

Embedded Software

C++ programming in a Linux environment

Agenda

- Code compilation - what's to happen
 - ▶ source, object & library code to executable
- Build tool - why?
- Make and how it works

A crash course in compilation

Source files



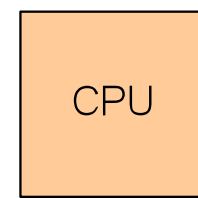
C++ compiler

Object files



C++ linker

Executable



Static library files



```
root@raspberrypi0-wifi:~# ./hello
Hello World
root@raspberrypi0-wifi:~#
```

C++ programming in a Linux environment – g++

- You have everything you need: GNU development tools
- An example C++ program: Good old “Hello World!”
- To compile and run in a shell:

C++ programming in a Linux environment – g++

- You have everything you need: GNU development tools
- An example C++ program: Good old “Hello World!”

```
// hello.cpp
#include<iostream>
using namespace std;

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

Invoke compiler **g++** to create executable **hello** from source file **hello.cpp**

- To compile and run in a shell:

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

Run hello

Programming

- This is a direct invocation of g++
- You can use this on more complex programs
 - ▶ Manually produce object files and manually combine to executable
 - ▶ Using flags
 - ▶ Debug
 - ▶ Release

```
$ g++ -o hello  
hello.cpp
```

Programming

- This is a direct invocation of g++
- You can use this on more complex programs
 - ▶ Manually produce object files and manually combine to executable
 - ▶ Using flags
 - ▶ Debug
 - ▶ Release

```
$ g++ -o hello  
hello.cpp
```

```
$ g++ -c part1.cpp  
$ g++ -c part2.cpp  
$ g++ -o hello part1.o  
part2.o
```

```
$ 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
```

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

Programming

- This is a direct invocation of g++
- You can use this on more complex programs
 - ▶ Manually produce object files and manually combine to executable
 - ▶ Using flags
 - ▶ Debug
 - ▶ Release

```
$ g++ -o hello  
hello.cpp
```

```
$ g++ -c part1.cpp  
$ g++ -c part2.cpp  
$ g++ -o hello part1.o  
part2.o
```

```
$ 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
```

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


Building tool

- Provides the ability to repeat and guarantee builds between builds
 - ▶ Reproducible results
 - ▶ Simplification of complexity
- Which one? Many exists
 - ▶ In this course
 - ▶ ***makefile*** - the most used in the unix world
 - ▶ Others include
 - ▶ ant
 - ▶ scons
 - ▶ rake
 - ▶ CMake

Programming - make

- Make is a scripting language in its own right
- make uses a `makefile` (default name is `makefile`) to determine dependencies, build rules, etc.

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

build: ${OBJECTS}
    ${CXX} -o ${EXECUTABLE} ${OBJECTS}

main.o : main.cpp
    ${CXX} -c main.cpp
```

- Seeking help on make
 - ▶ [http://www.cs.cmu.edu/cgi-bin/info2www?\(make.info\)Quick%20Reference](http://www.cs.cmu.edu/cgi-bin/info2www?(make.info)Quick%20Reference)

Programming - make

- Make is a scripting language in its own right
- make uses a `makefile` (default name is `makefile`) to determine dependencies, build rules, etc.

```
$ make [build_target]
```

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

build: ${OBJECTS}
    ${CXX} -o ${EXECUTABLE} ${OBJECTS}

main.o : main.cpp
    ${CXX} -c main.cpp
```

- Seeking help on make
 - ▶ [http://www.cs.cmu.edu/cgi-bin/info2www?\(make.info\)Quick%20Reference](http://www.cs.cmu.edu/cgi-bin/info2www?(make.info)Quick%20Reference)

Programming - make

- Make is a scripting language in its own right
- make uses a `makefile` (default name is `makefile`) to determine dependencies, build rules, etc.

```
$ make [build_target]
```

```
$ make -f makefile.other  
[build_target]
```

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

- Seeking help on make
 - ▶ [http://www.cs.cmu.edu/cgi-bin/info2www?\(make.info\)Quick%20Reference](http://www.cs.cmu.edu/cgi-bin/info2www?(make.info)Quick%20Reference)

make, makefiles and rules

- A makefile consists of rules of the following shape:

- Invocation

make, makefiles and rules

- A makefile consists of rules of the following shape:

```
target1 : prereq1 prereq2 ...  
    command1  
    command2  
    ...  
  
target2 : prereq1 prereq2 ...  
    command1  
    command2  
    ...
```

- Invocation

make, makefiles and rules

- A makefile consists of rules of the following shape:

Build target: Usually a file
that must be generated
(object file or executable)

```
target1 : prereq1 prereq2 ...  
    command1  
    command2  
    ...  
  
target2 : prereq1 prereq2 ...  
    command1  
    command2  
    ...
```

- Invocation

make, makefiles and rules

- A makefile consists of rules of the following shape:

Prerequisites: Files that are needed as input to create the build target

```
target1 : prereq1 prereq2 ...  
    command1  
    command2  
    ...  
  
target2 : prereq1 prereq2 ...  
    command1  
    command2  
    ...
```

Build target: Usually a file that must be generated (object file or executable)

- Invocation

make, makefiles and rules

- A makefile consists of rules of the following shape:

Prerequisites: Files that are needed as input to create the build target

```
target1 : prereq1 prereq2 ...  
    command1  
    command2  
    ...  
  
target2 : prereq1 prereq2 ...  
    command1  
    command2  
    ...
```

Build target: Usually a file that must be generated (object file or executable)

Commands: Actions that **make** carries out to build the target – can be any shell command, typically a compiler invocation

- Invocation

make, makefiles and rules

- A makefile consists of rules of the following shape:

Prerequisites: Files that are needed as input to create the build target

```
target1 : prereq1 prereq2 ...  
    command1  
    command2  
    ...  
target2 : prereq1 prereq2 ...  
    command1  
    command2  
    ...
```

Build target: Usually a file that must be generated (object file or executable)

Commands: Actions that **make** carries out to build the target – can be any shell command, typically a compiler invocation

- Invocation

Defaults to making first target in makefile(target1)

```
$ make  
$ make target2
```

make, makefiles and rules

- A makefile consists of rules of the following shape:

Prerequisites: Files that are needed as input to create the build target

```
target1 : prereq1 prereq2 ...  
    command1  
    command2  
    ...  
target2 : prereq1 prereq2 ...  
    command1  
    command2  
    ...
```

Build target: Usually a file that must be generated (object file or executable)

Commands: Actions that **make** carries out to build the target – can be any shell command, typically a compiler invocation

- Invocation

Defaults to making first target in makefile(target1)

```
$ make  
$ make target2
```

Makes a specific target

Programming – make and makefiles

- make will recursively check if a target needs to be built
 - ▶ if a prerequisite is more recent than its target, the target must be rebuilt
 - ▶ rebuilds are based on *specified* dependencies
- If a build of a target is necessary, make will build it first...

Programming – make and makefiles

- make will recursively check if a target needs to be built
 - ▶ if a prerequisite is more recent than its target, the target must be rebuilt
 - ▶ rebuilds are based on *specified* dependencies
- If a build of a target is necessary, make will build it first...

```
target1 : prereq1 prereq2 ...  
        command1  
        command2  
        ...  
  
target2 : prereq1 prereq2 ...  
        command1  
        command2  
        ...
```

Programming – make and makefiles

- Minimal makefile for “Hello World”
what will we see on console when we run “make”? Why?

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

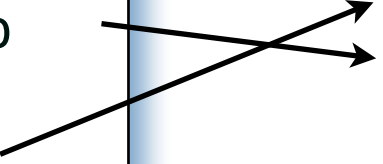
hello.o: hello.cpp
    g++ -c hello.cpp
```

Programming – make and makefiles

- Minimal makefile for “Hello World”
what will we see on console when we run “make”? Why?

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

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



```
$ make
g++ -c hello.cpp
g++ -o hello hello.o
$ ./hello
hello World!
$
```

Programming – make and makefiles

- A more complex example

- *What happens if we do...*

Programming – make and makefiles

- A 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

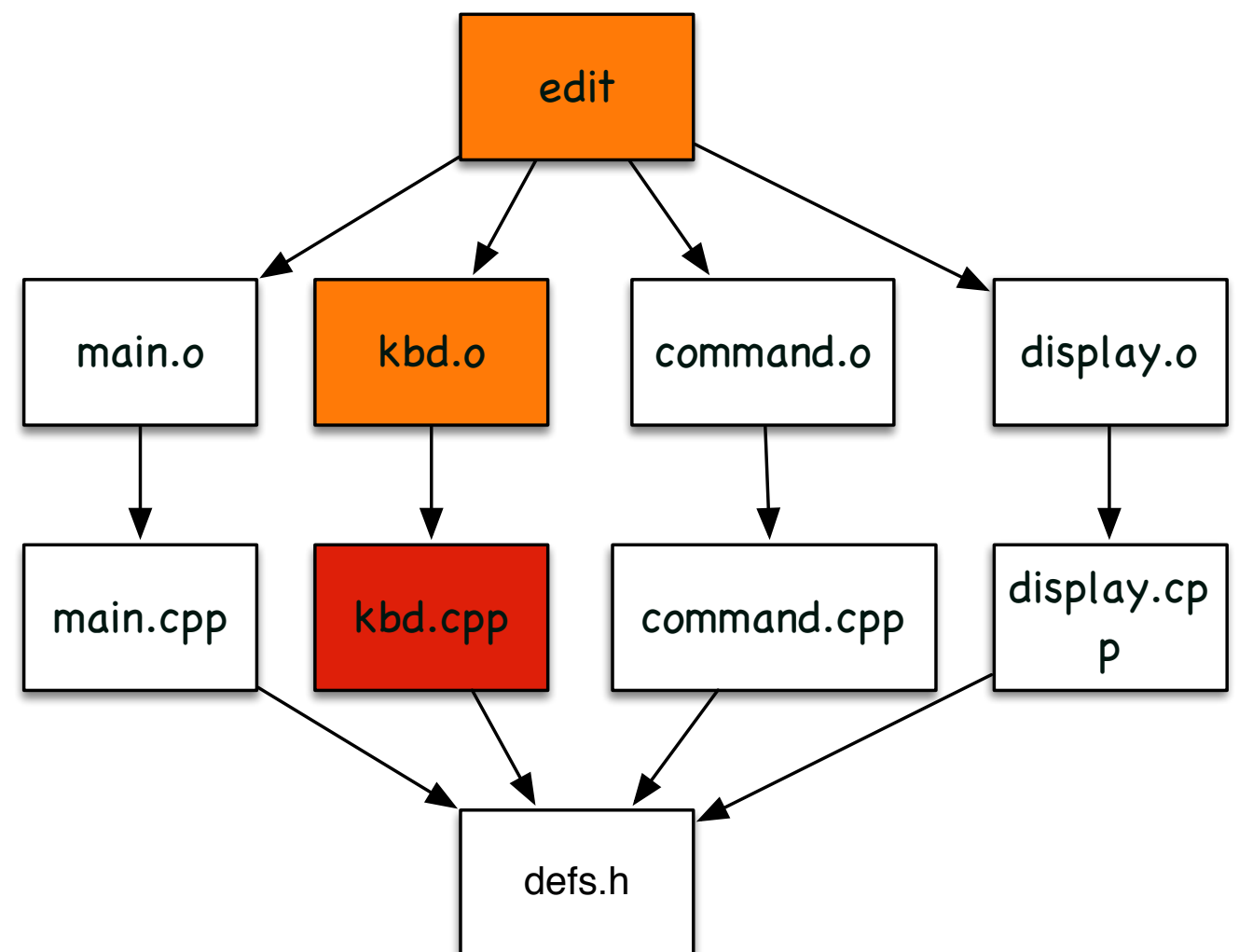
display.o : display.cpp defs.h buffer.h
      g++ -c display.cpp
```

- *What happens if we do...*

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

Programming – make and makefiles

- Dependency tree for *edit* makefile
 - *kbd.cpp* has changed resulting in a rebuild of
 - *kbd.o*
 - *edit*
- If *defs.h* is changed a rebuild of all would be necessary



make using variables & pattern matching rules

- Makefiles can use variables to simplify reading
 - ▶ Using variables simplifies reading and future changes/extension

```
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 using variables & pattern matching rules

- Makefiles can use variables to simplify reading
 - Using variables simplifies reading and future changes/extension

SOURCES contains names of source files

```
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 using variables & pattern matching rules

- Makefiles can use variables to simplify reading
 - ▶ Using variables simplifies reading and future changes/extension

SOURCES contains names of source files

OBJECTS defines object files by means of SOURCES, where ".cpp" is replaced by ".o"

```
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 using variables & pattern matching rules

- Makefiles can use variables to simplify reading
 - ▶ Using variables simplifies reading and future changes/extension

SOURCES contains names of source files

OBJECTS defines object files by means of SOURCES, where ".cpp" is replaced by ".o"

Pattern matching rule

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

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

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

make using variables & pattern matching rules

- Makefiles can be used to perform “service tasks” by running normal shell commands

```
$ make clean  
$ make install  
$ make 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}
```