

# Mobile Robot Control exercises tools

During the Mobile Robot Control (MRC) course you will encounter many tools, systems and concepts that you are currently unfamiliar with. This may be daunting at first, but soon you will notice the strengths of each of the tools and find out how they work together to allow you to program a real, physical robot. The following tutorials are aimed at getting you up speed with these tools as fast as possible.

Let's start with an overview of the tools we will be using and the roles they play within your project:

- Ubuntu: the Operating System we will be using. Ubuntu is a popular Linux distribution.
- C++: is the programming language we will be using. This means that your program, or code, will be written in C++.
- git: is a software versioning and revision control system. You will use it to share your project code between different group members, while maintaining a file version history. Think of it as Dropbox.
- VSCode: an Integrated Development Environment (IDE) for C++. All you need to create a C++ program is a simple text editor and a C++ compiler. However, it can become difficult to manage large projects, trace back where compile errors are coming from, etc. Think of VSCode as a very advanced text editor that understands C++ and makes programming C++ a lot nicer.

Alright, let's get our hands dirty. Time to install Ubuntu on your computer.

## Installation

### Installing VirtualBox

Ubuntu is an operating system, much like windows and MacOS. It is what makes a computer usable. Running a different operating system is like using a different computer altogether. In this course we recommend using a virtual machine to run ubuntu.

On your laptop, download virtualbox from their website:

<https://www.virtualbox.org/wiki/Downloads>

run the installer, and follow the instructions.

Virtualbox is the program that will create and run our virtual machine. A virtual machine is best seen as a piece of software that behaves as a virtual computer, which will thus allow us to install (and use) Ubuntu within windows.

After installing virtualbox, make sure you download a copy of the desktop image of Ubuntu to a folder on your windows machine:

<https://releases.ubuntu.com/20.04/>

## Setting up your virtual machine

Start creating your VM by opening virtualbox and selecting the blue star-like icon labeled "New".

1. In the menu that pops up you will see four tabs. Under the "Name and Operating System" you can give your VM a descriptive name. You can leave the Machine Folder at its default value. Select the ubuntu image you downloaded in the previous step. Make sure you select the proper type and version in the dropdowns below. The Type should be set to Linux, the Version should be set to Ubuntu (64-bit). Also check the box to skip unattended installation.
2. We dont need to do anything in the tab "Unattended Install", after all we chose to skip it.
3. In the tab "Hardware" you can choose the Memory Size. Select the amount of memory available to your VM. Make sure you select a value greater than 4096 MB. While making your selection, make sure you stay within the green boundary indicated underneath the slider. Select the amount of processors available to your VM, depending on the hardware of your PC. Generally, more is better.
4. In the tab "Hard Disk", select "Create a virutal Hard Disk Now". Select the size and location of your virtual hard disk. It is recommended to allocate at least 40 GB, however, more is generally better. Make sure you have enough disk space at the specified location. You may want to move your virtual disk to a different physical drive. Select VDI. for the File Type.
5. Click finish.

To finish the configuration of your VM take the following steps:

1. Right click on your VM
2. Select settings, indicated by an orange cog, in the context menu
3. Go to Display in the sidebar
4. Set the amount of Video Memory to (at least) 64 MB
5. Go to Network in the sidebar
6. set "Attached to" to Bridged Adapter. Under Name select your wireless connection. e.g. Intel(R) Wi-Fi 6 AX200 160MHz

Confirm your changes by clicking OK.

## Installing Ubuntu

Start your VM by clicking Start. The VM will boot to the installation screen. Make sure you follow the instructions to install Ubuntu.

### Erase disk ▾

When prompted you can select Erase disk and install Ubuntu without affecting your windows install since you are working on a virtual disk.

## Post-install configuration

When you are done installing Ubuntu, you reach the Ubuntu login screen and will be prompted to type in your password. Login to reach the Ubuntu desktop.

After login you will be prompted to perform some final configuration, like logging in to your Online Accounts. Configure these settings if you like, you can also skip them.


When prompted to upgrade to a newer version of Ubuntu. Choose **Don't Upgrade**.

When prompted to install updated software, choose "Install Now". This can also be done later by typing the following command into the terminal:

```
sudo apt-get update && sudo apt-get upgrade
```

After configuration, we need to do a few final steps:

1. In the taskbar of VirtualBox click devices
2. Click Insert Guest Additions CD image
3. On the menu that appears within Ubuntu click Run, and type your password.
4. Reboot your virtual machine.
5. Set the size of the window of your virtual machine by right clicking on the desktop and selecting display options. Choose the resolution that matches your monitor.

 **sometimes the approach above fails at step 3. In that case type the following command in a terminal and press enter. You can skip step 3** ▾

```
sudo apt-get install virtualbox-guest-additions-iso
```

You've now successfully installed Ubuntu. To check that everything has gone according to plan, go to your desktop. When you haven't changed your wallpaper, it should show you a picture that looks like a leopard. If you see a picture of a jellyfish or a beaver (or anything else) you've installed the incorrect version. We ask you kindly to reinstall the correct version of ubuntu (20.04).

# The terminal

We will now introduce you a tool called the terminal. On Ubuntu press ( ctrl + alt + t ). Do you feel like a hacker already? This is the terminal, it allows you to interact with your computer via text. For example try typing

```
ls
```

It will show you all the folders in your current directory, including the folders *Desktop Documents Downloads Music Pictures Public Templates Videos*

Lets make a folder for your course files:

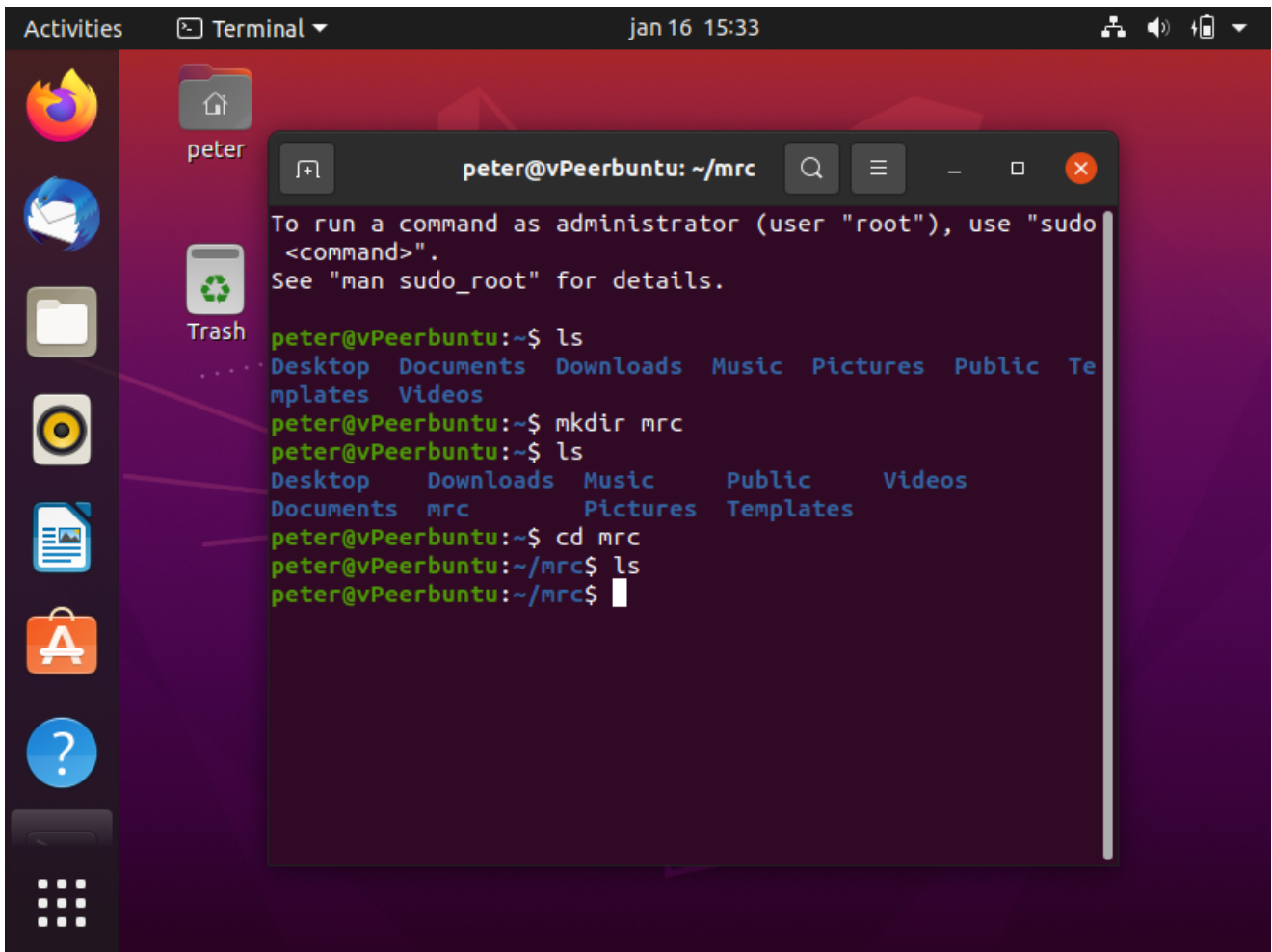
```
mkdir mrc
```

mkdir stands for "make directory". We are telling the computer to make a directory called "mrc". To check that it has indeed been created use `ls`

To navigate to this new directory we use the command `cd` which stands for "change directory". In the terminal type:

```
cd mrc
```

You will notice that the prompt has changed from `<username>@<computername>:~$` to `<username>@<computername>:~/mrc$`. This indicates the current directory. `~` is short for `/home/<current user>`.



## Terminal Colors

By default, the terminal in Ubuntu only uses white letters on a dark background. However, it can be quite convenient to allow the use of multiple text colors. For example, folders, files and executables are then displayed in different colors when using the command `ls`.

To enable text color in a terminal:

1. Open the file `.bashrc` in your homefolder, for example using: `gedit ~/.bashrc` in a terminal.
2. Uncomment the following line: `#force_color_prompt=yes` by removing the `#`.
3. Save and exit. Now if you start a new terminal, you will have a colored prompt.

## Terminator

You will soon find out that you will have to work in multiple terminals in parallel. A convenient tool to avoid having a large amount of terminals is Terminator, a program that allows you to have multiple terminals in one window. You can install it via the Ubuntu software center, or from the terminal:

```
sudo apt-get install terminator
```

Now close the terminal and open a new one. What is this!? It looks like a terminal but it is a little different. press ( ctrl + shift + e ). The terminal splits vertically! Wow, try it again. (ctrl + shift + e), and it splits into more. You now have three terminals to work with.

You can move between them with your mouse or by pressing ( alt + <arrow\_key> ). Your active window is highlighted red. That is the one you will type into.

But our terminals are getting quite narrow, it would be nice if we could also split them horizontally. Try selecting the largest terminal and pressing ( ctrl + shift + o )

When you need many terminals at once Terminator can help you to keep them organized.

## installing the EMC environment

 emc? ▾

the mrc course used to be called emc, that is why the software framework we use is still called emc

To get you up speed as soon as possible, we created a nice little installer script that will set-up your computer in no-time. It will download, install and compile all necessary software and if all goes well, you can start using our robot simulator within 30 minutes. Just do the following:

1. Open a terminal
2. Download the install script:

```
wget https://raw.githubusercontent.com/tue-robotics/emc-env/master/install.bash
```

3. And run it

```
source install.bash
```

That's it! The installer may ask for confirmation a few times because it has to install some programs system wide. Just enter your password, and you'll be good to go!

## Using the simulator

During the course, we have many groups and only one robot, so test time on the robot is scarce. Fortunately, we have a virtual (software) representation of the robot that can be used to simulate the robot. At the moment it:

- Simulates the movement of the robot
- Simulates the laser data of the robot, created by the virtual environment

- Provides a simple visualization

This should already be enough to get you started, and more will be added later (odometry, moving doors, better dynamics, etc).

## Updating the simulator

Before you start working with the simulator, make sure you have our latest version by running

```
mrc-update
```

This will download the latest changes and compile the updated software (framework and simulator). We will notify you when we made changes to the software such that you can run the updater, but feel free to run the command whenever you want.

### A word on ROS ▾

As was already stated before, you will not be using ROS in this course, unless you really want to use it yourself. However, secretly the provided tools are build on top of that; the inter-process communication to be more specific. Don't worry about it too much. Enough about ROS, let's start the simulator!

## Starting the simulator

Open a terminal and run

```
mrc-sim
```

That's it! The simulator has now been started, although we can't see it yet.

## Visualization

To visualize the simulator and our robot just run

```
sim-rviz
```

This pops up a visualization window showing the robot in the virtual world. This shows how the world is (at least in simulation...), your robot will not always have access to this information.

We can also see how the robot perceives the world through its sensors. You can see the LRF data projected in the world and you can probably guess what surface each points

reflects. See how the laser data changes even if the robot is standing still: this is the simulated noise added to the data.

## Controlling the robot

Now, as you can see, not much is happening. The robot is standing still in a static environment. Let's change that! A first simple way to test the simulator is by controlling the robot using your keyboard. Just run:

```
mrc-teleop
```

and you will be able to move the robot forward and backward with 'w' and 's', you can rotate the robot using 'a' and 'd' and you can stop movement with any other key.

The simulator is not an exact match for reality. For example, try driving your robot through a wall.

Try driving the robot around using 'mrc-teleop' and notice how the laser data changes, and how it differs from the actual virtual world.

## Stopping the simulator and visualization

You can stop the simulator and visualization by pressing (ctrl + C) in the terminal. (ctrl + C) is how you stop most processes in a terminal, also your own code.

## Loading a custom heightmap

By default, the simulator loads an indoor environment. Fortunately, you can change the simulation environment very easily! You just have to create a heightmap: an image that specifies for each pixel how 'high' the world has to be at that point. Since the laser only scans at one height, we can use two extremes here: flat (ground level) or high enough to be detected by the laser.

To create an image, simply open your favourite image editor and create a black-and-white image. If you don't know how to create an image, have a look at the section below. White corresponds to the floor, black to the walls (i.e., which will be detected by the laser). You have to keep the following things in mind:

The robot always starts in the center of the image. So, if you want to robot to start at the edge of your maze / corridor, just create a bigger image and move the black pixels to the upper part of the image.

You'll get the best result if the lines drawn are at least 2 pixels wide

Store your image lossless, i.e. using the png format (which is recommended by the way!), instead of the jpg format.

We will also need to create a file with meta information. An example is shown below



```
image: <YOUR_IMAGE_FILE>.png
resolution: 0.025
origin: [0.0, 0.0, -1.570796]
occupied_thresh: 0.9
free_thresh: 0.1
negate: 0
```

Here we specify how the computer should interpret our heightmap. This is a .yaml file with the following fields.

- image : Path to the image file containing the occupancy data; can be absolute, or relative to the location of the YAML file
- resolution : Resolution of the map, meters / pixel
- origin : The 2-D pose of the lower-left pixel in the map, as (x, y, yaw), with yaw as counterclockwise rotation (yaw=0 means no rotation). This parameter only affects visualization.
- occupied\_thresh, free\_thresh, negate: these parameters are currently unused by the simulator but they are part of the standard map metadata.

Once you have created an image, simply run the simulator and provide the yaml file as argument:

```
mrc-sim --map ./relative/path/to/<YOUR_METADATA_FILE>.yaml
```

That's it!

## Create a heightmap image

There are many linux applications that you use to create images. We suggest using Gimp, an open-source alternative to Photoshop. Although it might be a bit overkill to use for our application, it has great support and the thing we want to do (draw black lines on a white background) isn't hard. To install Gimp, run:

```
sudo apt-get install gimp
```

And run it using:

```
gimp
```

Then to create a simple image:

- Select File -> New (or ctrl-N)

- Specify the size of your image. Remember, 40 pixels = 1 meter, and the robot starts in the center
- Select the Pencil Tool. (left click on the paintbrush tool, or press 'N' on your keyboard)
- Set the pencil size in the lower left pane to something sensible, e.g. 2 pixels

Now, if you click left on the image, a dot is drawn, but we want lines! To draw a line:

- left click, then hold shift, then left click again.
- While holding the shift button, you can click more times to create a sequence of lines
- To create nice, axis-aligned lines, also hold ctrl

There is two types of saving in Gimp. The first one is the using File -> Save. However, this will only generate an xcf-file, something that can only be read by Gimp. Instead, you should use the File -> Export option:

- File -> Export
- Provide a name for your file. If you put '.png' as extension, it will be saved as png
- Use the default png export settings

That's it! Don't forget to make a yaml file for your map.

## Adding clutter objects and simulating wheelslip

In the real world, the odometry that is returned by the robot is not perfect and will have a certain amount of drift. Furthermore, the internal representation of the map never matches 100% with the real world. To add these discrepancies to the simulator (which are disabled by default), a JSON config file can be supplied. In this file it is easy to specify an array of objects that will be added to the world and to enable wheelslip. An example of this file is shown below:

```
{
  "uncertain_odom":true,
  "objects":[
    {
      "length": 1.0,
      "width": 1.0,
      "trigger_radius": 1.0,
      "repeat": false,
      "velocity": 0.6,
      "init_pose": [-1.0, 0.0, 0.0],
      "final_pose": [-1.0, -2.0, 0.0]
    },
    {
      "length": 0.2,
      "width": 0.4,
      "trigger_radius": 1.0,
```

```
    "repeat": true,  
    "velocity": 1.3,  
    "init_pose": [1.4, 0.0, 0.0],  
    "final_pose": [1.4, 1.0, 0.0]  
  }  
]  
}
```

In this file, the "uncertain\_odom" is enabled. Furthermore, two objects are added, which will start to move from their "init\_pose" to (approximately) their "final\_pose", supplied in (x,y,theta) coordinates. This movement is triggered when the robot comes within the "trigger\_radius" of the object, and the second object will keep repeating its movement. To supply such a file to the simulator, use the following argument:

```
mrc-sim --config ./relative/path/to/<YOUR_CONFIG_FILE>.json
```

or

```
mrc-sim --map ./relative/path/to/<YOUR_METADATA_FILE>.yaml --config  
./relative/path/to/<YOUR_CONFIG_FILE>.json
```

The wheelslip is simulated by random sampling a slip factor every few seconds, making the simulator stochastic. The result is that, just like with the real robot, no two trials will result in exactly the same robot position and odometry.

## Adding doors

The real environment might contain a door that the robot needs to open. The simulator is capable of simulating these doors, such that you can test your software before it gets real! To add doors to the simulated world, simply edit your heightmap and add grey lines to it. To be specific:

- The door should be a straight line (but not necessarily axis-aligned), with a minimum thickness of 2 pixels. To be more specific, all pixels in the door line should be connected
- The average RGB value should be between 25 and 230 (on a scale of 0-255). So for example, the RGB value (100, 100, 100) is a door, while (20, 20, 20) is a wall (and (255, 255, 255) is open space).
- You can have multiple doors in your world. However, make sure that they are always separated by at least one pixel, otherwise the simulator will interpret them as one big door.

The doors should be visible as green blocks in the simulator. Now, you can use the `emc::IO` object from your code to open these doors (but only if you are standing in front). You will

learn more about the `emc::IO` object later in these tutorials.

```
// ... Make sure robot is in front of a possible door
io.sendRequestOpenDoor();
```

To manually open a door in the simulator open a terminal and type

```
mrc-open-door
```

## Hello world

### Setting up your project

Of course, we not only want to use software during this course, but we want to create some! Let's start off with a simple example project. Go inside the `~/mrc` directory, and create a new folder with the name `my_project`:

```
cd ~/mrc
mkdir my_project
```

Often, the code files are not put directly in the root of a folder, but in a directory called *src*. This stands for "source", and is called this way because the files in there are the source of the compilation process, and are converted into binaries (files that are no longer human-readable, but are understandable for the machine). So, let's go. Remember that when using `cd` (and many other commands in linux) you can use tab-completion to type quicker, i.e, try:

```
cd m<<< now push the TAB key >>>
```

You will see that the terminal fills out the rest, because `my_project` is the only directory in the current directory that starts with an `m`. Ok, create the `src` directory, and go inside:

```
mkdir src
cd src
```

Finally, let's do some programming! You should already be familiar with the C++ language, so you know how to create a basic C++ program. Let us do it now. Open your favourite editor to create a file called `example.cpp` (e.g. `gedit example.cpp`) and put some code inside:

```
#include <iostream>

int main()
{
```

```
std::cout << "Hello world!" << std::endl;
return 0;
}
```

Once you have saved your C++ program it can be compiled from a terminal using:

```
g++ -o example example.cpp
```

This will create a file called example which you can run. To run the file use:

```
./example
```

Your computer should now greet the world.

Now, actually, our nice src is already not as clean as it should be. Check with `ls`. It should contain only source files, not binaries! No worries, let's go one directory up:

```
cd ..
```

In case you are wondering `..` means one directory up. Create a bin folder for our binaries:

```
mkdir bin
```

And run compilation as follows:

```
g++ -o bin/example src/example.cpp
```

Now our binary is created in the bin directory, while the source is in src: nicely separated! You can run the program using:

```
bin/example
```

By the way, just remove the example binary we created 'wrongly' in the src directory using:

```
rm src/example
```

And we are good to go.

## Using the EMC framework

So, we've got a C++ source file that we can compile, but it is still not very useful. We have to build software that runs on a robot and can perform a complex task. Starting from scratch would take a lot of time, but fortunately a lot is already provided! Actually it was already

secretly sitting on your computer, being installed by the install script. This software that is provided is not something that is runnable on its own, but a set of functions and C++ classes that we can use in our own project. Such a set of reusable software parts is called a software library. Now, we have to include this installed library in the project.

Open the `example.cpp` file, and change it to the following:

```
#include <emc/io.h>
#include <unistd.h>

int main()
{
    emc::IO io;
    while( io.ok() )
    {
        sleep(1);
        io.speak("test " );
    }

    return 0;
}
```

The include statement on top includes the `emc` framework in your source file, which means that all functions, classes, etc declared there can be used by your project. The `io` object is something we will use to build our application with. Don't worry about it now, we'll get back to that later.

Try to compile the project (make sure to 'be' in your project root, i.e.: `~/emc/my_project`):

```
g++ -o bin/example src/example.cpp
```

Woah, errors! Note that the error states something about undefined reference. We included `emc/engine.h` so we should be fine right? No: often `*.h`-files only declare functions etc, but they do not define them, that is: they tell the compiler something with that name is out there, but they do not provide the actual implementation. We need to tell the compiler where the implementation, which is already compiled into binary form, is. This is called linking, and we need to specify it in the `g++` command:

```
g++ -o bin/example src/example.cpp -lemc-framework
```

Here, the `-l` specifies that `g++` should link the program to the library that is called `emc-framework`. For those who are wondering, the compiler does not grab `emc-framework` out of thin air. Take a look in the `/usr/lib` directory: you will find `libemc-framework.so` sitting there, along with many other libraries. The extension `so` stands for shared object: it is a piece of

software that can be shared across different applications. The *h-files*, which are called *header files* can be found in */usr/include* (e.g., look for */usr/include/emc/io.h*).

To run this example, you will first need to start a ROS master. We can do this by launching the simulator.

## CMake

So, we compiled a source file into a binary in another directory, while linking against the *emc-framework* library. Imagine that by the end of the course you will use more libraries, and every time you need to remember the `g++` command. That's quite a nuisance! Fortunately, there are tools that will help you out. In this course, we will use [CMake](#).

### History of CMake ▾

A very short history: when Linux programmers started to become annoyed with typing the `g++` (or rather `gcc` back in those days, the compiler for plain C), they invented [Make](#), a "tool which controls the generation of executables and other non-source files of a program from the program's source files". Make allows you to specify compiler options, linking, etc of your project in a file, and once that was done, you only had to run `make` to compile your project. However, other Operating Systems created their own build systems (that is what these tools are called), e.g., Microsoft uses Visual Studio for Windows. Then some people came up with *CMake* which is a cross-platform (that is what the C stand for) build tool: it can be used on Linux, Mac OS and Windows. In fact, it builds on top of the OS-specific build tool. For example, on Linux, CMake generates Make-files, while on Windows it generates files that can be used by Visual Studio. That's quite useful! It allows programmers from all over the world to collaborate on software projects, even if they are using different Operating Systems!

Enough talking, let's start using CMake to *build* our project, such that compilation becomes easier. All you have to do is create a file in the root (`~/mrc/my_project`) of you project that can be read by CMake, called *CMakeLists.txt*. In it, you specify the instructions that are needed to compile the project. Create a file called `CMakeLists.txt` (e.g. `gedit CMakeLists.txt`) with this text:

```
cmake_minimum_required(VERSION 2.8)
project(my_project)

add_executable(example src/example.cpp)
```

This file is probably quite understandable at first sight: it specifies the minimum required version of CMake to read the file, the name of your project, and states that an executable called *example* should be created from the source file *src/example.cpp*.

Now, how should we use this thing? As was already said, CMake does not directly call the compiler. Instead, it generates *Makefiles* which can be used by the Linux-dependent *Make* tool. These *Makefiles*, or more generically called *build files*, are created in a separate folder, often called *build*. Go to the root of your project (`cd ~/mrc/my_project`), create a build directory and go inside:

```
mkdir build
cd build
```

Now, to generate the build files, we only have to call CMake and refer to the directory in which the CMakeLists file is we just created:

```
cmake ..
```

(remember .. stands for 'one directory up')

Have a look inside the build directory: CMake generated a lot of files, one of which is a Makefile. Now while 'being' in the build directory, call Make:

```
make
```

You will see that the compiler is called (as if we started g++ ourselves). Oh whoops... Again the *undefined reference* error. But this makes sense: we did not specify yet that we need to use the emc-framework library. Doing so in CMake is easy. Edit the CMakeLists.txt file, and add below the *add\_executable* statement:

```
target_link_libraries(example emc-framework)
```

Now run cmake and make again:

```
cd build
cmake ..
make
```

Success! However, the binary is also created in the directory where you called make, i.e. in the build directory. It would be nice to have it in the bin directory we created earlier. Well, we can. Just add the following line to your CMakeList.txt:

```
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
```

This sets the CMake variable EXECUTABLE\_OUTPUT\_PATH to be the bin directory in your project directory. The \${...} simply returns the value of the variable inside. The CMake variable PROJECT\_SOURCE\_DIR is set by CMake and refers to the directory in which your



CMakeLists.txt is placed (the name is somewhat confusing...). Try to run `cmake ..` and `make` again. You should see the executable appear in your bin directory.

By the way, you only need to run the `cmake` command if you change your *CMakeLists.txt*. Once you start programming, i.e, editing your source files, you can simply go to the build directory and run `make`.

### A note on CMake ▼

You might be thinking: this still is a lot of hassle. The `g++` command wasn't that bad, and now I need an extra directory, some extra commands, understand CMake, etc. Indeed, for the small example above CMake is a bit overkill. However, once your project grows and source files are added, more libraries are used, etc, you will see that it is quite handy to have your project set-up defined in one simple file. Furthermore, you won't have to tell your teammates what command to run if you add a file or library, as you can simply update the *CMakeLists.txt*. Also, CMake is a widely used language, and supported by many Integrated Development Environments (IDE), which can be thought of as really smart programming editors that not only provide editing, but also support compiling and even running and debugging your program. Having a *CMakeLists.txt* is having a project definition that can be used by many IDEs to 'understand' what is going on in your project.

## Towards an autonomous robot

So far, we have seen how to create a simple C++ project, run the simulator, show some visualizations and drive the simulated robot around using the keyboard. That's nice and all, but we don't want to manually drive around a virtual robot. We want an autonomous, real robot!

As was already stated during the lecture, we won't expose you to the (sometimes somewhat frustrating) low-level details of connecting software to hardware. Instead, we provide you with an abstraction layer that can be easily used within your program to read sensor data and send goals to the base. Under the hood this layer communicates using ROS, but unless things go horribly wrong you won't have to worry about that.

## The loop

When we want to control and monitor a piece of hardware, we often want to perform a series of steps, computations, procedures, etc., in a repetitive manner. When we talk about doing something repeatedly in software, the first 'control flow statement' that comes to mind is a loop. So, let's create a loop!

```
#include <iostream>
```

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

    return 0;
}
```

Remember that while the condition in the while statement is true, the body of the while loop will be executed. In this case, that is forever! Fortunately you can always interrupt the program using ctrl-C from the command line. By the way, the default behavior is that this directly kills your program, so all statements after the while loop (if there were any) would never be executed. You can verify this by putting a print statement there. You will see it is never called...

So, it's a nice loop, but there's at least three things wrong with it:

1. It runs forever, never 'gracefully' shutting down (only by user interruption)
2. It runs as fast as possible!
3. It doesn't do much useful except for flooding the screen...

For now, let's focus on point 2). Your operating system schedules the execution of programs: if multiple programs are running simultaneously, it gives each program a short period of time to perform their executions and then jumps to the next. What our program does in that time slice is printing 'Hello World!' as fast as it can! It is like a horrible, zappy kid taking up all of your time as soon as you give it some attention. We can be better than that.

## Let's wait a little

In fact, you can tell the operating system that you're done for some time. This allows it to schedule other tasks, or just sit idle until you or another program wants to do something again. This is called sleeping. It's like setting an alarm clock: you tell the operating system: wake me up in this-and-this much time.

So, let's add a sleep statement:

```
#include <iostream>
#include <unistd.h> // needed for usleep

int main()
{
    while(true)
    {
        std::cout << "Hello World!" << std::endl;
        usleep(1000000); // sleep period of time (specified in microseconds)
    }
}
```

```
    }  
  
    return 0;  
}
```

Ahhh, that's much better! Now approximately every second a statement is printed. This will use way less CPU power than the previous implementation. Note that we explicitly stated 'approximately'. The loop runs at approximately 1 Hz because:

1. The other statements in the loop also take time (in this case the printing).
2. The operating system can not guarantee that it will wake you up in exactly the time specified. This has to do with program priorities, schedules, etc. In high-performance mechatronics system, it is often needed that this frequency can be specified as 'hard' or 'strict' as possible. Therefore these machines often run real-time operating systems that will guarantee that, or at least to some extent. Don't worry about it, you won't notice Ubuntu is not hard real-time.

Although you shouldn't worry about the second point, it is important to take the first into account. As you will put more and more code into the body of your application, it will take more and more time to process it. Sleeping for a fixed amount of time causes your system to start lagging behind at some point.

Fortunately, we created something for you: a class that can be used to keep track of the time spent since the last sleep statement, and which will only sleep the remaining loop time. Use it like this:

```
#include <iostream>  
#include <emc/rate.h>  
  
int main()  
{  
    emc::Rate r(3); // set the frequency here  
  
    while(true)  
    {  
        std::cout << "Hello World!" << std::endl;  
        r.sleep(); // sleep for the remaining time  
    }  
  
    return 0;  
}
```

## Control the robot

Now finally, let's connect to the robot (even though it is a simulated one...)! As was already stated, we can use two types of inputs from the robot:

- The laser data from the laser range finder
- The odometry data from the wheels

And, in fact, we only have to provide one output:

- Base velocity command

That's it! All we have to do is create a mapping from these inputs to this output! We provide an easy to use IO object (IO stands for input/output) that can be used to access the robot's laser data and odometry, and send commands to the base. Let's take a look at an example:

```
#include <emc/io.h>
#include <emc/rate.h>

int main()
{
    // Create IO object, which will initialize the io layer
    emc::IO io;

    // Create Rate object, which will help keeping the loop at a fixed frequency
    emc::Rate r(10);

    // Loop while we are properly connected
    while(io.ok())
    {
        // Send a reference to the base controller (vx, vy, vtheta)
        io.sendBaseReference(0.1, 0, 0);

        // Sleep remaining time
        r.sleep();
    }

    return 0;
}
```

First, try to understand what is going on. You should already be able to derive what will happen if this program is executed.

Now, note a few things: The IO connection is created by just constructing an `emc::IO` object, it's that easy! We will loop as long as the connection is OK. Then, we can send a command to the base by simply calling a function on the io object. We do this at a fixed frequency of 10 Hz.

Now, fire up the simulator and visualization, and run the example. And what do you see? Voila, we move the robot using our application! Try modifying the `sendBaseReference` arguments, and see how they affect the robot behavior.

## Making the robot aware

So, the robot moves, but it's still pretty stupid. The simulator doesn't have collision detection, so once the robot is near a wall it just goes through it. We don't want that to happen to the real robot, because it will crash! Let's see if we can do something smart. Maybe using the laser?

The io object can not only be used to send commands, but also to read data from the sensors (more detailed information about the different sensor data, i.e., the laser data, odometry data, is given in the next tutorial step). Here, we will treat a reading of the laser data, which can be done as follows:

```
...
emc::LaserData scan;
if (io.readLaserData(scan))
{
    // We got new data, so do something with it
}
...
```

First, you have to create an object / variable that will hold the laser data. Then you call `readLaserData` with this object, and two things may happen: either new laser data was received and the function returns true. We can then directly start processing it. Or, the function returned false, which means there is no new data.

The `emc::LaserData` type is in fact a struct that holds all kind of data. Inside you will find information about the maximum range the sensor can measure, the minimum and maximum angle, and of course, the measured distances, or ranges themselves. Try putting the snippet above in the loop you created earlier. Add code that will be executed if new laser data is received, and which prints the minimum angle of the received data:

```
...
#include <iostream> // Add this line under the other includes
...
std::cout << scan.angle_min << std::endl; // Add this line in the loop
...
```

See, you can use the period ('.') to access members of the scan object. This is very similar to accessing the functions `sendBaseVelocity` and `readLaserData` of the io object. Now if you run the example, and your simulator is running, you will see a value being printed. This represents the minimum scanning angle of the Laser Range Finder. It doesn't change, and that makes sense: the minimum angle doesn't change because it is a fixed value.

Now let's try accessing a more interesting member of the scan object. The ranges member is not simply a single value but an array of values, or rather, an `std::vector`. These values

represent the measured distances at different angles. The scan object specifies a minimum angle and the `angle_increment` per array index, which means that you can calculate for each range index to which angle it belongs. Accessing a range value can be done using `[ ]` and `std::endl`, e.g.:

```
...
std::cout << scan.ranges[0] << std::endl;
...
```

prints the first distance in the ranges vector (i.e., the range belonging to the minimum angle). Note that all distances are in meters. Now, finally, we can make the robot a bit smarter. We can use it to loop over the values in the range vector:

```
...
for(unsigned int i = 0; i < scan.ranges.size(); ++i)
{
    if (scan.ranges[i] < some_value)
    {
        ...
    }
}
...
```

Alright!! Now you have all the ingredients to create an application that drives HERO forward, but stops if an obstacle is near!! We can create a main loop at a fixed frequency, send base commands, read the laser data and do something sensible with it. Just know that you can stop the base by simply calling `sendBaseReference(0, 0, 0)`.

## Obtaining laser, odometry and bumper data

This page provides a short description of the laser data, odometry data, and bumper data that can be obtained through the IO object introduced earlier.

### Laser Data

To obtain the laser data, do the following:

```
emc::LaserData scan;
if (io.readLaserData(scan))
{
    // ... We got the laser data, now do something useful with it!
}
```

The `LaserData` struct is defined as follows:

```
struct LaserData
{
    double range_min;
    double range_max;

    double angle_min;
    double angle_max;
    double angle_increment;

    std::vector<float> ranges;

    double timestamp;
};
```

The `range_min` and `range_max` values define what the smallest and largest measurable distances are. If a distance reading is below `range_min` or above `range_max`, that reading is invalid. The values `angle_min` and `angle_max` determine the angle of the first and last beam in the measurement. The value `angle_increment` is the angle difference between two beams. Note that it is actually superfluous, as it can be derived from `angle_min`, `angle_max` and the number of beams.

The actual sensor readings are stored in `ranges`. It is an `std::vector`, a vector of values which, in this case, stores floats. Each vector element corresponds to one measured distance in meters at a particular angle. That angle can be calculated from `angle_min`, `angle_increment` and the index of the element in the vector.

Finally, the `timestamp` specifies at which point in time the data was measured. The timestamp is in Unix time, i.e., the number of seconds since 1 January 1970. Note that the absolute value is not necessarily important, but that the timestamp can be handy to keep track of laser data over time, or to synchronize it with other input data (e.g., the odometry data).

## Odometry Data

The robot has a holonomic wheel base which consists of two wheels in a diff drive configuration and a rotational joint which rotates the entire body. The specific configuration of the wheels allows the robot to move both forwards and sideways, and enables it to rotate around its axis. Both wheels and the joint have an encoder which keeps track of the rotations of that wheel. By using all three encoders and knowing the wheel configuration, the displacement and rotation of the robot can be calculated. In other words: we can calculate how far the robot drove and how far it rotated since its initial position. This translation and rotation based on the wheel encoders is called odometry. However, note that this information is highly sensitive to drift: small errors caused by measurement errors and wheel slip are accumulated over time. Therefore, relying on odometry data alone over longer periods of

time is not recommended! Also note that the odometry data does not start at coordinates (0,0).

To obtain the odometry information, do the following:

```
emc::OdometryData odom;
if (io.readOdometryData(odom))
{
    // ... We got the odom data, now do something useful with it!
}
```

The OdometryData struct is defined as follows:

```
struct OdometryData
{
    double x;
    double y;
    double a;
    double timestamp;
};
```

Here x, y and a define the displacement and rotation of the robot since the previous measurement, according to the wheel rotations. The translation (x, y) is in meters. The rotation, a is in radians between  $-\pi$  and  $\pi$ . Like the laser data, the odometry data also contains a timestamp which is in seconds (Unix time).

## Lazy Boolean Evaluation

The functions shown above are boolean functions, i.e., they return True or False to indicate whether a new measurement of the sensor is available. If you want something to happen when either laser or odometer data is found you may be tempted to use something like:

```
emc::LaserData scan;
emc::OdometryData odom;
if (io.readLaserData(scan) || io.readOdometryData(odom))
{
    // ... We got sensor data, now do something useful with it!
}
```

However C++ uses lazy boolean evaluation. Meaning it won't evaluate the second argument in an OR statement if the first is already True. Similarly it won't evaluate the second argument in an AND statement if the first is False. In this example the laser data will be read but the odom struct remains empty! Consider instead this approach:



```
emc::LaserData scan;
emc::OdometryData odom;
bool new_laser_data = io.readLaserData(scan);
bool new_odom_data = io.readOdometryData(odom);
if (new_laser_data || new_odom_data)
{
    // ... We got sensor data, now do something useful with it!
}
```

## Exercise 1: the art of not crashing

### 📌 Don't crash ▾

Wow finally an actual exercise! We saw how we can control the robot and how we can read the laser data. Now lets make the robot do something.

1. Create a folder called exercise1 in the mrc folder. Create an src folder within the exercise1 folder. Within this folder make a new file called `dont_crash.cpp`.
2. Add a CMakeLists.txt to the exercise1 folder and set it up to compile the executable `dont_crash`.
3. Q 1 Think of a method to make the robot drive forward but stop before it hits something. **Document your designed method on the wiki.**
4. write code to implement your designed method. Test your code in the simulator. Use the Once you are happy with the result, **make sure to keep these files as we will need them later.**
5. Make a screen capture of your experiment in simulation and **upload it to your wiki page.**

### 📌 Multiple choice ▾

Your group will have found and implemented multiple methods of making a non-crashing robot. Describe them all on your wiki. Do credit the team members that worked on each one.

**On your wiki. Compare the different methods to one another. What situations do these methods handle well? Is there one "Best" method?**

## Sharing your project through git

Creating a software solution together means sharing a project, which in our case means sharing code. We will be using git for this. Git is "a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency":

- version control means that the files and changes you make to them are 'tracked' over time, i.e., their 'history' is stored. You can always go back to previous versions, show the differences between a certain point in time and now, and much more.
- distributed means that you don't necessarily need a central server to store your changes. Every team member has the full history available, and can always save his or her changes or go back to previous versions, even if he or she is offline. However, note that it is possible to use a central server that every team member writes his or her changes to, and receives the changes others made from. In this project, we will be using such a central server.

## Clone a project

The examples you worked on up until now are stored on your own computer, which is fine if you are the only person working on your project. However when working with a team you need every member to have access to the most recent version of the code base. Well, sharing is easy in git! Your teammates can get a local copy of the repository. This is called 'cloning', and works as follows.

The MRC team has already initialized a git repository for you which includes a small example. To share the repository with your group we will use GitLab. GitLab is open source software to collaborate on code. The TU/e has its own gitlab server which we will make use of. Now:

- Browse to [gitlab.tue.nl](https://gitlab.tue.nl) and click the TU/e login button
- Take the steps you usually take to login to your TU/e account
- E-mail your username (that starts with @) to Peter van Dooren (only one group member has to send their username)
- Once we have added you to your group's project, you will receive an e-mail with the link to your repository
- You can then add your colleagues to your project and start collaborating!

Before you can clone the repository we will have to tell gitlab it can trust your computer. For this we use a process called ssh. In a terminal type

```
ssh-keygen
```

Don't enter anything when given prompts. Simply press enter two or three times. We don't really need a passphrase for this ssh key and we can use the default location. After doing this type

```
cat ~/.ssh/id_rsa.pub
```

This will print your public ssh key to the terminal. Next, on gitlab select edit profile on your own profile. In the tab SSH Keys select `Add new key`. In the key field copy paste your public

ssh key. (remember that in a terminal copy is done via ctrl+shift+C). Give your key a descriptive name such as "MRC-virtual-ubuntu" and add the key.

## SSH keys ▾

SSH is communication protocol. The keys we use are a way for computers to verify eachothers identity. When you communicate with gitlab on your virtualbox the computer will automatically authenticate itself with this key. Think of it like a password for computers.

Once you added your ssh key to your account you can clone your groups repository using the following command (fill in the your group name for <GROUP\_NAME>, or copy the whole link from the GitLab page of your group):

```
cd ~/mrc
git clone https://gitlab.tue.nl/mobile-robot-control/2024/<GROUP_NAME>.git
shared_project
```

This will create a directory called 'shared\_project' in which you will find the files that someone else made. Now, this is your local copy of the repository to which you can make changes. We will learn here how we can save those changes and safely store these changes to the central server.

In git terminology, we commit changes to our local repository and then push these changes to the remote. So, let's do this!

## Telling git who you are

Before you start using git, you have to tell git who you are. That way, git can 'stamp' all the changes you make with your name and email address. That way it is easy to keep track of who did what, and when. Just run the following (with your real name and email address):

```
git config --global user.name "Put your name here"

git config --global user.email "Put your email address here"
```

## Making changes in a git repository

Navigate to your project directory, which is most likely the directory you just created while cloning the repository. For this tutorial, we will use '~/mrc/shared\_project', but you will use the actual name of your project.

```
cd ~/mrc/shared_project
```

Lets add the files of our previous exercise to this folder. Copy the folder exercise1 to your shared folder.

```
cp -r ~mrc/exercise1 ~mrc/shared_project/exercise1
```

Lets also make some changes to the README.md. Add your name to the list of group members. Open a file editor with

```
gedit README.md
```

After opening the file, make your changes and hit save.

If you want to check what changes you have made since your last commit use

```
git status
```

You will see a list of modifications and untracked files, e.g., something like this:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        exercise1
```

Modified means that the file has been changed since the last commit. Untracked means that these files and folders are not under git version control. If you do not remember what you changed since the last commit you can inspect it with

```
git diff README.md
```

This will show you the changes you made to README.md. If you leave out the argument git diff will show you the changes in every file in the repository.

## gitignore

Note that we only want some of the files to be tracked by git. For example, the bin and build folder are automatically generated by CMake and compilation. It is not necessary to share these files with your team members. We should therefore only add the source files, and the CMakeLists.txt (and possibly other files that you want to share). Since we

know that we will never want to commit the build and bin folders we have specified these folders in a process called [gitignore](#)

We should tell git which files we want to keep. To make sure git will keep track of files, use git add:

```
git add README.md
```

Now, run git status again. You should see something like this:

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    exercise1
```

As you can see, you now have some 'changes to be committed'. Committing means we store the added files as they are now. From that point on we can make new modifications, and we can always roll back to this version, see the changes since this version, and, we can share the changes with others. To commit the changes:



```
git commit -m 'Type a meaningful sentence here'
```

The part after '-m' is the commit message. It is important that this message is a meaningful message that describes what the added changes are, for example 'fixed this-and-this bug' or 'added this-and-this feature'. That way, you can look in the 'git log' and easily see what was added when, and to which version you need to roll back if you have to.

So, we have committed our changes. Let's see the status using git status. It no longer indicates changes to README.md. That is correct, we just committed them! Now, have a look at the git log:

```
git log
```

It outputs an overview of who committed what and when. You should see your name and commit message, and some long code with letters and numbers. This is a commit hash, and can be thought of as a unique id for this commit.

 [add your work](#) 

Now add the files from exercise1 and commit the changes using these same steps. You should commit multiple files at once if their changes are related.

## Storing your changes on the server

Alright, your work is now safely stored on your computer. You can make changes, and commit these as well (remember, first use 'git add', then 'git commit'). But you also want to share these changes with your team mates!

Since we have cloned the repository from gitlab, git will assume that this is also where it should send the commits to (and where it can pull them from later). But we still have to tell git to actually send them there. This is called 'pushing'. To do this you have to use this command:

```
git push
```

and enter your username and password.

Optional: A nice alternative to typing your username and password every time is to configure a SSH key for git and gitlab. A nice tutorial to accomplish this can be found [here](#), follow the tutorial from How to generate an RSA SSH key pair onward.

We won't go into details here, but the push command basically states that the current commits should be send to the origin (on Gitlab) and that the git repository should 'keep an eye' on what happens on the server. This means that if one of your team mates also pushes commits to this address, we can 'pull' them to your local repository. This is exactly how the synchronization works. You push your changes, and pull changes that others made.

## Retrieving changes from the server

Up until now, you have seen how you can commit your changes to the server. One thing is not yet explained, and that is how you can get the changes that others pushed to the server since you cloned it. Well, it's easy:

```
git pull
```

This will 'pull in' the latest commits and your files will be updated accordingly. Note that this won't work if you have modified files, i.e., file changes that you have not yet committed. You don't want these changes to be overwritten by a 'pull', so git doesn't allow it. To get the latest changes, you have to commit your changes and try pulling again.

## Git GUI

If you have completed the tutorial up to this point, you have learned how to use the basic functionality of git in the command line. However, while many people are satisfied with using git in the command line, plenty of graphical user interfaces (GUI) for git are available for free. These GUI generally offer the same (basic-)functionality as the command line you have seen earlier. However, do not require you to type and remember the syntax of every command. An important remark is, of course, that finding the right button that corresponds to the command you want to execute can be as difficult as remembering that command in the first place.

A popular git GUI is SmartGit. To install and try SmartGit, run:

```
wget https://www.syntevo.com/downloads/smartgit/smartgit-21_1_0.deb
sudo apt install ./smartgit-21_1_0.deb
```

Then to use SmartGit, hit the windows key on your computer, type smartgit, and launch the program. In the setup windows that will launch select Non-Commercial (Academic) use, and follow further instructions. In the window "Welcome to SmartGit" select the repository you want to use, or clone a repository by providing SmartGit with a link to your repository.

After setup, SmartGit will show a graph of all past commits, shows a window to make a new commit, and will show you a window with changed files. If you're planning to use SmartGit in this course, experiment a bit, try to find out if you can use SmartGit to do all the steps you have done in the first part of this tutorial.

## Recap

The tutorial above showed how to clone a repository in GitLab, how to push changes to it, and how to pull changes from it. From now on, keep working in this directory and commit all your changes, and pull the changes from your team members. After working through the tutorial above, you can also use the [GIT Cheatsheet](#) in case you need a refresh: click on e.g. workspace to see all related commands.

Again, you can commit the change once you've done something useful, but first you need to tell git again which files you want to commit by using 'git add'. Then once you've created the commit, you can push it to the server, this time simply using 'git push'. So, to sum up:

- Pull the changes your teammates made:

```
git pull
```

- Make your changes
- Add the files of which you want to commit the changes:

```
git add .\path\to\file
```

- Commit your changes:

```
git commit -m '... message ...'
```

- Push your changes to the server:

```
git push
```

## Exercise 2: Testing your don't crash

### ✓ Todo ▾

In the repository that you just cloned you will find a folder called test. This contains 2 maps for the simulator. Test your don't crash solution with these two maps. **report your findings on the wiki**

## Setting up an IDE

To keep the code in your packages clear and manageable, it is advised to use an Integrated Development Environment (IDE) to edit your C++ code. VScode is such an IDE. It has the advantage of understanding your code up to some extent. This means VScode can be used to, e.g., auto-complete names of variables and functions, get compilation error messages, and can even allow you to debug your code in a nice way. Also, as was stated in the previous tutorial, it can be made to understand CMake, which allows it do be used to compile your project, and even run the resulting executables.

## Install and Configure VScode

To make the use of VScode more comfortable for you we've created a script, which if you run it not only installs VScode, but also installs a few of its many extensions. These extensions extend the functionality of VScode. There are extensions to make it understand CMake, Git and C++.

To install VScode:

```
wget -O install.bash  
https://raw.githubusercontent.com/KdVos/MRC_vscode_install/main/vscode_install.ba  
sh  
source ./install.bash
```



## Setting up VScode for your project

Now you have installed a proper IDE, you can start to do some real programming! In previous tutorials, we created a C++ project called `my_project` and went through a little bit of work to get it to build using CMake. Now, that work will pay off: VScode 'understands' CMake, so we can directly load the project.

In a terminal go to your project. Within the root directory type the following:  
(the root directory is the directory containing `CMakeLists.txt`)

```
code .
```

This will open vscode in the Current directory. It will ask you whether you trust the authors of this directory, since we are the authors we will trust ourselves. Afterwards, when you are prompted to "select a kit" select the [unspecified] option. In the sidebar the current directory and the files contained in it are shown. Click on your `CMakeLists.txt` to open this file.

One of the most powerful features of vscode is the command palette. Almost all functionality of your IDE and its extensions are available through this window. To open it type:

`ctrl+shift+p`. (sometimes this doesn't work directly, you can try restarting vscode in this case)

To tell vscode that you like to use your cmake file, open the command palette and type `Cmake: Configure:` and hit enter. VScode will now remember that you are using CMake to build your program.

If we now open our `.cpp` file, we can try to compile our program within vscode. Again open up your command palette, but this time type `Cmake: Build`. In the terminal in the lower part of your screen you will see that our project is successfully built.

To run your program type `Cmake: Run without debugging` and our program will execute.

Since typing our commands in the command palette can get boring after some time, key bindings were created to make our lives easier. If we again open the command palette and type `Cmake:Build` we see that we could have also used the F7 key to build our project. If we close the command palette again and use the F7 key we see that this is indeed the case.

## Using VScode

From this point on you don't really have to leave VScode any more. You can edit the source file in VScode, edit the `CMakeLists.txt`, run CMake and even run your program. That's pretty awesome!

But you ain't seen nothing yet. Double click on `example.cpp` to edit it. Remove the `#include` statement at the top, and type it again, but slowly. You will see that as soon as you start typing `#include`, VScode pops up a window with some suggestions. The more you type, the

more specific it becomes. You can select an option using the arrow keys, and press enter or tab to confirm. This is called auto-completion and will save you a lot of typing. Continue typing. When you get to `<em`, you will see that VScode even understands the location of header files, and auto-completes them for you.

Now lets make a mistake on purpose: misspell return and use F7 to compile the project. You will see a list of issues pop up on the bottom. You can double click on issues to directly jump to the error, which is especially useful if your project gets bigger.

## Debugging your project

As you might know, the life of a programmer and software developer sometimes largely consists of debugging our own code. Ofcourse, VScode gives you the tools to do this.

You can add a breakpoint to your file by clicking the space before a line number.

You can run your program till an error occurs. Or you can pause execution during a run.

First, make sure you have built your project in debug mode using "Cmake: Select Variant". Afterwards, you can start debugging by typing "CMake: Debug" in your command palette.

Note, however, that your code performs better when you change your build variant back to Release mode.

## Making your life easier

So far we've only used the file explorer sidebar. But if you look closely at the left side of your vscode window you'll discover a range of other functionalities. This tutorial will highlight a few of them, but is definitley not exhaustive.

## GIT

In a later tutorial we will teach you how to use GIT. A version control system which enables easy collaboration on your code. VScode offers a built-in Graphical User Interface (GUI) which is accessed through the sidebar. Just click the third icon, which looks like a crossroads.

## TODO-tree

Since software development can sometimes become a bit of a complex operation, in which a large amount of things need to be implemented across multiple files and functions. We've added an extension so you can easily keep track of your todos, bugs and thing to be fixed. To access it just click the tree icon, it should be the second to last icon in your sidebar.

If we go back to our project files, and add a comment saying `// #TODO: this needs to be fixed ASAP`, you will see an item pop up in your todo-tree. This way it is easier to keep track of your todos, and you can make sure you don't forget about them.

## Concluding Remarks

As said before, Vscode offers you a library of hunderds of extensions, not all are useful to us, but some can improve your software developement experience significantly. This tutorial is, therefore, all but exhaustive. There are probably functionalities and extensions we missed, but experimenting a bit can probably lead to an even better experience.