Lecture 2. GNU Make Project Build Tool

Cross-Platform Application Development

September 22, 2017

Modular Approach to Software Development

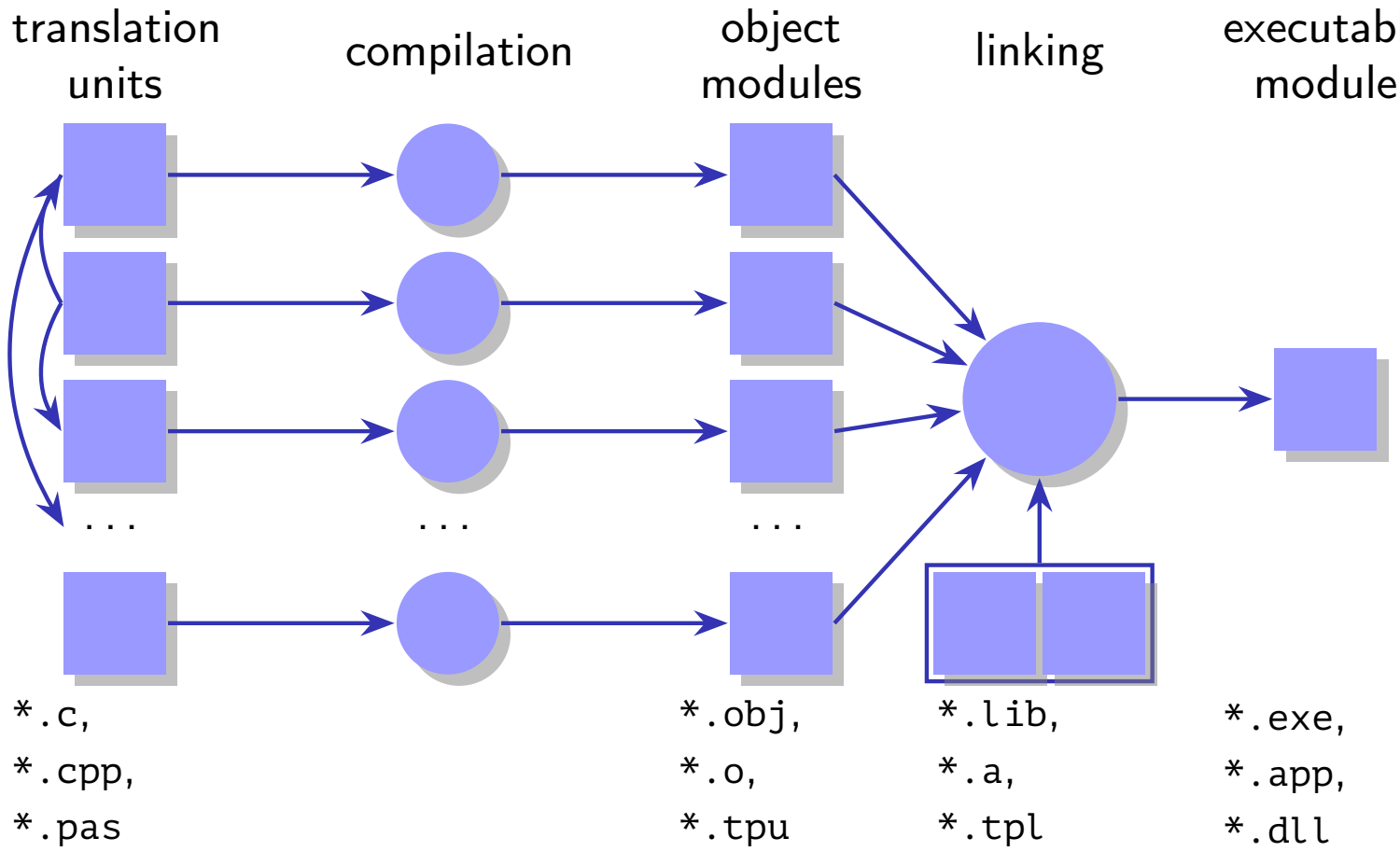

Figure 1: build process with modular approach

Main Advantages of Make Using

Advantages

- Convenient description of complex projects
- Potential of build automation
- Incremental build support
- Versatility
- Ability of small computational resources use

Main Implementations

Make Utility Implementations

- Unix make (1977, Stuart Feldman, AT&T Bell Labs).
- GNU make (1988, Richard Stallman, Roland McGrath, FSF) — Linux, MinGW, Cygwin.
- BSD make — FreeBSD, NetBSD, OpenBSD.
- Microsoft nmake.
- Borland make.
- Watcom wmake.
- ...

Sample Application Project

a.cpp

```
#include <iostream>

void f()
{
  std::cout << "f() from a.cpp" << std::endl;
}
```

b.cpp

```
#include "a.h"

int main()
{
  f();
}
```

a.h

```
void f();
```

Using Compiler Driver

Example (shell interaction)

```
$ g++ -o test_make a.cpp b.cpp
$ ./test_make
f() from a.cpp
$
```

Project Structure

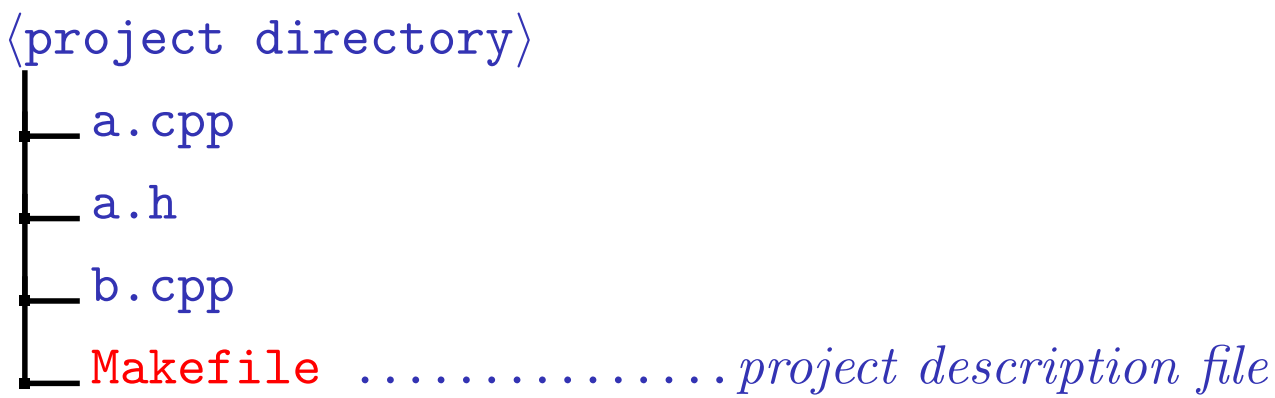

Figure 2: structure of project directory

Makefile Structure

Target Description in a Makefile

```
# Comment
```
$\langle target \rangle$ [$\langle target \rangle$ ...]: [$\langle prerequisite \rangle$ ...]
[$\langle TAB \rangle \langle command_1 \rangle$]
...
[$\langle TAB \rangle \langle command_n \rangle$]

Makefile Example

Example (Makefile)

```
# Makefile

all: test_make

test_make: a.o b.o
	g++ -o test_make a.o b.o

a.o: a.cpp a.h
	g++ -c -o a.o a.cpp

b.o: b.cpp a.h
	g++ -c -o b.o b.cpp
```

Example (shell interaction)

```
$ make
g++ -c -o a.o a.cpp
g++ -c -o b.o b.cpp
g++ -o test_make a.o b.o
$ ./test_make
f() from a.cpp
$ touch a.cpp
$ make
g++ -c -o a.o a.cpp
g++ -o test_make a.o b.o
$
```

Automatic Variables

Variable	Value
$@	target file name for the current rule;
$+	all prerequisite names delimited with spaces;
$^	all prerequisite names delimited with spaces, without duplicates;
$<	first prerequisite name;
$|	all order-only prerequisite names.

Table 1: main automatic variables

Using Automatic Variables

Example (Makefile)

```
test_make: a.o b.o
        g++ -o $@ $^

a.o: a.cpp a.h
        g++ -c -o $@ a.cpp

b.o: b.cpp a.h
        g++ -c -o $@ b.cpp

clean:
        rm -f *.o
        rm -f test_make
```

Example (shell interaction)

```
$ make clean
rm -f *.o
rm -f test_make
$ make
g++ -c -o a.o a.cpp
g++ -c -o b.o b.cpp
g++ -o test_make a.o b.o
$
```

Using Patterns

Example (Makefile)

```
out/test_make: out/a.o out/b.o
        g++ -o $@ $^

out/%.o: %.cpp a.h
        mkdir -p out
        g++ -c -o $@ $<

clean:
        rm -rf out
```

Using String Substitution Functions

Example (Makefile)

```
SOURCES := a.cpp b.cpp

out/test_make: $(patsubst %.cpp, out/%.o, ${SOURCES})
#          or: $(SOURCES:%.cpp=out/%.o)
        g++ -o $@ $^

out/%.o: %.cpp a.h
        mkdir -p out
        g++ -c -o $@ $<

clean:
        rm -rf out
```

Assignment Operators

Operator	Action	Right side substitution
:=	value assignment	in the place of assignment;
=	value assignment	in the place of left-side variable use;
?=	value assignment, if not already assigned	in the place of left-side variable use;
+=	concatenation after a space	depends on the previous assignment for the variable;

Table 2: assignment operators

Differencies in Assignments

Example (Makefile)

```
A   = abc
BA := $(A)def
BB  = $(A)def
A   = x

all:
        echo $(BA)
        echo $(BB)
```

Using Variable Assignments

Example (Makefile)

```
         SOURCES  := a.cpp b.cpp
override CXXFLAGS += -std=c++11
override LDFLAGS  += -lm

out/test_make: $(SOURCES:%.cpp=out/%.o)
        g++ ${LDFLAGS} -o $@ $^

out/%.o: %.cpp a.h
        mkdir -p out
        g++ ${CPPFLAGS} ${CXXFLAGS} -c -o $@ $<

clean:
        rm -rf out
```

Output of Make Utility

Example (shell interaction)

```
$ make CXXFLAGS=-O2
mkdir -p out
g++  -O2 -std=c++11 -c -o out/a.o a.cpp
mkdir -p out
g++  -O2 -std=c++11 -c -o out/b.o b.cpp
g++ -lm -o out/test_make out/a.o out/b.o
$ make clean
rm -rf out
$
```

Standard Variables

Variable	Value
CC	command to call C compiler;
CXX	command to call C++ compiler;
AR	command to call library archiver.

Table 3: main variables for commands

Variable	Value
CFLAGS	C compiler command arguments;
CXXFLAGS	C++ compiler command arguments;
CPPFLAGS	preprocessor command arguments;
LDFLAGS	linker command arguments.

Table 4: main variables for command arguments

Example of Compiler Command Setting

Example (Makefile)

```
CC := gcc-4.3

# ...
```

Example (shell interaction)

```
$ make CC=gcc-4.4
...
```

Multiple Targets and Rules

Example (Makefile)

```
# ...

out/%.o: %.cpp
        mkdir -p out
        ${CXX} ${CPPFLAGS} ${CXXFLAGS} -c -o $@ $<

out/a.o out/b.o: a.h    # repeated rules

# ...
```

Phony Targets

Example (Makefile)

```
.PHONY: clean          # declaration of a phony target

clean:
        rm -rf out
```

Conventions of Phony Target Names

Target	Actions
`all`	(default target) compile all the project without documentation and installation;
`check`	start self tests; the project must be previously built, but not necessarily installed;
`clean`	delete all files generated as a result of normal build — no need to delete directories created with `mkdir -p` command;
`install`	compile the project and copy all its files to the system directories where they should be for the real use;
`uninstall`	delete all installed files.

Table 5: names of the main frequently used targets in GNU projects

Order-Only Prerequisites

Target Description in a Makefile

⟨*targets*⟩ : [⟨*usual prerequisites*⟩] [| ⟨*order-only prerequisites*⟩]
[⟨*recipes*⟩]

Example of an Order-Only Prerequisite

Example (Makefile)

```
# ...

out/%.o: %.cpp | out
	${CXX} ${CPPFLAGS} ${CXXFLAGS} -c -o $@ $<

# ...

out:
	mkdir -p out

# ...
```

Output of Make Utility

Example (shell interaction)

```
$ make
mkdir -p out
g++   -c -o out/a.o a.cpp
g++   -c -o out/b.o b.cpp
g++ -lm -o out/test_make out/a.o out/b.o
$
```

Structure of a Project with a Library

Figure 3: structure of project directory

Output of Make Utility

Example (shell interaction)

```
$ make
mkdir -p out
g++ -c -o out/main.o main.cpp
g++ -c -o out/a.o a.cpp
g++ -c -o out/b.o b.cpp
ar cr out/libab.a out/a.o out/b.o
g++ -o out/main out/main.o out/libab.a
$
```

Example of External Makefiles Inclusion

Example (`Makefile`)

```
include  ../common.mak ../variables.mak
-include may_not_exist.mak

# ...
```

Example of Recursive Makefile

Example (Makefile)

```
SUBDIRS = project1 project2 project3

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
	$(MAKE) -C $@

project1: project3
```

Output of Make Utility

Example (shell interaction)

```
$ make
make -C project3
make[1]: Entering directory ‘/home/stu003/work/cross/projects/project3’
echo project3 > /dev/null
make[1]: Leaving directory ‘/home/stu003/work/cross/projects/project3’
make -C project1
make[1]: Entering directory ‘/home/stu003/work/cross/projects/project1’
echo project1 > /dev/null
make[1]: Leaving directory ‘/home/stu003/work/cross/projects/project1’
make -C project2
make[1]: Entering directory ‘/home/stu003/work/cross/projects/project2’
echo project2 > /dev/null
make[1]: Leaving directory ‘/home/stu003/work/cross/projects/project2’
```

Structure of a Project with Subdirectories

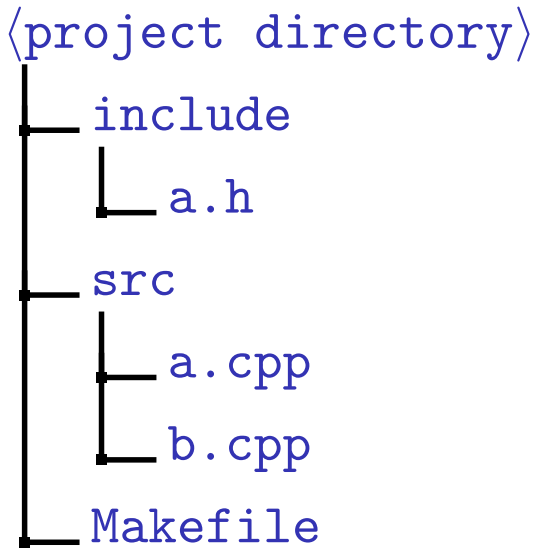

Figure 4: structure of project directory

Example of **vpath** Directive Use

Example (Makefile)

```
        SOURCES  := a.cpp b.cpp
override CPPFLAGS += -I include

vpath %.h   include
vpath %.cpp src
vpath %.o   _out

_out/test_make: $(SOURCES:%.cpp=_out/%.o) | _out
        $(CXX) $(LDFLAGS) -o $@ $^

_out/%.o: %.cpp a.h | _out
        $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o $@ $<
```

Example of Automatic Dependencies Search

Example (Makefile)

```
        SOURCES   = a.cpp b.cpp
override CPPFLAGS += -MMD -MP -I include

# ...
vpath %.d   _out

_out/hello_world: $(SOURCES:%.cpp=_out/%.o) | _out
        $(CXX) $(LDFLAGS) -o $@ $^

_out/%.o: %.cpp | _out
        $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o $@ $<

-include $(SOURCES:%.cpp=_out/%.d)
```