

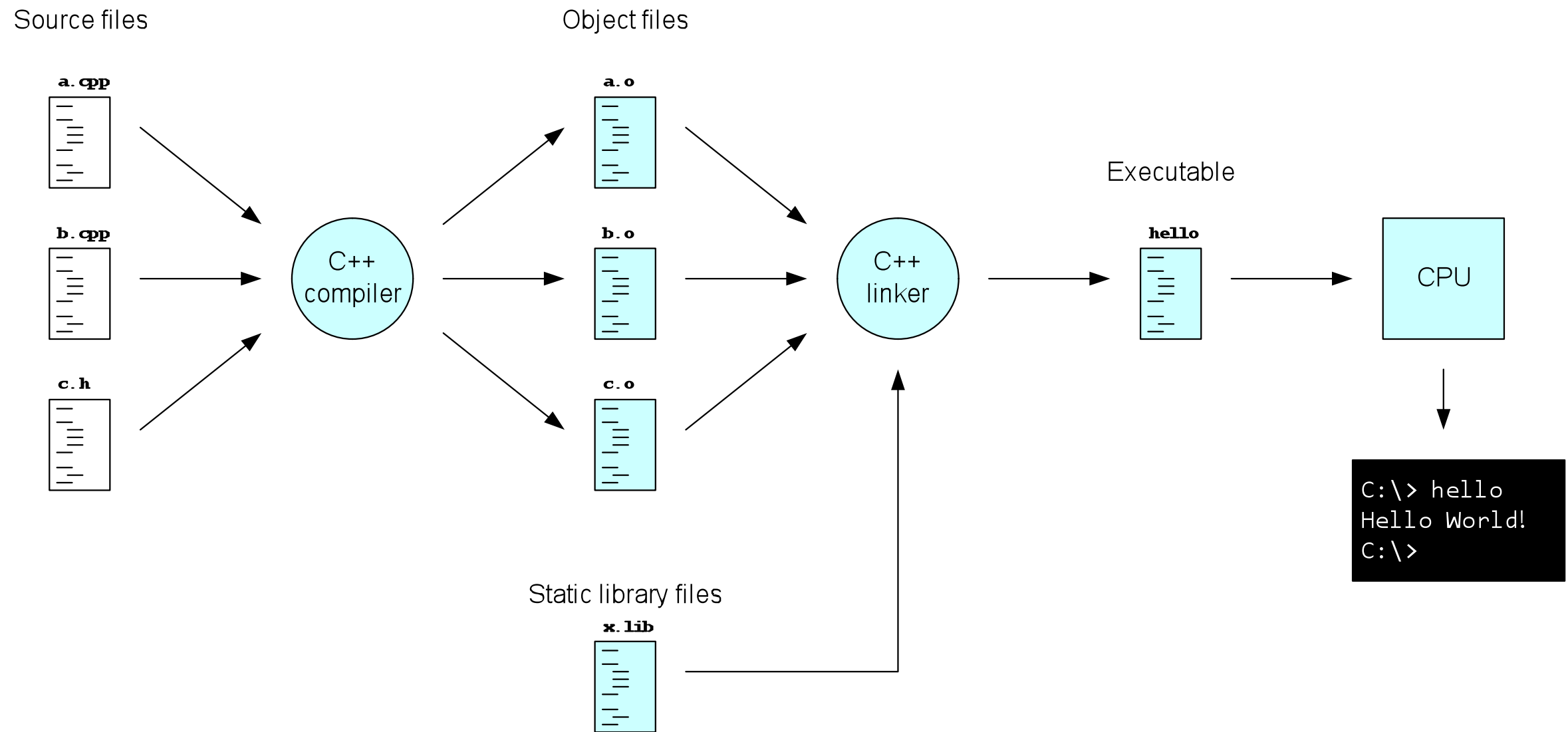
Embedded Software

Programming in Linux

Agenda

- Compilation
- Host and Target
- SW Development for Embedded targets - Howto make it

Compilation



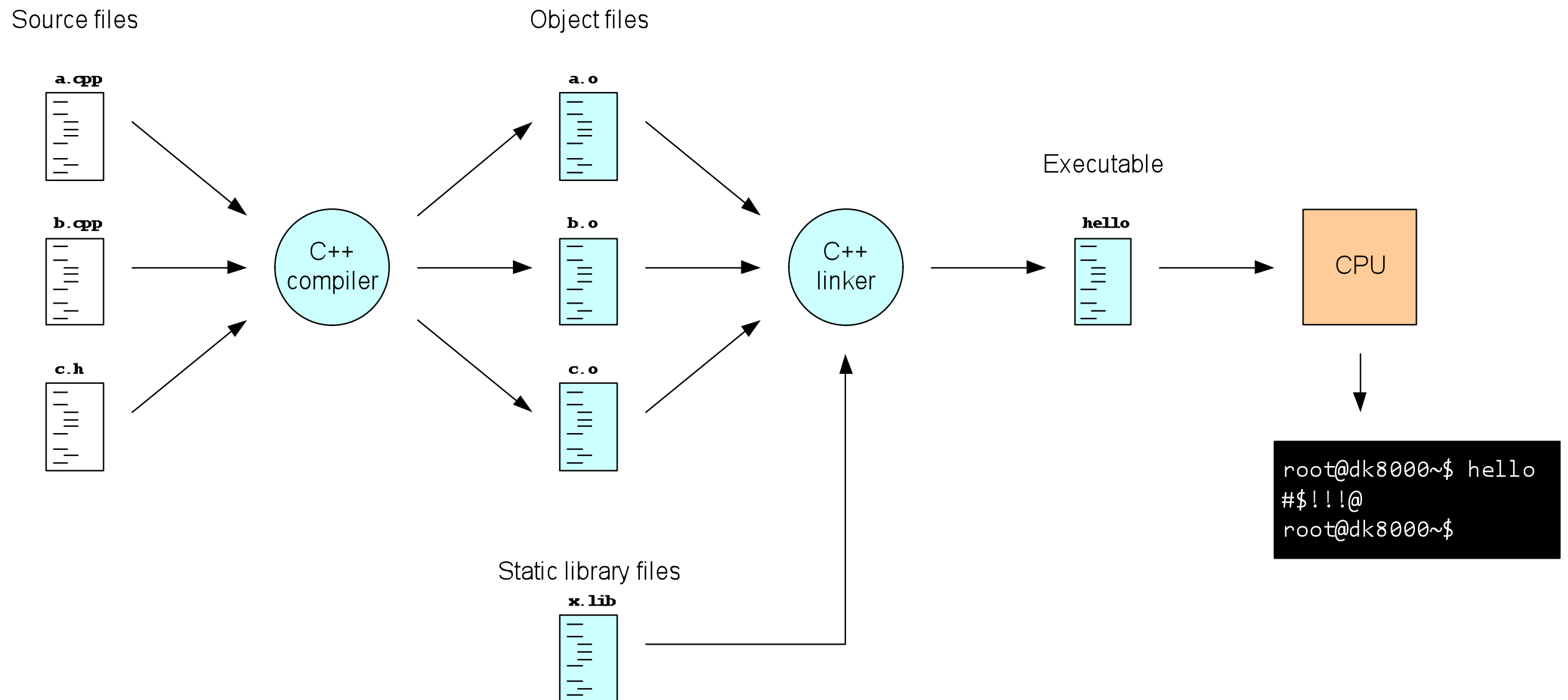
Host and target

- Any executable (“binary”) is generated on a *host* for a specific *target*

```
stud@ubuntu:~/$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV) ...
stud@ubuntu:~/$
```

- Until now, host == target
- What if you wish to generate programs for an **embedded** target, e.g. an ARM processor?
 - ▶ An ARM processor does not understand an x86 binary!

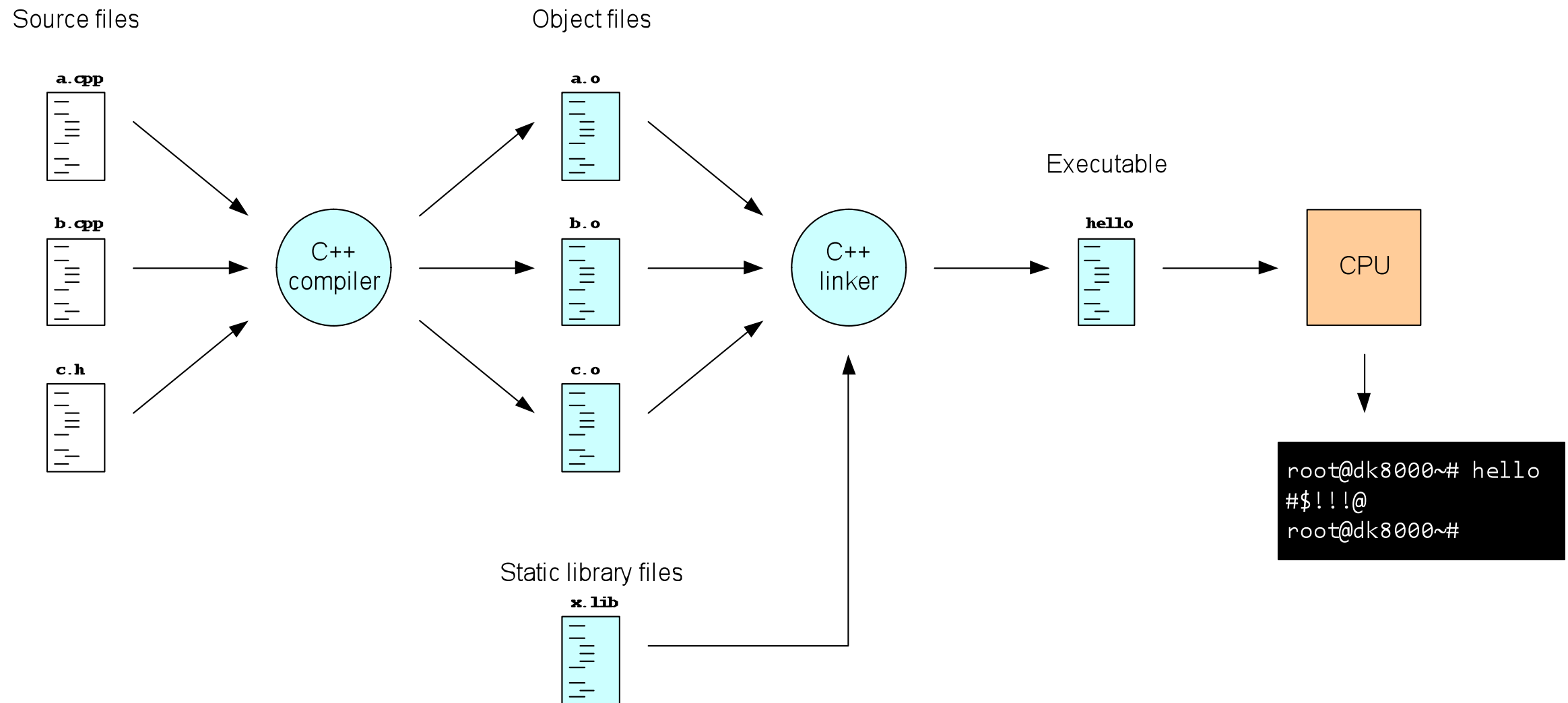
Host and target



SW development for embedded targets

- how to *make* it

- To make SW for an embedded target, you use *cross compilation*
 - You compile the program **on** the host, but **for** the target



SW development for embedded targets

- how to *make* it

- For the DevKit8000 (which is an ARM target) we use the Angstrom C/C++ compiler suite of tools

```
stud@ubuntu: ls /opt/poky/1.8/sysroot/...$  
arm-poky-linux-gnueabi-addr2line  arm-poky-linux-gnueabi-gprof  
arm-poky-linux-gnueabi-ar         arm-poky-linux-gnueabi-ld  
arm-poky-linux-gnueabi-as         arm-poky-linux-gnueabi-nm  
arm-poky-linux-gnueabi-c++        arm-poky-linux-gnueabi-objcopy  
arm-poky-linux-gnueabi-c++filt    arm-poky-linux-gnueabi-objdump  
arm-poky-linux-gnueabi-cpp        arm-poky-linux-gnueabi-ranlib  
arm-poky-linux-gnueabi-g++        arm-poky-linux-gnueabi-readelf  
arm-poky-linux-gnueabi-gcc        arm-poky-linux-gnueabi-size  
arm-poky-linux-gnueabi-gcc-4.9.2 arm-poky-linux-gnueabi-sprite  
arm-poky-linux-gnueabi-gcov       arm-poky-linux-gnueabi-strings  
arm-poky-linux-gnueabi-gdb        arm-poky-linux-gnueabi-strip  
arm-poky-linux-gnueabi-gdbtui
```

SW development for embedded targets

- how to *make* it

- Unfortunately the correct invocation is

- ```
arm-poky-linux-gnueabi-gcc -march=armv7-a -marm -mthumb-interwork
-mfloat-abi=hard -mfpu=neon -mtune=cortex-a8
-sysroot=/opt/poky/1.8/sysroots/cortexa8hf-vfp-neon-poky-linux-gnueabi
```

- To simplify matters a simple alias has been made

- ▶ For C

```
arm-devkit-gcc
```

- ▶ For C++

```
arm-devkit-g++
```



# SW development for embedded targets

## - how to *make* it

---

- To invoke the compiler we specify the name of the compiler

- ▶ For host:

```
stud@ubuntu:~$ g++ -o hello_host hello.cpp
stud@ubuntu:~$ file hello_host
hello_host: ELF 32-bit LSB executable, Intel 80386, ...
stud@ubuntu:~$./hello_host
Hello world!
stud@ubuntu:~$
```

- ▶ For target:

```
stud@ubuntu:~$ arm-devkit-g++ -o hello_tgt hello.cpp
stud@ubuntu:~$ file hello_tgt
hello_tgt: ELF 32-bit LSB executable, ARM, version 1 (SYSV), ...
stud@ubuntu:~$./hello_tgt
bash: ./hello_tgt: cannot execute binary file
stud@ubuntu:~$
```

# Native and cross compiler - Include (&lib) handling?

---

- How does a compiler know which include to use?

```
#include <iostream>

int main(int argc, char* argv[])
{
 std::cout << "Hello World" << std::endl;
}
```

Where and how is this one found?  
Magic?

**Its decided by the sysroot path**

**Usually compiled in - not in our case thus the reason for the alias**

**Can be discovered by g++ -v (or cross compiler edition -v)**

**This includes libraries**

*The **sysroot** path is thus responsible for ensuring that the compilers don't miks up files*

# SW development for embedded targets

## - how to *make* it

---

- Testing embedded SW can be very difficult – ***why?***
  - ▶ Very few resources (CPU, memory, keyboard, monitor, ...) for testing
- To the extent possible, you can use a simulated environment
  - ▶ If your target and host runs Linux, then it is *relatively* easy – compile and test on your host, then recompile for target
- Anything you need to think of in the simulated environment?
  - ▶ Time
  - ▶ Peripheral
  - ▶ Memory and CPU constraints
  - ▶ ...
- So...what can you test?