Basics of Linux terminal and C++

using gcc, git, make and cmake

by Christos Gkantidis,
Software R&D Engineer at Mentor, a Siemens business

Instructor:

Giorgos Dimitrakopoulos

Overview

- **Preparation**
  - You can do it on Windows too!
  - Install the necessary tools for development
  - Choose an editor
  - Create a GitHub repository
- **Write our first program**
- **Refactoring 1: Move our function**
- **Refactoring 2: Create a header file**
- **Refactoring 3: Create a static library**
- **Add some automation with make**
- **Add extra automation with cmake**
- **Template functions**

You can do it on Windows too!

- You don't need a dual boot or a virtual machine to compile your code on the terminal
- Microsoft offers the “Windows Subsystem for Linux”
- WSL allows developers to run a GNU/Linux environment, directly on Windows, unmodified, without the overhead of a traditional virtual machine
- Head to https://docs.microsoft.com/en-us/windows/wsl/ and you will have a working terminal in less than 30 minutes

Install the necessary tools for development

- Assuming you have Ubuntu installed, either as your main OS or in WSL, use the following command in the terminal to install the necessary tools.
- In case of a different OS use the respective package manager for the installation

```
linux> sudo apt install git gcc make cmake
```

Choose an editor

- The first step to writing C++ code, is to choose an editor you are comfortable with, and that it offers good C++ syntax highlighting
- If you want to write your code and compile it in the terminal, choose vim
- If you are intimidated by the fact that vim does not offer a GUI and has an initial steep learning curve, choose Visual Studio Code
- Even a simple editor like Sublime Text 3 or Notepad++ should be enough to get you started

Choose an editor

- In case you opted for WSL, install the editor on the host OS so you can have a GUI, by following the instructions at the editors website
- In case you want to use vim or you have a full-fledged Linux OS, install your choice through the terminal

```
# installing vim
linux> sudo apt install vim

# installing visual studio code
linux> sudo apt install snap
linux> sudo snap install --classic code

# install sublime text 3
linux> wget -qO - https://download.sublimetext.com/sublimehq-pub.gpg | sudo
         apt-key add -
linux> sudo apt install apt-transport-https
linux> echo "deb https://download.sublimetext.com/ apt/stable/" | sudo tee
         /etc/apt/sources.list.d/sublime-text.list
linux> sudo apt update
linux> sudo apt install sublime-text
```

Choose an editor

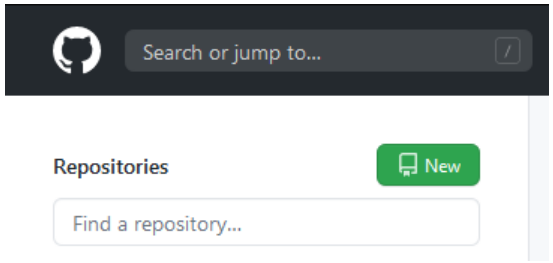
- Try to stay away from a full fledged IDE (Integrated Development Environment) for small projects (<1000 Lines of Code)
- They have a lot of options and will confuse you with their complexity
- They hide a lot of details behind the scenes, and you will never learn the underlying concepts of compiling your own code
- Most of them are not free
- Popular C++ IDEs: Visual Studio, Eclipse, CLion, NetBeans

Create a GitHub repository

- **Go to github.com and create an account if you don't have one already**
  - GitHub is a free, online repository host, which you can use to sync your project among many computers and collaborate with others
  - You can create public repositories, which are viewable by others, or private repositories that are viewable only by you
- **Other free and well-known repository hosts are: gitlab.com and bitbucket.org**

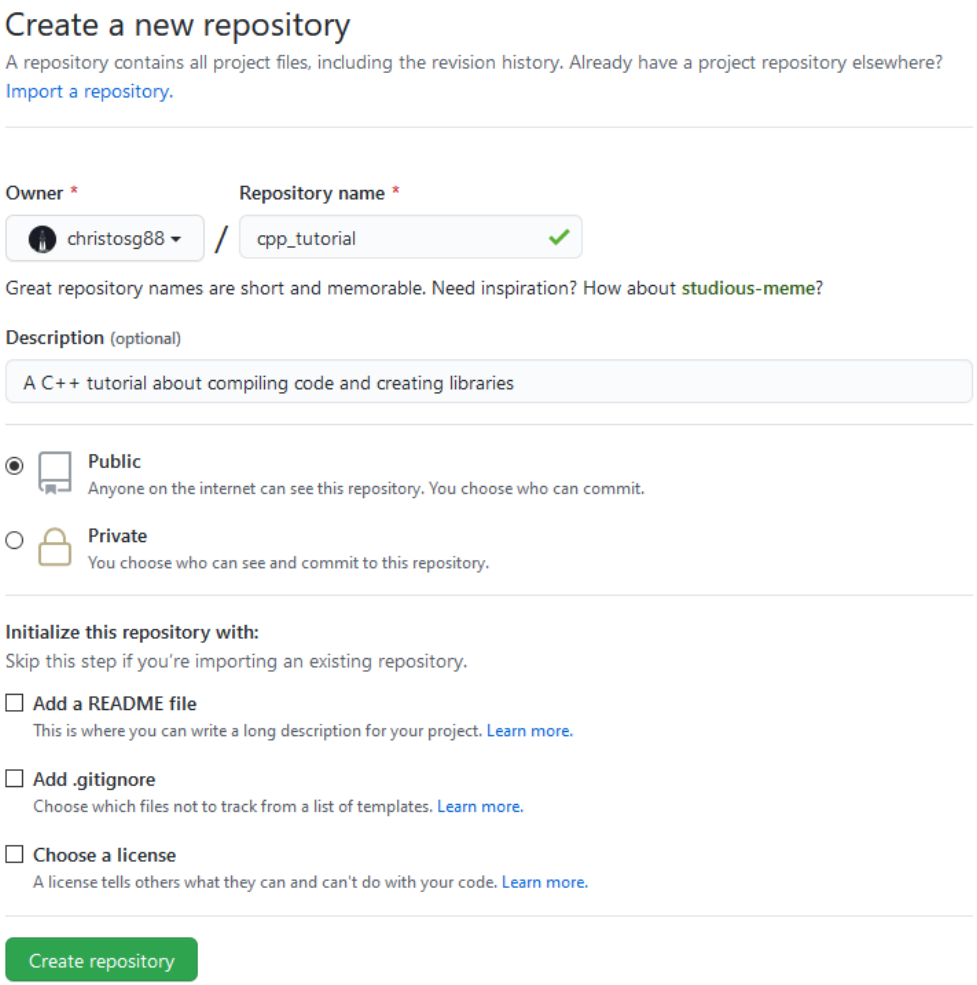
Create a GitHub repository

- Click on the green “New” button on the top left



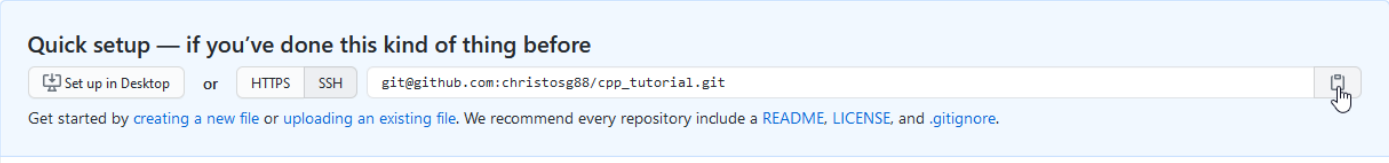
- Enter a name for your repo (something like “cpp_tutorial”)
- Optionally, add a small description of what your project is about. The description will show up next to your repo when people search for public repos
- You can also choose if you repository will be public or private
- We will use a public repo since we want to show our git-fu
- Finally click the green “Create repository” at the bottom

Create a GitHub repository



Create a GitHub repository

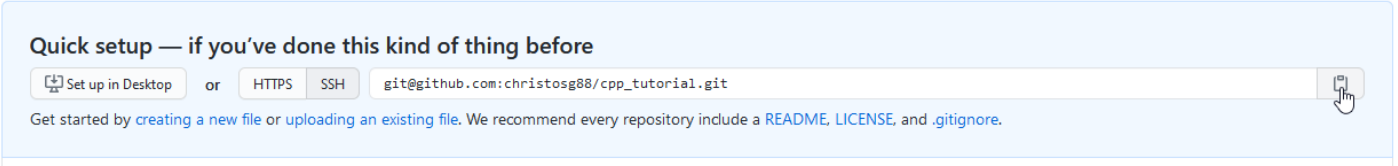
- After you get redirected to your new repo area, click on the clipboard icon to copy your repo's URL



- Enter a name for your repo (something like "cpp_tutorial")
- Optionally, add a small description of what your project is about. The description will show up next to your repo when people search for public repos
- You can also choose if you repository will be public or private
- We will use a public repo since we want to show our git-fu
- Finally click the green "Create repository" at the bottom

Create a GitHub repository

- After you get redirected to your new repo area, click on the clipboard icon to copy your repo's URL



- We need this URL to link our local repo with our online repo

Create a GitHub repository

- **Some git glossary for future reference:**
  - remote: an online repo which we can use to sync our project with other computers and collaborators
  - origin: this is a name chosen by convention, to refer to the main remote repo used for the project. You can have multiple remotes for a project, and not all of them need to be on the same repository host (e.g. you can sync your project both to GitHub and to GitLab)
  - commit: a commit is a snapshot of changes performed on the code
  - branch: a branch is a sequence of commits
  - master: this is the name of the main branch of your project. You can have multiple branches per project. The master branch should always have code that perform as expected with no surprises.
- **Learn more about git:**
  https://guides.github.com/introduction/git-handbook/

Overview

- **Preparation**
  - You can do it on Windows too!
  - Install the necessary tools for development
  - Choose an editor
  - Create a GitHub repository
- **Write our first program**
- **Refactoring 1: Move our function implementation**
- **Refactoring 2: Create a header file**
- **Refactoring 3: Create a static library**
- **Add some automation with make**
- **Add extra automation with cmake**
- **Template functions**

Write our first program

- We are going to write a small demo program, that will create two vectors and add them element-wise to produce a new vector
- We will start with a single source file, and abstract it step by step so that by the end we have a library which we can reuse in multiple projects
- We will call this project *cpp_tutorial*

Write our first program

- Create a new directory in your home directory to store your project and cd into it

```
linux> mkdir -p $HOME/workspace/cpp_tutorial/
linux> cd $HOME/workspace/cpp_tutorial/
```

- Initialize an empty git repository and link it to our GitHub repository

```
linux> git init
linux> git remote add origin <repo_url>
```

Write our first program

- **Create a file called main.cpp under src/**
- **As a convention, programmers prefer to store their source code in a directory called src/**
- **Other conventional directories inside the project directory are:**
  - bin/ where the executable binaries are stored
  - lib/ where the libraries are stored
  - include/ where the header files for the library APIs are stored
- **Create the remaining directories when needed**
- **Open the main.cpp file in your editor and fill in the code provided in the next slide**

Write our first program

```cpp
/// main.cpp
#include <iostream>
#include <vector>

/// return a vector that is the element-wise addition of left and right
std::vector<int> sum_vecs(std::vector<int> const &left, std::vector<int> const &right) {
  int size = left.size();
  std::vector<int> result(size);

  for (int i = 0; i < size; ++i) {
    result[i] = left[i] + right[i];
  }
  return result;
}

int main() {
  std::vector<int> vec1{1, 2, 3, 4, 5};
  std::vector<int> vec2{6, 7, 8, 9, 10};
  std::vector<int> vec3 = sum_vecs(vec1, vec2);

  // print the resulting vector
  int size = vec3.size();
  for (int i = 0; i < size; ++i) {
    std::cout << vec3[i] << " ";
  }
  std::cout << "\n";
  return 0;
}
```

Write our first program

- **In our main() function we create two vectors called vec1 and vec2**
- **Then, we call the function sum_vecs() and assign the result to a third vector called vec3**
  - The sum_vecs function expects two vectors of int elements, adds them element-wise and returns the result
- **Finally in main() we print the resulting vector element-by-element in one line**

Compile

- On the terminal, go into the project directory

```
linux> cd $HOME/workspace/cpp_tutorial
```

- Create the bin/ directory where we will put our executable

```
linux> mkdir bin/
```

- Run g++ to compile our source code into an executable

```
linux> g++ -o bin/main src/main.cpp
```

- If you copied the code correctly from the previous slide the compilation should complete without any messages
- Let's break down the compilation command on the next slide

Compile

- g++: this is the executable name of the GNU C++ compiler which we use to compile our source file
- -o bin/main: The -o option tells the compiler where to place the executable. In our case we tell the compiler to place the executable inside the bin/ directory and name it main
- src/main.cpp: this passes our source file as input to the compiler for compilation

Compile

- We can now run our executable

```
linux> ./bin/main
7 9 11 13 15
```

- As expected the result printed is the element-wise sum of the two vectors
- Since everything works as expected, let's commit our work so far in git

Commit

- Inside our project directory, execute the following

```
linux> git add src/main.cpp
```

- This will add the main.cpp file under src/, to the list of files that git will track for changes
  - Notice the we are not adding the executable we created, because it can be generated by compiling the source code
- Commit the change with a short message to describe it

```
linux> git commit -m 'Initial commit of summing two vectors'
```

- Push our commit to GitHub

```
linux> git push origin master
```

Overview

- **Preparation**
  - You can do it on Windows too!
  - Install the necessary tools for development
  - Choose an editor
  - Create a GitHub repository
- **Write our first program**
- **Refactoring 1: Move our function**
- **Refactoring 2: Create a header file**
- **Refactoring 3: Create a static library**
- **Add some automation with make**
- **Add extra automation with cmake**
- **Template functions**

Refactoring 1: Move our function

- If we want to use the sum_vecs() function to other projects, we would like to have access to our implementation without having to write it again and again
- In order to that, we will refactor our function into a library
- As a first step, we will move our function to a different source file
- Create a new file called vector_math.cpp under the src/ directory and open it in your editor

Refactoring 1: Move our function

```cpp
/// main.cpp
#include <iostream>
#include <vector>

/// function prototype
std::vector<int> sum_vecs(std::vector<int> const &left, std::vector<int> const &right);

int main() {
  std::vector<int> vec1{1, 2, 3, 4, 5};
  std::vector<int> vec2{6, 7, 8, 9, 10};
  std::vector<int> vec3 = sum_vecs(vec1, vec2);

  // print the resulting vector
  int size = vec3.size();
  for (int i = 0; i < size; ++i) {
    std::cout << vec3[i] << " ";
  }
  std::cout << "\n";
  return 0;
}
```

- We have removed the implementation of our function from main.cpp
- We still have to leave the prototype of our function, which is like a “promise” to the compiler that we are going to give the implementation somewhere else

Refactoring 1: Move our function

```cpp
/// vector_math.cpp
#include <vector>

/// return a vector that is the element-wise addition of left and right
std::vector<int> sum_vecs(std::vector<int> const &left, std::vector<int> const &right) {
  int size = left.size();
  std::vector<int> result(size);

  for (int i = 0; i < size; ++i) {
    result[i] = left[i] + right[i];
  }
  return result;
}
```

- We have placed the implementation of our function in vector_math.cpp
- Notice that the function prototype in main.cpp must match exactly the function signature of sum_vecs() in vector_math.cpp

Compile

- On the terminal, inside the project directory, run g++ to compile our refactored code

```
linux> g++ -o bin/main src/main.cpp src/vector_math.cpp
```

- Notice that everything is like our previous compilation, except that now we have to also pass as input to g++ our second source file which contains our function
- If you copied the code correctly you will see no messages displayed
- We can again run our program and get the same result as before

```
linux> ./bin/main
7 9 11 13 15
```

Commit

- Inside our project directory, execute the following

```
linux> git add src/main.cpp src/vector_math.cpp
```

- This will record the changes in main.cpp and vector_math.cpp under src/
- Commit the change with a short message to describe it

```
linux> git commit -m 'Refactor sum_vecs() into src/vector_math.cpp'
```

- Push our commit to GitHub

```
linux> git push origin master
```

Overview

- **Preparation**
  - You can do it on Windows too!
  - Install the necessary tools for development
  - Choose an editor
  - Create a GitHub repository
- **Write our first program**
- **Refactoring 1: Move our function**
- **Refactoring 2: Create a header file**
- **Refactoring 3: Create a static library**
- **Add some automation with make**
- **Add extra automation with cmake**
- **Template functions**

Refactoring 2: Create a header file

- **What if our library has multiple functions? Do we have to add a function prototype for all of them in main.cpp?**
- **Thankfully not, and this leads us to our second refactoring**
- **We will put all our function prototypes in a new file called vector_math.h under the include/ directory**
- **This file will contain all our function prototypes from vector_math.cpp and will serve two purposes**
  - It will be the API of our library, informing all users of what functions are available in our library and how to use them
  - The users of our library don't have to write the function prototype above the code when they want to use it

Refactoring 2: Create a header file

```cpp
/// main.cpp
#include "vector_math.h"

#include <iostream>
#include <vector>

int main() {
  std::vector<int> vec1{1, 2, 3, 4, 5};
  std::vector<int> vec2{6, 7, 8, 9, 10};
  std::vector<int> vec3 = sum_vecs(vec1, vec2);

  // print the resulting vector
  int size = vec3.size();
  for (int i = 0; i < size; ++i) {
    std::cout << vec3[i] << " ";
  }
  std::cout << "\n";
  return 0;
}
```

- **We have removed the function prototype, and replaced it with a directive to include the vector_math.h header file, which contains all the function prototypes of the library we want to use**

Refactoring 2: Create a header file

```
/// vector_math.h
#include <vector>

/// return a vector that is the element-wise addition of left and right
std::vector<int> sum_vecs(std::vector<int> const &left, std::vector<int> const &right);
```

```
/// vector_math.cpp
#include "vector_math.h"

std::vector<int> sum_vecs(std::vector<int> const &left, std::vector<int> const &right) {
  int size = left.size();
  std::vector<int> result(size);

  for (int i = 0; i < size; ++i) {
    result[i] = left[i] + right[i];
  }
  return result;
}
```

- Notice that vector_math.h contains only the prototype for our function
- vector_math.cpp must also include its own header file

Compile

- On the terminal, inside the project directory, run g++ to compile our refactored code

```
linux> g++ -I include/ -o bin/main src/main.cpp src/vector_math.cpp
```

- Notice that everything is like our previous compilation, except that now we also pass a `-I` option
  - The `-I` option tells the compiler where to search for header files that are included by our source files with `#include "file.h"`
- If you copied the code correctly you will see no messages displayed
- We can again run our program and get the same result as before

```
linux> ./bin/main
7 9 11 13 15
```

Commit

- Inside our project directory, execute the following

```
linux> git add src/main.cpp src/vector_math.cpp include/vector_math.h
```

- This will record the changes in main.cpp and vector_math.cpp under src/ and vector_math.h under include/
- Commit the change with a short message to describe it

```
linux> git commit -m 'Refactor function prototypes in vector_math.h'
```

- Push our commit to GitHub

```
linux> git push origin master
```

Overview

- **Preparation**
  - You can do it on Windows too!
  - Install the necessary tools for development
  - Choose an editor
  - Create a GitHub repository
- **Write our first program**
- **Refactoring 1: Move our function**
- **Refactoring 2: Create a header file**
- **Refactoring 3: Create a static library**
- **Add some automation with make**
- **Add extra automation with cmake**
- **Template functions**

Refactoring 3: Create a static library

- What if our library is implemented in multiple source files and not just src/vector_math.cpp? Do we have to pass all of the source files to the compiler?
- Thankfully not, and this leads us to our third refactoring
- We will create an archive, aka a static library, which other projects can link to their executable, without having to pass all source files to the compiler
- To better show this we will create a new function called print_vector() in a file called src/vector_helpers.cpp

Refactoring 3: Create a static library

- First, add the prototype of the new function in include/vector_math.h

```
/// vector_math.h
#include <vector>

/// return a vector that is the element-wise addition of left and right
std::vector<int> sum_vecs(std::vector<int> const &left, std::vector<int> const &right);

/// print a vector element by element in one line
void print_vec(std::vector<int> const &vec);
```

Refactoring 3: Create a static library

- Then create a new file under src/ called vector_helpers.cpp and add the implementation of the print_vec() function

```
/// vector_helpers.cpp
#include "vector_math.h“
#include <iostream>

void print_vec(std::vector<int> const &vec) {
  int size = vec.size();
  for (int i = 0; i < size; ++i) {
    std::cout <<vec[i] << " ";
  }
  std::cout << "\n";
}
```

- Notice that we again need to include the header file with the prototype of this function

Refactoring 3: Create a static library

- **Now update main.cpp to make use of the new function**

```cpp
/// main.cpp
#include "vector_math.h"
#include <vector>

int main() {
  std::vector<int> vec1{1, 2, 3, 4, 5};
  std::vector<int> vec2{6, 7, 8, 9, 10};
  std::vector<int> vec3 = sum_vecs(vec1, vec2);

  // print the resulting vector
  print_vec(vec3);

  return 0;
}
```

Compile

- Make sure that our program still compiles as we have done so far
- On the terminal, inside the project directory, run g++ to compile our refactored code

```
linux> g++ -I include/ -o bin/main src/main.cpp src/vector_math.cpp
         src/vector_helpers.cpp
```

- If you copied the code correctly you will see no messages displayed
- We can again run our program and get the same result as before

```
linux> ./bin/main
7 9 11 13 15
```

Compile

- To create an archive of our library, we must first compile our library source code, into object files
- First, from inside the project directory, create a lib/ directory to place the library

```
linux> mkdir lib/
```

- Now compile the library's source files into object files (.o)

```
linux> g++ -I include/ -c -o lib/vector_math.o src/vector_math.cpp
linux> g++ -I include/ -c -o lib/vector_helpers.o src/vector_helpers.cpp
```

- Notice that the compilation commands look like what we have been doing so far, with two differences
  - In the -o option, we tell the compiler to place the files in lib/, and we specify the extension .o to denote they are object files
  - We pass the -c option which tells the compiler to compile the source files, but to stop before the linking step

Compile

- Now create an archive (think of it like a package) of the object files we created, using the ar tool

```
linux> ar rs lib/libvector_math.a lib/vector_math.o lib/vector_helpers.o
```

- This will create a static library called libvector_math.a under lib/, which contains the object files vector_math.o and vector_helpers.o under lib/
  - The r option tells ar to insert the member object files in the archive, and replace any object files already in there
  - The s option tells ar to add an index to the archive along with the object files

Compile

- Now compile the executable linking the static library we created like so

```
linux> g++ -I include/ -o bin/main src/main.cpp lib/libvector_math.a
```

- or like so

```
linux> g++ -I include/ -o bin/main src/main.cpp -L lib/ -l vector_math
```

- On the first way we just have to pass our archive as another input file to g++, which will use it at the linking stage
- On the second way, we pass two extra options
  - -L tells the compiler where to search for the static libraries we will try to link
  - -l tells the compiler to link the vector_math library
    - Notice that we don't include the lib and the .a parts in this way, the compiler knows to add them itself

Compile

- If you copied the code correctly you will see no messages displayed
- We can again run our program and get the same result as before

```
linux> ./bin/main
7 9 11 13 15
```

Commit

- Inside our project directory, execute the following

```
linux> git add src/main.cpp src/vector_math.cpp src/vector_helpers.cpp
         include/vector_math.h
```

- This will record the changes in main.cpp, vector_math.cpp and vector_helpers.cpp under src/ and vector_math.h under include/
- Commit the change with a short message to describe it

```
linux> git commit -m 'Add vector_helpers.cpp that supports vector printing'
```

- Push our commit to GitHub

```
linux> git push origin master
```

Overview

- **Preparation**
  - You can do it on Windows too!
  - Install the necessary tools for development
  - Choose an editor
  - Create a GitHub repository
- **Write our first program**
- **Refactoring 1: Move our function**
- **Refactoring 2: Create a header file**
- **Refactoring 3: Create a static library**
- **Add some automation with make**
- **Add extra automation with cmake**
- **Template functions**

Add some automation with make

- It's hard to remember and type each time all the options needed to compile our library and our executable
- To automate this process we can use the make program
- The make program reads a file named Makefile, which contains targets and their dependencies, and executes the necessary commands to create the targets for us
- Inside the project directory, create a file named Makefile, open it in your editor and add the contents of the following slide

Add some automation with make

```
# Makefile

bin/main : src/main.cpp lib/libvector_math.a Makefile
    g++ -I include/ -o bin/main src/main.cpp -L lib/ -l vector_math

lib/libvector_math.a : lib/vector_math.o lib/vector_helpers.o Makefile
    ar rs lib/libvector_math.a lib/vector_math.o lib/vector_helpers.o

lib/vector_math.o : src/vector_math.cpp Makefile
    g++ -I include/ -c -o lib/vector_math.o src/vector_math.cpp

lib/vector_helpers.o : src/vector_helpers.cpp Makefile
    g++ -I include/ -c -o lib/vector_helpers.o src/vector_helpers.cpp

.PHONY: clean
clean:
    rm -f bin/main lib/vector_math lib/vector_helpers.o lib/libvector_math.a
```

- The filename before a colon is called a target
- The filenames after the colon are the dependencies of this target
- The indented lines bellow a target is called the recipe for the target and can constitute of multiple commands

Add some automation with make

- The recipe for a target is run only if any of the dependencies is newer than the target, otherwise the target is considered up-to-date
- Imagine that you have compiled your project, and then you add something more in src/main.cpp
- Since src/main.cpp is not a dependency in any of our library files, the source files of our library will not get compiled and the library will not be re-archived
- This is very helpful and can save a lot of compilation time in big projects

Compile

- To use the Makefile we wrote execute the following

```
linux> make bin/main
```

- or equivalently

```
linux> make
```

- Since bin/main is the first target in our Makefile, it's considered the default target, and you don't need to specify its name
- If you copied the code correctly you will see the recipes executed by make
- We can again run our program and get the same result as before

```
linux> ./bin/main
7 9 11 13 15
```

Commit

- Inside our project directory, execute the following

```
linux> git add Makefile
```

- This will record the changes in Makefile
- Notice that we don't need to add any of the other files, since we didn't change any of them
- Commit the change with a short message to describe it

```
linux> git commit -m 'Add Makefile for automation'
```

- Push our commit to GitHub

```
linux> git push origin master
```

Overview

- **Preparation**
  - You can do it on Windows too!
  - Install the necessary tools for development
  - Choose an editor
  - Create a GitHub repository
- **Write our first program**
- **Refactoring 1: Move our function**
- **Refactoring 2: Create a header file**
- **Refactoring 3: Create a static library**
- **Add some automation with make**
- **Add extra automation with cmake**
- **Template functions**

Add extra automation with cmake

- For large projects it might be tedious to write Makefiles to compile all source files
- To help ourselves, we can use cmake, which is a Makefile generator
- So instead of writing the Makefiles ourselves, we just describe to cmake the dependencies between our files, and it generates the necessary Makefiles
- The description to cmake must be in a file called CMakeLists.txt in our project directory, so create it and open it in your editor

Add extra automation with cmake

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.17)

project(cpp_tutorial)

add_library(vector_math src/vector_math.cpp src/vector_helpers.cpp)
target_include_directories(vector_math PRIVATE include/)

add_executable(main src/main.cpp)
target_include_directories(main PRIVATE include/)
target_link_libraries(main vector_math)

install(TARGETS main vector_math
  RUNTIME DESTINATION ${CMAKE_SOURCE_DIR}/bin
  ARCHIVE DESTINATION ${CMAKE_SOURCE_DIR}/lib
)
```

- **To produce the Makefiles, create a new directory inside our project directory called build/ and cd into it**

```
linux> mkdir build
linux> cd build
```

Compile

- Now call cmake passing as input the path where the CMakeLists.txt file is stored

```
linux> cmake ..
```

- This will show some information messages about the process of cmake
- At the end a Makefile will have been created in the current directory
- Now we run make as before

```
linux> make
```

- You will notice that the output of the Makefile generated by cmake is much more informative and colorful than the one we wrote by hand

Compile

- As a lest step, in order to place the executable inside bin/ and the library inside lib/ run the following

```
linux> make install
```

- cd back into the root of the project directory
- We can again run our program and get the same result as before

```
linux> ./bin/main
7 9 11 13 15
```

- We can now remove the Makefile we wrote by hand since we will be using cmake to build our code

```
linux> rm Makefile
```

Commit

- Inside our project directory, execute the following

```
linux> git add CMakeLists.txt Makefile
```

- This will record the addition of CMakeLists.txt and the removal of Makefile
- Commit the change with a short message to describe it

```
linux> git commit -m 'Replace Makefile with CMakeLists.txt'
```

- Push our commit to GitHub

```
linux> git push origin master
```

Overview

- **Preparation**
  - You can do it on Windows too!
  - Install the necessary tools for development
  - Choose an editor
  - Create a GitHub repository
- **Write our first program**
- **Refactoring 1: Move our function**
- **Refactoring 2: Create a header file**
- **Refactoring 3: Create a static library**
- **Add some automation with make**
- **Add extra automation with cmake**
- **Template functions**

Template functions

- Let's go back to the beginning
- We created the sum_vecs() and print_vec() function to accept vectors that contain elements of type int
- What if we wanted our library to support vectors of all types that support the operator +?
- To achieve that we will use template functions

Template functions

- When template functions are used, they expect an additional argument, which is the type of the objects they will manipulate
- Because we want the user to be able to use our library with whatever data types support the operator +, and because the compiler needs to know the implementation of the functions in order to substitute the user defined type, we can no longer create a static library, and our library has to be “header only”
- Change your include/vector_math.h as follows

Template functions

```
/// vector_math.h
#include <vector>
#include <iostream>

/// return a vector that is the element-wise addition of left and right
template<typename T>
std::vector<T> sum_vecs(std::vector<T> const &left, std::vector<T> const &right) {
  int size = left.size();
  std::vector<T> result(size);

  for (int i = 0; i < size; ++i) {
    result[i] = left[i] + right[i];
  }
  return result;
}

/// print a vector element by element in one line
template<typename T>
void print_vec(std::vector<T> const &vec) {
  int size = vec.size();
  for (int i = 0; i < size; ++i) {
    std::cout << vec[i] << " ";
  }
  std::cout << "\n";
}
```

Template functions

- We also need to update our src/main.cpp as follows to provide the type to the function templates

```cpp
/// main.cpp
#include "vector_math.h"
#include <vector>

int main() {
  std::vector<int> vec1{1, 2, 3, 4, 5};
  std::vector<int> vec2{6, 7, 8, 9, 10};
  std::vector<int> vec3 = sum_vecs<int>(vec1, vec2);

  // print the resulting vector
  print_vec<int>(vec3);

  return 0;
}
```

Template functions

- Also, since we are no longer compiling a static library, we must update our CMakeLists.txt

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.17)

project(cpp_tutorial)

add_executable(main src/main.cpp)
target_include_directories(main PRIVATE include/)
target_link_libraries(main vector_math)

install(TARGETS main
  RUNTIME DESTINATION ${CMAKE_SOURCE_DIR}/bin
)
```

- We can also remove the no-longer-used cpp files of our library

```
linux> rm src/vector_math.cpp src/vector_helpers.cpp
```

Compile

```
linux> cd build
linux> cmake ..
```

- At the end a new Makefile will have been created in the current directory
- Now we run make as before

```
linux> make
linux> make install
```

- cd back into the root of the project directory
- We can again run our program and get the same result as before

```
linux> cd ..
linux> ./bin/main
7 9 11 13 15
```

Commit

- Inside our project directory, execute the following

```
linux> git add -u
```

- This will record all the changes to the files git is already tracking, and since we didn't add any new files this makes our job much easier
- Commit the change with a short message to describe it

```
linux> git commit -m 'Refactor our library to a template, header-only one'
```

- Push our commit to GitHub

```
linux> git push origin master
```

THE END