Robotics Lab - Lecture 2

Robotics programming technologies
(Linux, c++, make & git)

Jonathan Cacace

March 3, 2020

Contents

Preface

This document contains the lecture notes for the *Robotics Lab* class taken at University of Naples Federico II for automation engineering master degree. The aim of this course is to give an overview of the fundamental tools and techniques used to program advanced robotics systems (both industrial and mobile). After a brief introduction of the technologies commonly used to program robots (e.g. Linux, c++, git), the Robot Operating System (ROS) framework is introduced and deeply studied. Simulation software will help the course attenders to implement and test state-of-art robotic algorithms and their own robot control software.

In this lesson, an overview of the technologies used to program robots is provided. In particular, this lesson will focus on an introduction to linux basics, to c++ programming and its compilation. Finally, an overview of version control software, with particular case of GIT is also provided.

1

Robotics programming technologies

Nowadays, there exist different setup to program robotic systems. Some of these rely on proprietary languages developed by robot providers, like `KRL` and sunrise for kuka industrial robots, or `RoboDK` for Universal Robots. However, these languages don't allow to develop intelligent robotic application to perform complex tasks, but they are considered to be implement cyclical motion in a completely known environment. Differently, sometimes developers need to program robot considering different high level or low level prospective. For this reason, to develop advanced robotic applications in which the robot is able to plan new actions based on the state of its operative environment, standard programming languages must be used. In this lesson we mainly considered `C++` because is fast and versatile, can be used both for high level reasoning (geometrical reasoning, image elaboration, etc ...) and for low level control. Before to recall some basic concepts on c++, we briefly discuss some basic concepts and commands used in Linux.

1.1 Linux Operating System

Linux is a family of open source Unix-like operating systems based on the Linux kernel, an operating system kernel free and open-source, monolithic and Unix-like kernel, released on 1991. Typically different versions of Linux are packaged into distributions. Nowadays, exist hundreds of linux distributions usable on embedded, server and desktop devices. One of the most popular linux distributions for desktop computers is called *Ubuntu*. Ubuntu is an African sub-Saharan term meaning *humanity* or more specifically it is philosophy: 'the belief in a universal bond of sharing that connects all humanity' or 'I am because we are'. Ubuntu is released every six months, with long-term support (LTS) releases every two years. The most recent long-term support release at writing time is 18.04 LTS (Bionic Beaver), which is

supported until 2023 under public support.

Independently from the distributions, all versions o linux share a list of commands invokable via the linux console (terminal). Be able to use linux console for a robotics software developer is fundamental for several reasons. First of all, typically robot based on Linux OS are not endowed of a graphical interface or developers joint the robot remotely using the Secure Shell (ssh) connection. In addition, a list of commands from the command line can be extremely useful to properly configure the robotic system.

1.1.1 Install Linux

The convenient way to install linux is with multi boot options. This setup allows to install multiple operating system on the same hard disk. Another way to use linux is relying on Virtual Machine. This solution in our case presents some disadvantages, mainly because of the requirements needed by the simulation software.

1.1.2 Basic Linux commands

In this section, a list of recurrent important linux commands are reported.

- **pwd**: when you first open the terminal, you are in the home directory of your user. To know which directory you are in, you can use the "pwd" command. It gives us the absolute path, which means the path that starts from the root.

- **ls**: use the ls command to know what files are in the directory you are in. You can see all the hidden files by using the command ls -a.

- **cd**: Use the cd command to go to a directory. For example, if you are in the home folder, and you want to go to the downloads folder, then you can type in cd Downloads. Remember, this command is case sensitive, and you have to type in the name of the folder exactly as it is. To go back from a folder to the folder before that, you can type "cd .." . The two dots represent back.

- **mkdir & rmdir**: Use the mkdir command when you need to create a folder or a directory. To delete a directory containing files, use rmdir.

- **rm**: Use the rm command to delete files and directories. Use "rm -r" to delete just the directory. It deletes both the folder and the files it contains when using only the rm command.

- **touch**: the touch command is used to create a file. It can be anything, from an empty txt file to an empty zip file. For example, "touch new.txt".

- **cp**: Use the cp command to copy files through the command line. It takes two arguments: The first is the location of the file to be copied, the second is where to copy.

- **mv**: Use the mv command to move files through the command line. We can also use the mv command to rename a file. For example, if we want to rename the file "text" to "new", we can use "mv text new". It takes the two arguments, just like the cp command.

- **locate**: the locate command is used to locate a file in a Linux system. This command is useful when you don't know where a file is saved or the actual name of the file. Using the -i argument with the command helps to ignore the case (it doesn't matter if it is uppercase or lowercase). So, if you want a file that has the word "hello", it gives the list of all the files in your Linux system containing the word hello when you type in locate -i hello. If you remember two words, you can separate them using an asterisk (*). For example, to locate a file containing the words hello and this, you can use the command locate -i *hello*this. In order to have an update representation of the filesystem and be able to find even the newest files contained in your machine you should insert the following command:

  ```
  $ sudo updatedb
  ```

  The first time this command could take a bit of time.

- **echo**: the echo command helps us move some data, usually text into a file. For example, if you want to create a new text file or add to an already made text file, you just need to type in, echo hello, my name is alok » new.txt. You do not need to separate the spaces by using the backward slash here, because we put in two triangular brackets when we finish what we need to write.

- **cat**: use the cat command to display the contents of a file. It is usually used to easily view programs.

- **nano**: nano is a text editors already installed in your Linux command line. The nano command is a good text editor that denotes keywords with color and can recognize most languages. It is one of the simplest text editor usable via command line.

- **sudo**: a widely used command in the Linux command line, sudo stands for SuperUser Do. So, if you want any command to be done with administrative or root privileges, you can use the sudo command. For example, if you want to edit a file like viz. alsa-base.conf, which needs root permissions, you can use the command: sudo nano alsa-base.conf.

- **chmod**: use chmod to make a file executable and to change the permissions granted to it in Linux. Imagine you have a python code named numbers.py in your computer. You'll need to run `python numbers.py` every time you need to run it. Instead of that, when you make it executable, you'll just need to run numbers.py in the terminal to run the file. To make a file executable, you can use the command chmod +x numbers.py in this case. Another situation in which this command is particularly useful is when your application needs to access to `USB` devices. In such case the device should have executable privileges.

- **ping**: use ping to check your connection to a server.

- You can power off or reboot the computer by using the command sudo halt and sudo reboot.

- You can use the clear command to clear the terminal if it gets filled up with too many commands.

1.1.3 Basic Linux concepts

In this subsection we report some basic concepts common to all linux operating systems.

- Command auto completion: auto completion represents a fundamental feature of linux console. In particular, TAB key can be used to fill up in terminal. For example, You just need to type `cd Doc` and then TAB and the terminal fills the rest up and makes it `cd Documents`.

- filesystem: most Linux systems use a standard layout for files so that system resources and programs can be easily located. This layout forms a directory tree, which starts at the / directory, also known as the *root directory*. Directly underneath / are important subdirectories: /bin, `/etc`, `/dev`, and `/usr`, among others. These directories in turn contain other directories which contain system configuration files, programs, and so on. In particular, each user has a home directory, which is the directory set aside for that user to store his or her files. A user has completely control of its user space (/home/user). Differently, for the higher level of the filesystem the operations must be performed with the use of superuser privileges (sudo in ubuntu).

- .bashrc: the linux shell is called bash (Bourne Again SHell). It is a command processor that typically runs in a text window where the user types commands. Bash can also read and execute commands from a file, called a shell script. Like all Unix shells, it supports piping, here documents, command substitution, variables, and control structures for condition-testing and iteration. When a new interactive linux shell

is open a series of configuration files are elaborated. In particular, bash reads and executes `/etc/bash.bashrc` and then `~/.bashrc`. For this reason, all the system configuration that you want to automatically load can be placed in the `bashrc` file. This file is placed in the home directory of the user. in addition, it is a hidden file (in fact its name starts with a dot): `/home/user/.bashrc`.

- Environment variables: in linux based systems environment variables are a set of dynamic named values, stored within the system that are used by applications launched in shells. In simple words, an environment variable is a variable with a name and an associated or a list of values. Environment variables allow you to customize how the system works and the behavior of the applications on the system. For example, the environment variable can store information about the default text editor or browser, the path to executable files, or the system locale and keyboard layout settings. In bash a variable can be set in the following way:

```
$ export V=environment
```

While, to print the content of a variable you should refer to the variable name using the $ character:

```
$ echo $V
```

In the following a list of commands used to handle environment variables are reported:

```
$ echo $VARIABLE  #To display value of a variable
$ env  #Displays all environment variables
$ VARIABLE_NAME=variable_value  #Create a new variable
$ unset variable_value  #Remove a variable
$ export Variable=value  #To set value of an environment variable
```

- APT: to install new software a convenient way is to use the package manager. In ubuntu (derived from debian) the package manager is called APT (Advanced Packaging Tool). APT simplifies the process of managing software on Unix-like computer systems by automating the retrieval, configuration and installation of software packages, either from precompiled files or by compiling source code. In order to install a new software you should use the following syntax:

```
$ sudo apt-get install [PACKAGE NAME]
```

If the package exists in the apt repository, the list of dependencies of such package will be also automatically installed. If you want to check if a software is contained in the apt repository you could use the following command:

```
$ apt-cache search [PACKAGE NAME]
```

Otherwise, if you know only initial part of the package you can used the auto completion.

Other usage modes of apt and apt-get that facilitate updating installed packages include:

- **update** is used to resynchronize the package index files from their sources. The lists of available packages are fetched from the location(s) specified in /etc/apt/sources.list. For example, when using a Debian archive, this command retrieves and scans the Packages.gz files, so that information about new and updated packages is available.
- **upgrade** is used to install the newest versions of all packages currently installed on the system from the sources enumerated in /etc/apt/sources.list. Packages currently installed with new versions available are retrieved and upgraded; under no circumstances are currently installed packages removed, or packages not already installed retrieved and installed. New versions of currently installed packages that cannot be upgraded without changing the install status of another package will be left at their current version.

In order to add additional software list to the apt repository you should edit the files included in `/etc/apt` directory. In particular, on Ubuntu and all other Debian based distributions, the apt software repositories are defined in the `/etc/apt/sources.list` file or in separate files under the `/etc/apt/sources.list.d/` directory. If you already installed ROS in your system, you should know that the first instruction of ROS tutorial is the following:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list'
```

This line uses the `echo` command to create a file called `ros-latest.list` filled with the address of the repository of ROS packages. At this point, the apt command should be able to see the packages contained in the ROS repository. More information about how to add other repository to APT will be provided when the installation of ROS is discussed.

1.2 Introduction to C++ programming

C++ is a general-purpose programming language created as an extension of the C programming language. One of C++'s strengths is that it can be used to write programs for nearly any processor. It is a high-level language: when you write a program in it, the shorthand are sufficiently expressive that you don't need to worry about the details of processor instructions. In addition, C++ does give access to some lower-level functionality than other languages (e.g. memory addresses). For this reason, C++ can be used both for high level robot programming (reasoning, planning and so on) and for robot low level control. The source code of a C++ program is written in a text file. The process in which a program goes from text files to processor instructions is depicted in Fig. 1.1. In particular, object files are intermediate files that represent an incomplete copy of the program: each source file only expresses a piece of the program, so when it is compiled into an object file, the object file has some markers indicating which missing pieces it depends on. The linker takes those object files and the compiled libraries of predefined code that they rely on, fills in all the gaps, and spits out the final program, which can then be run by the operating system. The compiler and linker are just regular programs. The step in the compilation process in which the compiler reads the file is called *parsing*. In C++, all these steps are performed ahead of time, before you start running a program. In some languages, they are done during the execution process, which takes time. This is one of the reasons C++ code runs far faster than code in many more recent languages.

```
1 /*
2  * First C++ program that says hello (hello.cpp)
3  */
4
5 #include <iostream>      //IO operations
6 using namespace std;
7
8 // Program entry point
9 int main() {
10   // Say Hello
11   cout << "hello, world" << endl;
12   // Terminate main()
13   return 0;
14 } // End of main function
```

The classical form of a single process C++ program is reported in the above

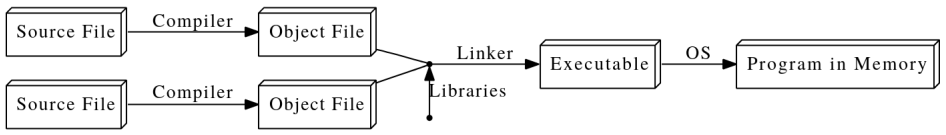

Figure 1.1: C++ compilation process

algorithm. This is nothing more than a simple program that prints the string "hello, world". In such code, `cout` function print out some piece of text to the screen, while to specify e certain context the namespace keyword is used: In C++, identifiers can be defined within a context called a namespace. When we want to access an identifier defined in a namespace, we tell the compiler to look for it in that namespace using the scope resolution operator (::). Here, we're telling the compiler to look for cout in the std namespace, in which many standard C++ identifiers are defined. If we do this, we can omit the std::prefix when writing `cout`.

C++ syntax is very similar to C and other compiled languages. In this section, we highlight some useful functionalities of C++ using external libraries.

Pointers

After declared a variable in C++, the computer associates its name with a particular location in memory where the value of the variable is stored. When in the code such variable is referred, the computer firstly look up the address the correspond to the variable name, then go to the location in memory to retrieve the value it contains.

Mainly, C++ allows us to perform these steps independently:

- &x evaluates to the address of x in memory
- *(&x)takes the address of x and dereferences it – it retrieves the value at that location in memory. *(&x)thus evaluates to the same thing as x.

Pointers allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself. In particular in C++ pointers are particularly useful to:

- Pass-by-reference variables is more efficient.
- Manipulate complex data structures efficiently, even if their data is scattered in different memory locations.
- Return multiple values from a single function.

To declare a pointer variable named `ptr` that points to an integer variable named `x`:

```
1 int *ptr = &x;
```

`int *ptr` declares the pointer to an integer value, which we are initializing to the address of x. We can have pointers to values of any type.

The following block of code shows a simple example in which pointers are used to pass variables by reference.

```
1 void squareByPtr(int *numPtr){
2   *numPtr= *numPtr**numPtr;
3 }
4
5 int main() {
6   int x=5;
7   squareByPtr(&x);
8   std::cout << x << std::endl; //Prints 25
9 }
```

The usage of the * and & operators with pointers/references can be confusing. The * operator is used in two different ways: when declaring a pointer, * is placed before the variable name to indicate that the variable being declared is a pointer - say, a pointer to an int or char, not an int or char value. Then, when using a pointer that has been set to point to some value, *is placed before the pointer name to dereference it, to access or set the value it points to. A similar distinction exists for &, which can be used either to indicate a reference datatype (`int &x;`), or to take the address of a variable (`int *ptr=&x;`).

Smart pointers (Shared pointers)

A smart pointer represents a class of objects aiming at simplify the usage of pointers. In particular, smart pointers prevent most situations of memory leaks by making the memory deallocation automatic and providing feature like automatic memory management or bounds checking. Such features are intended to reduce bugs caused by the misuse of pointers, while retaining efficiency.

In C++, a smart pointer is implemented as a template class that mimics, by means of operator overloading, the behaviors of a traditional (raw) pointer, (e.g. dereferencing, assignment) while providing additional memory management features.

Among different features of smart pointers, in this document we are interested in the `std::shared_ptr`. C++11 introduces `std::shared_ptr`, defined in the header <memory>. A `shared_ptr` is a container for a raw pointer. It maintains reference counting ownership of its contained pointer in cooperation with all copies of the `shared_ptr`. An object referenced by the contained raw pointer will be destroyed when and only when all copies of the `shared_ptr` have been destroyed.

```
1 //Allocates 1 integer and initialize it with value 5.
2 std::shared_ptr<int> p0(new int(5));
3 //Valid, allocates 5 integers.
4 std::shared_ptr<int[]> p1(new int[5]);
5 //Both now own the memory.
6 std::shared_ptr<int[]> p2 = p1;
7 //Memory still exists, due to p2.
8 p1.reset();
```

```
9 //Deletes the memory, since no one else owns the memory.
10 p2.reset();
```

One important feature of `shared_ptr` is that multiple threads can safely simultaneously access different `shared_ptr` that point to the same object. In particular, `shared_ptr` is considered when multiple owners should access to the same object in memory. A `shared_ptr` object effectively holds a pointer to the resource that it owns or holds a null pointer. A resource can be owned by more than one `shared_ptr` object; when the last `shared_ptr` object that owns a particular resource is destroyed, the resource is freed.

Classes

A class represents a user-defined data type which groups together related pieces of information. If you consider a geometric vector, a vector consists of 2 points: a start and a finish, each point itself has an x and y coordinate. We can create the following class to represent different type of vectors:

```
1 class Vector {
2 private:
3   double xStart;
4   double xEnd;
5   double yStart;
6   double yEnd;
7 };
```

Of course, similar results can be obtained using a simple data structure. To improve the class functionalities we should implement some methods in the vector class. Some functions are closely associated with a particular class, like the calculation of the norm of a vector:

```
1 class Vector {
2 public:
3   float get_norm();
4 private:
5   double xStart;
6   double xEnd;
7   double yStart;
8   double yEnd;
9 };
```

In addition, a class need a constructor: a method that is called when an instance is created. In our case, we can consider to initialize the member of the class (the points of the vector) when an instance of the class is created:

```
1 class Vector {
2 public:
3   Vector( float xstart_, float xend_,
4    float ystart_, float yend_);
5   float get_norm();
6 private:
7   double xStart;
```

```
8   double xEnd;
9   double yStart;
10  double yEnd;
11 };
```

In this way, to initialize a new vector and get its norm, we must implement the functions of its class:

```
1 Vector::Vector( float xstart_, float xend_,
2 float ystart_, float yend_) {
3   xStart = xstart_;
4   xEnd = xend_;
5   yStart = ystart_;
6   yEnd = yend_;
7 }
8
9
10 float Vector::get_norm() {
11   return sqrt(pow((xEnd - xStart),2) + pow((yEnd - yStart),2))
12 }
13
14
15 int main() {
16   Vector v(0, 0, 2, 2);
17   std::cout << v.get_norm() << std::endl;
18   return 0;
19 }
```

As for the class modifiers, we can choose three different type for the class members:

- *public*: members are accessible from outside the class
- *private*: members cannot be accessed (or viewed) from outside the class
- *protected*: members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

Data hiding is a software development technique specifically used in object-oriented programming to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes. To implement data hiding you can follow two simple rules: make all the data members private and create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member.

```
1   void Vector::set_xend( float xend ) {
2     xEnd = xend;
3   }
4
5   float Vector::get_xend() {
```

```
6     return xEnd;
7   }
```

1.2.1 Compilation using make

To generate executable files we need to compile one or more source files. One of the most common way is to use the `GNU make` program and (under linux) the `GCC` compiler. GCC, formerly for "GNU C Compiler", has grown over times to support many languages such as C (gcc), C++ (g++), Objective-C, Objective-C++, Java (gcj), etc.... It is now referred to as "GNU Compiler Collection". In this section we will discuss different tools to compile C++ programs. In particular, we introduce the command line tool of gcc to compile programs, then we discuss the `make` utility used to automatize the compilation process.

GCC

The GNU C and C++ compiler are called gcc and g++, respectively. Considering the following block of code:

```
1 // hello.c
2 #include <stdio.h>
3
4 int main() {
5   printf("Hello, world!\n");
6   return 0;
7 }
```

To compile the hello.c:

```
$ gcc hello.c
```

This command generates an executable that by default under linux OS is called `a.out`. This file can be executed from unix console using the following commands:

```
$ chmod a+x a.out
$ ./a.out
```

To specify the output filename, use -o option:

```
$ gcc -o hello.exe hello.c
```

Consider that `gcc` and `g++` share the same syntax, so to compile a C++ program, just used `g++` instead of `gcc`.

In case your program has multiple source files, like f1.cpp and f2.ccp, you could compile them in a single command:

```
$ g++ -o program f1.cpp f2.cpp
```

with the option `-c` you can compile multiple different object source files separately into object file, and link them together in the later stage. In this case, changes in one file does not require re-compilation of the other files:

```
$ g++ -c f1.cpp
$ g++ -c f2.cpp
$ g++ -o program f1.o f2.o
```

An important element of program compilation is represented by the shared and static libraries that can be used in your source code. In particular, a library is a collection of pre-compiled object files that can be linked into your programs via the linker. Examples are the system functions such as printf() and sqrt(). There are two types of external libraries: static library and shared library.

- A static library has file extension of ".a" (archive file). When your program is linked against a static library, the machine code of external functions used in your program is copied into the executable.
- A shared library has file extension of ".so" (shared objects). When your program is linked against a shared library, only a small table is created in the executable. Before the executable starts running, the operating system loads the machine code needed for the external functions - a process known as dynamic linking. Dynamic linking makes executable files smaller and saves disk space, because one copy of a library can be shared between multiple programs. The shared library codes can be upgraded without the need to recompile your program.

When compiling the program, the compiler needs the header files to compile the source codes; the linker needs the libraries to resolve external references from other object files or libraries. This could be a tedious step because the compiler and linker will not find the headers/libraries unless you set the appropriate options. For each of the headers used in your source (via #include directives), the compiler searches the so-called include-paths for these headers. The include-paths are specified via `-Idir` option (or environment variable `CPATH`). Since the header's filename is known (e.g., iostream.h, stdio.h), the compiler only needs the directories.

As for the linker, it searches the so-called library-paths for libraries needed to link the program into an executable. The library-path is specified via `-Ldir` option (or environment variable `LIBRARY_PATH`). In addition, you also have to specify the library name. In Unix, the library libxxx.a is specified via -lxxx option. In this context, the linker needs to know both the directories as well as the library names. Hence, two options need to be specified. To summarize, GCC uses the following environment variables:

- `PATH`: For searching the executables and run-time shared libraries (.so).

- `CPATH`: For searching the include-paths for headers. It is searched after paths specified in -I<dir> options. `C_INCLUDE_PATH` and `CPLUS_INCLUDE_PATH` can be used to specify C and C++ headers if the particular language was indicated in pre-processing.

- `LIBRARY_PATH`: For searching library-paths for link libraries. It is searched after paths specified in -L<dir> options.

Make

The `make` utility automates building process of executable from source code. `make` uses a so-called `makefile`, which contains rules on how to generate executable. Let's see a first example to build the `hello.c` program into executable using make utility.

Create the following file named "makefile" (without any file extension), which contains rules to build the executable, and save in the same directory as the source file.

```
1 all: hello
2
3 hello: hello.o
4 gcc -o hello hello.o
5
6 hello.o: hello.c
7 gcc -c hello.c
8
9 clean:
10   rm hello.o hello
```

To compile the program, run the `make` command in the same directory of the `makefile`.

```
$ make
```

makefile is typically used when you have a complex compilation structure for your program (multiple sources, libraries and so on). For this reason, several variables can be used to simply the content of the makefile. Automatic variables are set by make after a rule is matched. There include:

- `$@`: the target filename.

- `$*`: the target filename without the file extension.

- `$<`: the first prerequisite filename.

- `$^`: the filenames of all the prerequisites, separated by spaces, discard duplicates.

- `$+`: similar to `$^`, but includes duplicates.

- `$?`: the names of all prerequisites that are newer than the target, separated by spaces.

The previous makefile can be re-written as:

```
all: hello

# $@ matches the target;
# $< matches the first dependent
hello: hello.o
gcc -o $@ $<

hello.o: hello.c
gcc -c $<

clean:
  rm hello.o hello
```

CMake

Using `make` to compile complex and multi-platform projects could be not an easy task. For this reason, a generator of build system is often used to simplify the work of a software developer. The most famous generator of build-systems and also the one adopted by ROS is called `CMake`. `CMake` is a cross-platform free and open-source software tool for managing the build process of software using a compiler-independent method. It is used in conjunction with native build environments such as Make, Qt Creator, Ninja, Apple's Xcode, and Microsoft Visual Studio.

The build process with `CMake` takes place in two stages. Simple configuration files placed in each source directory (called `CMakeLists.txt` files) are used to generate standard build files (e.g., `makefiles`) which are used in the usual way. Another nice feature of `CMake` is that it generates a cache file that is designed to be used with a graphical editor. For example, when `CMake` runs, it locates include files, libraries, and executables, and may encounter optional build directives. This information is gathered into the cache, which may be changed by the user prior to the generation of the native build files.

Considering `hello.c` source file of the previous example, to compile it in `CMake` you should create a *txt* file named `CMakeLists.txt`:

```
# Specify the minimum version for CMake

cmake_minimum_required(VERSION 2.8)

# Project's name

project(hello)
# Set the output folder where your program will be created
set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}/bin)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR})
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR})
```

```
12
13 # The following folder will be included
14 include_directories("${PROJECT_SOURCE_DIR}")
```

In this file we used the following global variables:

- `CMAKE_BINARY_DIR`: if you are building in-source, this is the same as `CMAKE_SOURCE_DIR`, otherwise this is the top level directory of your build tree
- `CMAKE_SOURCE_DIR`: this is the directory, from which cmake was started, i.e. the top level source directory
- `EXECUTABLE_OUTPUT_PATH`: set this variable to specify a common place where CMake should put all executable files (instead of `CMAKE_CURRENT_BINARY_DIR`)
- `EXECUTABLE_OUTPUT_PATH`: set this variable to specify a common place where `CMake` should put all executable files (instead of `CMAKE_CURRENT_BINARY_DIR`), for example `SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)`. `LIBRARY_OUTPUT_PATH`: set this variable to specify a common place where CMake should put all libraries
- `PROJECT_SOURCE_DIR`: contains the full path to the root of your project source directory, i.e. to the nearest directory where `CMakeLists.txt` contains the `PROJECT()` command.

Finally, to compile the source code you should add this final line to your `CMakeLists.txt`:

```
1 add_executable(hello ${PROJECT_SOURCE_DIR}/hello.c)
```

Now you are ready to compile the `hello.c` source file. At this point, you will have the folder with the following files:

```
$ ls
$ CMakeLists.txt hello.c
```

The common way to compile with `CMake` tools, is to create a temporary folder in which all the compilation file are put. In this way, you can delete the temporary compilation file to share the entire folder of your project:

```
$ mkdir build && cd build
$ cmake ..
$ make
```

Sometimes, you should need to install custom libraries in your system. This is particularly useful when you want to use functions and headers in different source files. In `CMake` the `install` command is used. This command generates installation rules for a project. Rules specified by calls to this command within a source directory are executed in order during installation.

```
#libs
add_library(mylib SHARED ${LIB_SRC})
#install
install(TARGETS mylib DESTINATION /usr/lib)
install(FILES ${LIB_HEADER} DESTINATION /usr/include/mylib)
```

In this case, after the `make` command, to perform the install step you should type the following command:

```
$ sudo make install
```

1.3 git

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Today, Git is the most widely used modern version control system in the world.

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. So ideally, we can place any file in the computer on version control. A Version Control System (VCS) allows you to revert files back to a previous state, revert the entire project back to a previous state, review changes made over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also means that if you screw things up or lose files, you can generally recover easily. The most famous

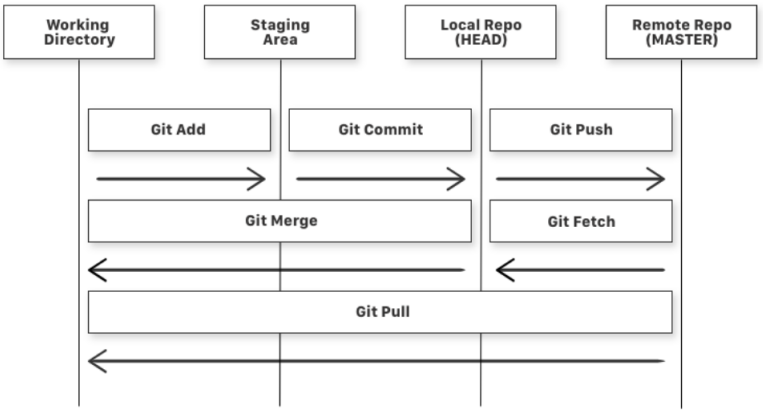

Figure 1.2: Git workflow

implementation of Git is github (`www.github.com`). In this section we introduce the basic concept of Git. First of all, the *Remote Repository* is where you send your changes when you want to share them with other people, and where you get their changes from, while the *Development Environment* is

what you have on your local machine: the three parts of it are your *Working Directory*, the *Staging Area* and the *Local Repository*. The workflow of git among these directories is shown in Fig. 1.2. First of all, what is a repository? A repository is nothing but a collection of source code. If you consider a file in your Working Directory, it can be in three possible states. The following commands are used to manage the workflow:

- `git add` is a command used to add a file that is in the working directory to the staging area.
- `git commit` is a command used to add all files that are staged to the local repository.
- `git push` is a command used to add all committed files in the local repository to the remote repository. So in the remote repository, all files and changes will be visible to anyone with access to the remote repository.
- `git fetch` is a command used to get files from the remote repository to the local repository but not into the working directory.
- `git merge` is a command used to get the files from the local repository into the working directory.
- `git pull` is command used to get files from the remote repository directly into the working directory. It is equivalent to a git fetch and a git merge .

As for the main commands to start with git, you can follow these commands:

- Create a new repository:

  ```
  $ git init
  ```

- Checkout a repository: create a working copy of a local repository by running the command

  ```
  $ git clone /path/to/repository
  ```

- You can propose changes (add it to the Index) using:

  ```
  $ git add <filename> or git add *
  ```

- To actually commit these changes use:

  ```
  $ git commit -m "Commit message"
  ```

- Your changes are now in the `HEAD` of your local working copy. To send those changes to your remote repository, execute:

  ```
  $ git push origin master
  ```

  Change `master` to whatever branch you want to push your changes to.

- If you have not cloned an existing repository and want to connect your repository to a remote server, you need to add it with:

  ```
  $ git remote add origin <server>
  ```

- branching: branches are used to develop features isolated from each other. The master branch is the "default" branch when you create a repository. Use other branches for development and merge them back to the master branch upon completion. To create a new branch named *branch* and switch to it using:

  ```
  $ git checkout -b branch
  ```

  To switch back to master

  ```
  $ git checkout master
  ```

  And delete the branch again

  ```
  $ git branch -d branch
  ```

  A branch is not available to others unless you push the branch to your remote repository

  ```
  $git push origin <branch>
  ```

- To update your local repository to the newest commit and fetch and merge remote changes in your working directory, execute:

  ```
  $ git pull
  ```

  while, to merge another branch into your active branch (e.g. master), use:

  ```
  $ git merge <branch>
  ```

Typically, a repository can be private or public. In the first case the owner and the developers of the project are the solely ones that can see the project and download its source code. In this context, you need to authenticate during the `clone` operations. This can be done in two different ways: with `https` authentication: just inserting user name and password of your account or using `SSH` authentication. Using the `SSH` protocol, you can connect and authenticate to remote servers and services. After you've checked for existing SSH keys, you can generate a new SSH key to use for authentication, then add it to the ssh-agent. To configure your GitHub account to use your new (or existing) SSH key, you'll also need to add it to your GitHub account.