The build pipeline

The build pipeline is as follows: **Preprocess** -> **Compile** -> **Link**. So at least three questions arise, they are:

- What is **Preprocess**?
- What is **Compile**?
- What is **Link**?

Let's look at what is **Preprocess**. In order to do preprocessing we need something to preprocess, so what this "thing"? We preprocess **source files**. Source files are just the files we type our code in, the ones which end with ".cpp". Preprocessing "looks" in the source files for:

- File inclusion - #include <iostream>, *note: this is called an **#include** directive below.*
- Macros - #define MAX_LENGTH 1000
- Other directives which do not concern us for now

Long story short, preprocessing takes files as input, transforms them using some rules and produces output files which are used as input for the compilator.

For example, when the **preprocessor** finds an **#include** directive it finds the file which is included and slaps the whole content of that file on that line on which it found the **#include** directive.

Another example of preprocessing is how macros as handled. Let's take look at this example:

Before preprocessing:

```
#include <iostream>

// Eight-ball starts with 16 balls, strange right?
#define MAX_BALLS 16

int main()
{
  int currentBallsCount = MAX_BALLS;

  // Other code...
  return 0;
}
```

After preprocessing:

```
// All the code from iostream header file. Like a lot of lines..

int main()
{
  int currentBallsCount = 16;

  // Other code...
  return 0;
}
```

The result of preprocessing are pure C++ source files.

Alright, lets continue onto **compiling.** The compilator gets preprocessed source files and one by one translates them to **object files**, **object file** is just a sequence of instructions written in computer language that the computer can execute or some other helpful information. At this point in time errors are shown if the code has problems in it.

The result of compiling are object files (i.e. files in binary format)

Onto **linking!** The **Linker** takes all the files that the compiler generates and resolves any missing symbols(links). What this means is this: If you have many files, and each file references other files in order to use the functionality that they provide, you need to know exactly where are the definitions and declarations of the functionality that you use. The linker takes care of resolving this issue.

The result of linking is either library file or an executable file.

- You can think of **library files** as the implementation of the header files. A nice analogy is phone number and a person – the header file is a phone number, you know it so you can call the person, but talking to the person and getting to know him requires *him* **–** the library file.
- **Executable file** are instructions to for the computer to execute stored in binary format.

Have a nice day! 😊