C++ PROGRAMMING IN LINUX

AARHUS
UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

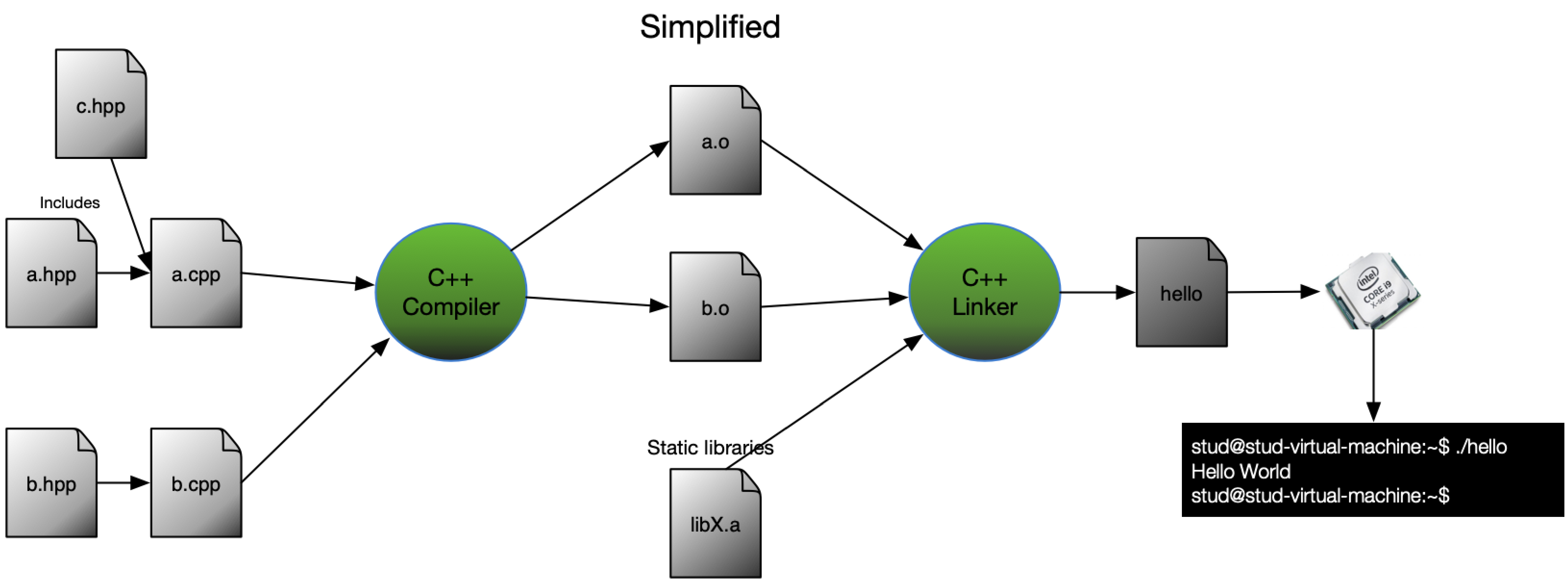
AGENDA

- Linux
  - Code compilation - whats to happen
    - source, object & library code to executable
  - Build tool - why?
  - Make and how it works
- Cross compliling
  - To target
- SW development for embedded targets
  - How to make it

CODE COMPILATION - WHATS TO HAPPEN

CODE COMPILATION - WHATS TO HAPPEN

FROM CODE TO EXECUTABLE



BUILD TOOL - WHY?

BUILD TOOL - WHY?

- Provides the ability to repeat and guarantee builds between builds
  - Reproducible results
  - Simplification of complexity
- What complexity???
  - Is it not simple???

BUILD TOOL - WHY?

- Simple invocation of g++
  - Generates executable from source in one go
  - `-o name` - where *name* is the name of the output file

SOURCE - HELLO.CPP

```
#include<iostream>
using namespace std;

int main()
{
     cout << "Hello World!" << endl;
     return 0;
}
```

COMPILE & LINK + RUN

```
$ g++ -o hello hello.cpp
$ ./hello
Hello world!
```

BUILD TOOL - WHY?

- Simple invocation of g++
  - Generates executable from source in one go
  - `-o name` - where *name* is the name of the output file

SOURCE - HELLO.CPP

```cpp
#include<iostream>
using namespace std;

int main()
{
    cout << ”Hello World!” << endl;
    return 0;
}
```

COMPILE & LINK + RUN

```
$ g++ -o hello hello.cpp
$ ./hello
Hello world!
```

Build tool???

BUILD TOOL - WHY?

- Simple invocation of g++
  - Generates executable from source in one go
  - `-o name` - where *name* is the name of the output file

SOURCE - HELLO.CPP

```
#include<iostream>
using namespace std;

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

COMPILE & LINK + RUN

```
$ g++ -o hello hello.cpp
$ ./hello
Hello world!
```

Build tool???

- What happens if 100 files needs to be compiled?
- What if only *one* file was changed?
  - You would still recompile the rest - 99 files :-(

BUILD TOOL - WHY?

- Instead of one big compilation
  - Many small
- Compiles each source to an object file (remember diagram)
- Links all objects (no libs here) to an exe file

```
$ g++ -c part1.cpp
$ g++ -c part2.cpp
$ g++ -o hello part1.o part2.o
$ ./hello
Hello world!
```

BUILD TOOL - WHY?

- Instead of one big compilation
  - Many small
- Compiles each source to an object file (remember diagram)
- Links all objects (no libs here) to an exe file

```
$ g++ -c part1.cpp
$ g++ -c part2.cpp
$ g++ -o hello part1.o part2.o
$ ./hello
Hello world!
```

- Much better
  - Only sources that have changed need to recompile
  - Linking always needed
- By hand?
  - What if we have 100 source files? :-/

BUILD TOOL - WHY?

- A compilation process can be controlled
  - compiler options
    - Optimisation
      - -O0 - None
      - -O1 - Level 1
      - -O2 - Level 2
      - -O3 - Level 3
    - Debug info
      - -ggdb
    - Thread library
      - -pthread
    - Release / Debug build
    - *And MANY MANY more*

DEBUG BUILD

```
$ g++ -Iinc -Wall -ggdb -O0 -pedantic -c part1.cpp
$ g++ -Iinc -Wall -ggdb -O0 -pedantic -c part2.cpp
$ g++ -o hello part1.o part2.o
$ ./hello
Hello world!
```

RELEASE BUILD

```
$ g++ -Iinc -Wall -O2 -pedantic -c part1.cpp
$ g++ -Iinc -Wall -O2 -pedantic -c part2.cpp
$ g++ -o hello part1.o part2.o
$ ./hello
Hello world!
```

- Please - ***A build tool :-)***

MAKE!

THE 3 MAIN PARTS OF A MAKEFILE

- Receipt - describes how something is build
  - target1
    - A 'file' that should be created by the commands
  - prereq1 prereq2
    - Prerequisites are files that are needed to build 'target1'
    - If any of these change - e.g. are newer than target then target ('target1') is rebuild
  - command1
    - May be one command - e.g. one line
    - May be many lines (command2 & command3 ...)
    - Important: The file 'target1' must have been created when all commands have been run

```
# Receipt for target1
target1: prereq1 prereq2 ...
	command1
	command2
	...

# Receipt for target2
target2: prereq1 prereq2 ...
	command1
	command2
	...
```

MAKE!

RUNNING MAKE

- To *make* a target like 'target1'

```
$ make target1
```

```
target1: prereq1 prereq2 ...
	command1
	command2
	...

target2: prereq1 prereq2 ...
	command1
	command2
	...
```

MAKE!

MINIMAL MAKEFILE

- Minimal makefile for "Hello World" program
  - What will we see on the console when we run "make"
    - Why and why is it important to understand?

```
hello: hello.o
    g++ -o hello hello.o

hello.o: hello.cpp
    g++ -c hello.cpp
```

MAKE!

MINIMAL MAKEFILE

- Minimal makefile for "Hello World" program
  - What will we see on the console when we run "make"
    - Why and why is it important to understand?

```
01 hello: hello.o
02     g++ -o hello hello.o
03
04 hello.o: hello.cpp
05     g++ -c hello.cpp
```

```
$ make
g++ -c hello.cpp            # Line 5 from makefile
g++ -o hello hello.o        # Line 2 from makefile
$ ./hello
Hello World!
$
```

MAKE!

MORE COMPLEX EXAMPLE

```
edit: main.o kbd.o command.o display.o
    g++ -o edit main.o kbd.o command.o display.o

main.o: main.cpp defs.h
    g++ -c main.cpp

kbd.o: kbd.cpp defs.h command.h
    g++ -c kbd.cpp

command.o: command.cpp defs.h command.h
    g++ -c command.cpp

display.o: display.cpp defs.h buffer.h
    g++ -c display.cpp
```

MAKE!

MORE COMPLEX EXAMPLE

```
edit: main.o kbd.o command.o display.o
    g++ -o edit main.o kbd.o command.o display.o

main.o: main.cpp defs.h
    g++ -c main.cpp

kbd.o: kbd.cpp defs.h command.h
    g++ -c kbd.cpp

command.o: command.cpp defs.h command.h
    g++ -c command.cpp

display.o: display.cpp defs.h buffer.h
    g++ -c display.cpp
```

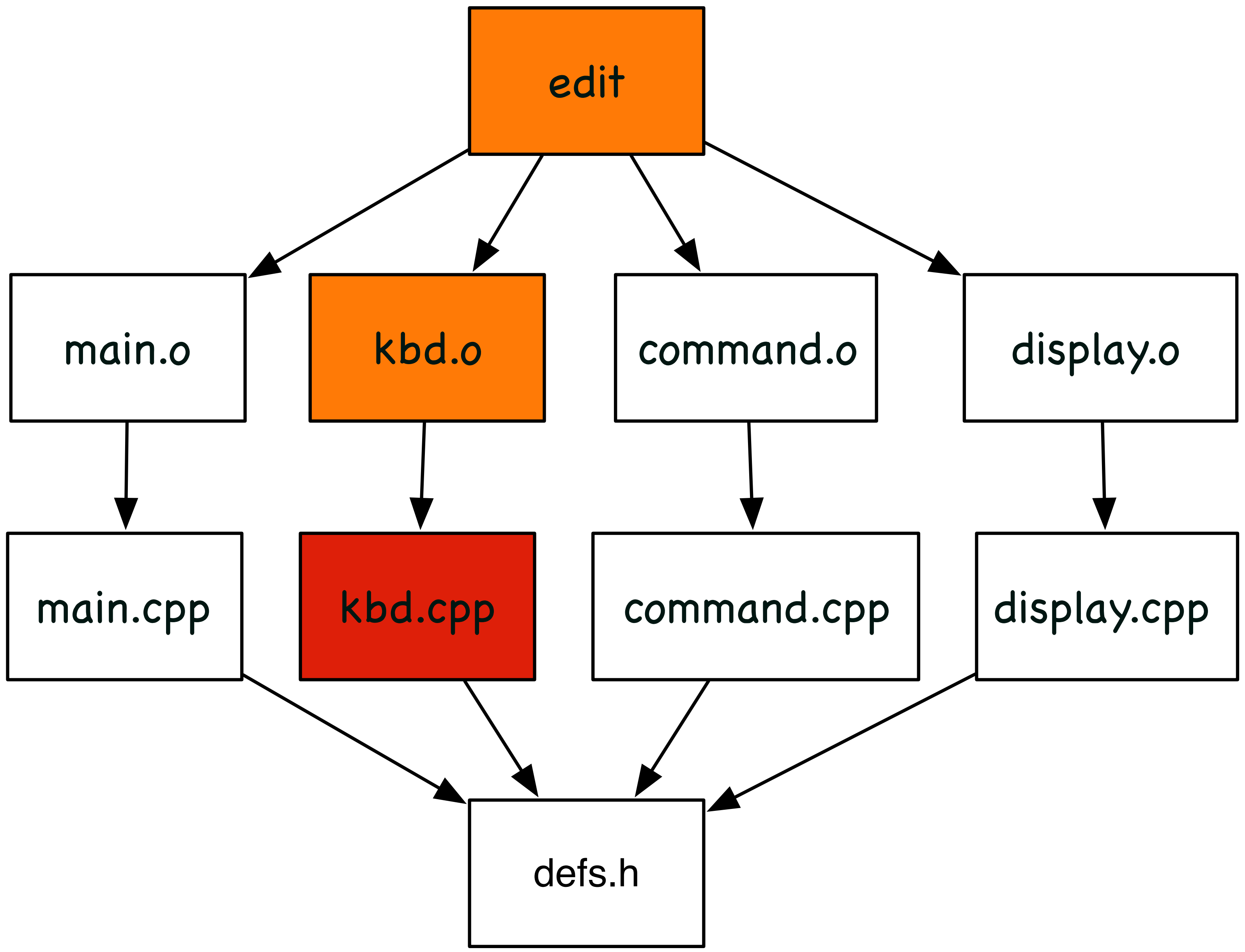
What happens if we do?

```
$ make
$ make edit
$ make display.o
```

MAKE!

HOW MAKE WORKS

- Make builds a dependency tree
  - Used to find out what to rebuild if file changes
- Fx. *kbd.cpp* changes a rebuild of
  - *kbd.o*
  - *edit* - the execuble
- From the tree its seen that changes to *defs.h* => total rebuild!



MAKE!

VARIABLES AND PATTERN MATCHING

- Variables
  - SOURCES contains names of source files
  - OBJECTS is same as SOURCES with .cpp replaced with .o
  - Important
    - Variables contains a string, not the file itself
    - Manipulations on vars are string manipulations

```
SOURCES=main.cpp kbd.cpp cmd.cpp disp.cpp
OBJECTS=${SOURCES:.cpp=.o}
EXECUTABLE=edit
CXX=g++
CXXFLAGS=-ggdb -I.


%.o: %.cpp
	${CXX} -c -o $@ $^ ${CXXFLAGS}

${EXECUTABLE}: ${OBJECTS}
	${CXX} -o $@ $^
```

MAKE!

VARIABLES AND PATTERN MATCHING

- Pattern matching
  - %.o
    - IF a dependency ends in .o then this receipt can be used
    - Ex. dependency like myfile.o generates the receipt below

```
myfile.o: myfile.cpp
    g++ -c -o myfile.o myfile.cpp -ggdb -I.
```

```
SOURCES=main.cpp kbd.cpp cmd.cpp disp.cpp
OBJECTS=${SOURCES:.cpp=.o}
EXECUTABLE=edit
CXX=g++
CXXFLAGS=-ggdb -I.


%.o: %.cpp
    ${CXX} -c -o $@ $^ ${CXXFLAGS}

${EXECUTABLE}: ${OBJECTS}
    ${CXX} -o $@ $^
```

MAKE!

STANDARD TARGETS

- Normally targets == files
- There are exceptions
  - clean
  - install
  - run

```
SOURCES = main.cpp kbd.cpp cmd.cpp disp.cpp
OBJECTS = ${SOURCES:.cpp=.o}
EXECUTABLE=edit
INSTALL_DIR=/home/me/exec
CXX=g++


...

${EXECUTABLE}: ${OBJECTS}
	${CXX} -o $< $@
...

clean:
	rm ${EXECUTABLE} ${OBJECTS}

install:
	cp ${EXECUTABLE} ${INSTALL_DIR}

run:
	 ${INSTALL_DIR}/${EXECUTABLE}
```

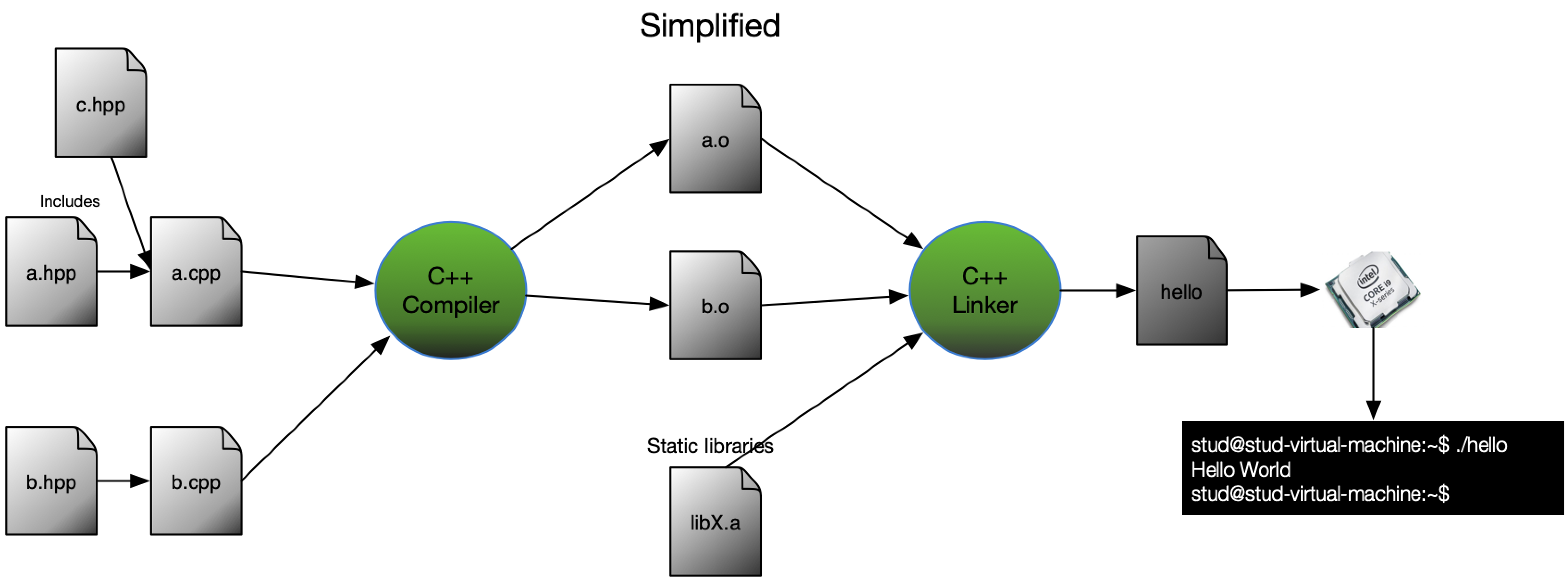
CROSS COMPLILING

CROSS COMPLILING

- Build process - recap
- How do you cross-compile
- Sanity check

BUILD PROCESS - RECAP

- Whether it be for host or target - the *build process* is the same



HOW DO YOU CROSS-COMPILE

- Use the correct compiler
  - Host
    - gcc/g++
  - Target
    - arm-poky-linux-gnueabi-gcc/arm-poky-linux-gnueabi-c++
- Toolchain tools include numerous different programs

```
stud@ubuntu: ls /opt/poky/2.4.1/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-gcov       arm-poky-linux-gnueabi-strings
arm-poky-linux-gnueabi-gdb        arm-poky-linux-gnueabi-strip
arm-poky-linux-gnueabi-gdbtui
```

HOW DO YOU CROSS-COMPILE

- arm compiler need numerous options
  - arm-poky-linux-gnueabi-g++ -march=armv6 -mfpu=vfp -mfloat-abi=hard -mtune=arm1176jzf-s -mfpu=vfp —sysroot=$SDKTARGETSYSROOT …
- To simplify, two scripts have been made
  - For the C compiler
    - arm-rpizw-gcc
  - For the C++ compiler
    - arm-rpizw-g++

HOW DO YOU CROSS-COMPILE

- Cross compiling your very Hello World
- Compile
  - arm-rpizw-g++ -o hello hello.cpp

HELLO.CPP

```cpp
#include

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

TERMINAL

```
stud@ubuntu:~$ arm-rpizw-g++ -o hello hello.cpp
stud@ubuntu:~$ ./hello
```

SANITY CHECK

- Compile or have a compiled file
  - What type is it
  - For which platform

Host

```
stud@ubuntu:~$ g++ -o hello_host hello.cpp
stud@ubuntu:~$ file hello_host
hello_host: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), …
stud@ubuntu:~$ ./hello_host
Hello world!
stud@ubuntu:~$
```

Target

```
stud@ubuntu:~$ arm-rpizw-g++ -o hello_tgt hello.cpp
stud@ubuntu:~$ file hello_tgt
hello_tgt: ELF 32-bit LSB executable, ARM, EABI5 version 1 (GNU/Linux), …
stud@ubuntu:~$ ./hello_tgt
bash: ./hello_tgt: cannot execute binary file
```

SW DEVELOPMENT FOR EMBEDDED TARGETS

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
  - Peripherial
  - Memory and CPU constraints
  - ...
- So...what can you test?