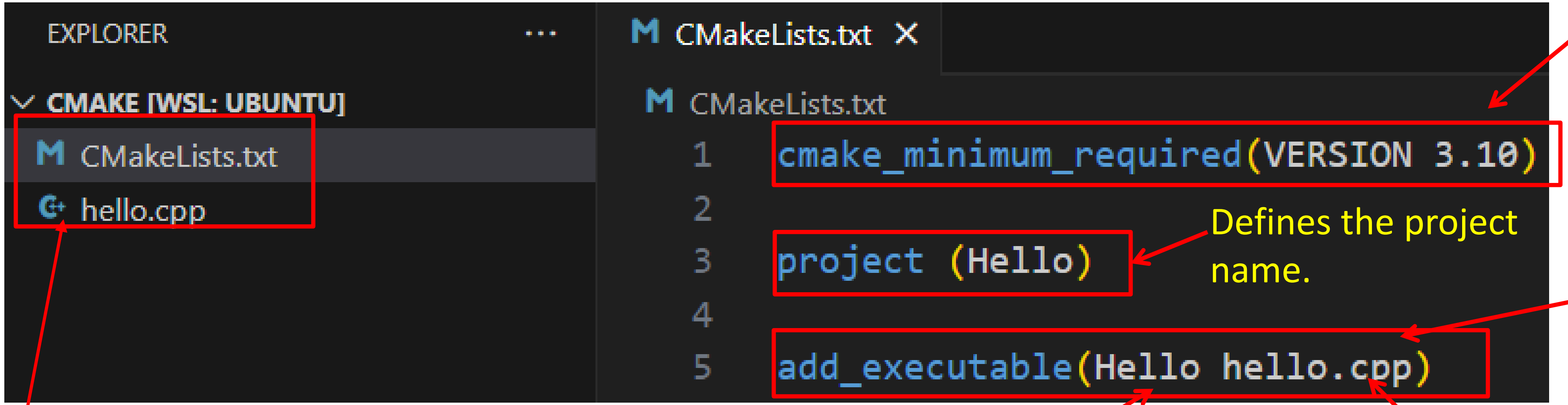
Step1: Writes the CMake configuration file **CMakeLists.txt**.

Step2: Executes the command **cmake PATH** to generate the **Makefile**.
(PATH is the directory where the CMakeLists.txt resides.)

Step3: Compiles using the **make** command.

1. A single source file in a project

The most basic project is an executable built from source code files. For simple projects, a three-line **CMakeLists.txt** file is all that is required.



```
cmake_minimum_required(VERSION 3.10)

project (Hello)

add_executable(Hello hello.cpp)
```

Specifies the minimum required version of CMake. Use **cmake --version** in Vscode terminal window to check the cmake version in your computer.

Defines the project name.

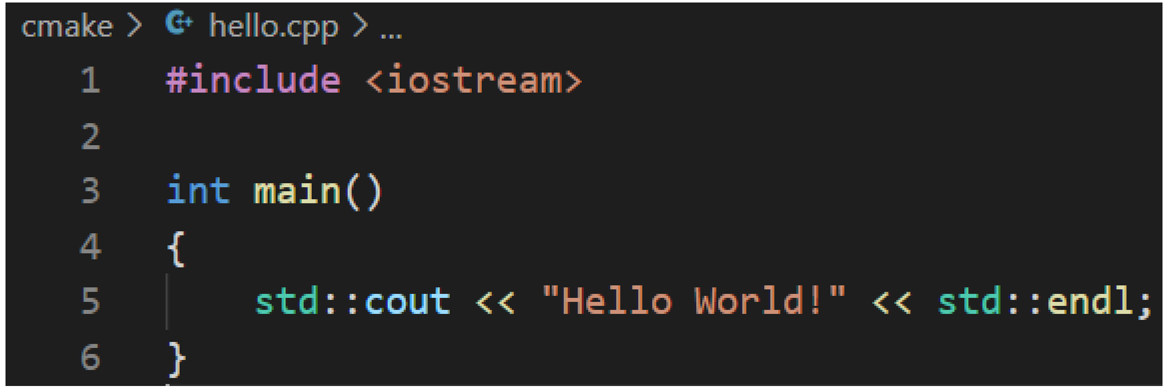
Adds the Hello executable target which will be built from hello.cpp.

The first parameter indicates the filename of executable file.

The second parameter indicates the source file.

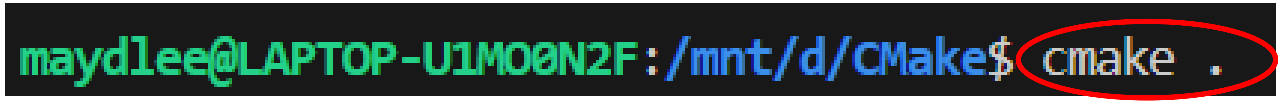
Store the CMakeLists.txt file in the same directory as the hello.cpp.

Suppose there is a hello.cpp

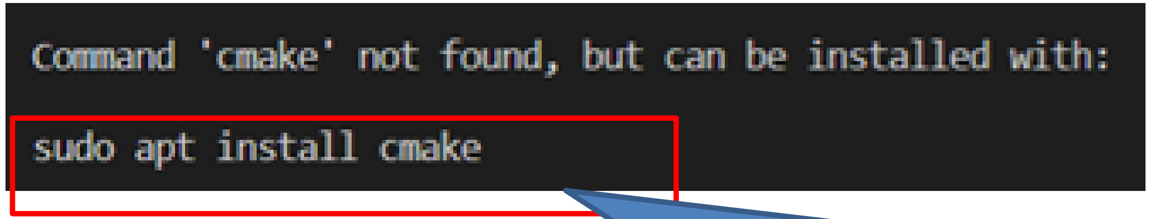


```
#include <iostream>

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

In current directory, type **cmake .** to generate makefile. If cmake does not be installed, follow the instruction to intall cmake.

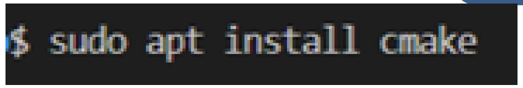


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ cmake .
```

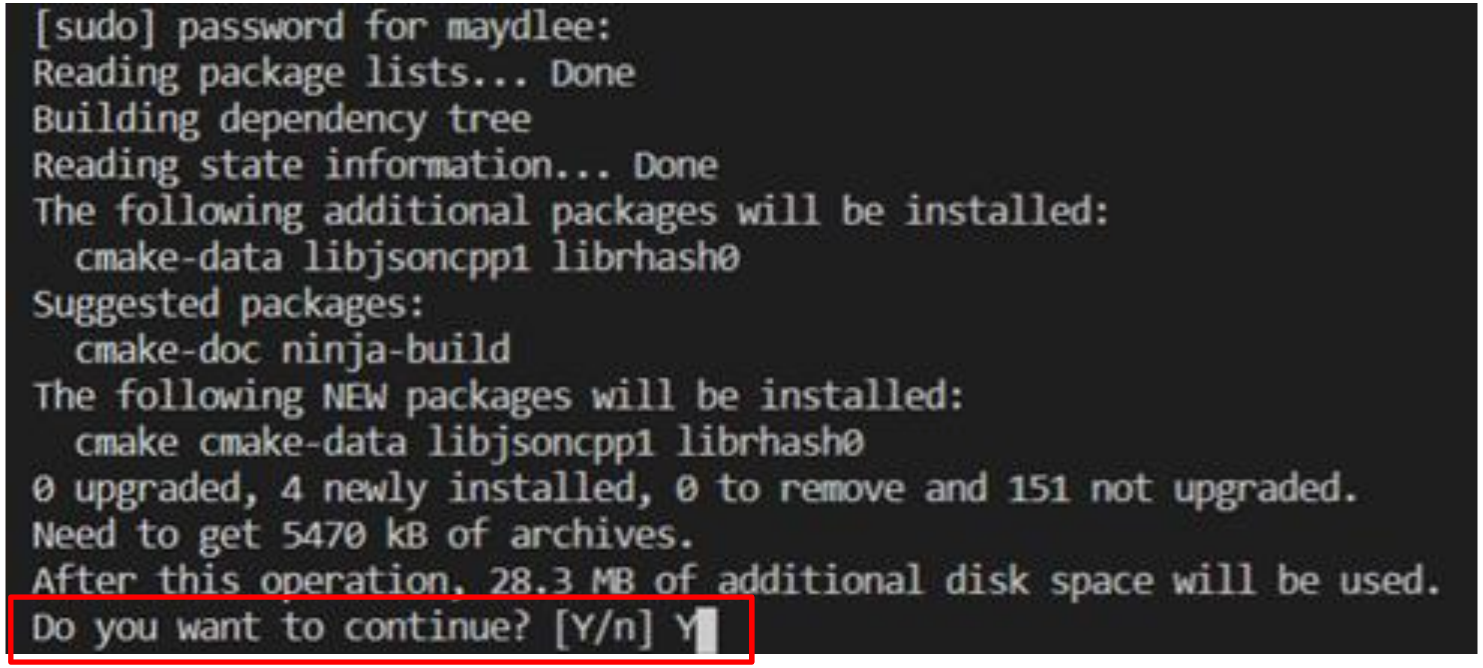


```
Command 'cmake' not found, but can be installed with:

sudo apt install cmake
```

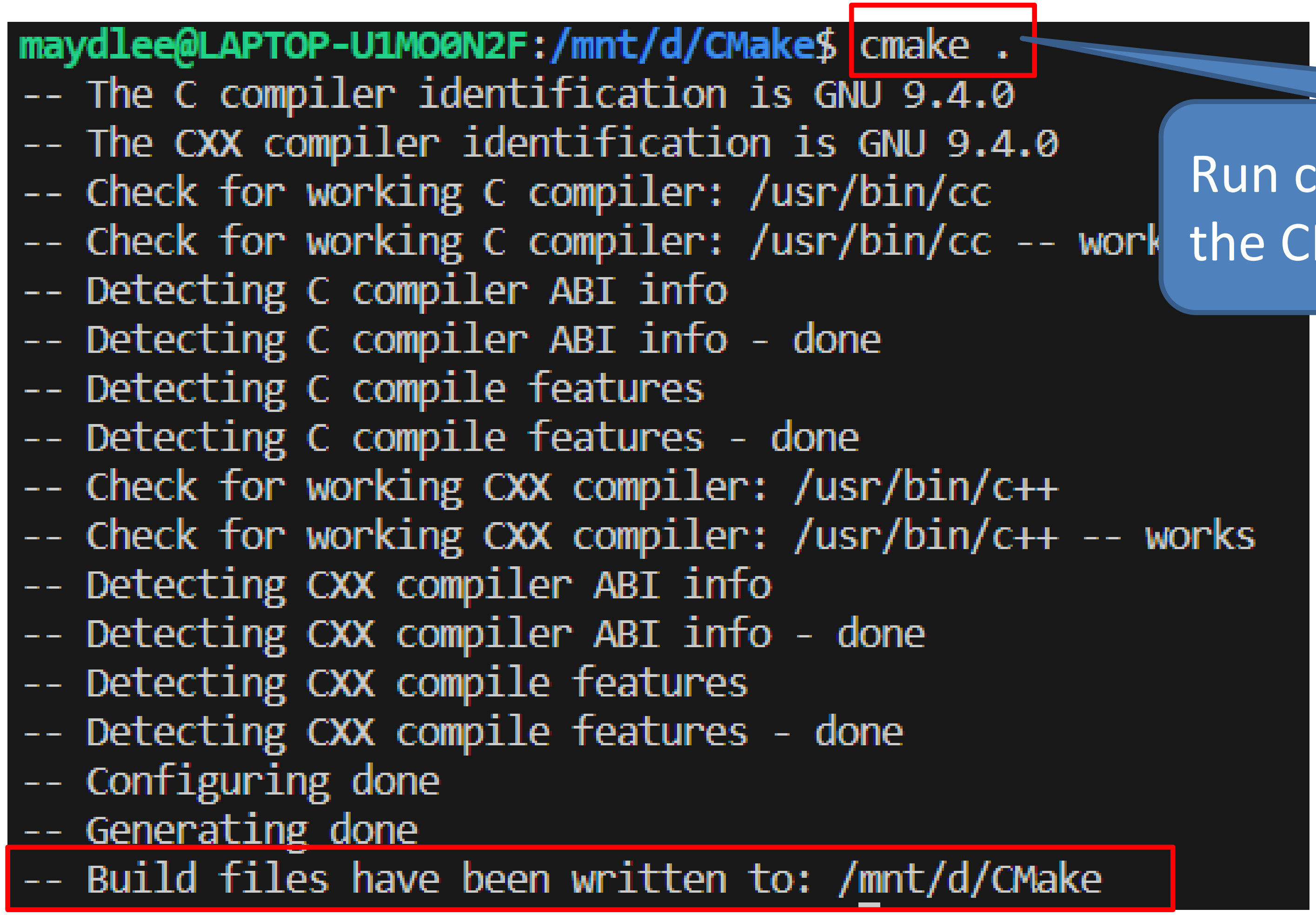
Install cmake first by instruction



```
$ sudo apt install cmake
```



```
[sudo] password for maydlee:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  cmake-data libjsoncpp1 librhash0
Suggested packages:
  cmake-doc ninja-build
The following NEW packages will be installed:
  cmake cmake-data libjsoncpp1 librhash0
0 upgraded, 4 newly installed, 0 to remove and 151 not upgraded.
Need to get 5470 kB of archives.
After this operation, 28.3 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

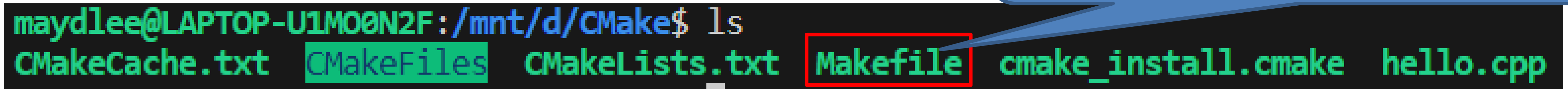


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ cmake .
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- work
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/CMake
```

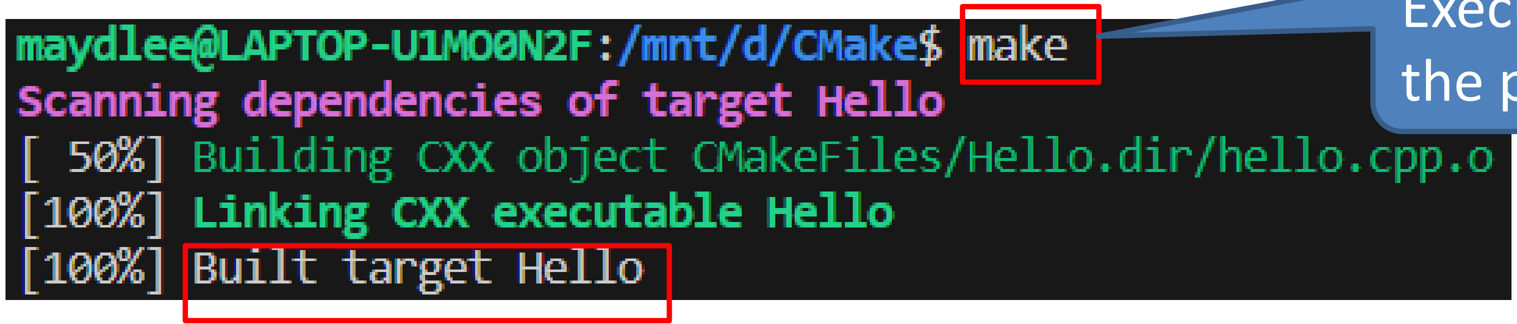
Run cmake to generate makefle, . indicates the CMakeList.txt is in the current directory.

生成的文件的位置

Makefile file is created automatically after running cmake in the current directory.

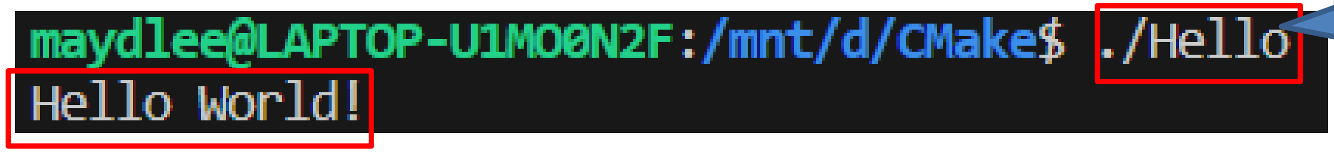


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ ls
CMakeCache.txt  CMakeFiles  CMakeLists.txt  Makefile  cmake_install.cmake  hello.cpp
```



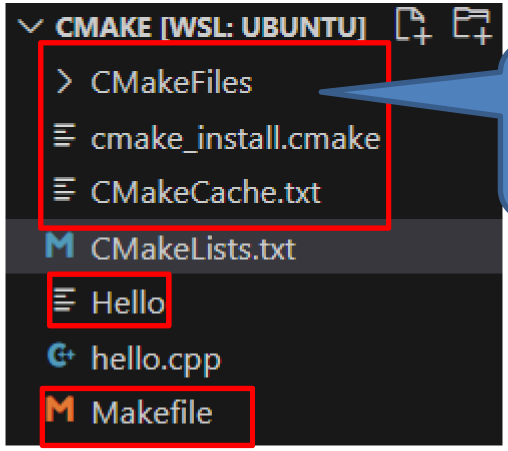
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ make
Scanning dependencies of target Hello
[ 50%] Building CXX object CMakeFiles/Hello.dir/hello.cpp.o
[100%] Linking CXX executable Hello
[100%] Built target Hello
```

Execute make to compile the program.



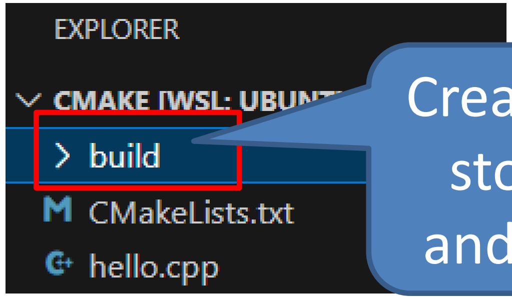
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ ./Hello
Hello World!
```

Run the program



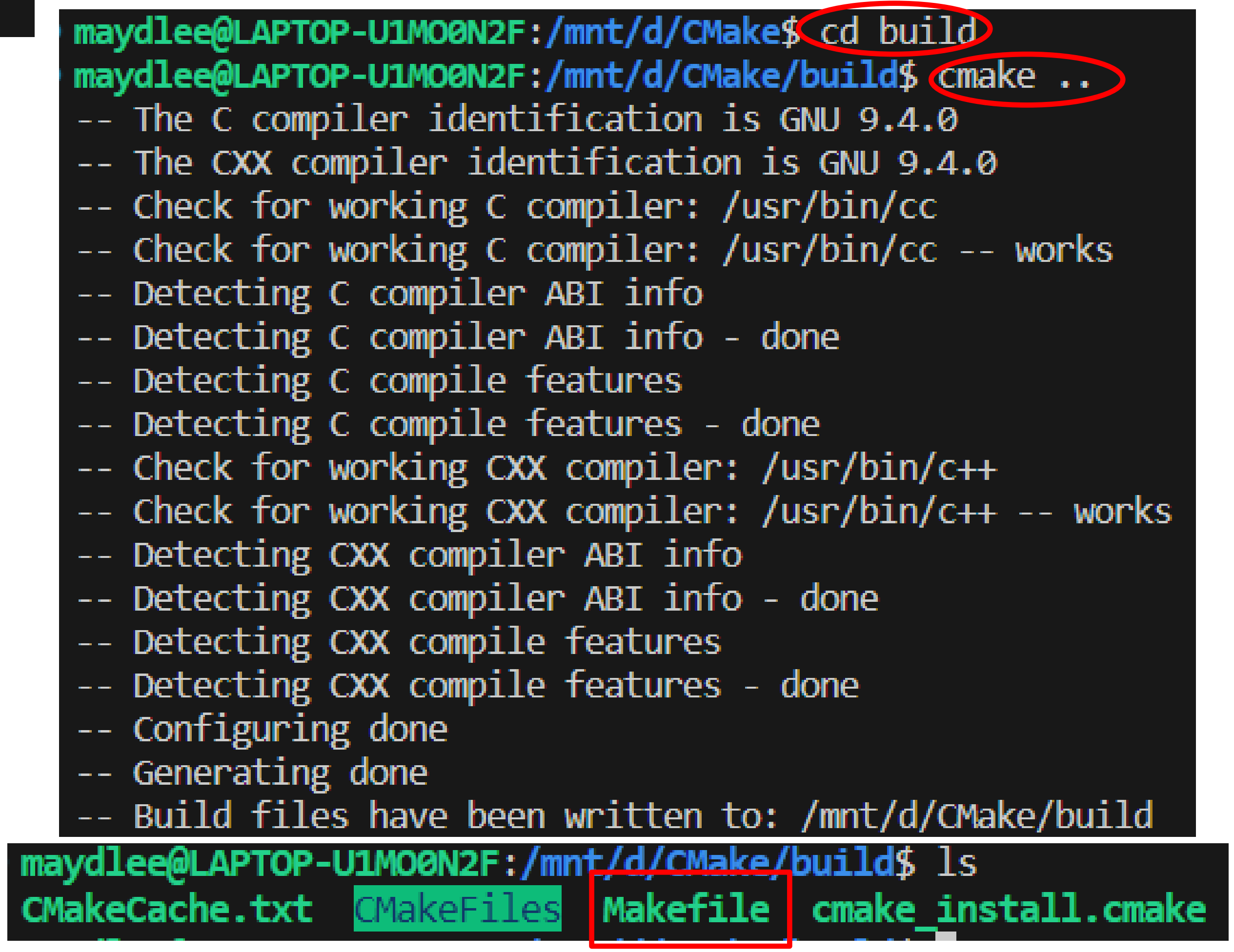
CMAKE [WSL: UBUNTU]
CMakeFiles
cmake_install.cmake
CMakeCache.txt
CMakeLists.txt
Hello
hello.cpp
Makefile

Delete all the building files and directory by CMake.

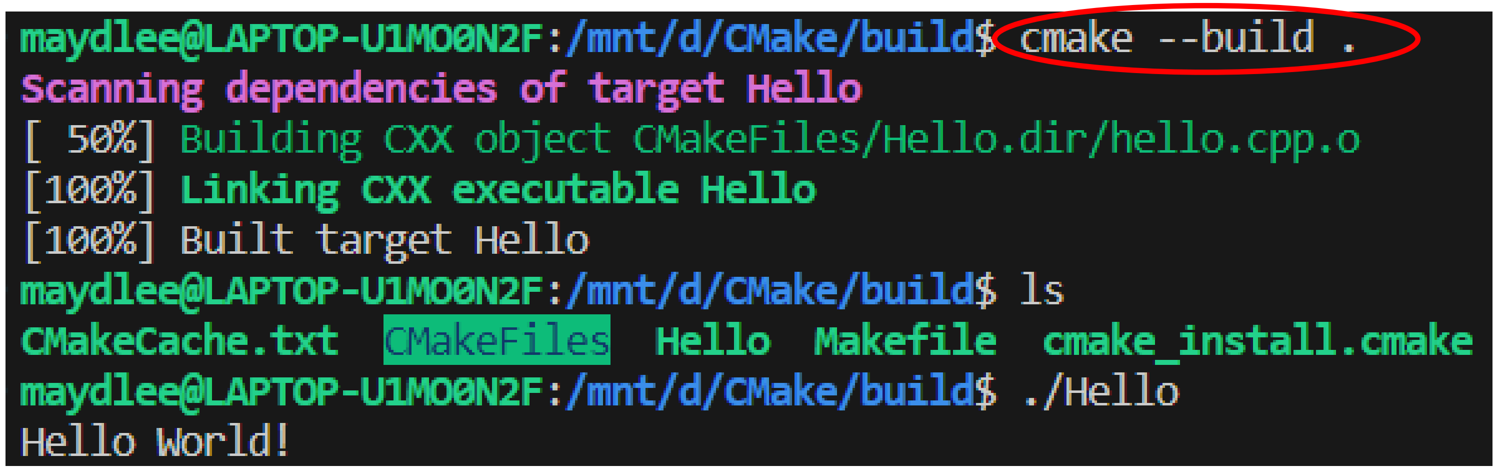


EXPLORER
CMAKE [WSL: UBUN
build
CMakeLists.txt
hello.cpp

Create an empty folder to store the building files and directory by CMake.

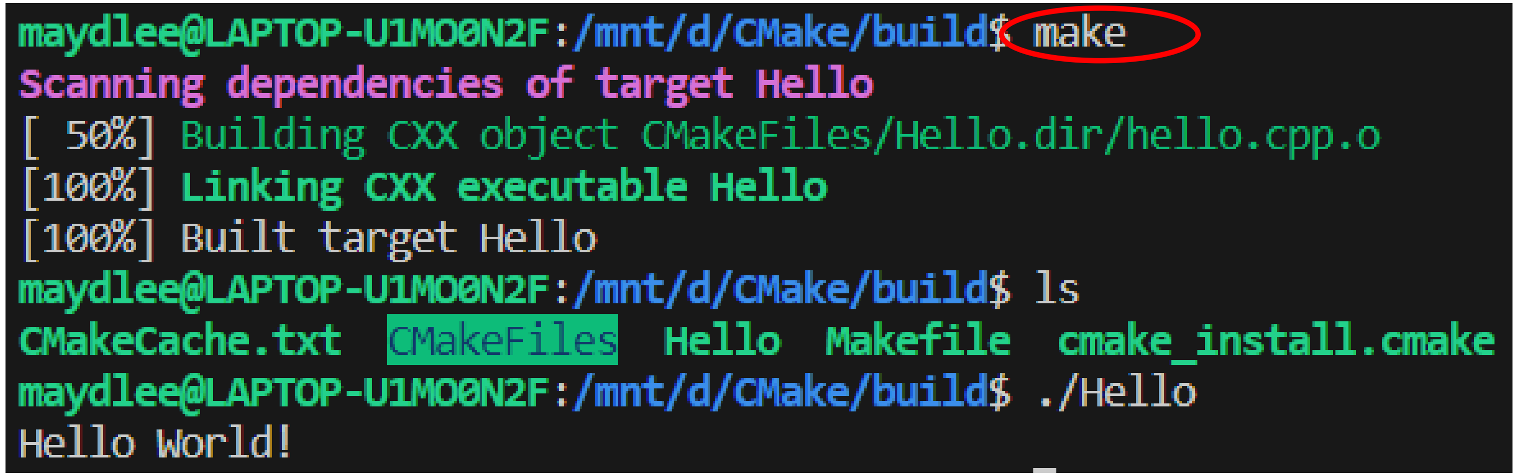


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ cd build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/build$ cmake ..
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/CMake/build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/build$ ls
CMakeCache.txt  CMakeFiles  Makefile  cmake_install.cmake
```



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/build$ cmake --build .
Scanning dependencies of target Hello
[ 50%] Building CXX object CMakeFiles/Hello.dir/hello.cpp.o
[100%] Linking CXX executable Hello
[100%] Built target Hello
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/build$ ls
CMakeCache.txt  CMakeFiles  Hello  Makefile  cmake_install.cmake
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/build$ ./Hello
Hello World!
```

or

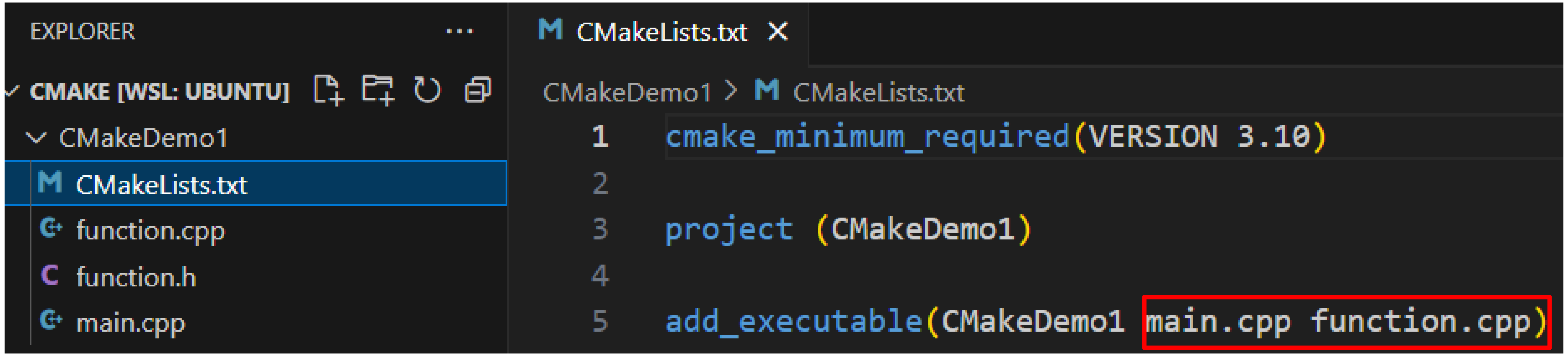


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/build$ make
Scanning dependencies of target Hello
[ 50%] Building CXX object CMakeFiles/Hello.dir/hello.cpp.o
[100%] Linking CXX executable Hello
[100%] Built target Hello
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/build$ ls
CMakeCache.txt  CMakeFiles  Hello  Makefile  cmake_install.cmake
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/build$ ./Hello
Hello World!
```

2. Multi-source files in a project

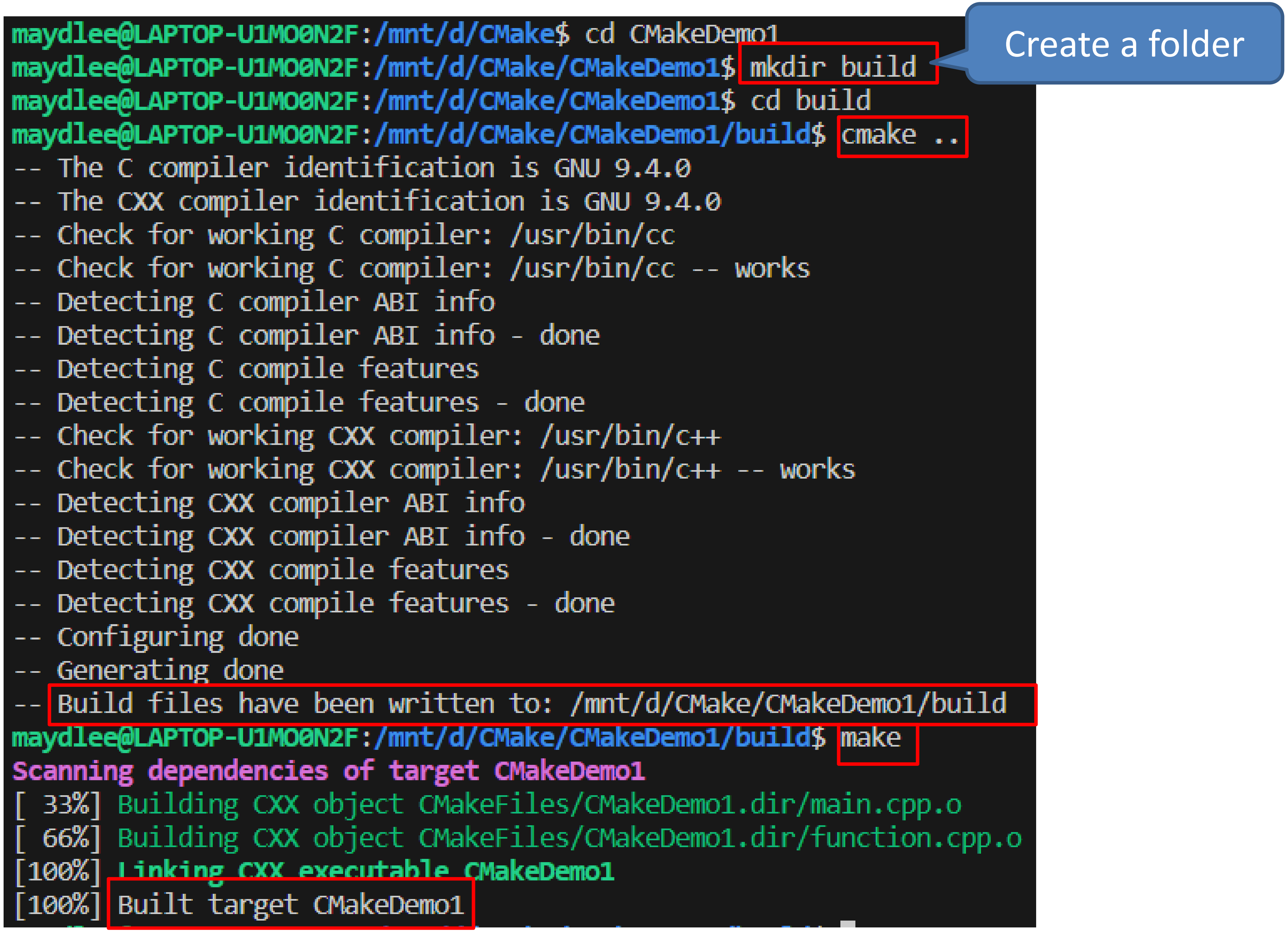
There are three files in the same directory.

```
./CmakeDemo1
      |
      +--- main.cpp
      |
      +--- function.cpp
      |
      +--- function.h
```



```
cmake_minimum_required(VERSION 3.10)

project (CMakeDemo1)

add_executable(CMakeDemo1 main.cpp function.cpp)
```

List all the source files using space as the separator.



Create a folder

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ cd CMakeDemo1
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo1$ mkdir build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo1$ cd build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo1/build$ cmake ..
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/CMake/CMakeDemo1/build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo1/build$ make
Scanning dependencies of target CMakeDemo1
[ 33%] Building CXX object CMakeFiles/CMakeDemo1.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/CMakeDemo1.dir/function.cpp.o
[100%] Linking CXX executable CMakeDemo1
[100%] Built target CMakeDemo1
```

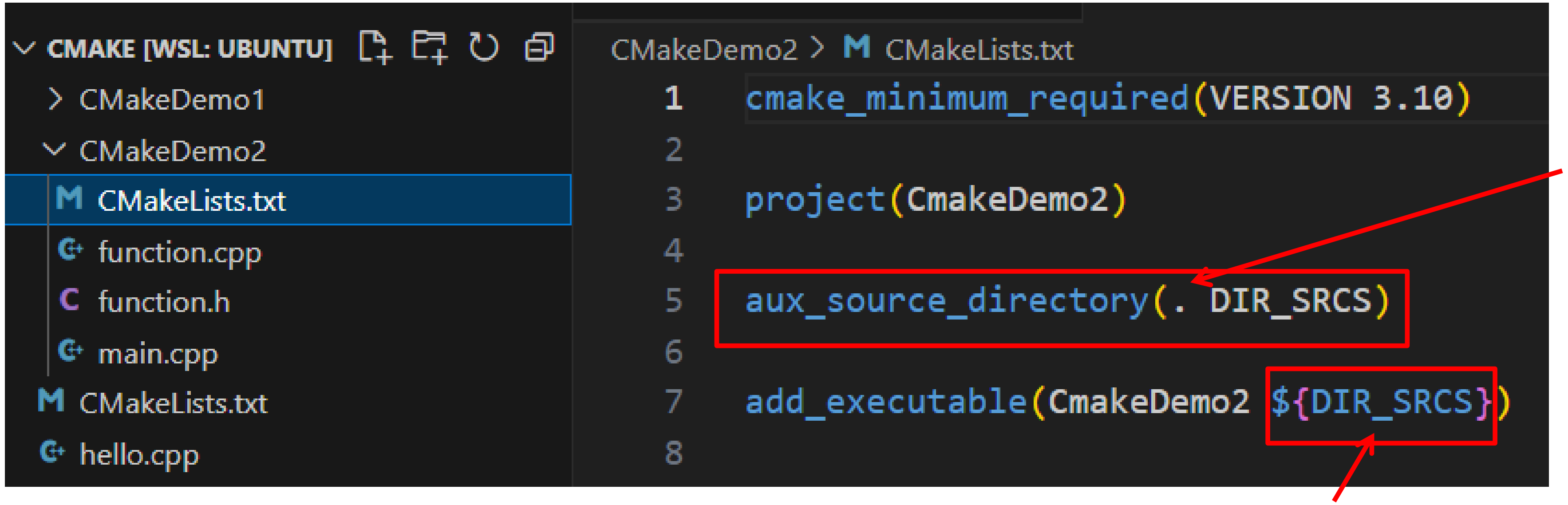
2. Multi-source files in a project

If there are several files in directory, put each file into the **add_executable** command is not recommended. The better way is using **aux_source_directory** command.

aux_source_directory (<dir> <variable>)

The command finds all the source files in the specified directory indicated by <dir> and stores the results in the specified variable indicated by <variable>.

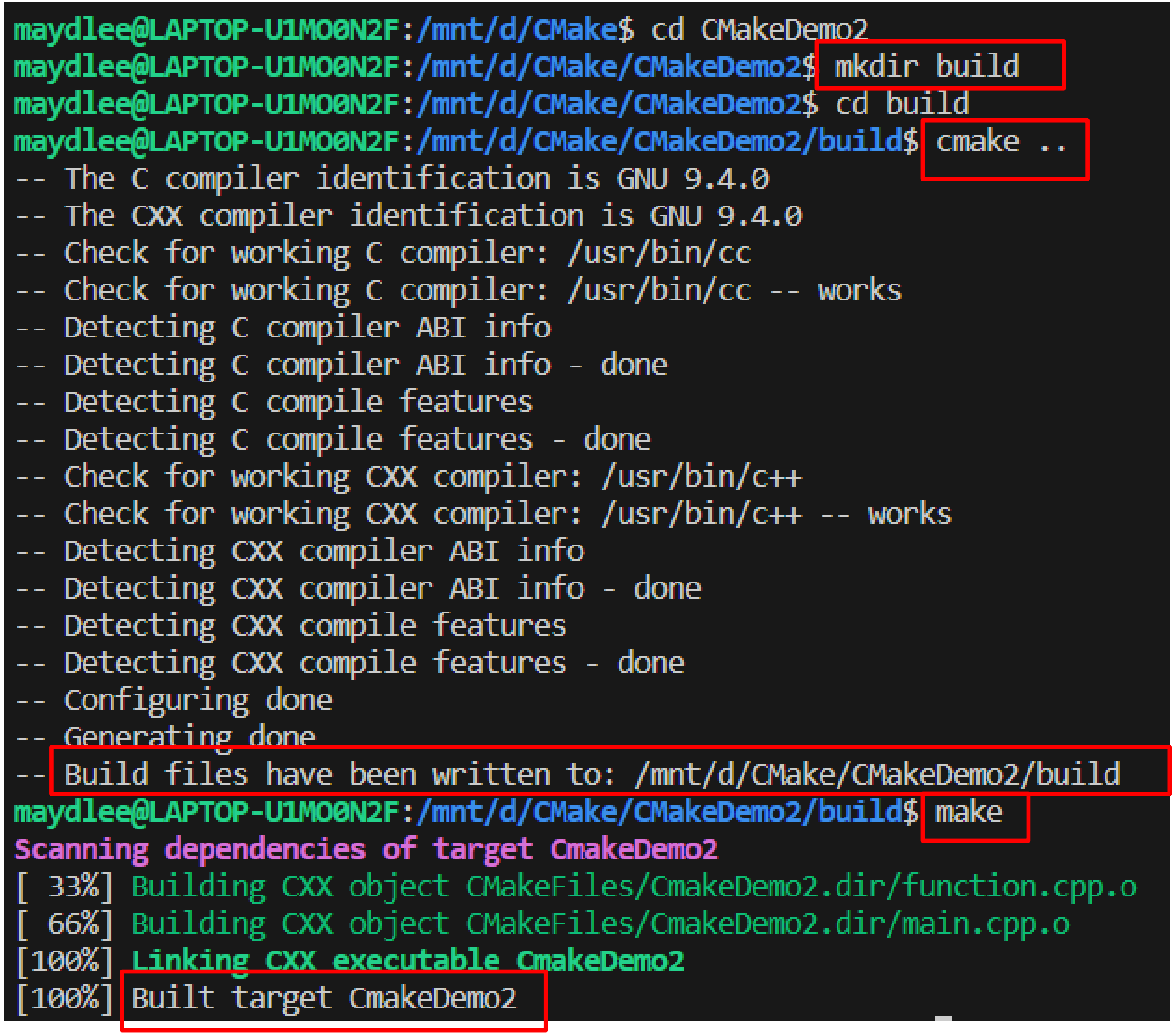
2. Multi-source files in a project



```
cmake_minimum_required(VERSION 3.10)

project(CmakeDemo2)

aux_source_directory(. DIR_SRCS)

add_executable(CmakeDemo2 ${DIR_SRCS})

```

Store all files in the current directory into DIR_SRCS variable.

Compile the source files in the variable by **${ }** into an executable file named CmakeDemo2



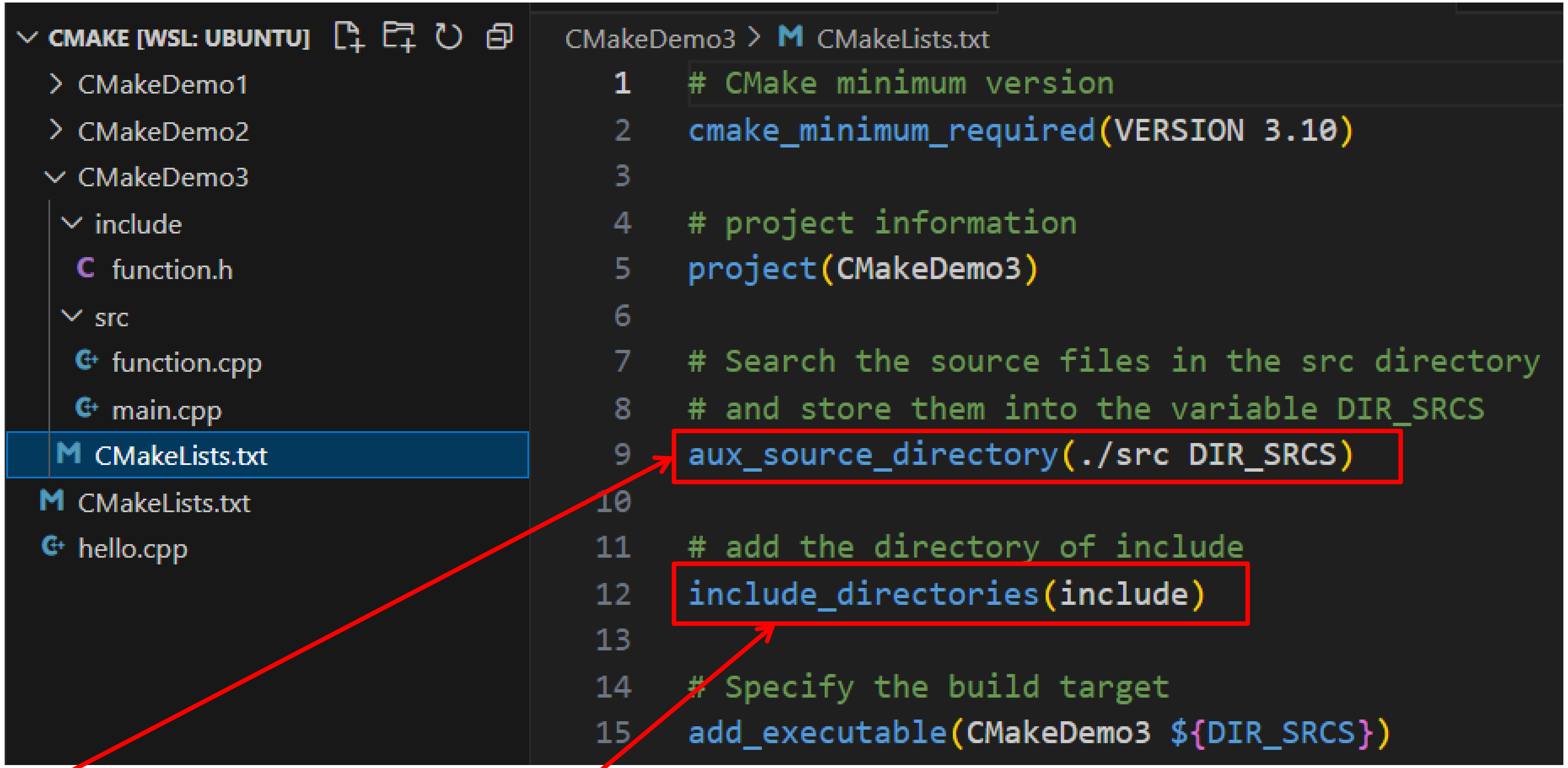
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ cd CMakeDemo2
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo2$ mkdir build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo2$ cd build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo2/build$ cmake ..
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/CMake/CMakeDemo2/build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo2/build$ make
Scanning dependencies of target CmakeDemo2
[ 33%] Building CXX object CMakeFiles/CmakeDemo2.dir/function.cpp.o
[ 66%] Building CXX object CMakeFiles/CmakeDemo2.dir/main.cpp.o
[100%] Linking CXX executable CmakeDemo2
[100%] Built target CmakeDemo2
```

3. Multi-source files in a project in different directories

./CMakeDemo3

```
|
+--- src/
|      |
|      +-- main.cpp
|      +-- function.cpp
|
+--- include/
      |
      +--- function.h
```

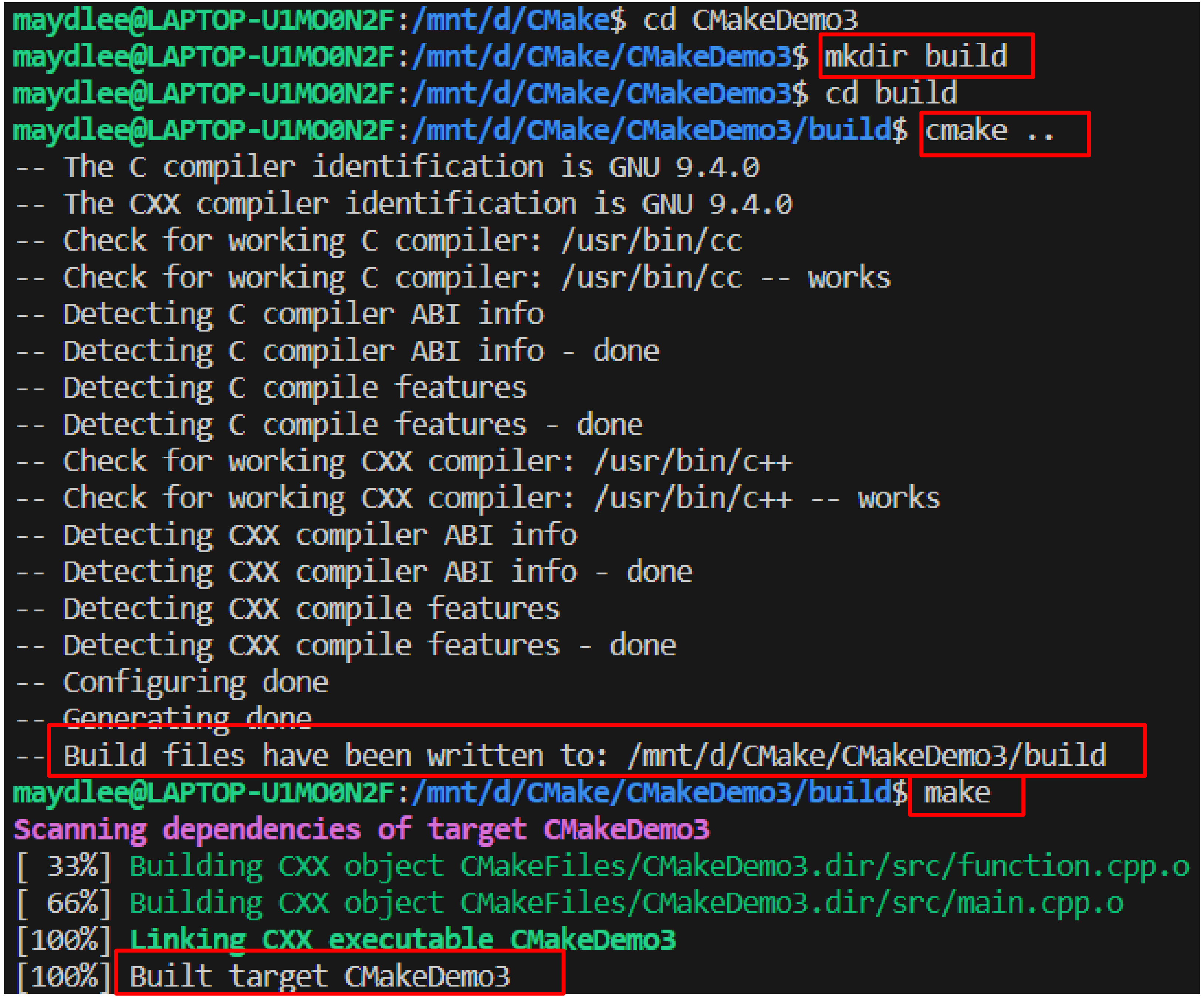
We write CMakeLists.txt in CmakeDemo3 folder.



```
# CMake minimum version
cmake_minimum_required(VERSION 3.10)

# project information
project(CMakeDemo3)

# Search the source files in the src directory
# and store them into the variable DIR_SRCS
aux_source_directory(./src DIR_SRCS)

# add the directory of include
include_directories(include)

# Specify the build target
add_executable(CMakeDemo3 ${DIR_SRCS})
```

All .cpp files are in the **src** directory

Include the header file which is stored in **include** directory.



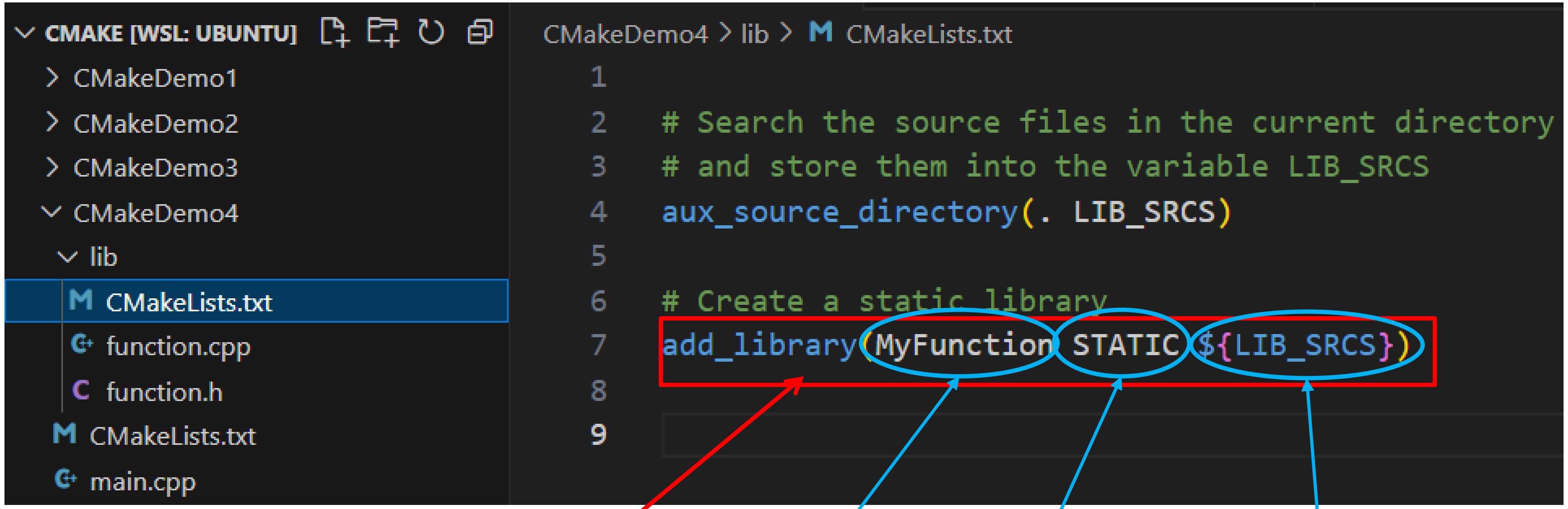
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake$ cd CMakeDemo3
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo3$ mkdir build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo3$ cd build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo3/build$ cmake ..
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/CMake/CMakeDemo3/build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo3/build$ make
Scanning dependencies of target CMakeDemo3
[ 33%] Building CXX object CMakeFiles/CMakeDemo3.dir/src/function.cpp.o
[ 66%] Building CXX object CMakeFiles/CMakeDemo3.dir/src/main.cpp.o
[100%] Linking CXX executable CMakeDemo3
[100%] Built target CMakeDemo3
```

4. Static library Dynamic library

We want to create a static(or dynamic) library by function.cpp and call the static(or dynamic) library in main.cpp. This time we write two CMakeLists.txt files, one in CmakeDemo4 folder and another in lib folder.

```
./CMakeDemo4
      |
      +--- main.cpp
      |
      +--- lib/
            |
            +--- function.h
            |
            +--- function.cpp
```

The CMakeLists.txt in lib folder creates a static library.



```

# Search the source files in the current directory
# and store them into the variable LIB_SRCS
aux_source_directory(. LIB_SRCS)

# Create a static library
add_library(MyFunction STATIC ${LIB_SRCS})

```

library file name

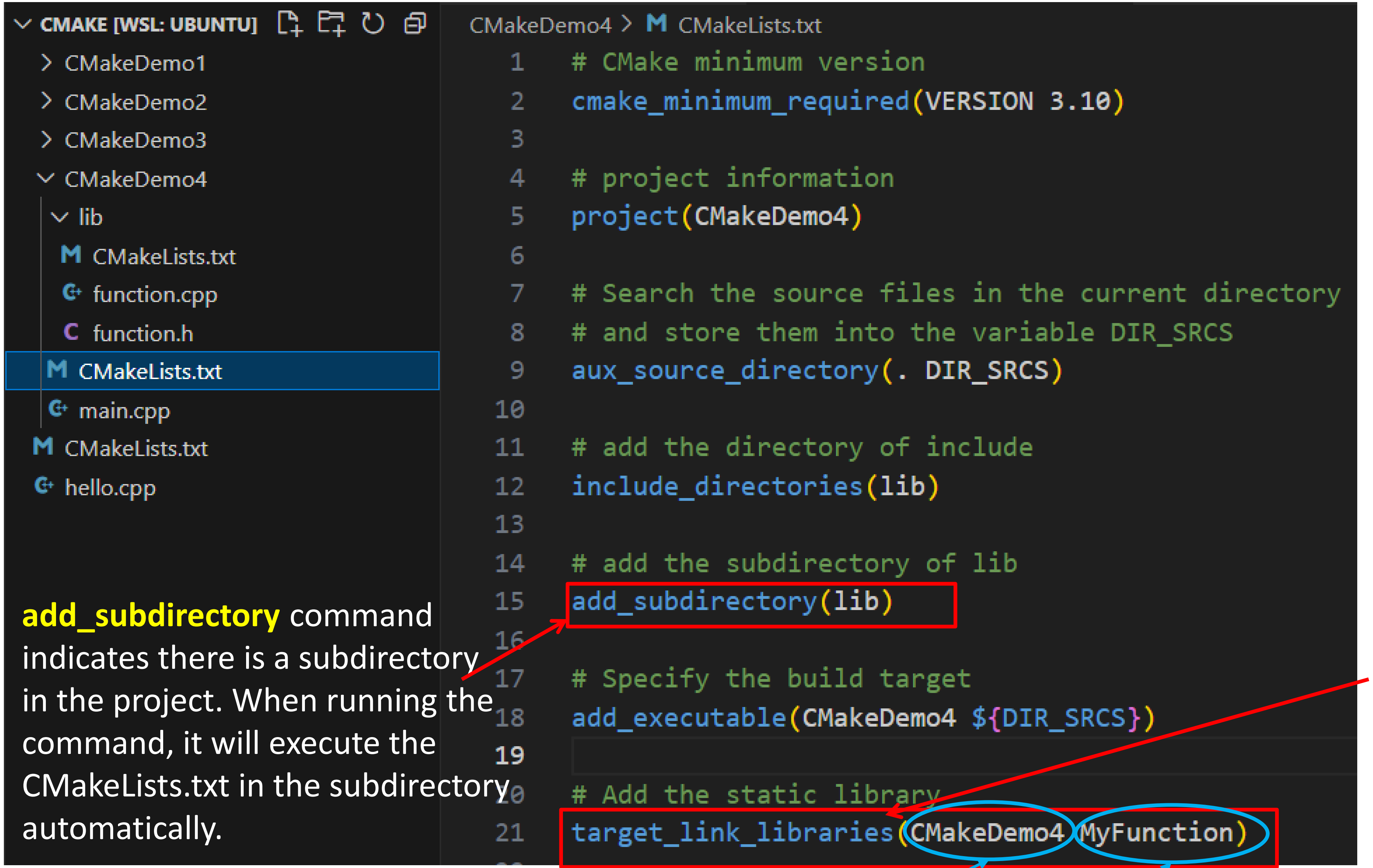
static library

The directory from which the library file originates.

Create a static library named libMyFunction.a by the files in the current directory.

Note: If we use **SHARED** instead of STATIC in **add_library** command, it will create a shared(dynamic) library file.

The CMakeLists.txt in CMakeDemo4 folder creates the project.



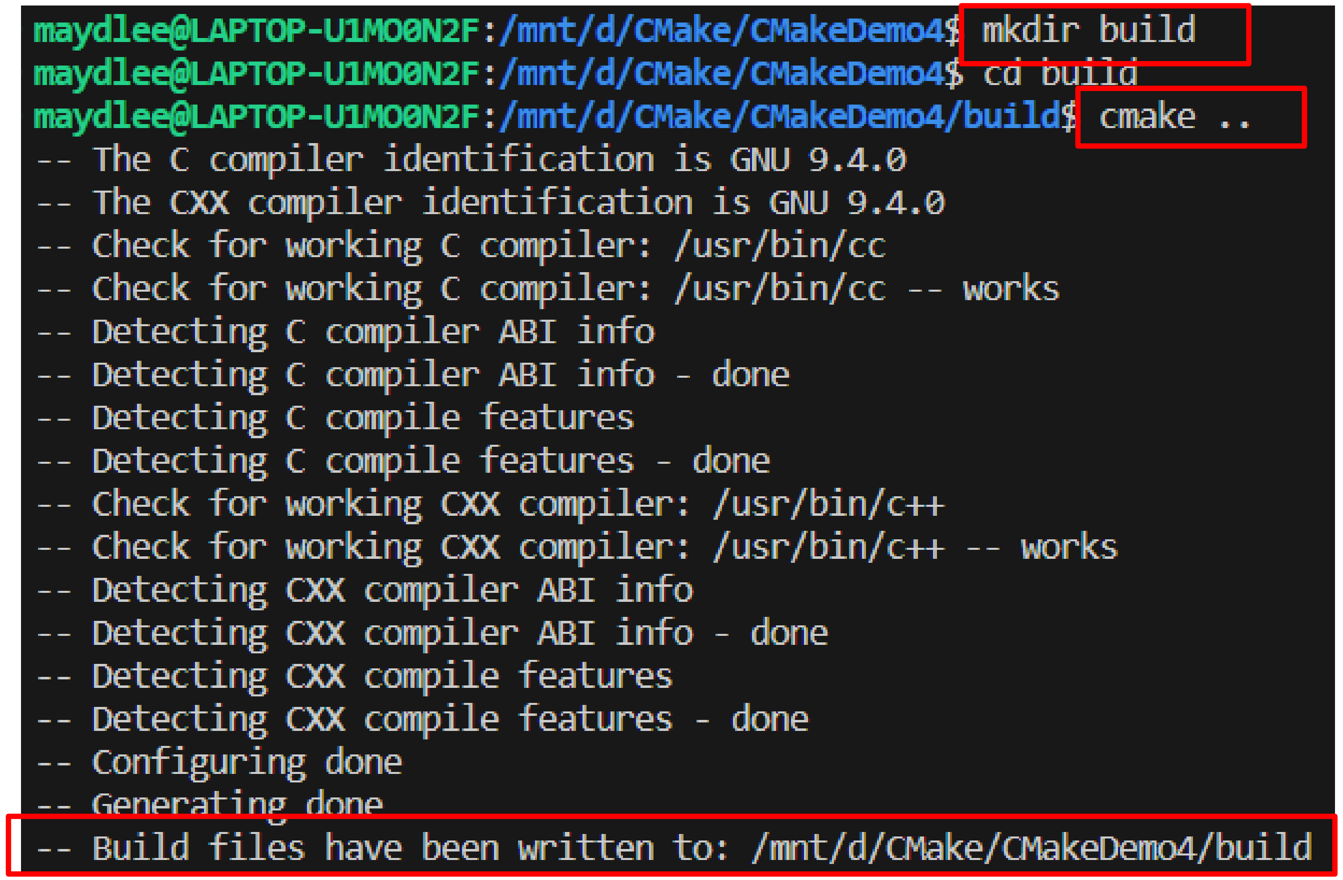
```
# CMake minimum version
cmake_minimum_required(VERSION 3.10)

# project information
project(CMakeDemo4)

# Search the source files in the current directory
# and store them into the variable DIR_SRCS
aux_source_directory(. DIR_SRCS)

# add the directory of include
include_directories(lib)

# add the subdirectory of lib
add_subdirectory(lib)

# Specify the build target
add_executable(CMakeDemo4 ${DIR_SRCS})

# Add the static library
target_link_libraries(CMakeDemo4 MyFunction)
```

add_subdirectory command indicates there is a subdirectory in the project. When running the command, it will execute the CMakeLists.txt in the subdirectory automatically.

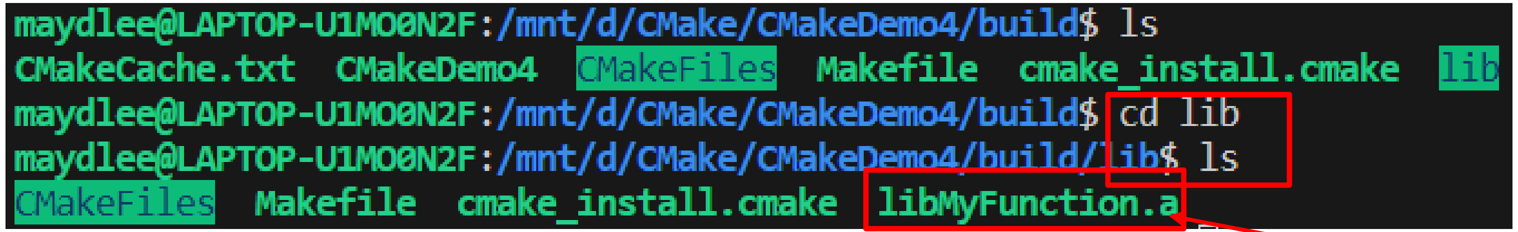
Indicates that the project needs link a library named **MyFunction**, MyFunction can be a static library file or a dynamic library file.

project name

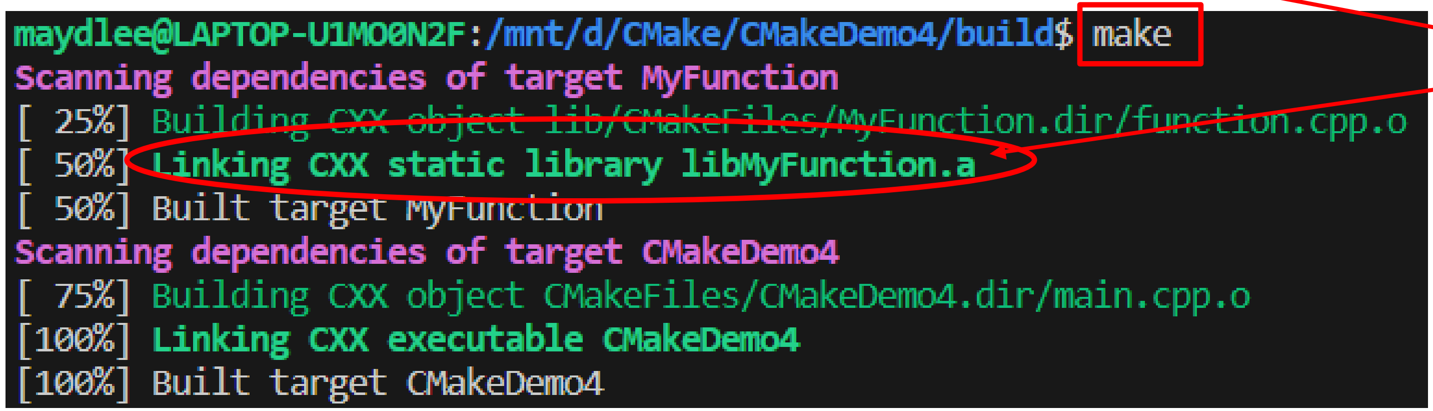
library file name
If there are more than one file, list them using space as the separator.



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo4$ mkdir build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo4$ cd build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo4/build$ cmake ..
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/CMake/CMakeDemo4/build
```



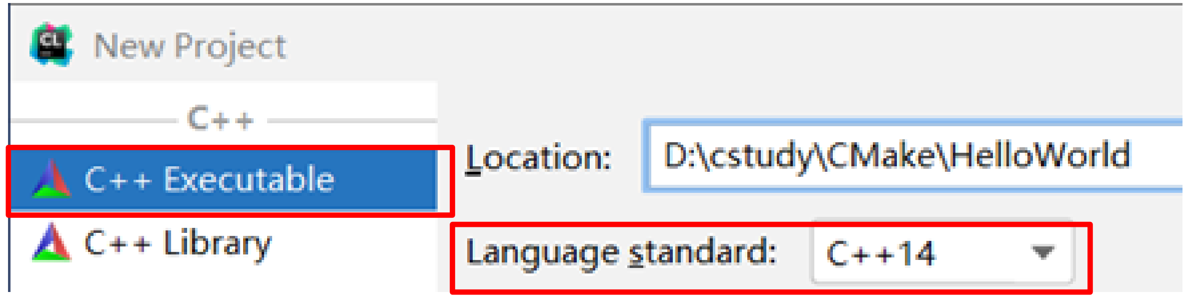
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo4/build$ ls
CMakeCache.txt  CMakeDemo4  CMakeFiles  Makefile  cmake_install.cmake  lib
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo4/build$ cd lib
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo4/build/lib$ ls
CMakeFiles  Makefile  cmake_install.cmake  libMyFunction.a
```



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo4/build$ make
Scanning dependencies of target MyFunction
[ 25%] Building CXX object lib/CMakeFiles/MyFunction.dir/function.cpp.o
[ 50%] Linking CXX static library libMyFunction.a
[ 50%] Built target MyFunction
Scanning dependencies of target CMakeDemo4
[ 75%] Building CXX object CMakeFiles/CMakeDemo4.dir/main.cpp.o
[100%] Linking CXX executable CMakeDemo4
[100%] Built target CMakeDemo4
```

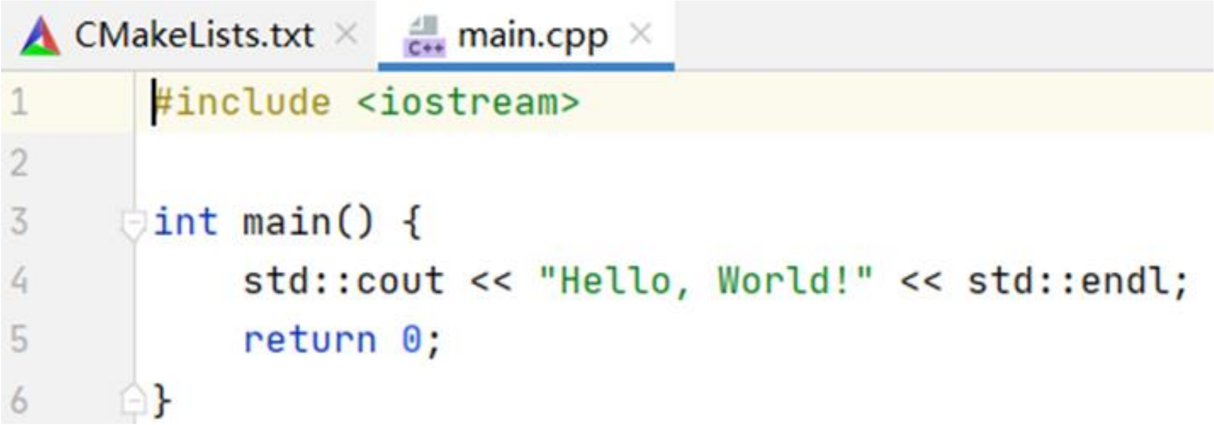
If we use **SHARED** in add_library command, there will creates a dynamic library named **libMyFunction.so** and link with it in main.

Create a C++ project by CLion, the CMakeLists.txt is created automatically.

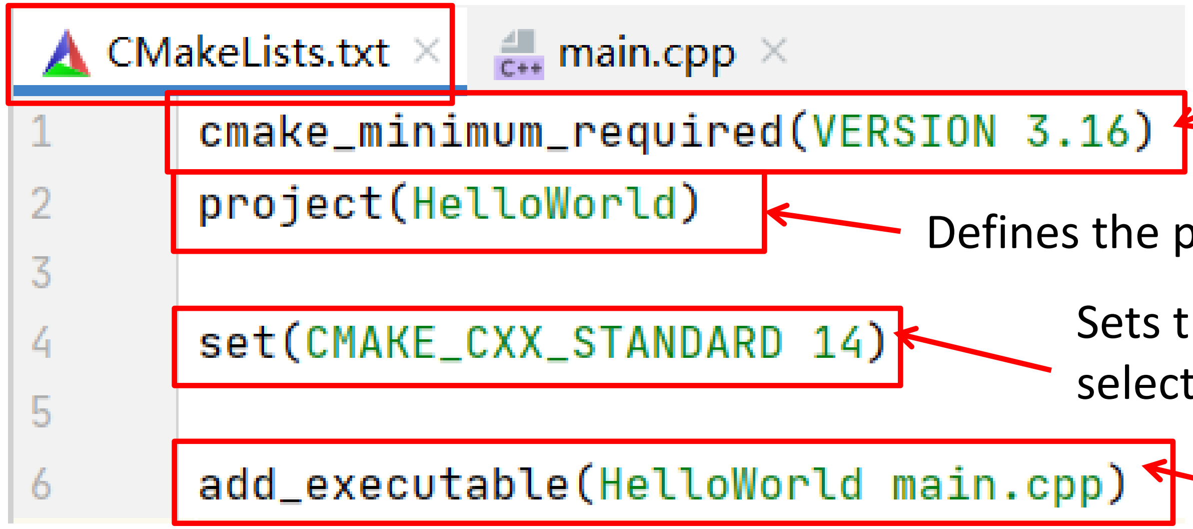


Location: D:\cstudy\CMake\HelloWorld

Language standard: C++14



```cpp
#include <iostream>

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



```cmake
cmake_minimum_required(VERSION 3.16)
project(HelloWorld)

set(CMAKE_CXX_STANDARD 14)

add_executable(HelloWorld main.cpp)
```

Specifies the minimum required version of Cmake. It is set to the version of Cmake bundled in Clion (always one of the newest versions available).

Defines the project name according to what we provided during project creation.

Sets the CMAKE_CXX_STANDARD variable to the value of 14, as we selected when creating the project.

Adds the HelloWorld executable target which will be built from main.cpp.

For more about Cmake(cmake tutorial):
https://cmake.org/cmake/help/latest/guide/tutorial/index.html
https://www.jetbrains.com/help/clion/2016.3/quick-cmake-tutorial.html

2.2 Storage duration, Scope and Linkage

- Scope describes the region or regions of a program that can access an identifier. An identifier has one of following scopes: **block scope**, **function prototype scope**, or **file scope**.

- Linkage describes how a name can be shared in different units. A variable has one of the following linkages: **external linkage**, **internal linkage**, or **no linkage**. A name with external linkage can be shared across files, and a name with internal linkage can be shared by functions within a single file. Names of automatic variables have no linkage because they are not shared.

- Storage duration describes the lifetime of a variable. A variable has one of the following storage durations: automatic storage duration, static storage duration, dynamic storage duration or thread storage duration.

C++ uses three separate schemes(four under C++11) for storing data. The different storage classes offer different combinations of scope, linkage and storage duration.

- Automatic storage duration: **Variables declared inside a function definition**(including function parameters)have automatic storage duration. They are created when program execution enters the function or block in which they are defined, and the memory used for them is freed when execution leaves the function or block.

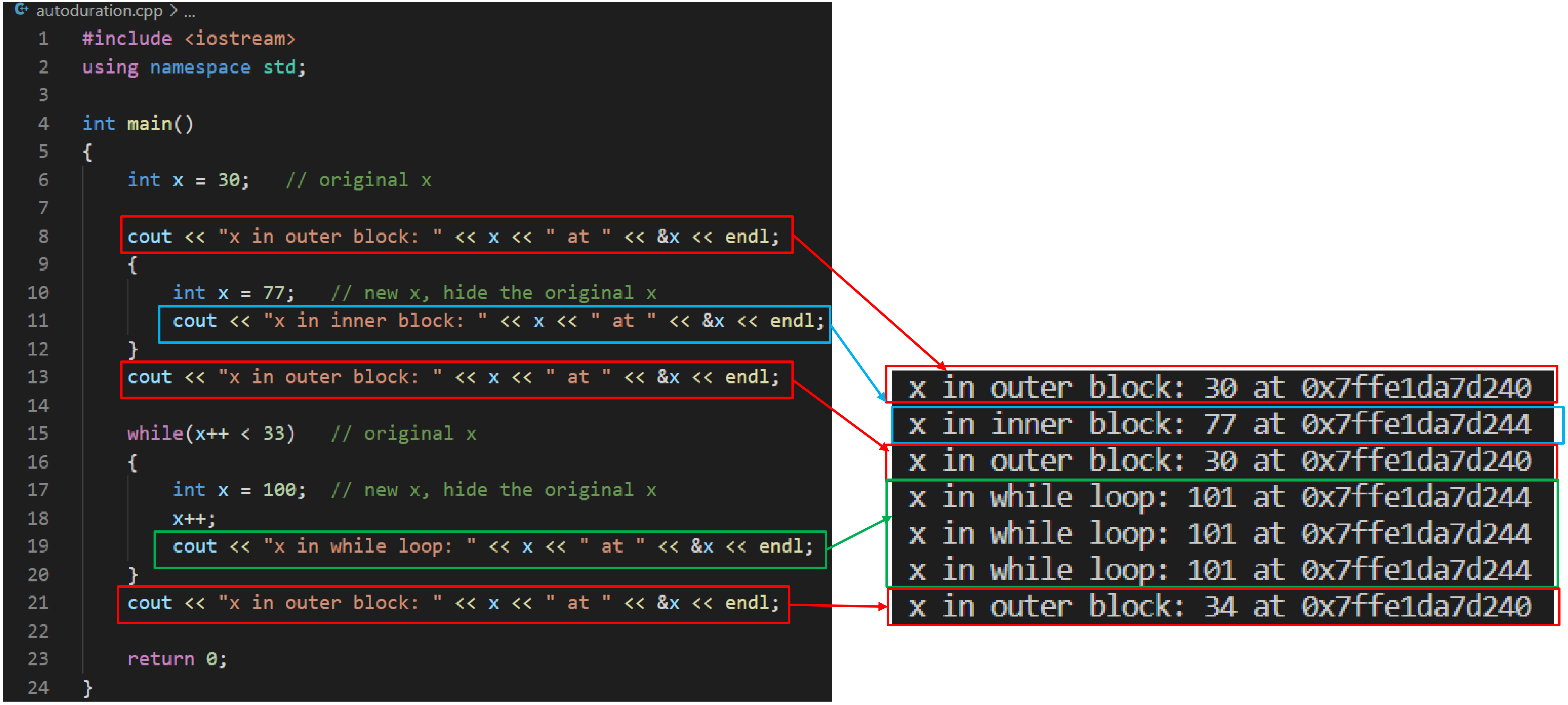
- Static storage duration: **Variables defined outside a function definition** or else by using the **keyword static** have static storage duration. They persist for the entire time a program is running.

- Dynamic storage duration: Memory allocated by the **new operator** persists until it is freed with the **delete operator** or until the program ends, whichever comes first. This memory has dynamic storage duration and sometimes is termed the free store or the heap.

- Thread storage duration(C++11): Variables declared with the **thread_local** keyword have storage that persists for as long as the containing thread lasts.

2.2.1 Automatic Storage Duration

Function parameters and variables declared inside a function have, by default, automatic storage duration. **They also have local scope and no linkage**.



autoduration.cpp > ...

```cpp
#include <iostream>
using namespace std;

int main()
{
    int x = 30;    // original x

    cout << "x in outer block: " << x << " at " << &x << endl;
    {
        int x = 77;    // new x, hide the original x
        cout << "x in inner block: " << x << " at " << &x << endl;
    }
    cout << "x in outer block: " << x << " at " << &x << endl;

    while(x++ < 33)    // original x
    {
        int x = 100;  // new x, hide the original x
        x++;
        cout << "x in while loop: " << x << " at " << &x << endl;
    }
    cout << "x in outer block: " << x << " at " << &x << endl;

    return 0;
}
```

```
x in outer block: 30 at 0x7ffe1da7d240
x in inner block: 77 at 0x7ffe1da7d244
x in outer block: 30 at 0x7ffe1da7d240
x in while loop: 101 at 0x7ffe1da7d244
x in while loop: 101 at 0x7ffe1da7d244
x in while loop: 101 at 0x7ffe1da7d244
x in outer block: 34 at 0x7ffe1da7d240
```

2.2.2 Static Storage Duration

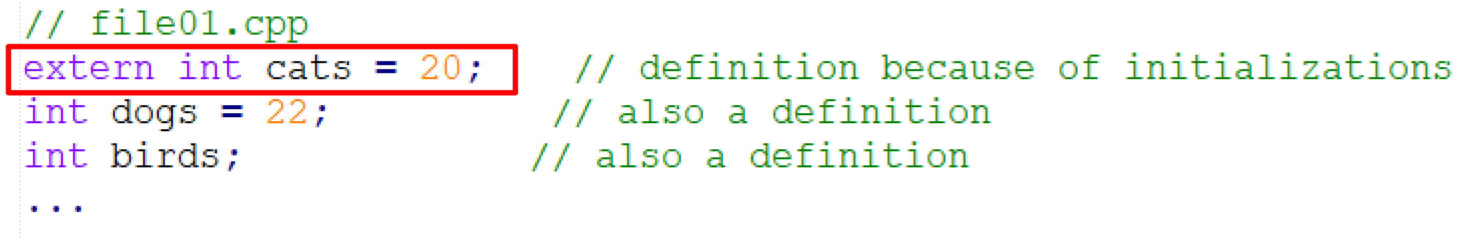
C++, like C, provides **static storage duration variables** with three kinds of linkage: **external linkage** (accessible across files), **internal linkage** (accessible to functions within a single file),and **no linkage** (accessible to just one function or to one block within a function).

All three last for the duration of the program. The static variables stay present as long as the program executes.

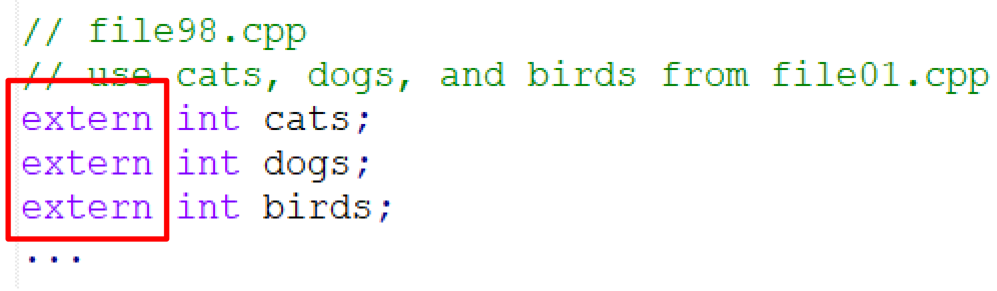
1.Static Duration, External Linkage

Variables with **external linkage** are often simply called **external variables(global variables)**. They necessarily have static storage duration and file scope. External variables are defined outside, and hence external to any function.

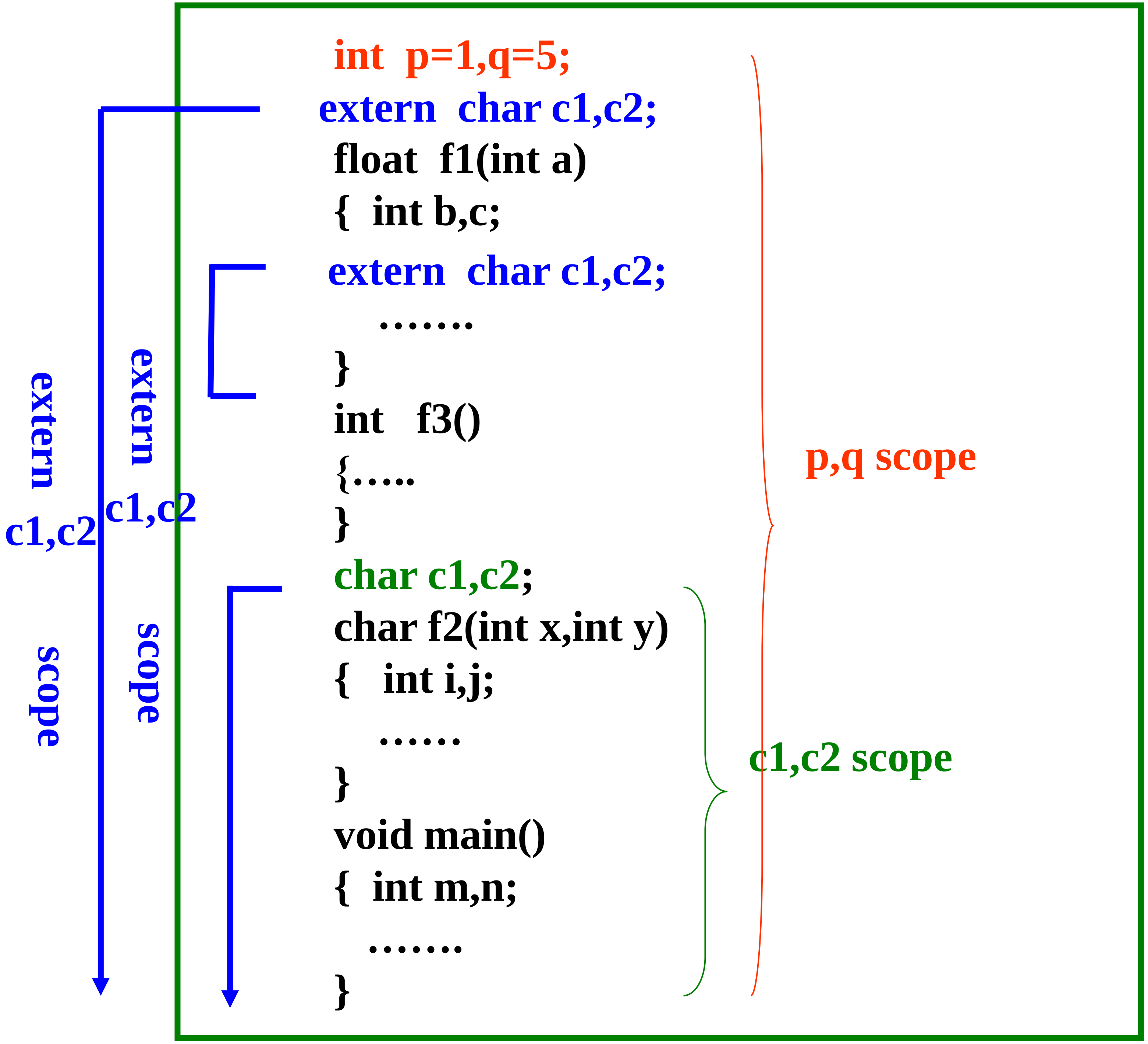
If you use an external variable in several files, only one file can contain a definition for that variable (per the one definition rule). But every other file using the variable needs to declare that variable using the keyword **extern**.



```
// file01.cpp
extern int cats = 20;    // definition because of initializations
int dogs = 22;          // also a definition
int birds;             // also a definition
...
```



```
// file98.cpp
// use cats, dogs, and birds from file01.cpp
extern int cats;
extern int dogs;
extern int birds;
...
```



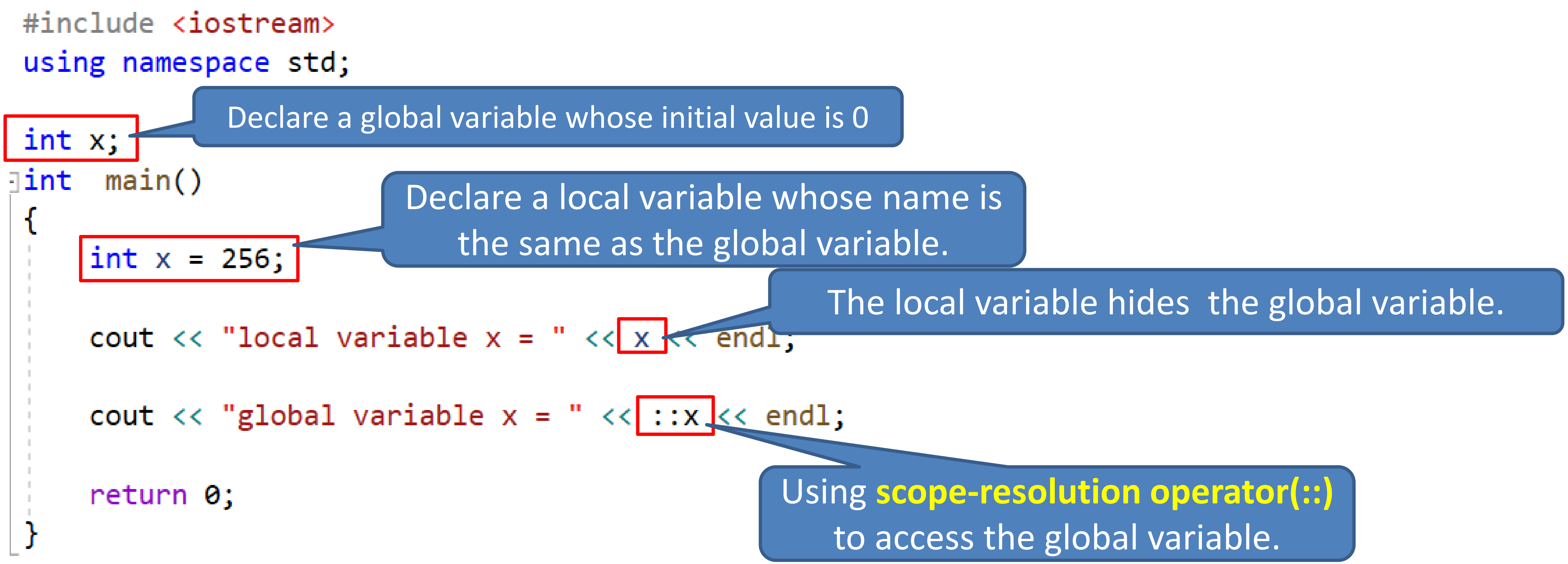
```
int  p=1,q=5;
extern  char c1,c2;
float  f1(int a)
{  int b,c;
extern  char c1,c2;
    …….
}
int   f3()
{…..
}
char c1,c2;
char f2(int x,int y)
{   int i,j;
    ……
}
void main()
{  int m,n;
   …….
}
```

extern c1,c2 scope

extern c1,c2 scope

p,q scope

c1,c2 scope



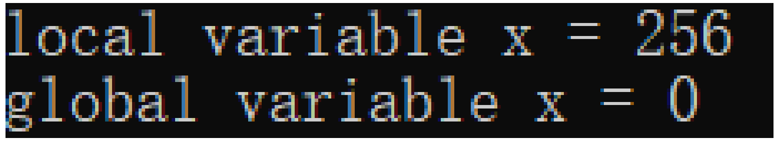
```cpp
#include <iostream>
using namespace std;

int x;
int  main()
{
    int x = 256;

    cout << "local variable x = " << x << endl;

    cout << "global variable x = " << ::x << endl;

    return 0;
}
```

Declare a global variable whose initial value is 0

Declare a local variable whose name is the same as the global variable.

The local variable hides the global variable.

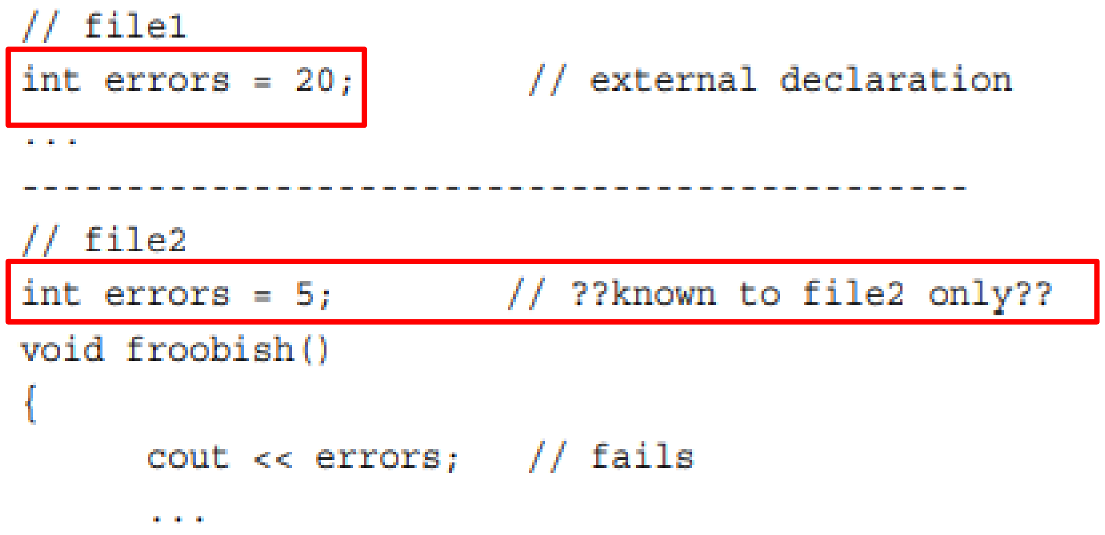
Using **scope-resolution operator(::)** to access the global variable.



```
local variable x = 256
global variable x = 0
```

2. Static Duration, Internal Linkage

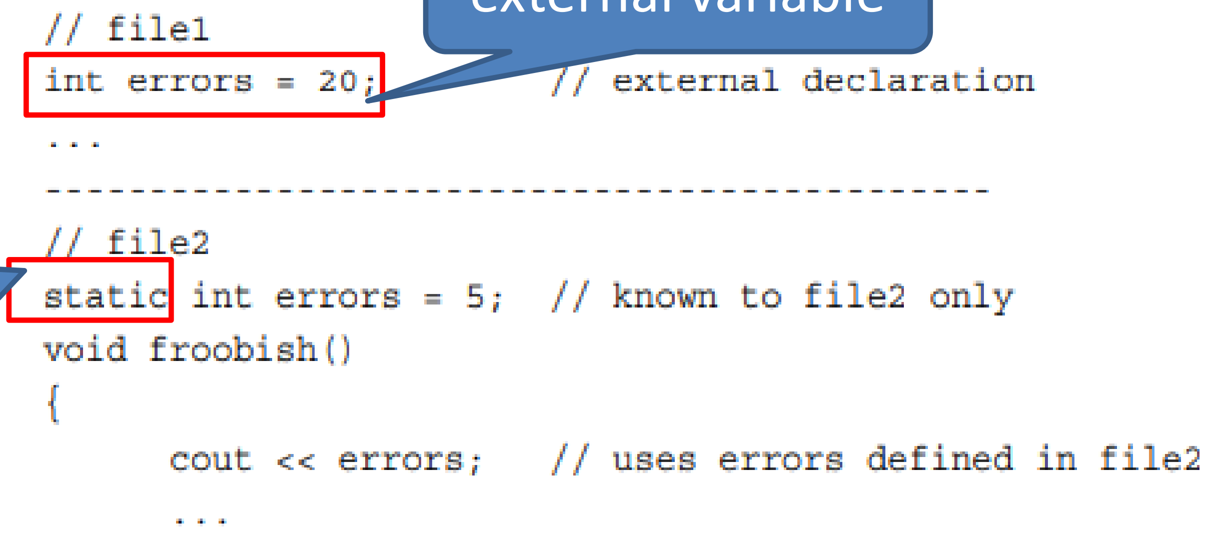
Variables of this storage class have static storage duration, file scope, and internal linkage. You can create one by defining it outside of any function (just as with an external variables) with the storage class specifier **static**. A variable with internal linkage is local to the file that contains it.



```
// file1
int errors = 20;          // external declaration
...
---------------------------------------------
// file2
int errors = 5;          // ??known to file2 only??
void froobish()
{
      cout << errors;    // fails
      ...
```

Using **static** to share data among functions found in just one file, avoiding name conflicting with external variable.

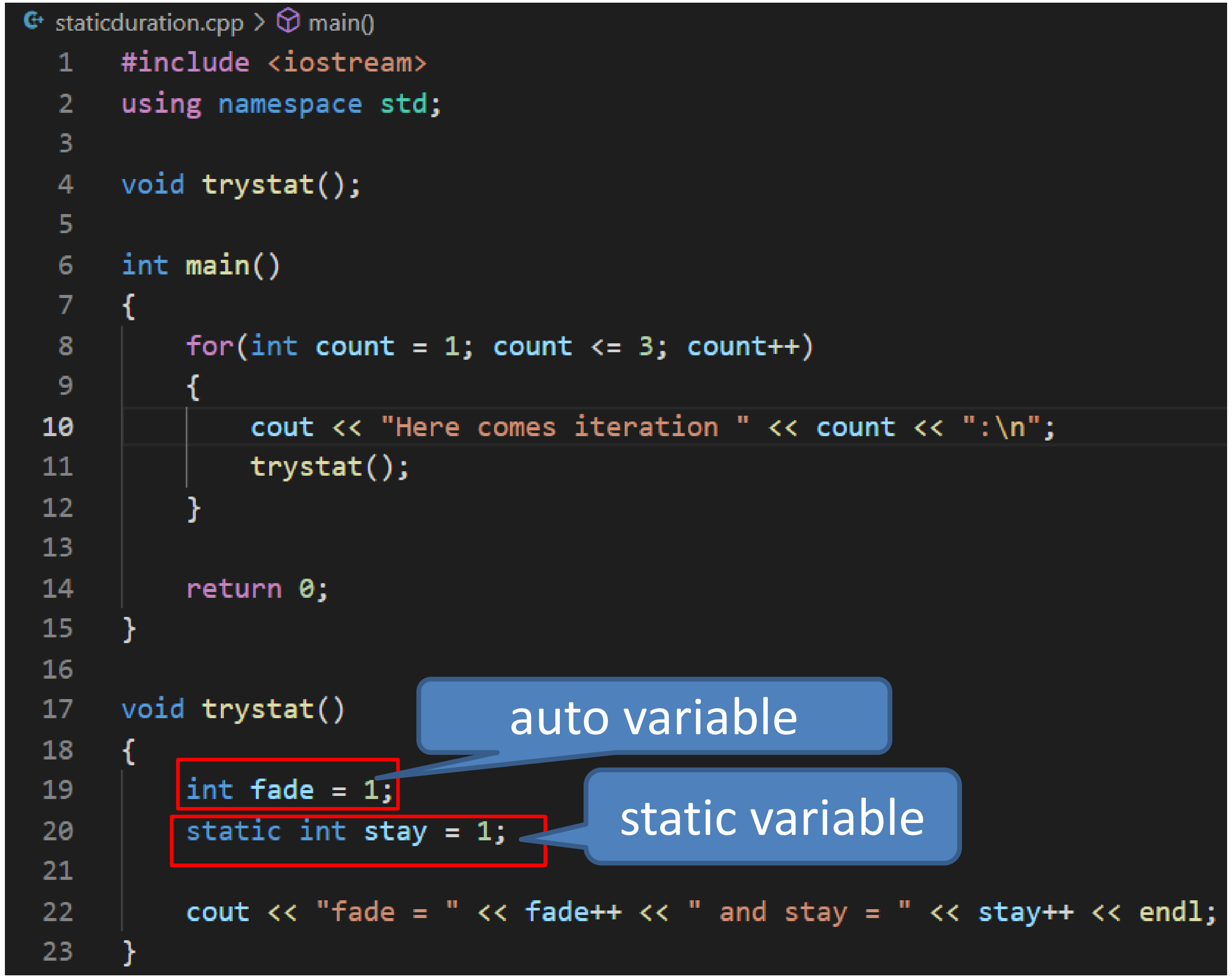
external variable



```
// file1
int errors = 20;            // external declaration
...
---------------------------------------------
// file2
static int errors = 5;   // known to file2 only
void froobish()
{
      cout << errors;    // uses errors defined in file2
      ...
```

3. Static Duration, No Linkage

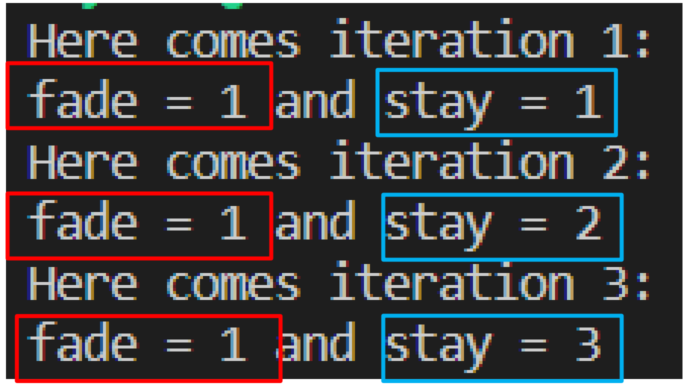
You create such a variable by applying the **static** modifier to a variable defined **inside a block**. When you use it inside a block, static causes a local variable to have static storage duration. If you initialize a static local variable, the program **initializes the variable once**.



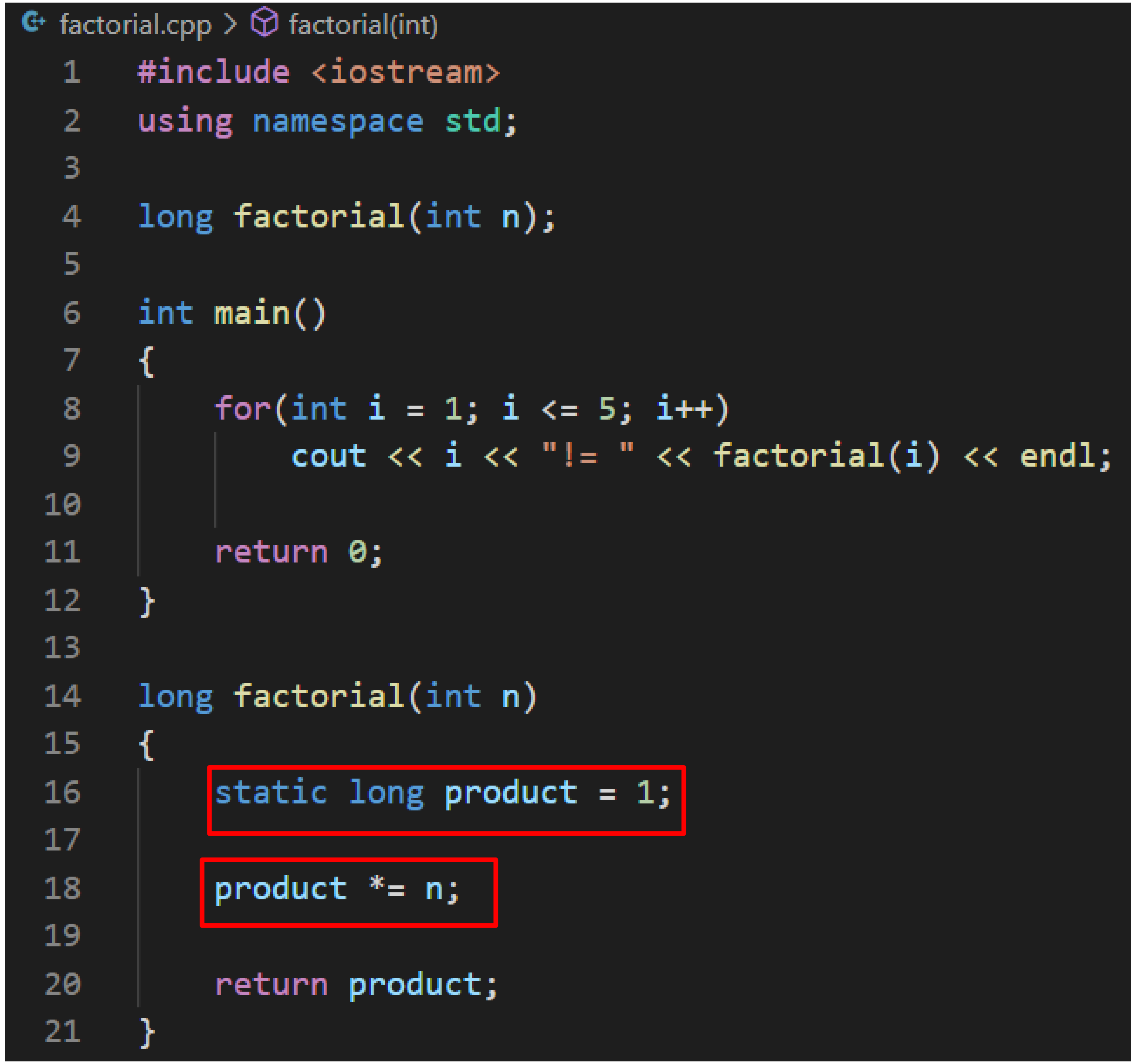
```cpp
#include <iostream>
using namespace std;

void trystat();

int main()
{
    for(int count = 1; count <= 3; count++)
    {
        cout << "Here comes iteration " << count << ":\n";
        trystat();
    }

    return 0;
}

void trystat()
{
    int fade = 1;
    static int stay = 1;

    cout << "fade = " << fade++ << " and stay = " << stay++ << endl;
}
```

auto variable

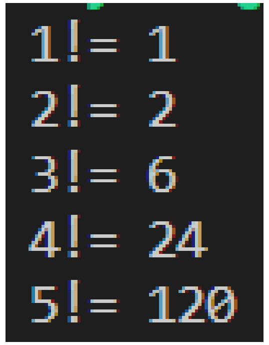
static variable



```
Here comes iteration 1:
fade = 1 and stay = 1
Here comes iteration 2:
fade = 1 and stay = 2
Here comes iteration 3:
fade = 1 and stay = 3
```



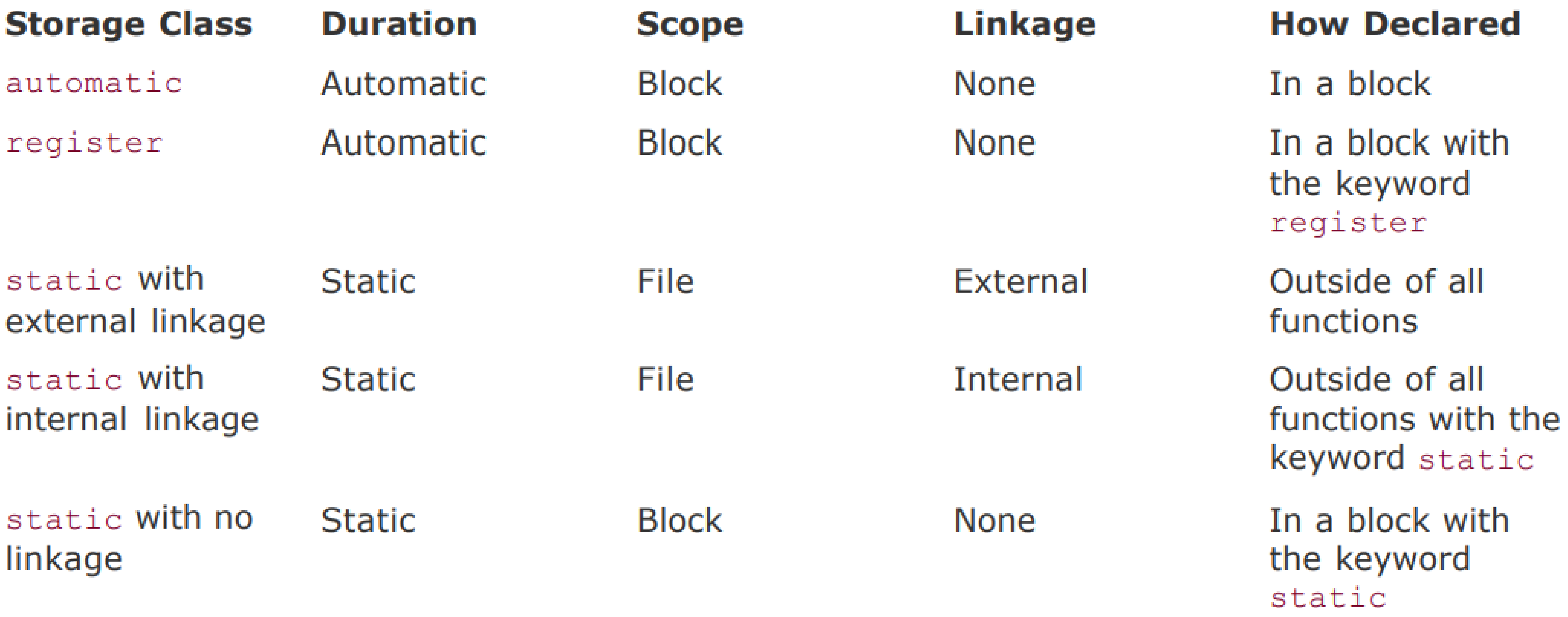
```cpp
#include <iostream>
using namespace std;

long factorial(int n);

int main()
{
    for(int i = 1; i <= 5; i++)
        cout << i << "!= " << factorial(i) << endl;

    return 0;
}

long factorial(int n)
{
    static long product = 1;

    product *= n;

    return product;
}
```



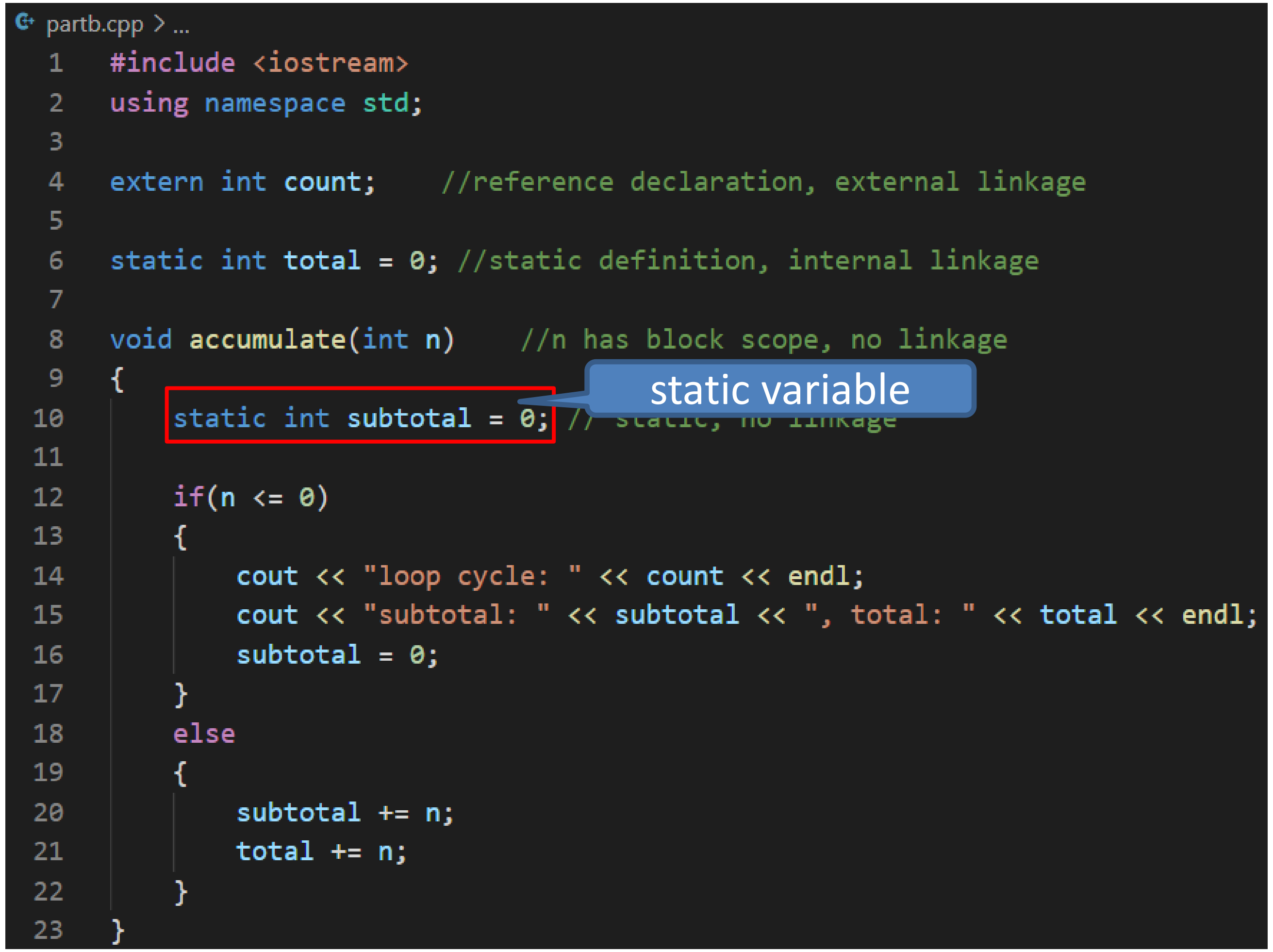
```
1!= 1
2!= 2
3!= 6
4!= 24
5!= 120
```

C and C++ use scope, linkage, and storage duration to define five storage classes: automatic, register, static with block scope, static with external linkage, and static with internal linkage.

Five Storage Classes



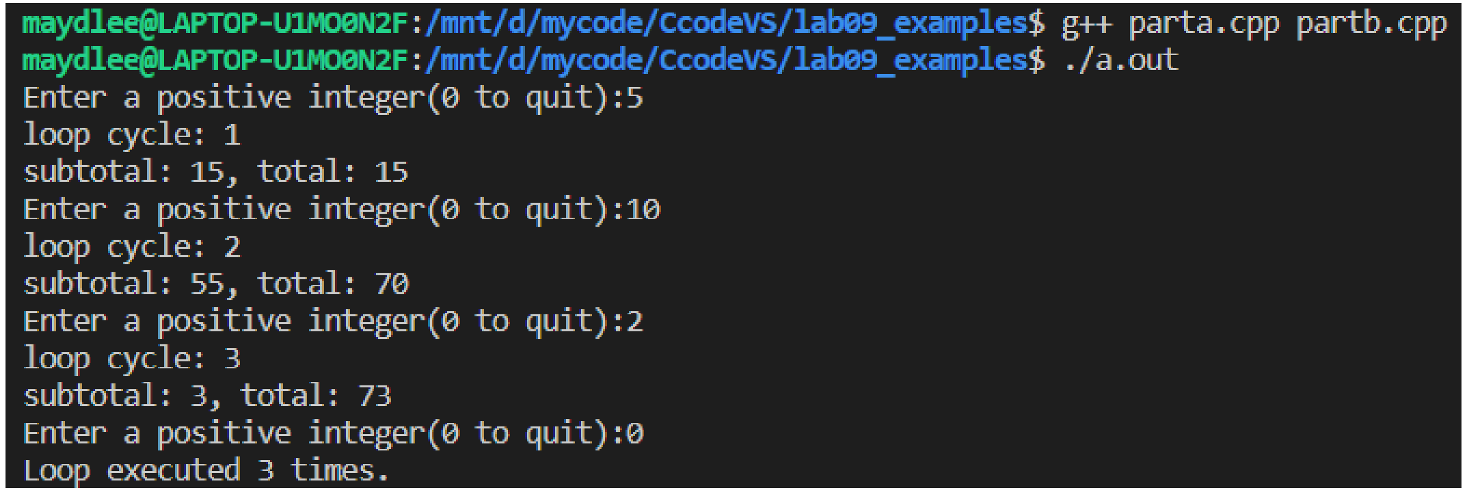
Storage Class	Duration	Scope	Linkage	How Declared
`automatic`	Automatic	Block	None	In a block
`register`	Automatic	Block	None	In a block with the keyword `register`
`static` with external linkage	Static	File	External	Outside of all functions
`static` with internal linkage	Static	File	Internal	Outside of all functions with the keyword `static`
`static` with no linkage	Static	Block	None	In a block with the keyword `static`



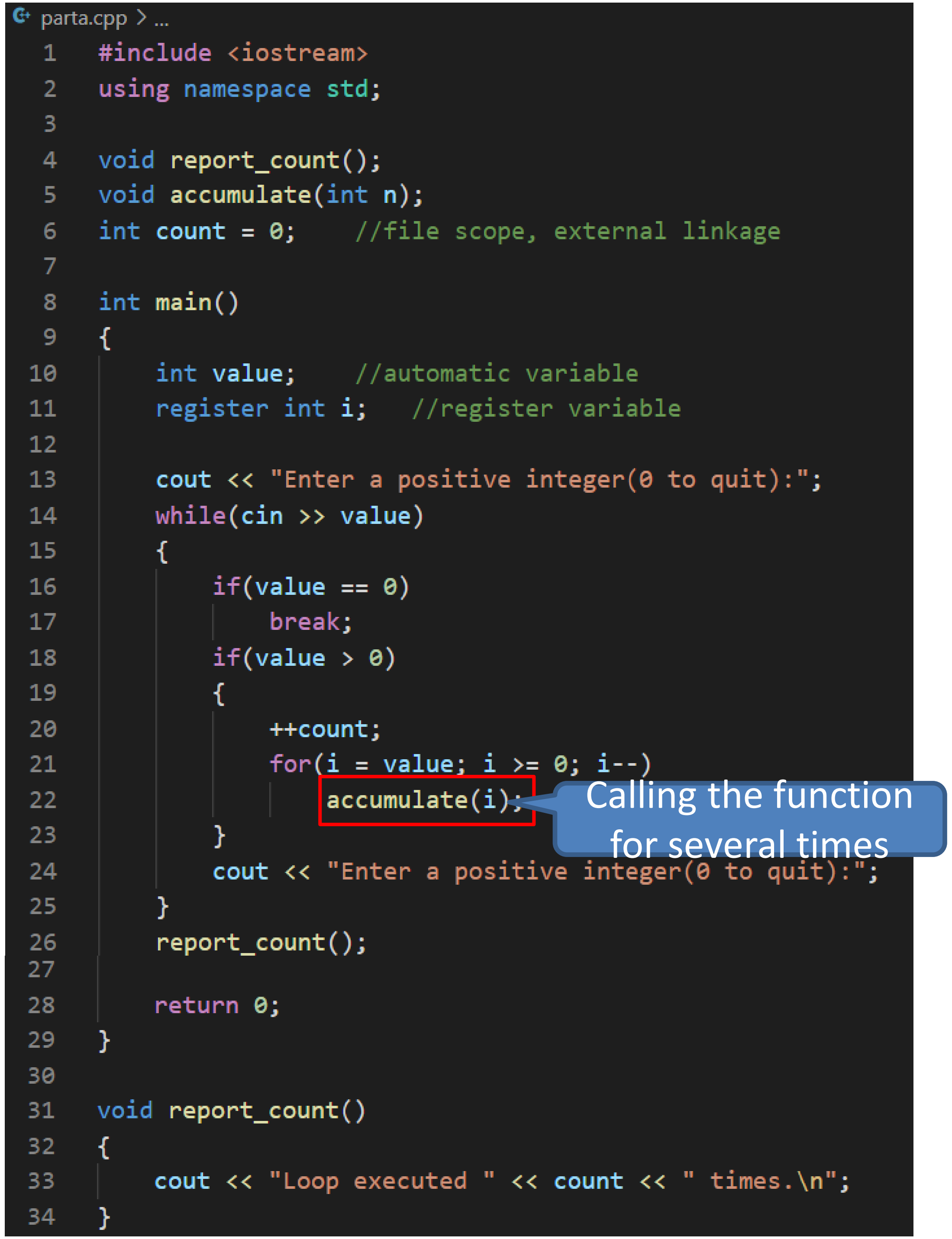
partb.cpp > ...

```cpp
#include <iostream>
using namespace std;

extern int count;    //reference declaration, external linkage

static int total = 0; //static definition, internal linkage

void accumulate(int n)    //n has block scope, no linkage
{
    static int subtotal = 0; // static, no linkage

    if(n <= 0)
    {
        cout << "loop cycle: " << count << endl;
        cout << "subtotal: " << subtotal << ", total: " << total << endl;
        subtotal = 0;
    }
    else
    {
        subtotal += n;
        total += n;
    }
}
```

static variable



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples$ g++ parta.cpp partb.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples$ ./a.out
Enter a positive integer(0 to quit):5
loop cycle: 1
subtotal: 15, total: 15
Enter a positive integer(0 to quit):10
loop cycle: 2
subtotal: 55, total: 70
Enter a positive integer(0 to quit):2
loop cycle: 3
subtotal: 3, total: 73
Enter a positive integer(0 to quit):0
Loop executed 3 times.
```



parta.cpp > ...

```cpp
#include <iostream>
using namespace std;

void report_count();
void accumulate(int n);
int count = 0;    //file scope, external linkage

int main()
{
    int value;    //automatic variable
    register int i;   //register variable

    cout << "Enter a positive integer(0 to quit):";
    while(cin >> value)
    {
        if(value == 0)
            break;
        if(value > 0)
        {
            ++count;
            for(i = value; i >= 0; i--)
                accumulate(i);
        }
        cout << "Enter a positive integer(0 to quit):";
    }
    report_count();

    return 0;
}

void report_count()
{
    cout << "Loop executed " << count << " times.\n";
}
```

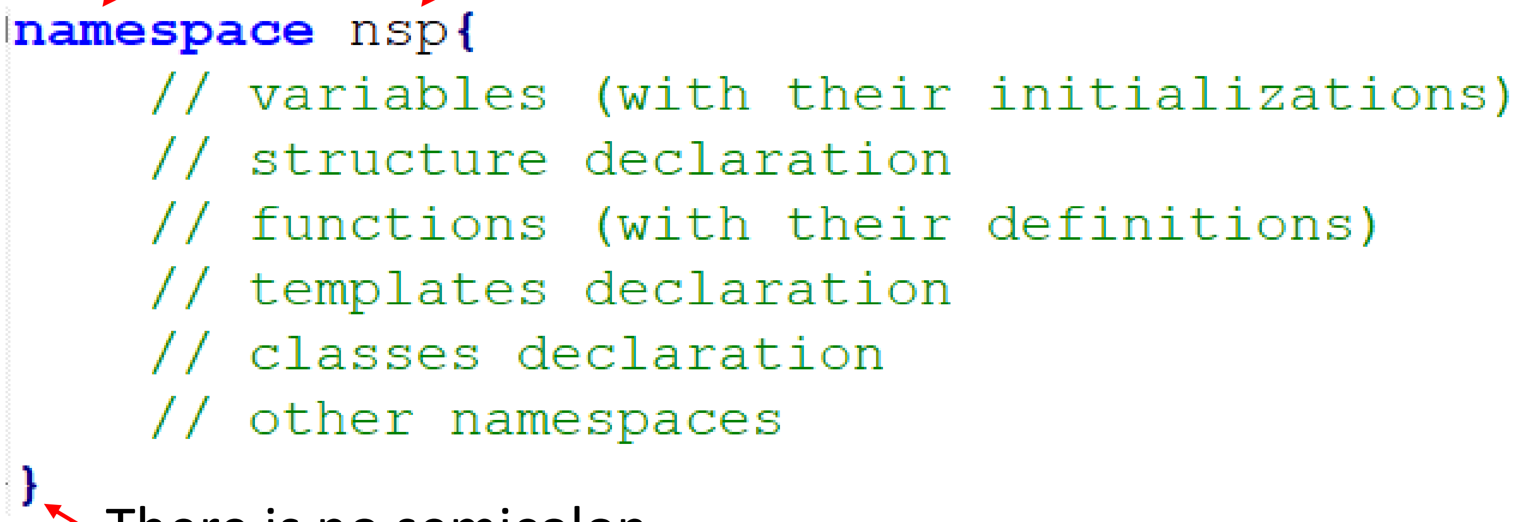
Calling the function for several times

2.3 Namespace

Namespaces provide a much more controlled mechanism for preventing name collisions. A namespace is a scope.

Namespace definition

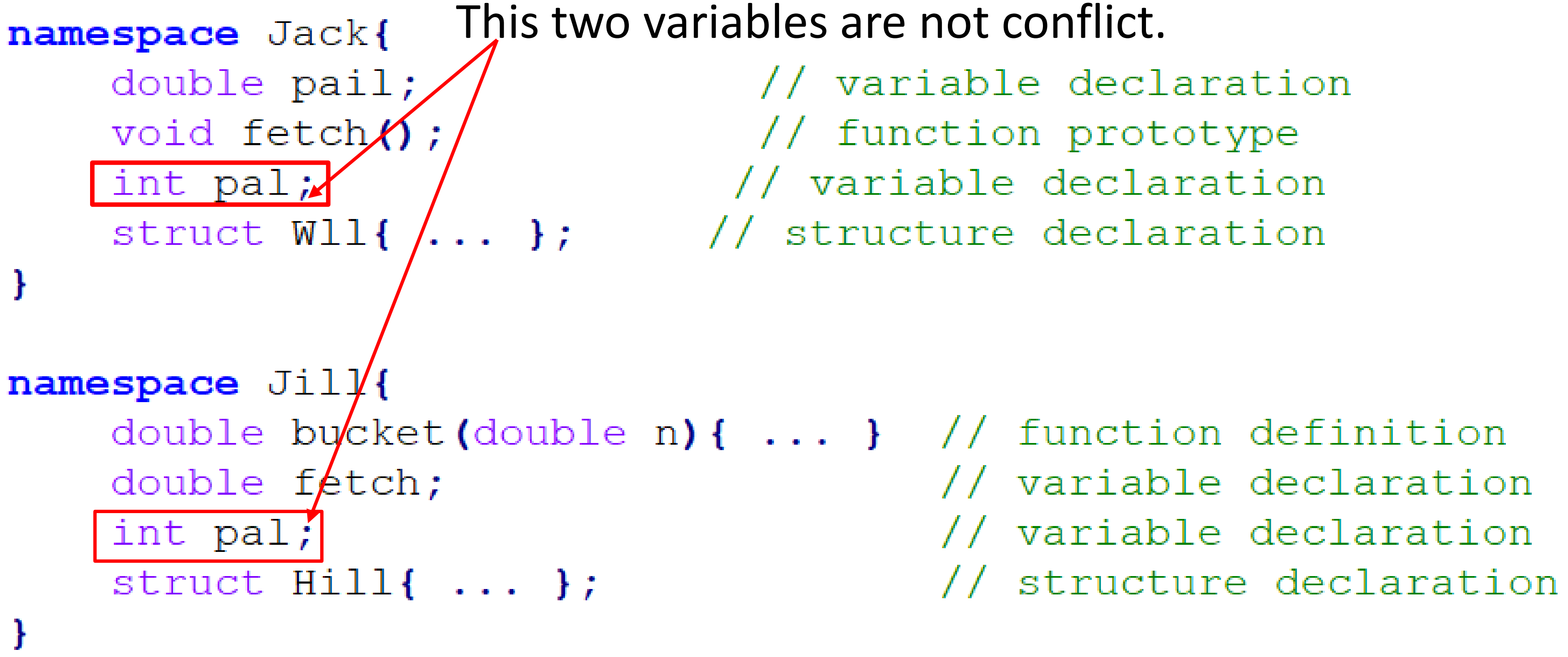
keyword namespace name



```
namespace nsp{
    // variables (with their initializations)
    // structure declaration
    // functions (with their definitions)
    // templates declaration
    // classes declaration
    // other namespaces
}
```

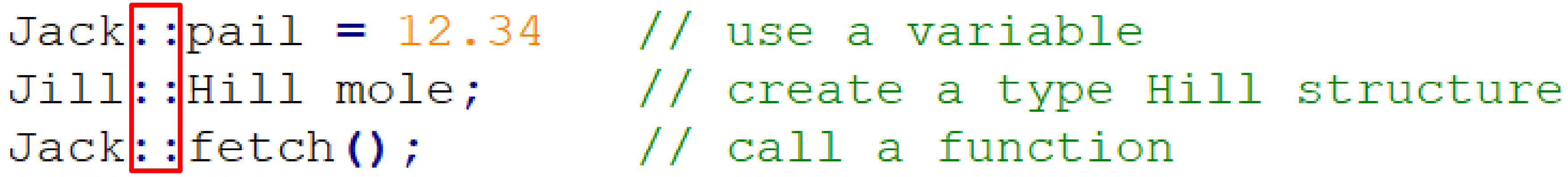
There is no semicolon.

This two variables are not conflict.



```
namespace Jack{
    double pail;                    // variable declaration
    void fetch();                   // function prototype
    int pal;                       // variable declaration
    struct Wll{ ... };            // structure declaration
}

namespace Jill{
    double bucket(double n){ ... }  // function definition
    double fetch;                   // variable declaration
    int pal;                        // variable declaration
    struct Hill{ ... };             // structure declaration
}
```

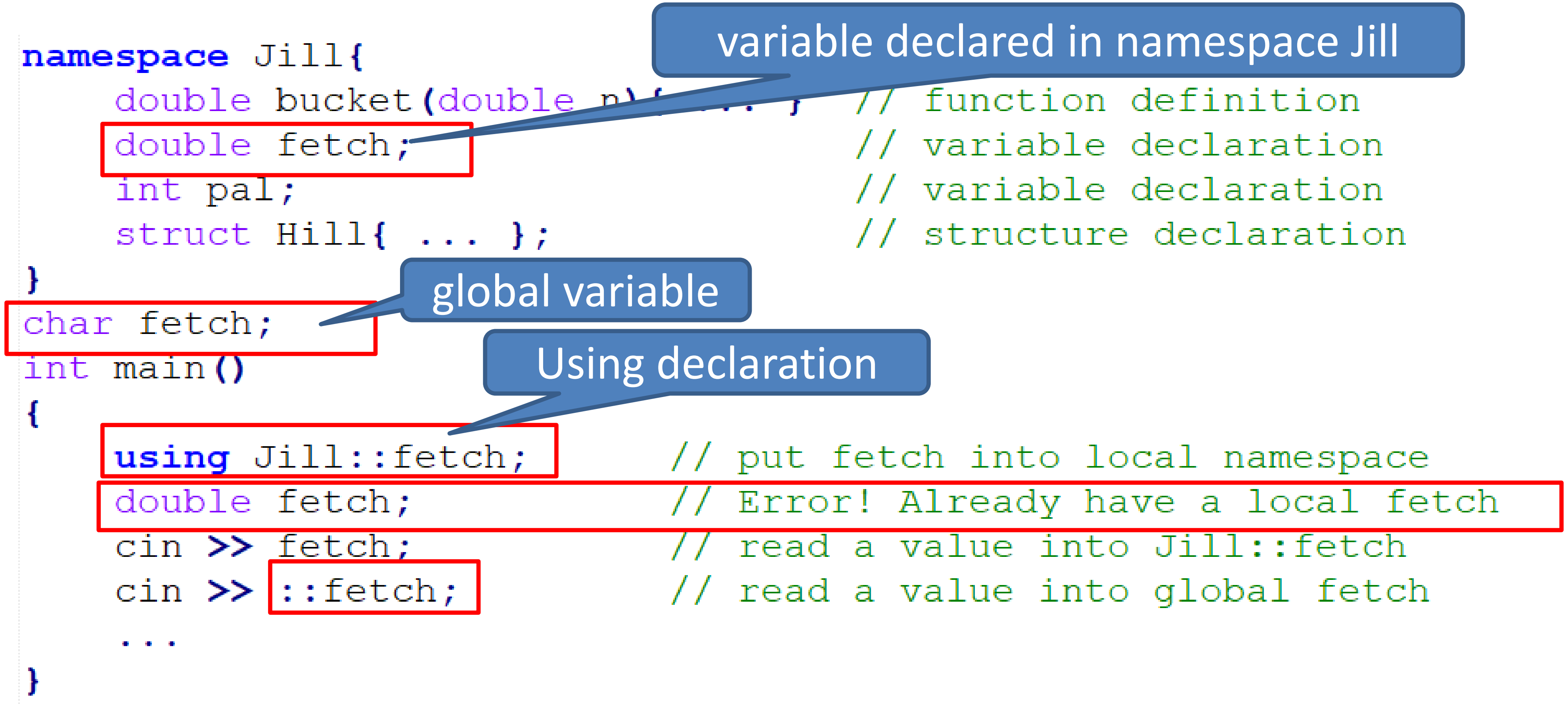
You can use **::**, the scope-resolution operator, to qualify a name with its namespace.



```
Jack::pail = 12.34    // use a variable
Jill::Hill mole;      // create a type Hill structure
Jack::fetch();        // call a function
```

using Declarations and using Directives

A **using declaration** introduces only **one namespace member** at a time. Names introduced in a using declaration obey normal scope rules: they are visible from the point of the using declaration to the end of the scope in which the declaration appears. Entities with the same name defined in an outer scope are hidden.



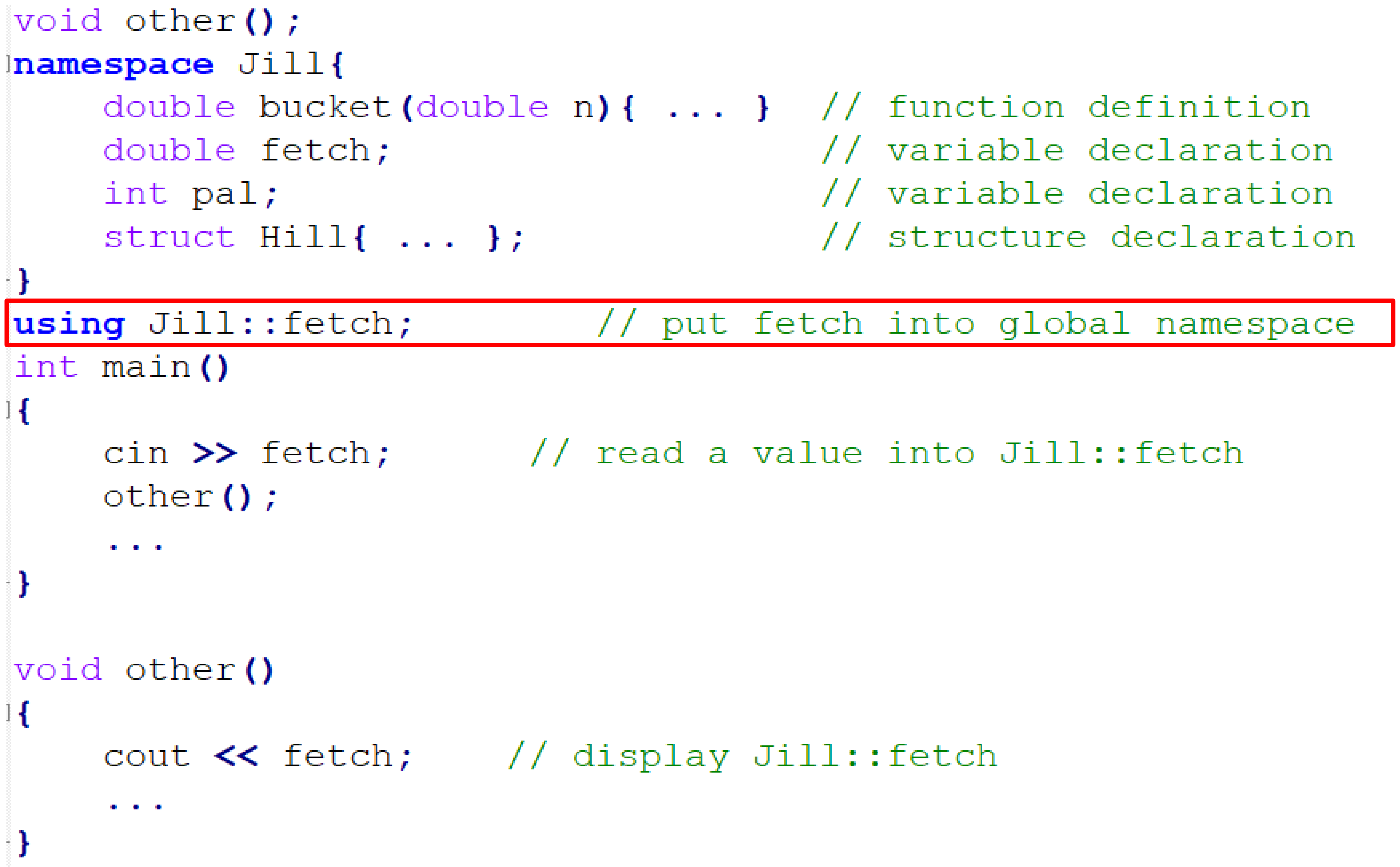
```
namespace Jill{
    double bucket(double n){ ... }         // function definition
    double fetch;                          // variable declaration
    int pal;                               // variable declaration
    struct Hill{ ... };                    // structure declaration
}
char fetch;
int main()
{
    using Jill::fetch;              // put fetch into local namespace
    double fetch;                   // Error! Already have a local fetch
    cin >> fetch;                   // read a value into Jill::fetch
    cin >> ::fetch;                 // read a value into global fetch
    ...
}
```

variable declared in namespace Jill

global variable

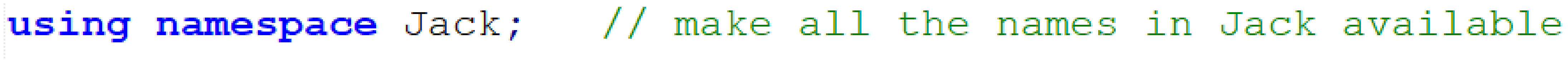
Using declaration

Placing a using declaration at the external level adds the name to the global namespace:

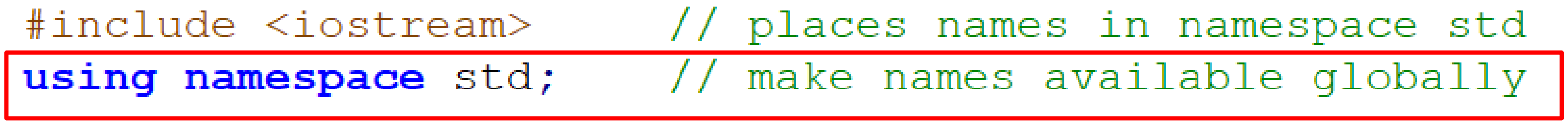


```
void other();
namespace Jill{
    double bucket(double n){ ... }  // function definition
    double fetch;                   // variable declaration
    int pal;                        // variable declaration
    struct Hill{ ... };             // structure declaration
}
using Jill::fetch;        // put fetch into global namespace
int main()
{
    cin >> fetch;      // read a value into Jill::fetch
    other();
    ...
}

void other()
{
    cout << fetch;    // display Jill::fetch
    ...
}
```

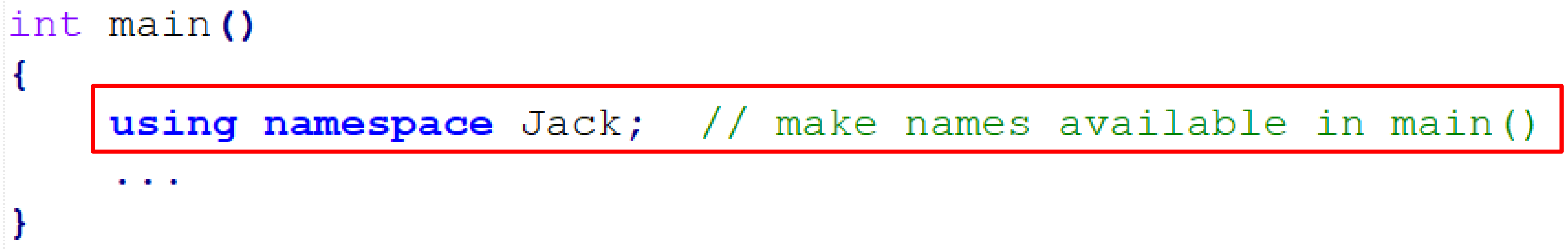
A **using declaration**, makes a single name available. In contrast, the **using directive** makes all the names available.



```
using namespace Jack;    // make all the names in Jack available
```

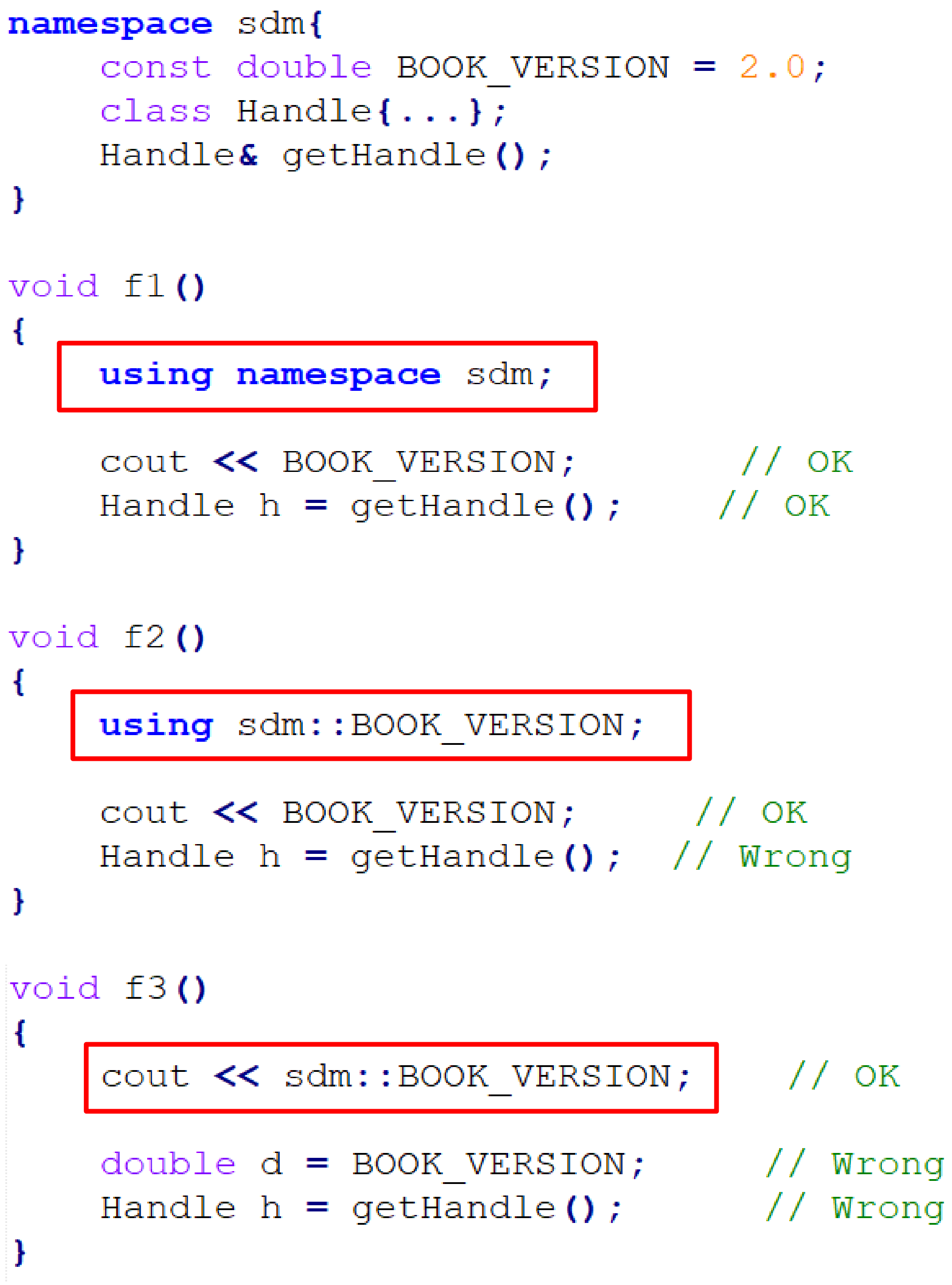


```
#include <iostream>       // places names in namespace std
using namespace std;      // make names available globally
```



```
int main()
{
    using namespace Jack;  // make names available in main()
    ...
}
```

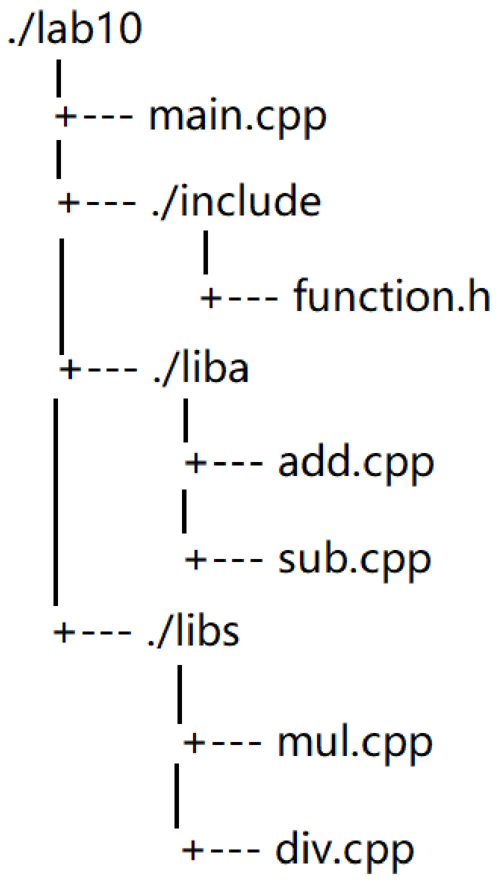
Generally speaking, the using declaration is safer to use than a using directive because it shows exactly what names you are making available.



```
namespace sdm{
    const double BOOK_VERSION = 2.0;
    class Handle{...};
    Handle& getHandle();
}

void f1()
{
    using namespace sdm;

    cout << BOOK_VERSION;       // OK
    Handle h = getHandle();    // OK
}

void f2()
{
    using sdm::BOOK_VERSION;

    cout << BOOK_VERSION;     // OK
    Handle h = getHandle();  // Wrong
}

void f3()
{
    cout << sdm::BOOK_VERSION;    // OK

    double d = BOOK_VERSION;     // Wrong
    Handle h = getHandle();      // Wrong
}
```

3 Exercises

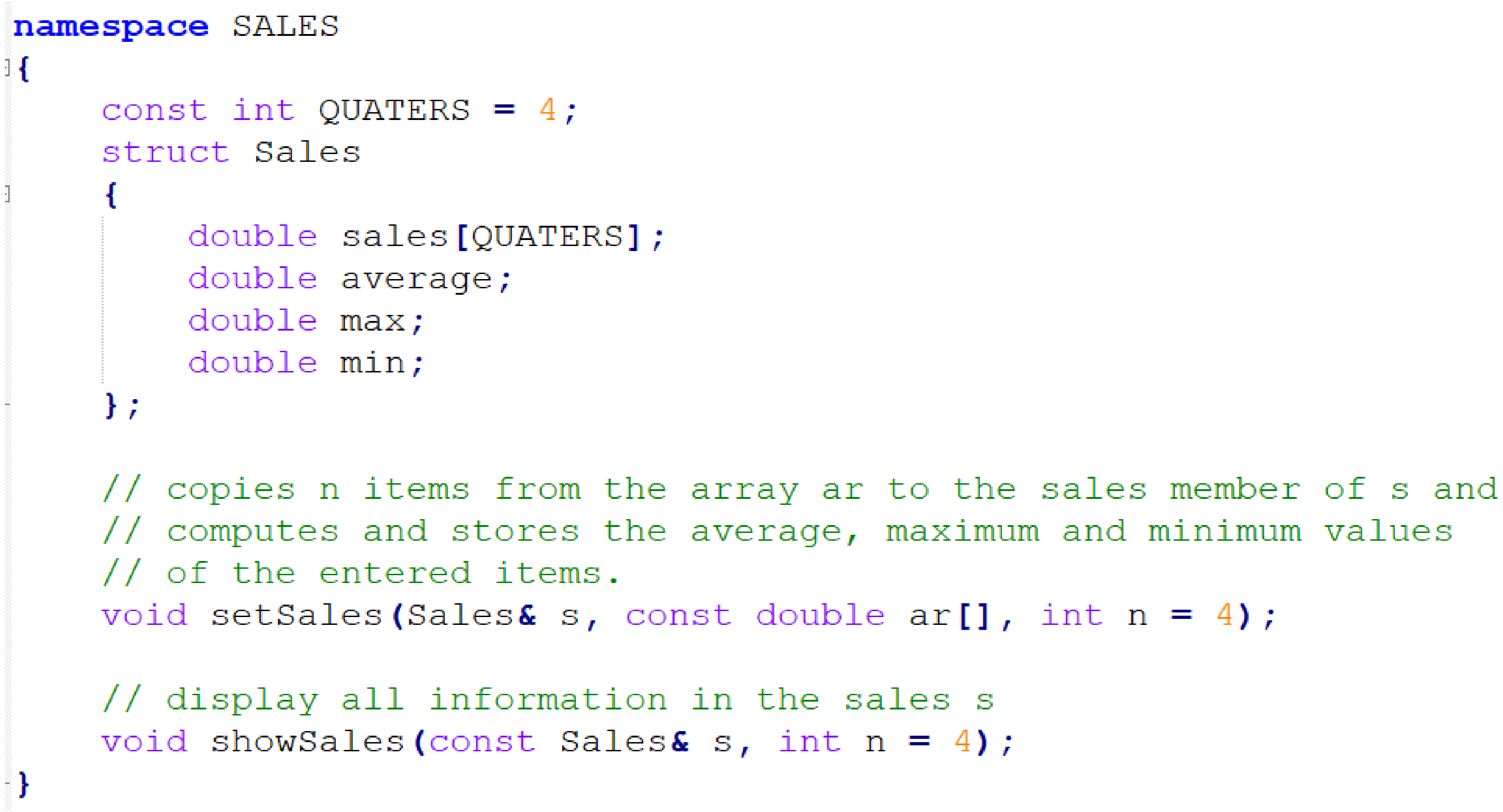
1.Define four functions that implement the operations of addition, subtraction, multiplication and division respectively.(one function one .cpp file) Write a test program to test these functions.



```
./lab10
  |
  +--- main.cpp
  |
  +--- ./include
  |        |
  |        +--- function.h
  +--- ./liba
  |      |
  |      +--- add.cpp
  |      |
  |      +--- sub.cpp
  +--- ./libs
         |
         +--- mul.cpp
         |
         +--- div.cpp
```

According to the tree structure of the files, creates a static library with the two files in the liba directory and a dynamic library with two files in the libs directory. And then link with main.cpp. Using cmake command to compile and build your project. At last run the program.

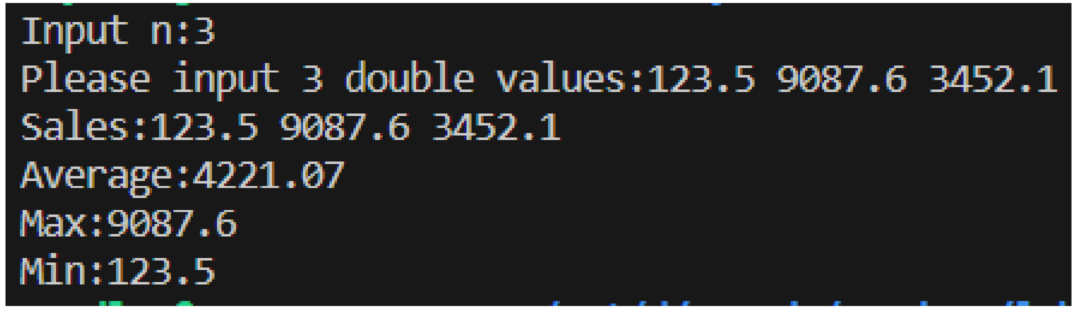
2. Write a three-file program based on the following namespace:



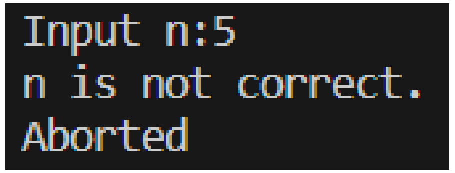
```
namespace SALES
{
    const int QUATERS = 4;
    struct Sales
    {
        double sales[QUATERS];
        double average;
        double max;
        double min;
    };

    // copies n items from the array ar to the sales member of s and
    // computes and stores the average, maximum and minimum values
    // of the entered items.
    void setSales(Sales& s, const double ar[], int n = 4);

    // display all information in the sales s
    void showSales(const Sales& s, int n = 4);
}
```

The **first file** should be a header file that contains the namespace. The **second file** should be a source code file that extends the namespace to provide definitions for the two prototyped functions. The **third file** should define a **Sales object**. It should use setSales() to provide values for the structure. And then it should display the contents of the structure by using showSales().

A sample runs might look like this:



```
Input n:3
Please input 3 double values:123.5 9087.6 3452.1
Sales:123.5 9087.6 3452.1
Average:4221.07
Max:9087.6
Min:123.5
```



```
Input n:5
n is not correct.
Aborted
```