

DASL-100.2

C++ Programming and Linux

Week 4-2

1. Headers
2. Library
3. Make and CMake

DASL-100.2

C++ Programming and Linux

1. Headers

- In C++, a header is a file that contains function and class declarations, constants, and other definitions that can be used by other parts of a program. It helps making programming easier and more efficient.
- Header files typically have the ".h" and they are included in a C++ source file using the `#include` directive.
- Header files allow the separation of interface and implementation in a C++ program. Therefore, it is easier to change the implementation without affecting the interface.
- There are two types of headers:
 - Pre-existing header files: Already exists in C/C++ compiler, we just need to import them. For example, `#include <iostream>` or `#include <string>`.
 - User-defined header files: Defined by the user and can be imported using "include".

1. Headers

- We will be exploring the user-defined header files.
- Example 1, a single header file (.h) without additional source file (.cpp).

header1.h - Documents - Visual Studio Code

```

File Edit Selection View Go Run Terminal Help
EXPLORER
DOCUMENTS
  cplusplus
    week4-1
    week4-2
      C header1.h
      headers
      headers.cpp
C++ class > week4-2 > C header1.h
1 #ifndef HEADER_1_H
2 #define HEADER_1_H
3
4 #include <iostream>
5
6 void function_hello_world(){
7     std::cout << "Hello world!" << std::endl;
8 }
9
10 #endif
  
```

headers.cpp - Documents - Visual Studio Code

```

File Edit Selection View Go Run Terminal Help
EXPLORER
DOCUMENTS
  cplusplus
    week4-1
    week4-2
      C header1.h
      headers
      headers.cpp
C++ class > week4-2 > C headers.cpp
1 #include "header1.h"
2
3 int main(){
4     function_hello_world();
5
6     return 0;
7 }
  
```

PROBLEMS OUTPUT TERMINAL ...

```

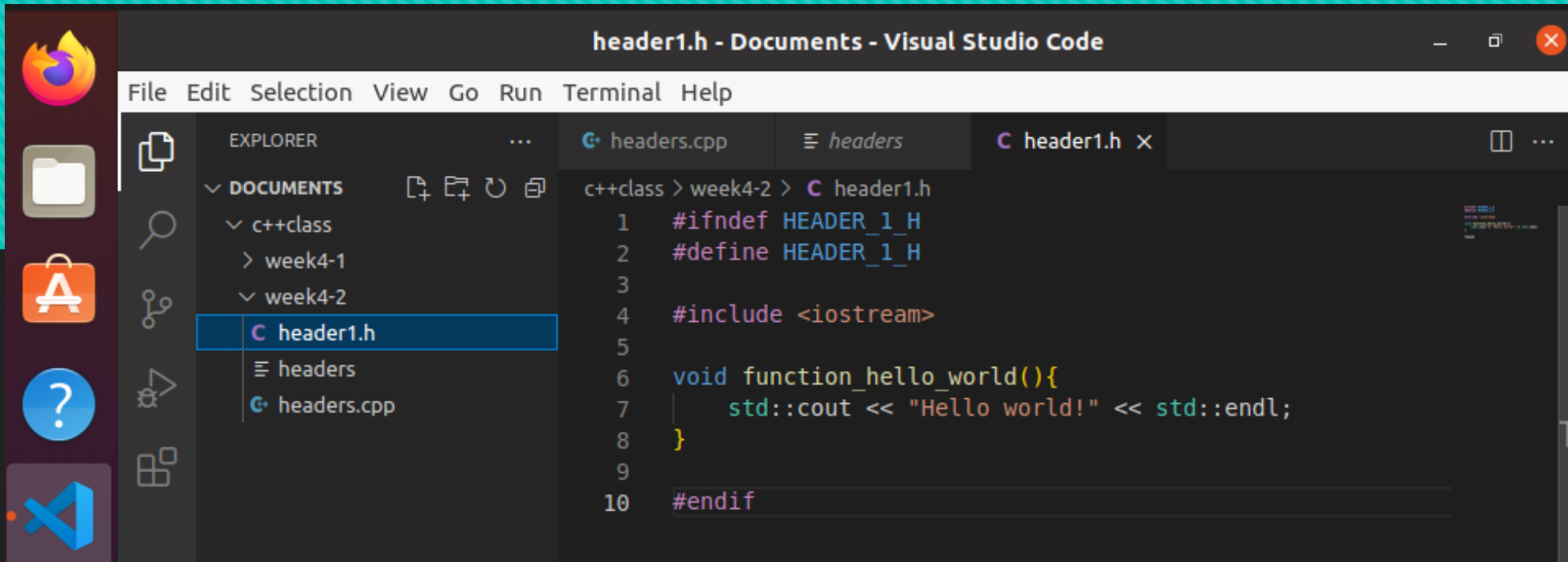
bash - week4-2
● ubuntu20045@ubuntu:~/Documents/cplusplus/week4-2$ g++ headers.cpp -o headers
● ubuntu20045@ubuntu:~/Documents/cplusplus/week4-2$ ls
header1.h headers headers.cpp
● ubuntu20045@ubuntu:~/Documents/cplusplus/week4-2$ ./headers
Hello world!
○ ubuntu20045@ubuntu:~/Documents/cplusplus/week4-2$
  
```

DASL-100.2

C++ Programming and Linux

1. Headers

- In the header file header1.h, #ifndef and #define are preprocessor directives in C++ that are commonly used together in header files to prevent multiple inclusions of the same header file.
- The #ifndef directive stands for "if not defined".
- The #define directive is used to define a macro or identifier.



```
header1.h - Documents - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
DOCUMENTS
  cplusplus
    week4-1
    week4-2
      C header1.h
      headers
      headers.cpp
c++class > week4-2 > C header1.h
1  #ifndef HEADER_1_H
2  #define HEADER_1_H
3
4  #include <iostream>
5
6  void function_hello_world(){
7  |   std::cout << "Hello world!" << std::endl;
8  | }
9
10 #endif
```

DASL-100.2

C++ Programming and Linux

1. Headers

- Example 2, a header file (.h) with additional source file (.cpp).

functions.cpp - Documents - Visual Studio Code

```

File Edit Selection View Go Run Terminal Help
EXPLORER
DOCUMENTS
  c++class
    week4-1
    week4-2
      functions.cpp
      header1.h
      headers.cpp
c++class > week4-2 > functions.cpp
1  #include "header1.h"
2
3  #include <iostream>
4
5
6  void function_hello_world() {
7      std::cout << "Hello world!" << std::endl;
8  }
9

```

headers.cpp - Documents - Visual Studio Code

```

File Edit Selection View Go Run Terminal Help
EXPLORER
DOCUMENTS
  c++class
    week4-1
    week4-2
      functions.cpp
      header1.h
      headers.cpp
c++class > week4-2 > headers.cpp
1  #include "header1.h"
2
3  int main(){
4      function_hello_world();
5
6      return 0;
7  }

```

header1.h - Documents - Visual Studio Code

```

File Edit Selection View Go Run Terminal Help
EXPLORER
DOCUMENTS
  c++class
    week4-1
    week4-2
      functions.cpp
      header1.h
      headers.cpp
c++class > week4-2 > header1.h
1  #ifndef HEADER_1_H
2  #define HEADER_1_H
3
4
5
6  void function_hello_world();
7
8
9  #endif

```


DASL-100.2

C++ Programming and Linux

1. Headers

- Note that if we use the traditional way “`g++ headers.cpp -o headers`” to compile this, we will get an error.
- Because the additional source file (.cpp) is not able to be linked to our executable file.
- Use “`g++ headers.cpp functions.cpp header1.h -o headers`” instead.

```
PROBLEMS OUTPUT TERMINAL ... bash - week4-2 + v [ ] [ ] ... ^ x
⊗ ubuntu20045@ubuntu:~/Documents/c++class/week4-2$ g++ headers.cpp -o headers
/usr/bin/ld: /tmp/ccjQKCE5.o: in function `main':
headers.cpp:(.text+0x9): undefined reference to `function_hello_world()'
collect2: error: ld returned 1 exit status
○ ubuntu20045@ubuntu:~/Documents/c++class/week4-2$ [ ]
```

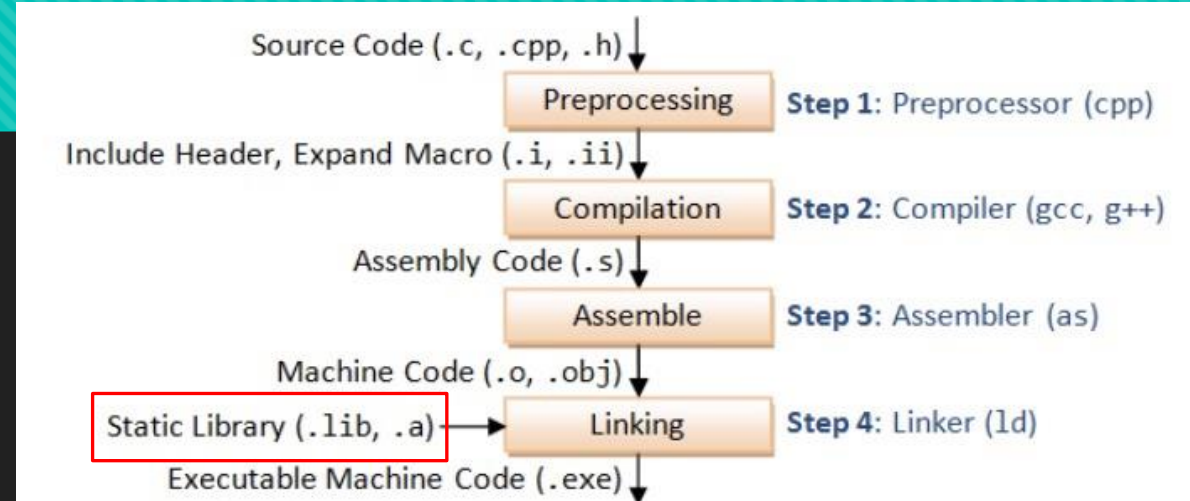
```
PROBLEMS OUTPUT TERMINAL ... bash - week4-2 + v [ ] [ ] ... ^ x
⊗ ubuntu20045@ubuntu:~/Documents/c++class/week4-2$ g++ headers.cpp -o headers
/usr/bin/ld: /tmp/ccjQKCE5.o: in function `main':
headers.cpp:(.text+0x9): undefined reference to `function_hello_world()'
collect2: error: ld returned 1 exit status
● ubuntu20045@ubuntu:~/Documents/c++class/week4-2$ g++ headers.cpp functions.cpp header1.h -o headers
● ubuntu20045@ubuntu:~/Documents/c++class/week4-2$ ls
functions.cpp header1.h headers headers.cpp
● ubuntu20045@ubuntu:~/Documents/c++class/week4-2$ ./headers
Hello world!
○ ubuntu20045@ubuntu:~/Documents/c++class/week4-2$ [ ]
```

DASL-100.2

C++ Programming and Linux

1. Libraries

- C++ class libraries are collections of pre-written C++ classes that provide developers with pre-built solutions for common programming tasks. These libraries can help to speed up development time and reduce the amount of code needed to build an application.
- The object library provides compiled functions and data that are linked with your program to produce an executable program. Types of Libraries include:
 - Standard Libraries: provides several generic, function objects, generic strings and streams (including interactive and file I/O), etc.
 - Static Libraries.
 - Dynamic (Shared) Libraries.

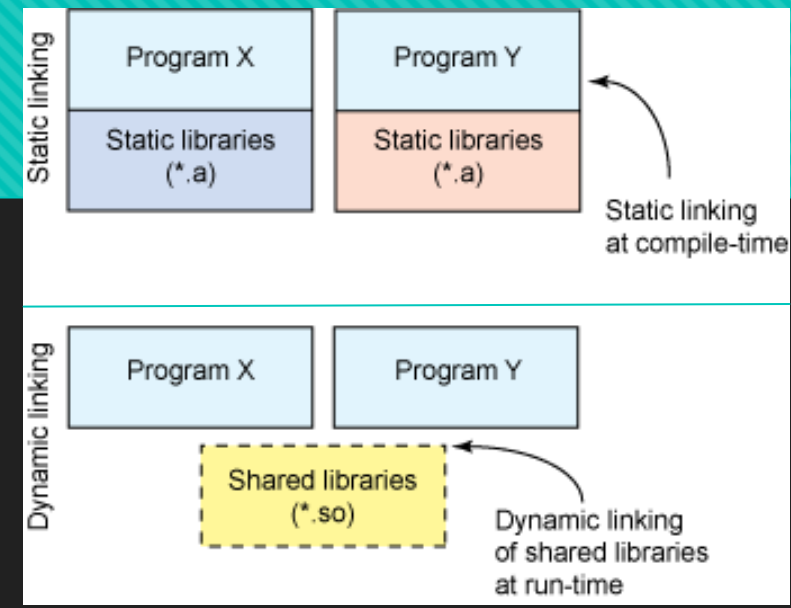


DASL-100.2

C++ Programming and Linux

1. Libraries

- A **static** library is linked directly into an executable during the build process, resulting in a larger executable file that includes all the library code. The library code is loaded directly into memory at runtime, making it more efficient.
- A **dynamic** library is loaded at runtime by an executable or another shared library, resulting in a smaller executable file that only contains a reference to the library code. The library code is loaded into memory at runtime by the operating system, making it more flexible.
- Overall, **static** libraries are good for small projects where performance is critical and library size is not a concern. **Dynamic** libraries are good for larger projects where library size and flexibility are important.
- **Static** libraries are .a files in Linux and .lib files in Windows.
- **Dynamic** libraries are .so in Linux and .dll in Windows.



DASL-100.2

C++ Programming and Linux

2. Libraries - Static

- Example 1, static libraries:
 - Step 1, create a main source code "staticlibexample.cpp".
 - Create a folder name "libraries".
 - Create functions.cpp file and header1.h file inside libraries folder.

```
staticlibexample.cpp - week4-2 - Visual Studio Code  
File Edit Selection View Go Run Terminal Help  
EXPLORER  
WEEK4-2  
  .vscode  
  libraries  
    functions.cpp  
    header1.h  
    staticlibexample.cpp  
staticlibexample.cpp  
1 #include "./libraries/header1.h"  
2  
3 int main() {  
4     function_hello_world();  
5  
6     return 0;  
7 }  
PROBLEMS TERMINAL  
bash - libraries  
ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$
```

```
functions.cpp - week4-2 - Visual Studio Code  
File Edit Selection View Go Run Terminal Help  
EXPLORER  
WEEK4-2  
  .vscode  
  libraries  
    functions.cpp  
    header1.h  
    staticlibexample.cpp  
functions.cpp  
1 #include "header1.h"  
2  
3 #include <iostream>  
4  
5 void function_hello_world(){  
6     std::cout << "Hello world!" << std::endl;  
7 }  
PROBLEMS TERMINAL  
bash - libraries  
ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$
```

```
header1.h - week4-2 - Visual Studio Code  
File Edit Selection View Go Run Terminal Help  
EXPLORER  
WEEK4-2  
  .vscode  
  libraries  
    functions.cpp  
    header1.h  
    staticlibexample.cpp  
header1.h  
1 #ifndef HEADER_1_H  
2 #define HEADER_1_H  
3  
4 void function_hello_world();  
5  
6 #endif  
PROBLEMS TERMINAL  
bash - libraries  
ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$
```

2. Libraries - Static

- Example 1, static libraries:
 - Step 2, navigate to libraries folder and generate an object file (.o) from the function.cpp file by using the command: `g++ -c functions.cpp -o functions.o`.

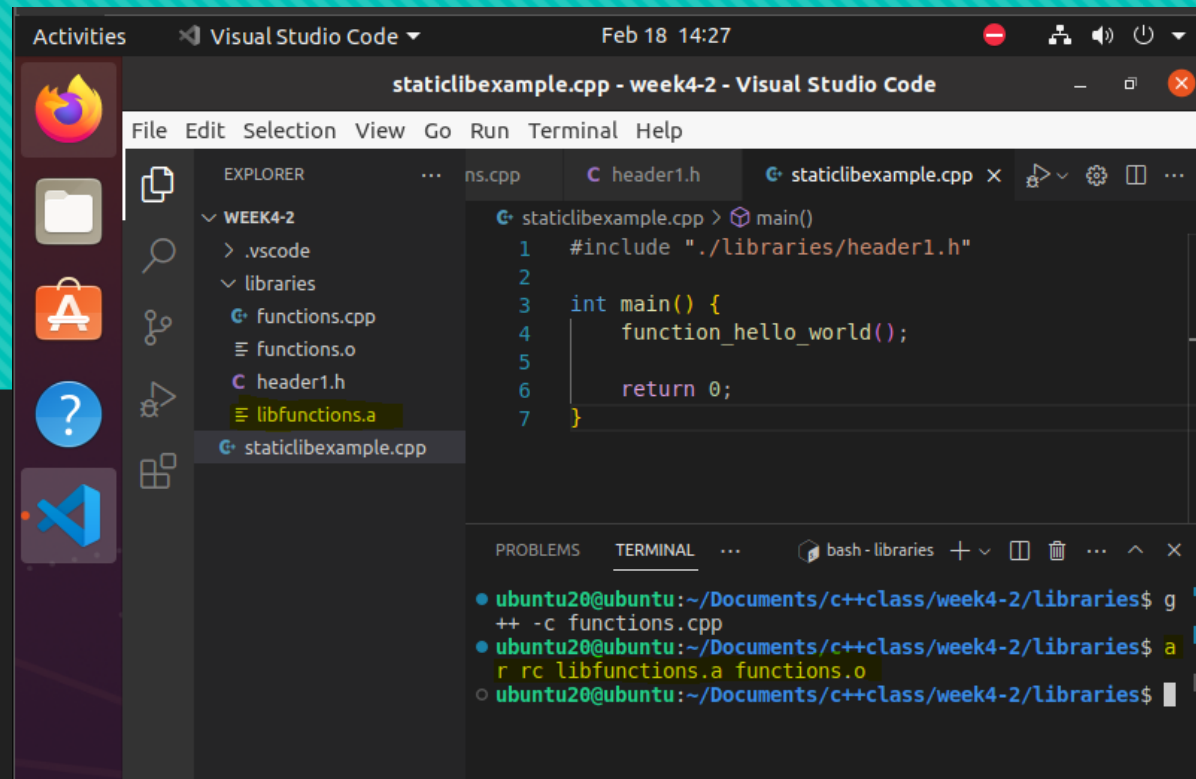
```
staticlibexample.cpp - week4-2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
WEEK4-2
  .vscode
  libraries
    functions.cpp
    functions.o
    header1.h
    staticlibexample.cpp
staticlibexample.cpp
1 #include "./libraries/header1.h"
2
3 int main() {
4     function_hello_world();
5
6     return 0;
7 }
TERMINAL
bash - libraries
● ubuntu20@ubuntu:~/Documents/c++class/week4-2$ ls
libraries staticlibexample.cpp
● ubuntu20@ubuntu:~/Documents/c++class/week4-2$ cd libraries/
● ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$ ls
functions.cpp header1.h
● ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$ g++ -c functions.cpp -o functions.o
○ ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$
```

DASL-100.2

C++ Programming and Linux

2. Libraries - Static

- Example 1, static libraries:
 - Step 3, generate a static library file (.a) to contain all the object files (.o) by using the command: “ar rc libfunctions.a functions.o”.



```
staticlibexample.cpp - week4-2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
WEEK4-2
  .vscode
  libraries
    functions.cpp
    functions.o
    header1.h
    libfunctions.a
    staticlibexample.cpp
staticlibexample.cpp
1 #include "./libraries/header1.h"
2
3 int main() {
4     function_hello_world();
5
6     return 0;
7 }
TERMINAL
bash - libraries
● ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$ g++ -c functions.cpp
● ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$ ar rc libfunctions.a functions.o
○ ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$
```

2. Libraries - Static

- Example 1, static libraries:
 - Step 4, navigate back to the main source code folder and generate an executable file with the static library file by using the command `g++ staticlibexample.cpp -o staticlibexample -L ./libraries/ -lfunctions`

The screenshot shows the Visual Studio Code interface with a terminal window. The terminal output is as follows:

```
ubuntu20@ubuntu:~/Documents/c++class/week4-2/libraries$ c
d ..
ubuntu20@ubuntu:~/Documents/c++class/week4-2$ ls
libraries  staticlibexample.cpp
ubuntu20@ubuntu:~/Documents/c++class/week4-2$ g++ staticl
ibexample.cpp -o staticlibexample -L ./libraries/ -lfunct
ions
ubuntu20@ubuntu:~/Documents/c++class/week4-2$ ls
libraries  staticlibexample  staticlibexample.cpp
ubuntu20@ubuntu:~/Documents/c++class/week4-2$ ./staticlib
example
Hello world!
ubuntu20@ubuntu:~/Documents/c++class/week4-2$
```

2. Libraries - Dynamic

- Example 2, dynamic libraries:
 - Step 1, create a main source code “dynamiclibexample.cpp”.
 - Create a folder name “libraries”.
 - Create avgfunction.cpp file and header1.h file inside libraries folder.

The image displays three Visual Studio Code windows illustrating the setup for dynamic libraries:

- dynamiclibexample.cpp - week4-2 - Visual Studio Code:** Shows the main source code. The Explorer sidebar shows a folder named 'libraries' containing 'avgfunction.cpp' and 'header1.h'. The code includes the header file and defines a main function that calls the average function.

```

1 #include "./libraries/header1.h"
2 #include <iostream>
3 int main() {
4
5     int array[5] = {3,4,5,6,8};
6     double avg = average(array);
7     std::cout<<"Results: "<<avg<<std::endl;
8
9     return 0;
10 }
11

```

- avgfunction.cpp - week4-2 - Visual Studio Code:** Shows the implementation of the average function. The Explorer sidebar shows the 'libraries' folder containing 'avgfunction.cpp' and 'header1.h'.

```

1 #include "header1.h"
2
3 double average(int* array)
4 {
5     double sum = 0;
6     for(int i=0;i<5;i++)
7     {
8         sum += array[i];
9     }
10    double average = sum/5;
11    return average;
12 }
13
14

```

- header1.h - week4-2 - Visual Studio Code:** Shows the header file for the average function. The Explorer sidebar shows the 'libraries' folder containing 'header1.h'.

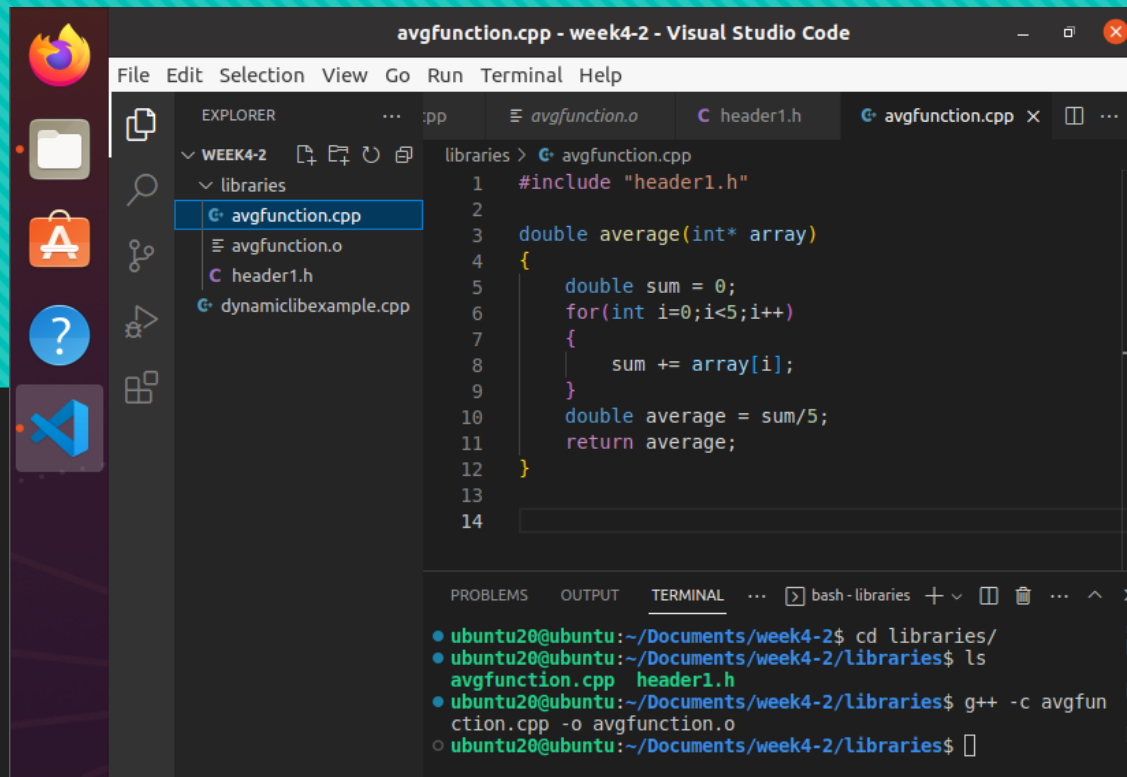
```

1 #ifndef HEADER_1_H
2 #define HEADER_1_H
3
4 double average(int* array);
5
6 #endif
7

```


2. Libraries - Dynamic

- Example 2, dynamic libraries:
 - Step 2, navigate to libraries folder and generate an object file (.o) from the sum.cpp file by using the command: `g++ -c avgfunction.cpp -o avgfunction.o`.



The screenshot shows the Visual Studio Code interface. The Explorer view on the left shows a project named 'WEEK4-2' with a subfolder 'libraries' containing 'avgfunction.cpp', 'avgfunction.o', and 'header1.h'. The main editor window shows the contents of 'avgfunction.cpp' with the following code:

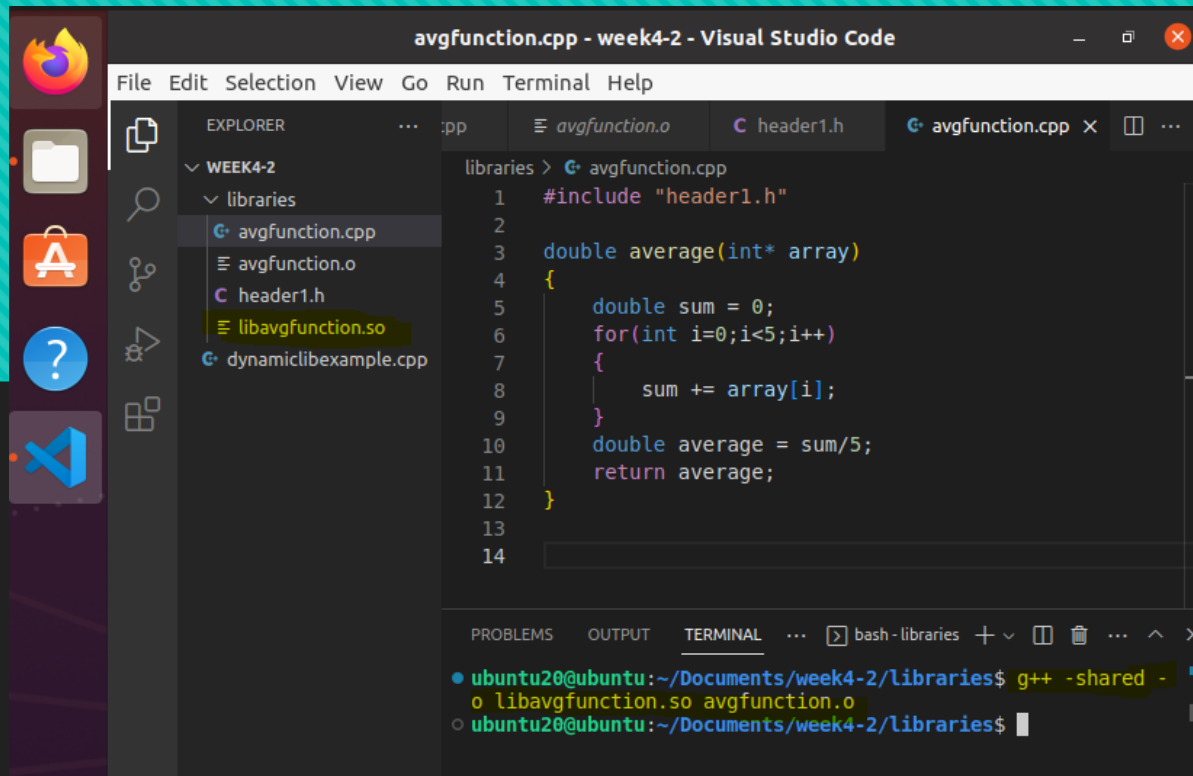
```
1 #include "header1.h"
2
3 double average(int* array)
4 {
5     double sum = 0;
6     for(int i=0;i<5;i++)
7     {
8         sum += array[i];
9     }
10    double average = sum/5;
11    return average;
12 }
13
14
```

The Terminal view at the bottom shows the following commands and output:

```
ubuntu20@ubuntu:~/Documents/week4-2$ cd libraries/
ubuntu20@ubuntu:~/Documents/week4-2/libraries$ ls
avgfunction.cpp  header1.h
ubuntu20@ubuntu:~/Documents/week4-2/libraries$ g++ -c avgfunction.cpp -o avgfunction.o
ubuntu20@ubuntu:~/Documents/week4-2/libraries$
```

2. Libraries - Dynamic

- Example 2, dynamic libraries:
 - Step 3, generate a dynamic library file (.so) to contain all the object files (.o) by using the command: “g++ -shared -o libavgfunction.so avgfunction.o”



The screenshot shows the Visual Studio Code interface with a project named 'week4-2'. The Explorer view on the left shows a folder 'libraries' containing 'avgfunction.cpp', 'avgfunction.o', 'header1.h', 'libavgfunction.so', and 'dynamiclibexample.cpp'. The main editor displays the contents of 'avgfunction.cpp', which includes a header file and a function definition for 'average'. The Terminal at the bottom shows the command being executed: `g++ -shared -o libavgfunction.so avgfunction.o`.

```
avgfunction.cpp - week4-2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
WEEK4-2
  libraries
    avgfunction.cpp
    avgfunction.o
    header1.h
    libavgfunction.so
    dynamiclibexample.cpp
  libraries > avgfunction.cpp
1  #include "header1.h"
2
3  double average(int* array)
4  {
5      double sum = 0;
6      for(int i=0;i<5;i++)
7      {
8          sum += array[i];
9      }
10     double average = sum/5;
11     return average;
12 }
13
14
TERMINAL
ubuntu20@ubuntu:~/Documents/week4-2/libraries$ g++ -shared -o libavgfunction.so avgfunction.o
ubuntu20@ubuntu:~/Documents/week4-2/libraries$
```

2. Libraries - Dynamic

- Example 2, dynamic libraries:
 - Step 4, navigate back to the main source code folder and generate an executable file with the dynamic library file by using the command :"g++ dynamiclibexample.cpp -o dynamiclibexample -L ./libraries/ -lavgfunction"
 - Note that when we try to run the executable file it gives us an error.

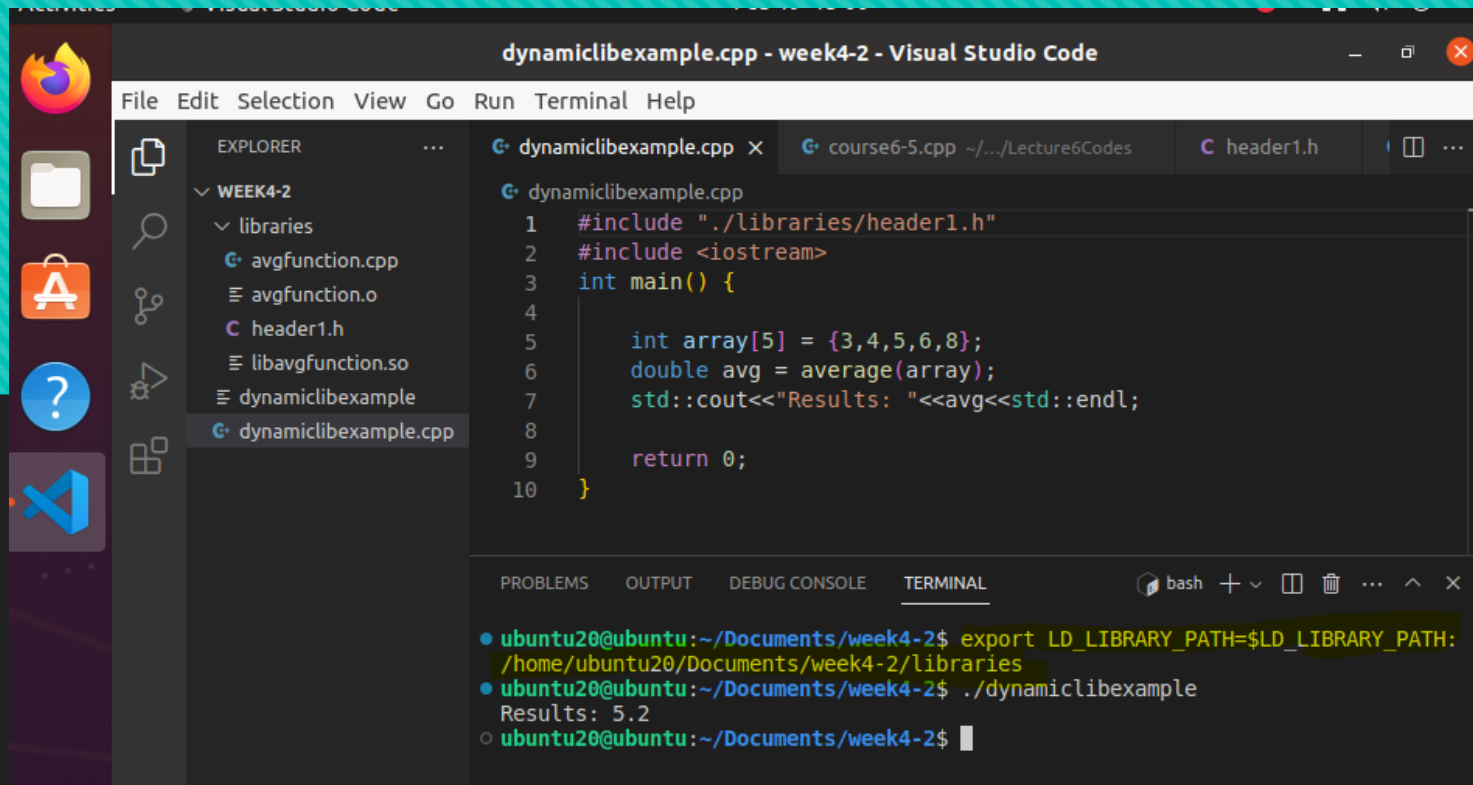
The screenshot shows the Visual Studio Code interface with a C++ file named `dynamiclibexample.cpp` open. The code includes a header file `header1.h` and a function `average` from `avgfunction.cpp`. The `main` function uses `average` and prints the results. The terminal window shows the following commands and output:

```

ubuntu20@ubuntu:~/Documents/week4-2/libraries$ cd ..
ubuntu20@ubuntu:~/Documents/week4-2$ ls
dynamiclibexample.cpp  libraries
ubuntu20@ubuntu:~/Documents/week4-2$ g++ dynamiclibexample.cpp -o dynamiclibexample -L ./libraries/ -lavgfunction
ubuntu20@ubuntu:~/Documents/week4-2$ ls
dynamiclibexample  dynamiclibexample.cpp  libraries
ubuntu20@ubuntu:~/Documents/week4-2$ ./dynamiclibexample
./dynamiclibexample: error while loading shared libraries: libavgfunction.so: cannot open shared object file: No such file or directory
ubuntu20@ubuntu:~/Documents/week4-2$
    
```

2. Libraries - Dynamic

- Example 2, dynamic libraries:
 - Step 5, link the path to the executable file by type in the command:
“export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:/home/ubuntu20/Documents/week4-2/libraries”



The screenshot shows the Visual Studio Code editor with a C++ file named `dynamiclibexample.cpp` open. The code in the editor is as follows:

```
1 #include "./libraries/header1.h"
2 #include <iostream>
3 int main() {
4
5     int array[5] = {3,4,5,6,8};
6     double avg = average(array);
7     std::cout<<"Results: "<<avg<<std::endl;
8
9     return 0;
10 }
```

The terminal at the bottom shows the following commands and output:

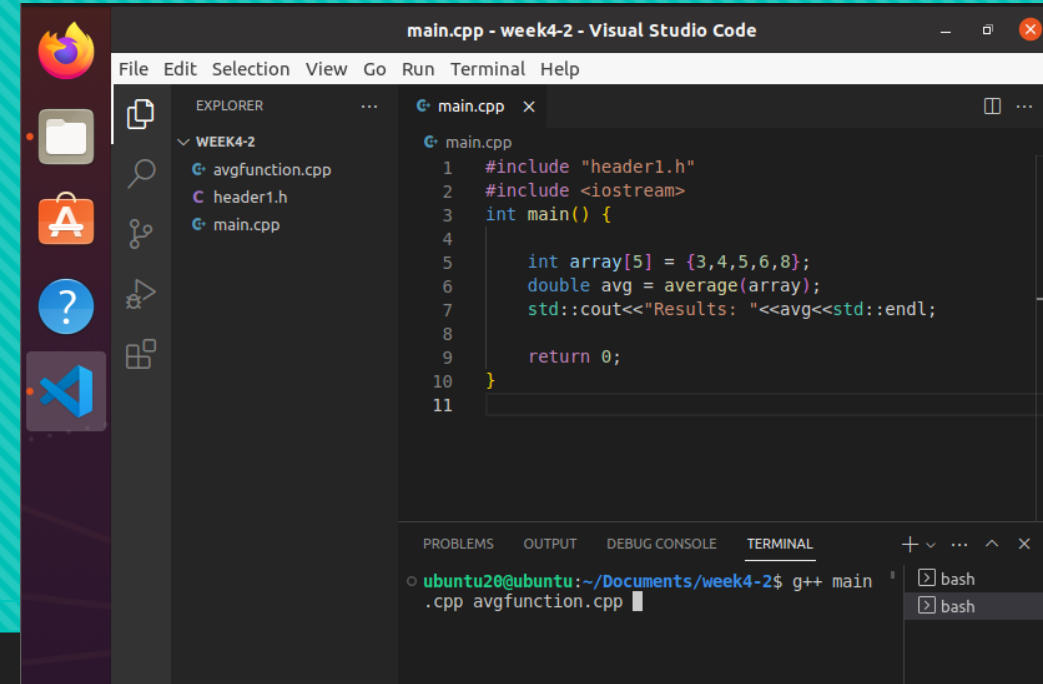
```
ubuntu20@ubuntu:~/Documents/week4-2$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
/home/ubuntu20/Documents/week4-2/libraries
ubuntu20@ubuntu:~/Documents/week4-2$ ./dynamiclibexample
Results: 5.2
ubuntu20@ubuntu:~/Documents/week4-2$
```

DASL-100.2

C++ Programming and Linux

3. Make

- Often, a program is considered of several files. It is painful to link and compile as we demonstrated in the libraries.
- Make is a build automation tool that is used to build software projects by specifying a set of rules and dependencies between files. Make reads a file called "Makefile" that contains the rules and dependencies for the project. The Makefile specifies the targets, dependencies, and commands needed to build the software.
- We can simply do "g++ main.cpp avgfunction.cpp" but if there are a lot of files, we would have to incorporate them all in the command line.



The screenshot shows the Visual Studio Code editor with a C++ file named `main.cpp` open. The code in the editor is as follows:

```
1 #include "header1.h"
2 #include <iostream>
3 int main() {
4
5     int array[5] = {3,4,5,6,8};
6     double avg = average(array);
7     std::cout<<"Results: "<<avg<<std::endl;
8
9     return 0;
10 }
11
```

The Explorer sidebar on the left shows a folder named `WEEK4-2` containing the files `avgfunction.cpp`, `header1.h`, and `main.cpp`. The Terminal at the bottom shows the command `g++ main.cpp avgfunction.cpp` being executed, with the prompt `ubuntu20@ubuntu:~/Documents/week4-2$` and the output `bash`.

DASL-100.2

C++ Programming and Linux

3. Make

- Install make, type in command : "sudo apt install make"
- Create a Makefile by click new file and name it as "Makefile"

The screenshot shows the Visual Studio Code interface with the 'main.cpp' file open. The terminal at the bottom displays the command `sudo apt install make` being executed. The Explorer sidebar shows the file structure for 'WEEK4-2', including 'main.cpp', 'Makefile', 'main.o', 'avgfunction.cpp', 'avgfunction.o', 'header1.h', and 'myexe'. The main editor shows the following C++ code in 'main.cpp':

```
1 #include "header1.h"
2 #include <iostream>
3 int main() {
4
5     int array[5] = {3,4,5,6,8};
6     double avg = average(array);
7     std::cout<<"Results: "<<avg<<std::endl;
8
9     return 0;
10 }
11
```

The screenshot shows the Visual Studio Code interface with the 'Makefile' file open. The Explorer sidebar shows the file structure for 'WEEK4-2', including 'main.cpp', 'Makefile', 'main.o', 'avgfunction.cpp', 'avgfunction.o', 'header1.h', and 'myexe'. The main editor shows the following Makefile content:

```
M Makefile
1  output: main.o avgfunction.o
2      g++ main.o avgfunction.o -o myexe
3
4  main.o: main.cpp
5      g++ -c main.cpp
6
7  avgfunction.o: avgfunction.cpp header1.h
8      g++ -c avgfunction.cpp
9
10 clean:
11     rm *.o myexe
```

The terminal at the bottom shows the execution of `make` and `./myexe`, resulting in the output: `Results: 5.2`.

DASL-100.2

C++ Programming and Linux

3. CMake

- Similar to *Make*, *CMake* is a cross-platform build system that is used to manage the build process for C++ projects. *CMake* generates native build files for various platforms such as Unix, Windows, and macOS. The build process for C++ projects typically involves compiling the source code, linking object files, and generating executables or libraries.
- *CMake* uses a file called "CMakeLists.txt" to define the build process for a project. The CMakeLists.txt file specifies the source files, libraries, and dependencies for the project, and defines how the project should be built. *CMake* can also be used to generate project files for various integrated development environments (IDEs), such as Visual Studio, Eclipse, and Xcode.

DASL-100.2

C++ Programming and Linux

3. CMake

- Install cmake, type in the command "sudo apt install cmake".
- Create a new file "CMakeLists.txt".

The image consists of two side-by-side screenshots of the Visual Studio Code editor interface. The left screenshot shows the 'header1.h' file being edited. The code in the editor is as follows:

```
1 #ifndef HEADER_1_H
2 #define HEADER_1_H
3
4 double average(int* array);
5
6 #endif
7
```

The terminal at the bottom shows the command `sudo apt install cmake` being executed, with the output `bash` and `bash` indicating successful execution.

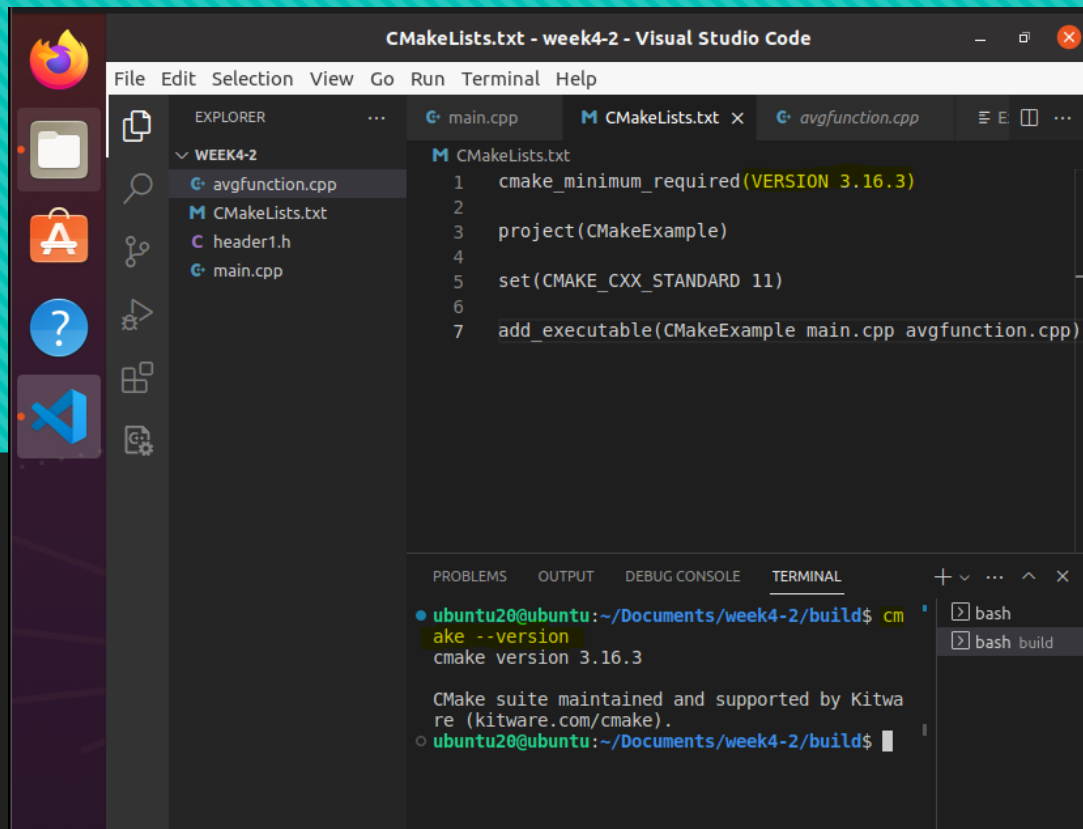
The right screenshot shows the 'CMakeLists.txt' file being created. The code in the editor is as follows:

```
1
```

The terminal at the bottom shows the command `bash` being executed, with the output `bash` indicating successful execution.

3. CMake

- Check CMake version by type in command : "cmake --version"
- Type in the CMake file as follow:



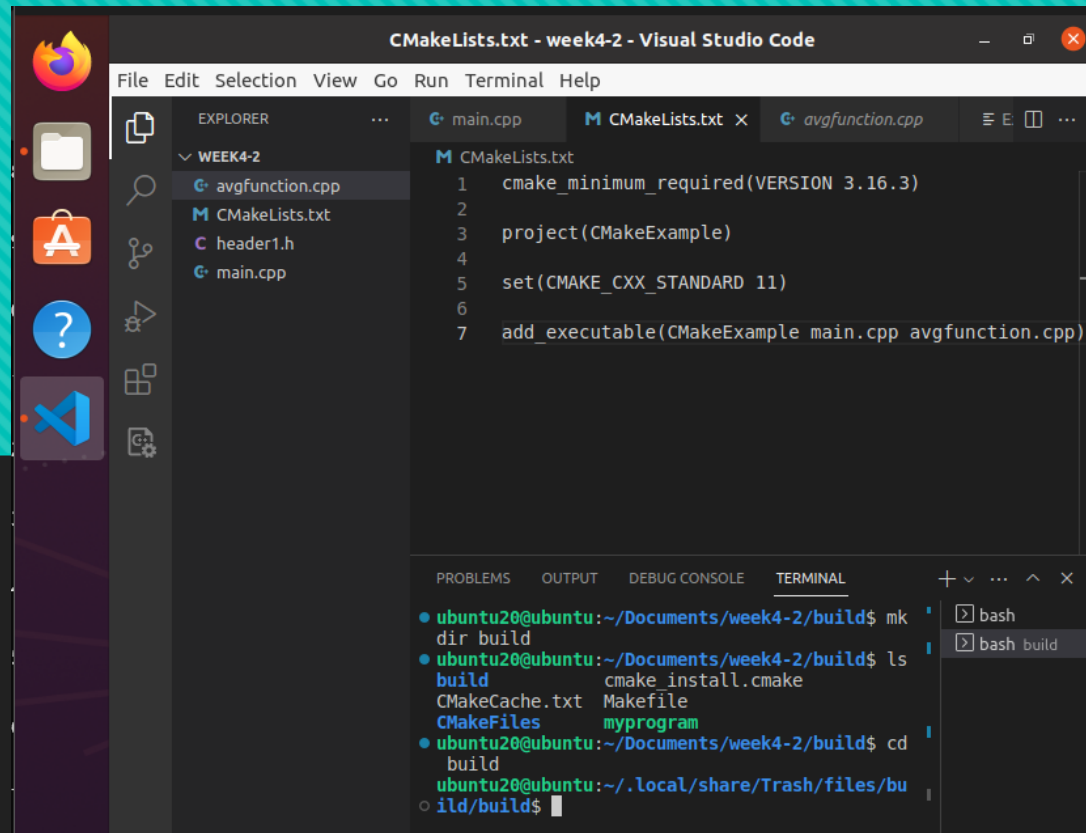
The screenshot shows the Visual Studio Code interface. The Explorer view on the left shows a project folder named 'WEEK4-2' containing files 'avgfunction.cpp', 'CMakeLists.txt', 'header1.h', and 'main.cpp'. The main editor displays the content of 'CMakeLists.txt':

```
1 cmake_minimum_required(VERSION 3.16.3)
2
3 project(CMakeExample)
4
5 set(CMAKE_CXX_STANDARD 11)
6
7 add_executable(CMakeExample main.cpp avgfunction.cpp)
```

The Terminal view at the bottom shows the command 'cmake --version' being executed in a terminal window, resulting in the output 'cmake version 3.16.3'. Below the terminal output, there is a note: 'CMake suite maintained and supported by Kitware (kitware.com/cmake).'

3. CMake

- Create a build folder by type in command : “mkdir build”
- Navigate to build folder.



The screenshot shows the Visual Studio Code interface. The Explorer view on the left shows a project folder named 'WEEK4-2' containing files 'avgfunction.cpp', 'CMakeLists.txt', 'header1.h', and 'main.cpp'. The main editor displays the content of 'CMakeLists.txt':

```
1 cmake_minimum_required(VERSION 3.16.3)
2
3 project(CMakeExample)
4
5 set(CMAKE_CXX_STANDARD 11)
6
7 add_executable(CMakeExample main.cpp avgfunction.cpp)
```

The Terminal view at the bottom shows the following commands and output:

```
ubuntu20@ubuntu:~/Documents/week4-2/build$ mkdir build
ubuntu20@ubuntu:~/Documents/week4-2/build$ ls
build          cmake_install.cmake
CMakeCache.txt Makefile
CMakeFiles     myprogram
ubuntu20@ubuntu:~/Documents/week4-2/build$ cd build
ubuntu20@ubuntu:~/Documents/week4-2/build/build$
```


3. CMake

- Type in the command : "cmake .."

The screenshot shows the Visual Studio Code interface with a CMakeLists.txt file open in the editor. The terminal window at the bottom shows the execution of the `cmake ..` command in the `~/Documents/week4-2/build` directory. The output indicates that the C and CXX compilers (GNU 9.4.0) are identified and working, and the build files have been generated.

```

CMakeLists.txt - week4-2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
WEEK4-2
  build
    CMakeFiles
    cmake_install.cmake
    CMakeCache.txt
    Makefile
    avgfunction.cpp
    CMakeLists.txt
    header1.h
    main.cpp
  main.cpp
  CMakeLists.txt
  avgfunction.cpp
  CMakeLists.txt
1 cmake_minimum_required(VERSION 3.16.3)
2
3 project(CMakeExample)
4
5 set(CMAKE_CXX_STANDARD 11)
6
7 add_executable(CMakeExample main.cpp avgfunction.cpp)

TERMINAL
bash-build
ubuntu20@ubuntu:~/Documents/week4-2/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: /home/ubuntu20/Documents/week4-2/build
ubuntu20@ubuntu:~/Documents/week4-2/build$
  
```

3. CMake

- Now, to generate executable file, type in the command : “make”.
- Executable file “CMakeExample” is generated.

CMakeLists.txt - week4-2 - Visual Studio Code

```

1 cmake_minimum_required(VERSION 3.16.3)
2
3 project(CMakeExample)
4
5 set(CMAKE_CXX_STANDARD 11)
6
7 add_executable(CMakeExample main.cpp avgfunction.cpp)

```

```

ubuntu20@ubuntu:~/Documents/week4-2/build$ make
Scanning dependencies of target CMakeExample
[ 33%] Building CXX object CMakeFiles/CMakeExample.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/CMakeExample.dir/avgfunction.cpp.o
[100%] Linking CXX executable CMakeExample
[100%] Built target CMakeExample
ubuntu20@ubuntu:~/Documents/week4-2/build$

```

CMakeLists.txt - week4-2 - Visual Studio Code

```

1 cmake_minimum_required(VERSION 3.16.3)
2
3 project(CMakeExample)
4
5 set(CMAKE_CXX_STANDARD 11)
6
7 add_executable(CMakeExample main.cpp avgfunction.cpp)

```

```

ubuntu20@ubuntu:~/Documents/week4-2/build$ make
Scanning dependencies of target CMakeExample
[ 33%] Building CXX object CMakeFiles/CMakeExample.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/CMakeExample.dir/avgfunction.cpp.o
[100%] Linking CXX executable CMakeExample
[100%] Built target CMakeExample
ubuntu20@ubuntu:~/Documents/week4-2/build$ ls
CMakeCache.txt  CMakeFiles      Makefile
CMakeExample    cmake_install.cmake
ubuntu20@ubuntu:~/Documents/week4-2/build$ ./CMakeExample
Results: 5.2
ubuntu20@ubuntu:~/Documents/week4-2/build$

```