The Build Process: Makefiles and beyond

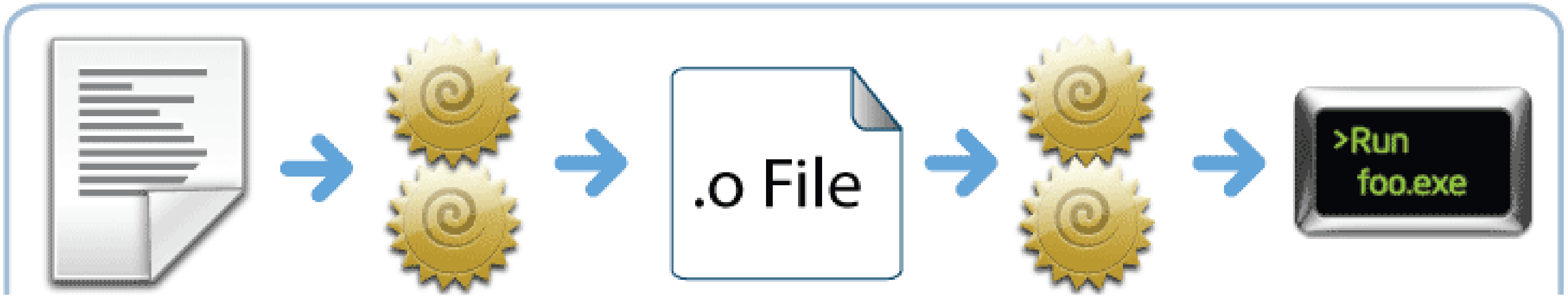
Ravikishore Kommajosyula

The build process

Creating executable from source code and/or libraries

Compilation: Compiling source code to object files

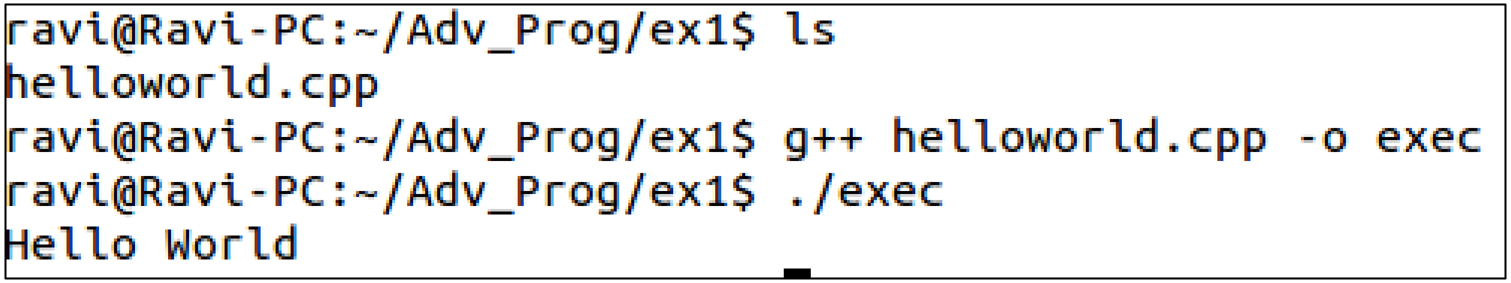
Linking: Linking several object files into a binary



- Libraries as pre-compiled object files

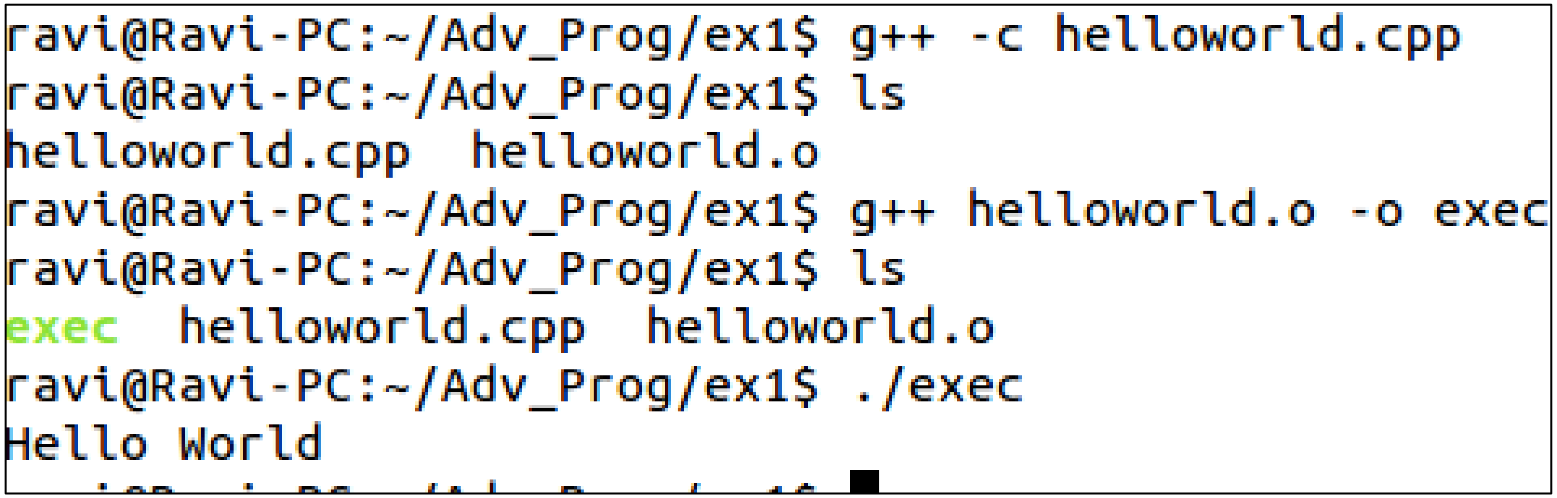
Build process in Terminal – Single file

- Compiling and Linking in one step:



```
ravi@Ravi-PC:~/Adv_Prog/ex1$ ls
helloworld.cpp
ravi@Ravi-PC:~/Adv_Prog/ex1$ g++ helloworld.cpp -o exec
ravi@Ravi-PC:~/Adv_Prog/ex1$ ./exec
Hello World
```

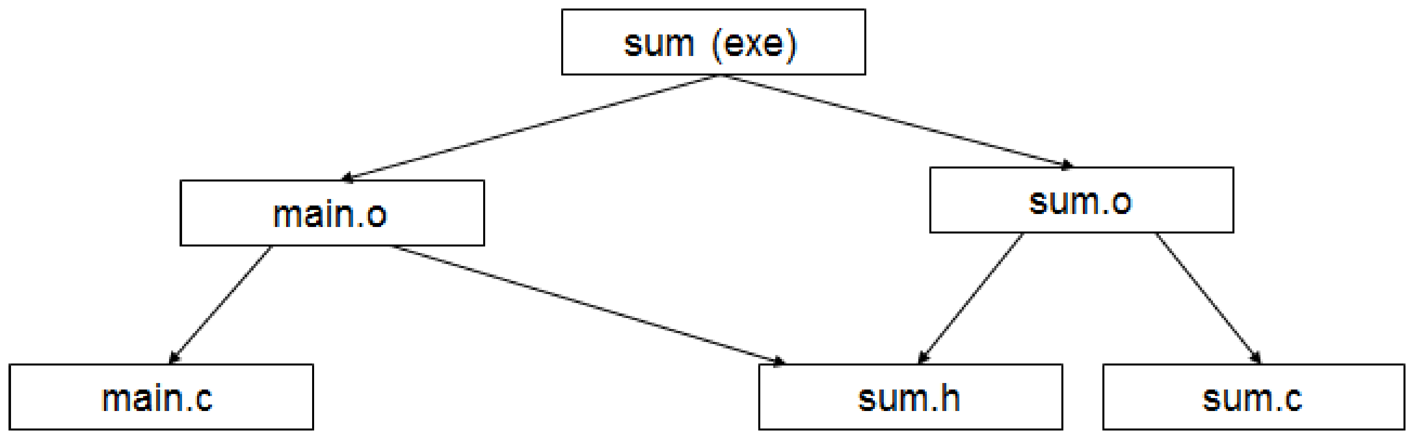
- Compile and Link separately:



```
ravi@Ravi-PC:~/Adv_Prog/ex1$ g++ -c helloworld.cpp
ravi@Ravi-PC:~/Adv_Prog/ex1$ ls
helloworld.cpp  helloworld.o
ravi@Ravi-PC:~/Adv_Prog/ex1$ g++ helloworld.o -o exec
ravi@Ravi-PC:~/Adv_Prog/ex1$ ls
exec  helloworld.cpp  helloworld.o
ravi@Ravi-PC:~/Adv_Prog/ex1$ ./exec
Hello World
```

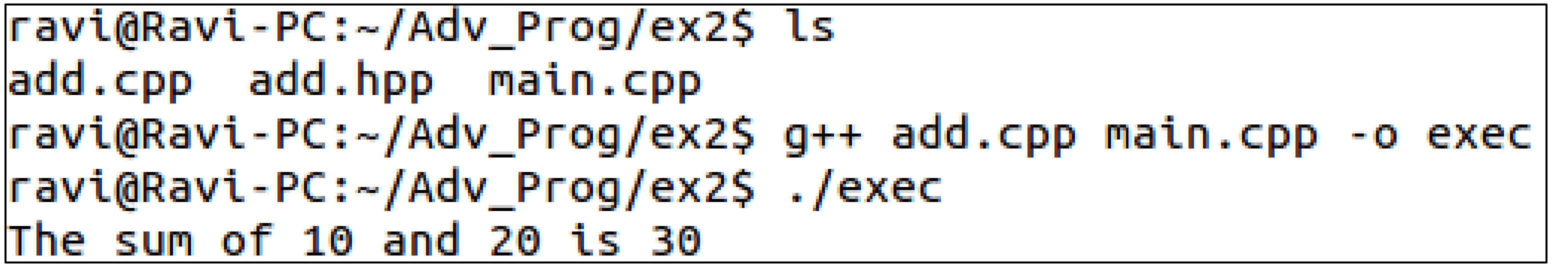
Build process – Multiple files

- Function declaration is needed at compile time (header files)
- Function definition is needed at linking time (object files)



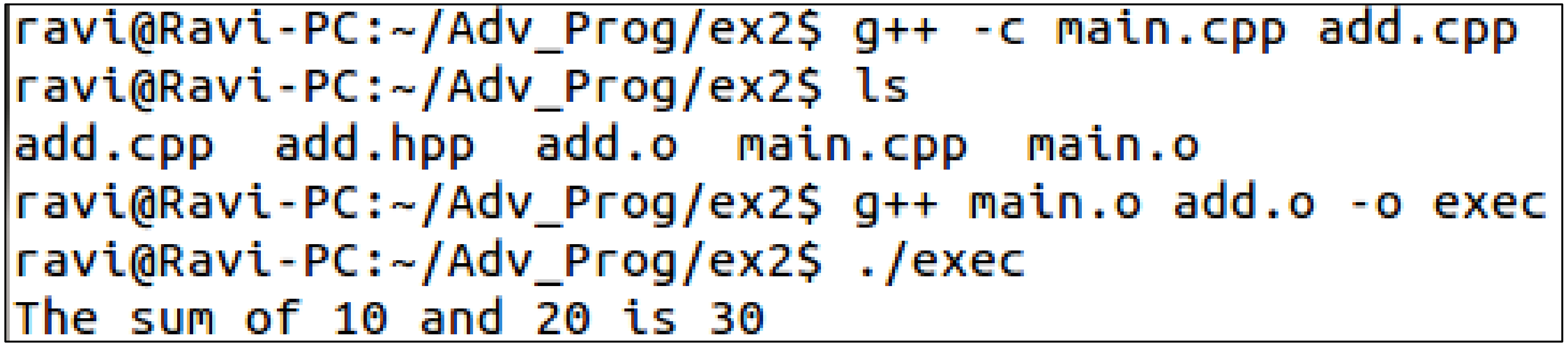
Build process in Terminal – Multiple files

- Compiling and Linking in one step:



```
ravi@Ravi-PC:~/Adv_Prog/ex2$ ls
add.cpp  add.hpp  main.cpp
ravi@Ravi-PC:~/Adv_Prog/ex2$ g++ add.cpp main.cpp -o exec
ravi@Ravi-PC:~/Adv_Prog/ex2$ ./exec
The sum of 10 and 20 is 30
```

- Compile and Link separately:



```
ravi@Ravi-PC:~/Adv_Prog/ex2$ g++ -c main.cpp add.cpp
ravi@Ravi-PC:~/Adv_Prog/ex2$ ls
add.cpp  add.hpp  add.o  main.cpp  main.o
ravi@Ravi-PC:~/Adv_Prog/ex2$ g++ main.o add.o -o exec
ravi@Ravi-PC:~/Adv_Prog/ex2$ ./exec
The sum of 10 and 20 is 30
```

Build process in Terminal – Observations

- Tedious efforts of typing involved
- Error prone
- Not suitable for large projects
- Any change in the code would need a re-compilation of the whole code
- Is there a smarter way??

Make is one answer

Make utility and Makefile

Make: A utility that automates the build process

Makefile: Special format file that contains Rules on how to build the executable (target)

- Widely used utility especially in Unix based OS
- Several tools available to generate Makefiles (CMake)
- Eclipse creates Makefiles and supports projects with user defined Makefiles

Our first Makefile

- Makefile works on a set of rules
- By default, builds the target of the first rule
- Rules define how "Targets" should be build from "Dependencies"
- Rule syntax:

```
Target  :  Dependencies
           Command
```

- Rule for building Helloworld:

```
exec  :  helloworld.c
         g++  helloworld.cpp  -o  exec
```

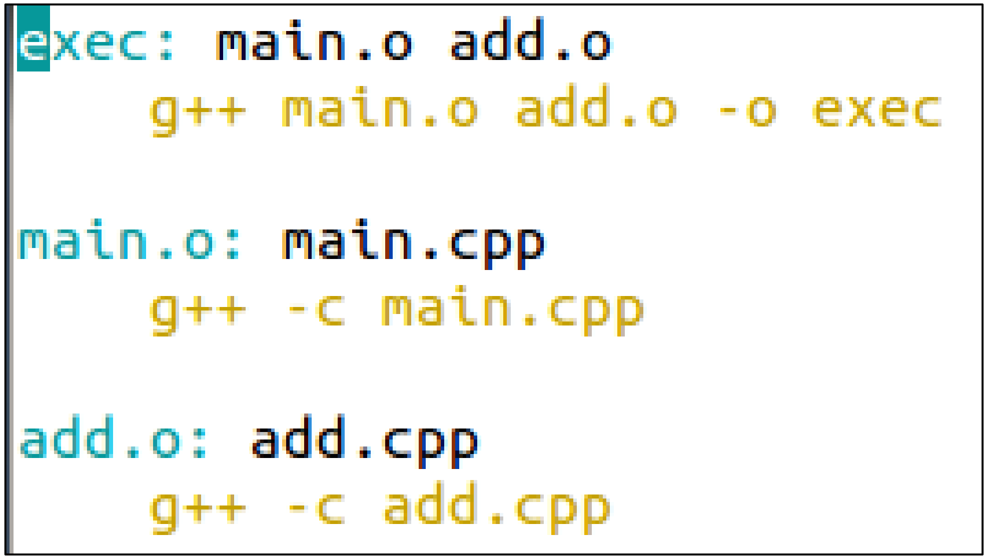
Syntax of a Makefile

Makefile contains the following 5 things:

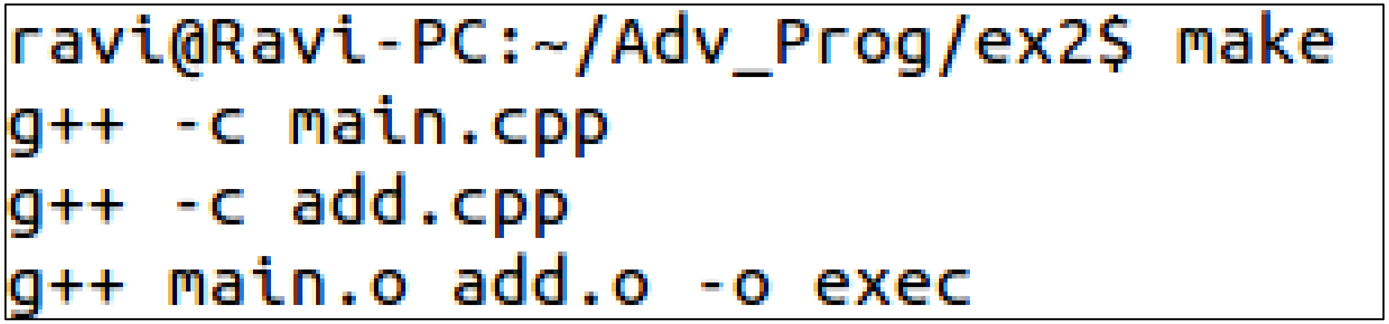
1) Explicit Rules – Defines how a target is to be built from dependencies
2) Implicit Rules – Routinely used customary techniques without specifying in detail
3) Variable Definitions – for text substitutions
4) Directives – Direct Makefile to do special things like reading another Makefile, conditional jumps
5) Comments – all line starting with a #

Makefile by examples

Compile and Link Multiple files – Only Explicit Rules:



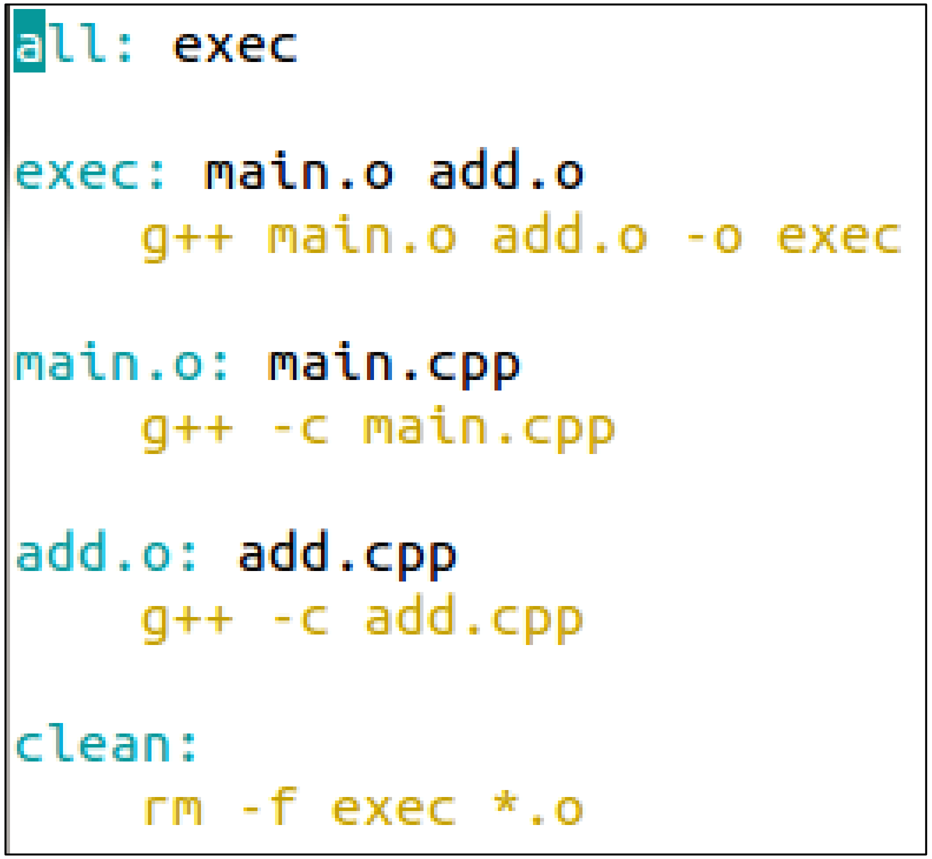
```
exec: main.o add.o
	g++ main.o add.o -o exec

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

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



```
ravi@Ravi-PC:~/Adv_Prog/ex2$ make
g++ -c main.cpp
g++ -c add.cpp
g++ main.o add.o -o exec
```

Makefile – Multiple targets

- Makefile can have several targets
- Standard practice to have "all", "release", "debug" and "clean" targets
- Debug target is a good way to hide print statements in release

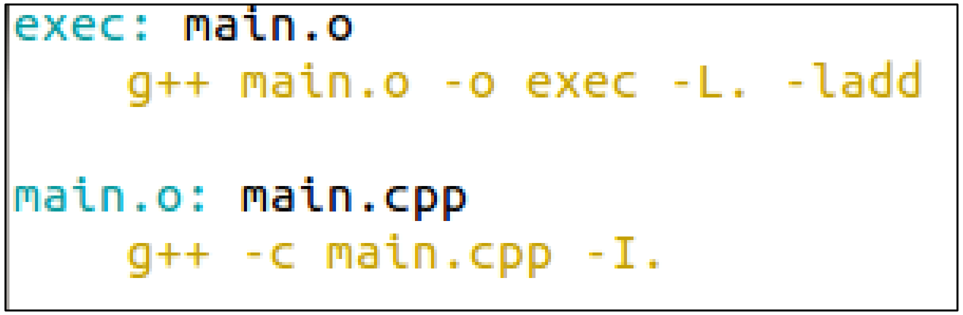


```
all: exec

exec: main.o add.o
	g++ main.o add.o -o exec

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

add.o: add.cpp
	g++ -c add.cpp

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

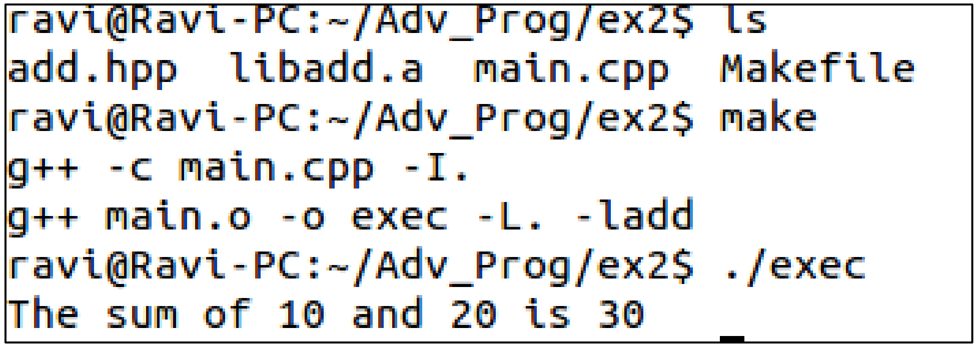
Makefile – Using external libraries

Recap: Header files at compilation, Object files at linking

- Specify the path of include files at compile time
- Specify the path of library and the library name at linking



```
exec: main.o
	g++ main.o -o exec -L. -ladd

main.o: main.cpp
	g++ -c main.cpp -I.
```



```
ravi@Ravi-PC:~/Adv_Prog/ex2$ ls
add.hpp  libadd.a  main.cpp  Makefile
ravi@Ravi-PC:~/Adv_Prog/ex2$ make
g++ -c main.cpp -I.
g++ main.o -o exec -L. -ladd
ravi@Ravi-PC:~/Adv_Prog/ex2$ ./exec
The sum of 10 and 20 is 30
```

Makefile – Using variables

- Normal variables - Text Substitutions

  CFLAGS = -O1 –Wall –Werror –std=c++11

- Value of variables can be modified / appended

  CFLAGS += -DDEBUG_FLAG

- Automatic variables – limited scope in the recipe

Variable	Functionality
$@	File name of the target of rule
$<	First dependency
$^	Name of all dependencies
$?	All dependencies newer than target

Makefile – Substitution references

- Substitutes the value of variable with alterations specified

```
$(var:a=b)
${var:a=b}
```

- Replaces every ‘a’ at the end of the word with ‘b’ for the variable *var*
- Example –

```
SRCS = a.c b.c c.c prog1.c
OBJS = $(SRCS:.c=.o)
```

Makefile – Inference rules

- One form of implicit rules to perform frequently done tasks

```
<filename>.o: <filename>.c
                $(CC) $(CFLAGS) <filename>.c
```

- Using inference rules, we can write –

```
.c.o:
        $(CC) $(CFLAGS) $<
```

Where $< refers to dependencies out of date

Makefile – Final Example

Define Variables

Append variables to link PAPI library

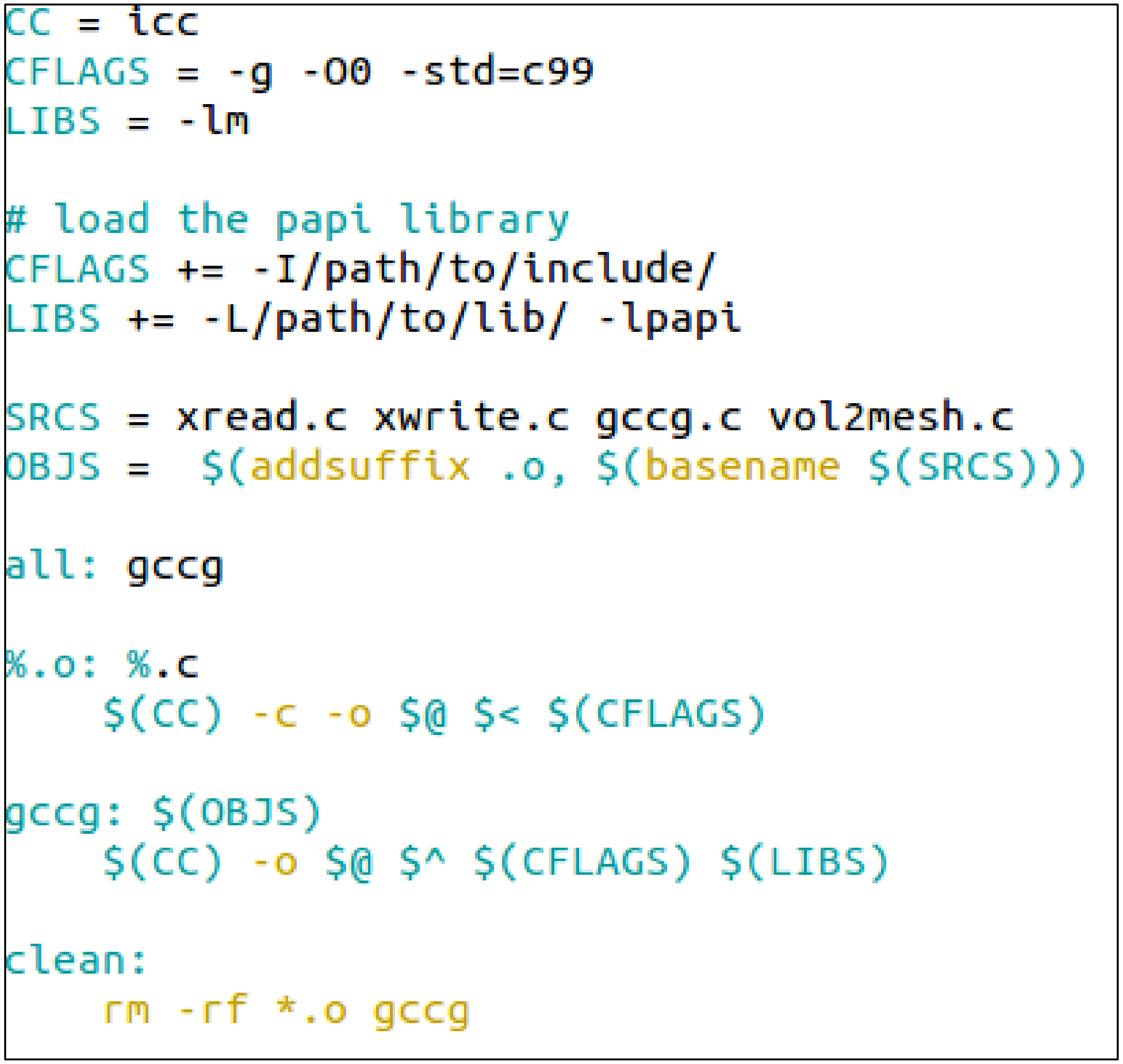
Create list of OBJS from SRCS

Make default target

Build object files

Link object files to get the binary

Make target (clean)



```
CC = icc
CFLAGS = -g -O0 -std=c99
LIBS = -lm

# load the papi library
CFLAGS += -I/path/to/include/
LIBS += -L/path/to/lib/ -lpapi

SRCS = xread.c xwrite.c gccg.c vol2mesh.c
OBJS =  $(addsuffix .o, $(basename $(SRCS)))

all: gccg

%.o: %.c
	$(CC) -c -o $@ $< $(CFLAGS)

gccg: $(OBJS)
	$(CC) -o $@ $^ $(CFLAGS) $(LIBS)

clean:
	rm -rf *.o gccg
```

Outlook: Beyond Makefiles

- Makefiles are almost the de-facto standard for build system, especially in UNIX based OS and have several advantages…….However……
- Makefiles have some disadvantages:
  - Cryptic syntax and not easy to code
  - Portability is an issue with Makefiles
  - Recursive Make with subfolders in a project could be dangerous
  - Environment variables affect the build process and in some cases is difficult to reproduce

Outlook: Make alternatives

- CMake: Cross-platform Makefile generator

  (+) Write directives to build the project at a higher level

  (+) Cross platform support

  (-) Re-inventing the wheel by developing a new language

  (-) Could be tedious to learn / migrate

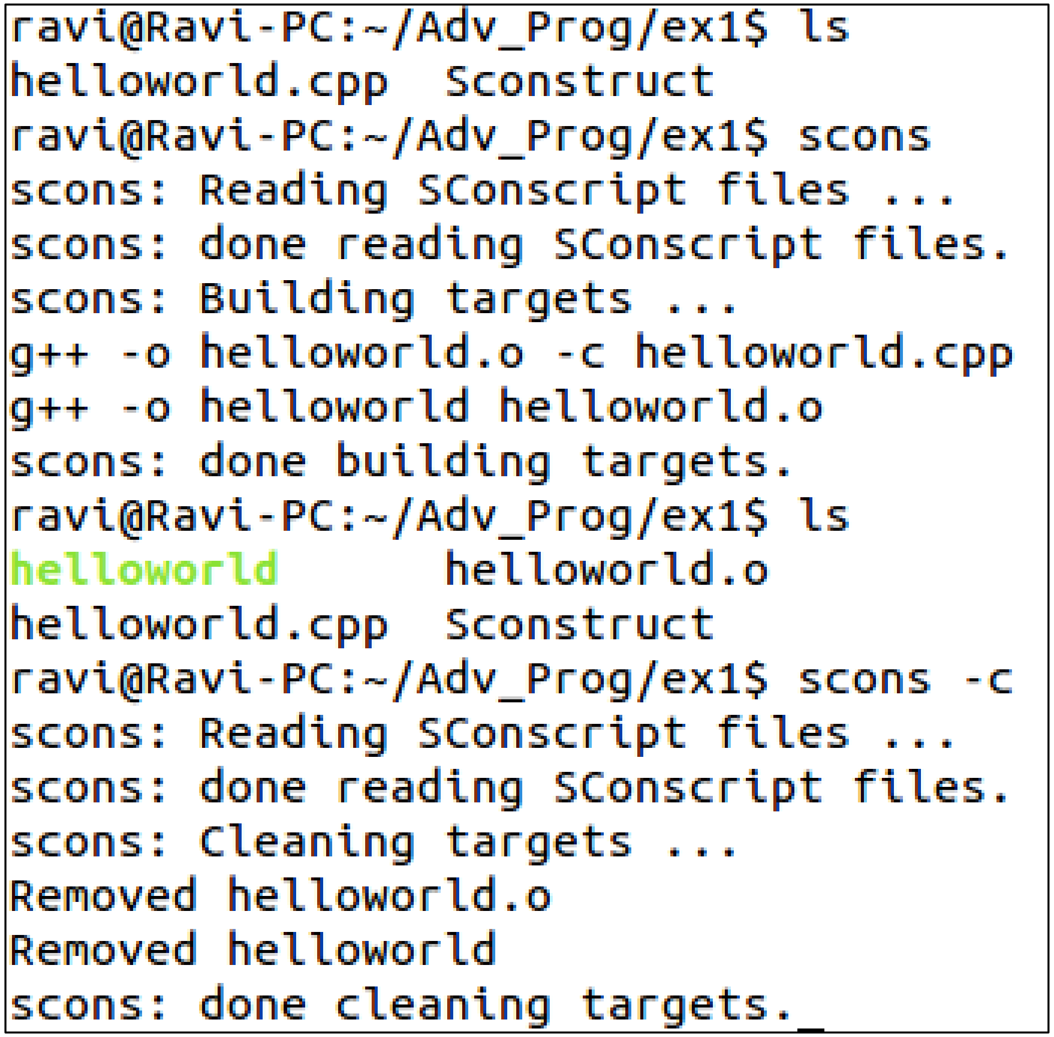
- Scons: Cross-platform software construction tool
  - Based on python
  - Automatically analyzes source code file dependencies and operating system adaptation requirements
  - Tipped to replace Make as the default build system

Outlook: Scons Helloworld

- Python script file with name *Sconstruct*
- Default build configuration given by

Program('helloworld.cpp')

- Provides default options for cleaning project
- In-order execution of the script



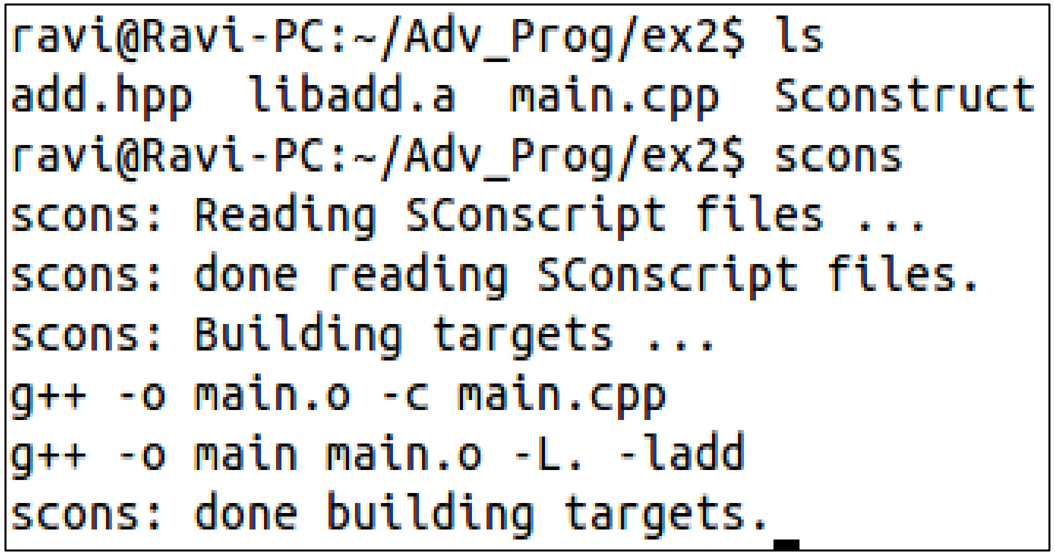
```
ravi@Ravi-PC:~/Adv_Prog/ex1$ ls
helloworld.cpp  Sconstruct
ravi@Ravi-PC:~/Adv_Prog/ex1$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o helloworld.o -c helloworld.cpp
g++ -o helloworld helloworld.o
scons: done building targets.
ravi@Ravi-PC:~/Adv_Prog/ex1$ ls
helloworld      helloworld.o
helloworld.cpp  Sconstruct
ravi@Ravi-PC:~/Adv_Prog/ex1$ scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed helloworld.o
Removed helloworld
scons: done cleaning targets._
```

Outlook: Scons with Libraries

- Libraries can be added to the build with the *LIBRARY* attribute
- Path to be specified using *LIBRARYPATH* attribute
- User-defined libraries can be built using the *LIBRARY* command

Library('foo', ['f1.c', 'f2.o'])

```
#Sconstruct
Program('main.cpp',  LIBRARY='add',
LIBRARYPATH = ' . ')
```



```
ravi@Ravi-PC:~/Adv_Prog/ex2$ ls
add.hpp  libadd.a  main.cpp  Sconstruct
ravi@Ravi-PC:~/Adv_Prog/ex2$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o main.o -c main.cpp
g++ -o main main.o -L. -ladd
scons: done building targets.
```

Outlook: Scons - Final remarks

- Build environment can be changed by creating a new Construction Environment and setting values of Construction Variables
- The ease of use and learning of *Scons* tool, together with power of Python scripting makes it a very powerful and usable build tool

```
#Sconstruct

import os

env = Environment(CC = ‘icc',

CCFLAGS = '-O2')

env.Program(‘helloworld.cpp')
```

Thank you for your attention!

Any questions?