Lesson 2: Robotics programming technologies

Jonathan Cacace

jonathan.cacace@unina.it

PRISMA Lab
Department of Electrical Engineering and Information Technology
University of Naples Federico II

www.prisma.unina.it

Robot Programming

Robot programming languages:

- Native robot programming language:
  - KRL: Kuka Robot Language
  - Kuka Sunrise
  - RoboDK
- With these languages develop intelligent robotic application performing complex tasks is not easy
- They are considered to be implement cyclical motion in a completely known environment
- Advanced robotic applications are programmed using standard programming languages

Robot Programming

System setup to program robots:

- OS: Unix based OS
- Programming language: C++
- Version control: git

Why C++?
C++: fast and versatile, it can be used both for high level reasoning and for low level control.

Linux OS

- Linux: family of open source Unix-like operating systems based on the Linux kernel (kernel first release 1991).
- Ubuntu: one of the most popular linux distributions
- 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.

Basic Linux commands

- **pwd** (print working directory): 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.

```
$ pwd
$ /home/jcacace
```

Basic Linux commands

- **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.

Basic Linux commands

- **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:

  ```
  $ 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.

Basic Linux commands

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

Basic Linux commands

- **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.

Basic Linux commands

- **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
  ```

Basic Linux commands

- **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.

Basic Linux commands

- **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.

Basic Linux commands

- **locate**: the locate command is used to find a file in a Linux system. This command is useful when you dont know where a file is saved or the actual name of the file. So, if you want a file that has the word "hello" gives the list of all the files in your Linux system containing the word hello when you type in locate -i hello. 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

Basic Linux commands

- **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 Jonathan" > new.txt
```

Basic Linux commands

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

Basic Linux commands

- **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.

Basic Linux commands

- **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 alsa-base.conf, which needs root permissions, you can use the command:

  ```
  $ sudo nano alsa-base.conf
  ```

Basic Linux commands

- **chmod**: use chmod to make a file executable and to change the permissions granted to it in Linux. To make a file executable, you can use the command

  ```
  $ chmod +x numbers.py
  ```

  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.

  ```
  $ chmod 777 /det/ttyUSB0
  ```

Basic Linux commands

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

Basic Linux concepts

filesystem: layout of Linux: directory tree, which starts at the / directory, also known as the root directory. Directly underneath / are important sub-directories: /bin, /etc, /dev, and /usr, among others. These directories in turn contain other directories which contain system configuration files, programs, and so on.

Basic Linux concepts

filesystem: Each user has a home directory, which is the directory dedicated for the 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).

Basic Linux concepts

.bashrc: the linux shell is called bash. 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`.

Basic Linux concepts

Environment variables: set of dynamic named values stored within the system that are used by applications launched in shells. Environment variables allow you to customize how the system works and the behavior of the applications on the system. 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
```

Basic Linux concepts

Environment variables: 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
```

Basic Linux concepts

APT package manager: In ubuntu 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. In order to install a new software you should use the following syntax:

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

Basic Linux concepts

APT package manager: 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.

Basic Linux concepts

APT:

- **update** is used to re-synchronize the package index files from their sources. The lists of available packages are fetched from the location(s) specified in `/etc/apt/sources.list`.
- **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`.

Basic Linux concepts

To add additional software list to the apt repository you should edit the files included in `/etc/apt` directory. 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.

Basic Linux concepts

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-lat
```

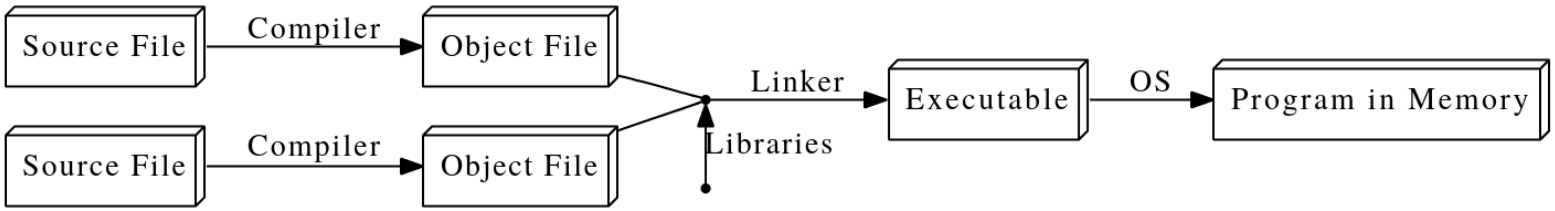
This line uses the echo command to create a file called `ros-latest.list` filled with the address of the repository of ROS packages. Now, the apt command should be able to see the packages contained in the ROS repository.

C++ compilation

C++:

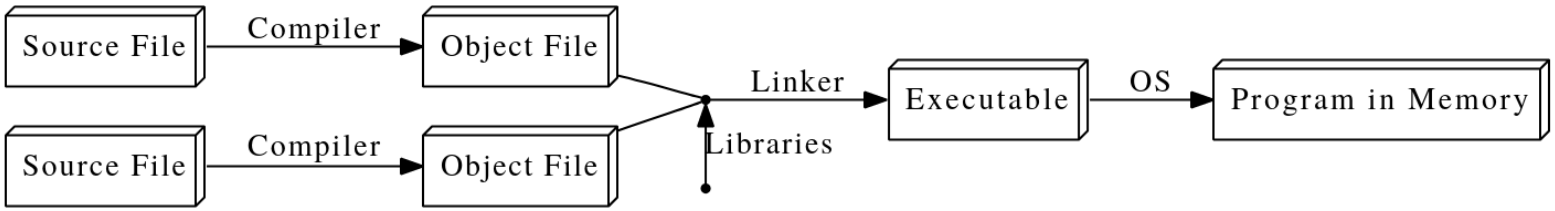
- General-purpose programming language
- Can be used to write programs for nearly any processor
- High-level language...
- ...give access to some lower-level functionality than other languages (e.g. memory addresses)
- both for high level robot programming (reasoning, planning and so on) and for robot low level control

Compilation:



The source code of a C++ program is written in a text file. Object files are intermediate files that represent an incomplete copy of the program. They have some markers indicating which missing pieces of code it depends on.

C++ compilation



- 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
- In C++, all these steps are performed ahead of time, before you start running a program. This is one of the reasons C++ code runs far faster than code in many more recent languages

Sample program:

```
1  #include <iostream>      //IO operations
2  using namespace std;
3  // Program entry point
4  int main() {
5      // Say Hello
6      cout << "hello, world" << endl;
7      // Terminate main()
8      return 0;
9  }
10 // End of main function
```

Pointers

In C++, after declared a variable...

- The computer associates its name with a particular location in memory where the value of the variable is stored
- When this 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

Pointers

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. Thus, *(&x) evaluates to the same thing as x.

Pointers

Pointers allow us to manipulate data much more flexibly. Manipulating the memory addresses of data can be more efficient than manipulating the data itself:

- 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.

Pointers

`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.

```
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  }
```

Pointers

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.
- 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

Pointers could be difficult to use (memory leaking)!
A **smart pointer** represents a class of objects aiming at simplify the usage of pointers:

- Smart pointer is implemented as a template class that mimics, by means of operator overloading, the behaviors of a traditional (raw) pointer
- Prevent most situations of memory leaks by making the memory deallocation automatic
- Provides feature like automatic memory management or bounds checking

Smart pointers

ROS uses smart pointers (we will see how!).

- ROS uses the `std::shared_ptr`: 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.

Smart pointers

ROS uses smart pointers (we will see how!).

- With `shared_ptr` multiple threads can safely simultaneously access different `shared_ptr` that point to the same object.
- `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.

Smart pointers

```
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();
```

Classes

Class represent user-defined data types grouping together related pieces of information.
Example: *geometric vector*.

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

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  };
```

Classes

A class needs of 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 };
```

Initialize and array and get its norm:

```
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  float Vector::get_norm() {
10     return sqrt(pow((xEnd - xStart),2) + pow((yEnd -
           yStart),2))
11 }
```

Initialize and array and get its norm:

```
1
2  int main() {
3      Vector v(0, 0, 2, 2);
4      std::cout << v.get_norm() << std::endl;
5      return 0;
6  }
```

Class modifiers

We can choose three different modifiers for 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

How to choose modifiers?
Data hiding: hide internal data members ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes:

- Make all the data members private
- Create public setter and getter functions for each data member

```
1  void Vector::set_xend( float xend ) {
2      xEnd = xend;
3  }
4  float Vector::get_xend() {
5      return xEnd;
6  }
```

C++ Compilation

To generate executable files we need to compile one or more source files.

- GCC (GNU Compiler Collection) has grown over times to support many languages such as C, C++, Objective-C, Objective-C++, Java, etc, ...
- GNU C compiler: gcc
- GNU C++ compiler: g++

Sample program `hello.c`.

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

```
$ gcc hello.c
$ chmod a+x a.out
$ ./a.out
```

To specify the output filename, use -o option:

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

C++ Compilation

A library is a collection of pre-compiled object files that can be linked into your programs via the linker:

- A **static library** has file extension of ".a". 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
  - Executable files smaller
  - No need to recompile your program

C++ Compilation

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, 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.

C++ Compilation

The compiler and linker will not find the headers/libraries unless you set the appropriate options!!

- The **linker** searches the so-called library-paths for libraries needed to link the program into an executable and can be specified via `-Ldir` option (or environment variable `LIBRARY_PATH`)
- You also have to specify the library name. In Unix, the library `libxxx.a` is specified via `-lxxx` option.
- The linker needs to know both the directories as well as the library names.

C++ Compilation

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

While GCC is a compiler, MAKE is a building tool that can use GCC:

- 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.

MAKE

Use a makefile to compile the `hello.c` source code:

```
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
```

MAKE

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)
- 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:

```
1  all: hello
2
3  # $@ matches the target;
4  # $< matches the first dependent
5  hello: hello.o
6      gcc -o $@ $<
7
8  hello.o: hello.c
9      gcc -c $<
10
11 clean:
12     rm hello.o hello
```

There is a bit more...
Using `make` to compile complex projects could be difficult:
CMake automatizes the generation of the `makefile`.

- GCC: Compiler
- MAKE: building tool
- CMake: generator of build-systems

CMake

The build process with `CMake` takes place in two stages:

- Generation of `makefile` by means of configuration (`CMakeLists.txt`).

  ```
  $ cmake
  ```

- Compilation using the `makefile` as shown before.

  ```
  $ make
  ```

`CMakeLists.txt` to compile `hello.c` makefile:

```
1  # Specify the minimum version for CMake
2  cmake_minimum_required(VERSION 2.8)
3  # Project's name
4  project(hello)
5  # Set the output folder where your program will be
       created
6  set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}/bin)
7  set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR})
8  set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR})
9  # The following folder will be included
10 include_directories("${PROJECT_SOURCE_DIR}")
```

In this file we used the following global variables:

- `CMAKE_BINARY_DIR`: binary sources
- `CMAKE_SOURCE_DIR`: source file directory
- `EXECUTABLE_OUTPUT_PATH`: bin output path
  `LIBRARY_OUTPUT_PATH`: lib output path
- `PROJECT_SOURCE_DIR`: project source directory

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

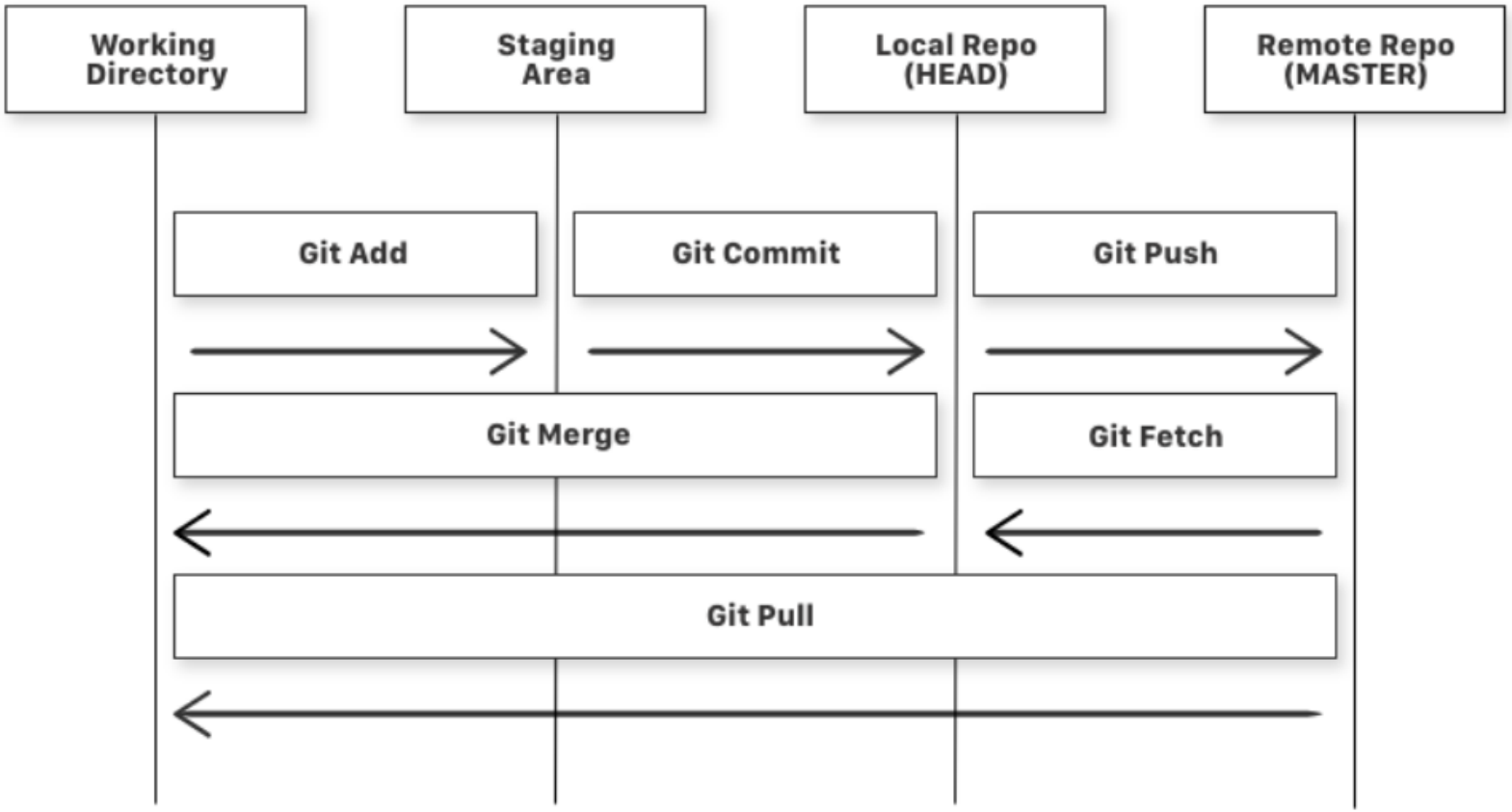
```
1 add_executable(hello ${PROJECT_SOURCE_DIR}/hello.c)
```

Version Control System:

- Is a system that records changes to a file or set of files over time so that you can recall specific versions later
- It 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
- Github (`www.github.com`) is one of the most used VCS based on GIT

What is a repository? A repository is nothing but a collection of source code contained in your *Development Environment*:

- *Working Directory*: files in your working directory
- *Staging Area*: a temp area that git add is placed into.
- *Local Repository*: the repository on your machine
- *Remote Repository*: the repository on the server


Working Directory
Staging Area
Local Repo (HEAD)
Remote Repo (MASTER)
Git Add
Git Commit
Git Push
Git Merge
Git Fetch
Git Pull

Start with repository:

- Create a new repository:

  ```
  $ git init
  ```

- If you want to connect a (local) repository to a remote server, you need to add it with:

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

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

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

Manage the repository:

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

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

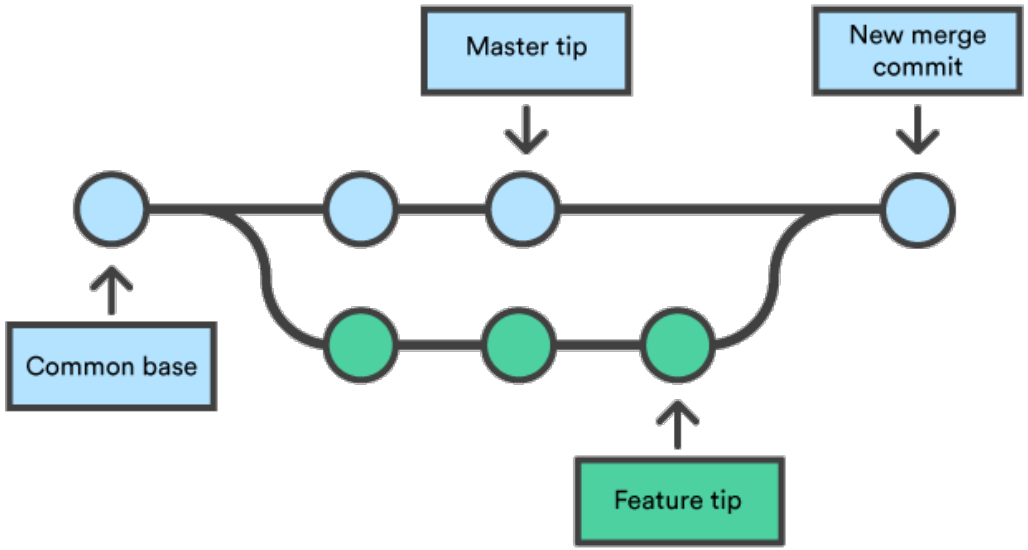
- To actually commit these changes use:

  ```
  $ 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
  ```

Branching




```
$ git checkout -b branch: to switch back to master
$ git checkout master: and delete the branch again
$ git branch -d branch: and delete the branch again
```