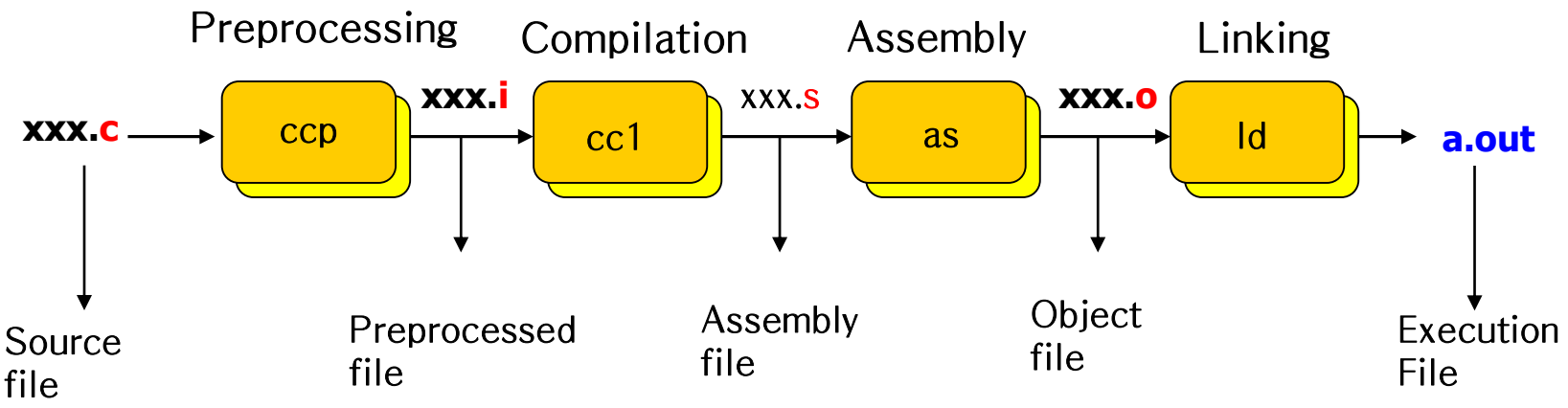
Gcc and Make

Chonnam National University
School of Electronics and Computer Engineering

Kyungbaek Kim

gcc

- Originally, gcc means “GNU C Compiler”
- From 1999, the meaning of gcc changes to “GNU Compiler Collection”
  - Handling C, C++, Objective C, Portran, Java



Handling C code with gcc

Extension	Type	Handling
.c	C source File	Preprocessing, Compile, Assembly, Linking
.s	Assembly File	Assembly, Linking
.S	Assembly File	Preprocessing, Assembly, Linking
.o	Object File	Linking
.a .so	Compiled Library File	Linking

〔example〕 a.c

```
#include <stdio.h>
int main()
{
        printf ("Hello Linux\n");

        return 0;
}
```

```
[root@localhost ~]$ gcc a.c
[root@localhost ~]$ ./a.out
```

Options in gcc

Option	Meaning
-c	Compiling a source file into an object file
-o filename	Set the output binary (execution) file name (default: a.out)
-I dir	Add the directory for searching header files
-S	Compiling a source file into an assembly file
-E	Preprocessing a source file
-L dir	Add the directories for searching libraries
-g	Add standard debugging information into the binary (execution) file
-ggdb	Add gdb debugging information into the binary (execution) file
-O	Optimizing compilation
-O*level*	Set the optimization level (1 ~ 3)
-D*FOO=BAR*	Defile preprocessing macro FOO with the value of BAR
-llib	Linking libraries while linking
-Wall	Show most of the warnings related to possibly incorrect code

Example

```
[root@localhost ~]$ gcc -o b b.c
```

➔ Binary file “b” is generated

```
[root@localhost ~]$ gcc -c b.c
```

➔ Object file “b.o” is generated

```
[root@localhost ~]$ gcc b.o
```

➔ Binary file “a.out” is generated

```
[root@localhost ~]$ gcc -S b.c
```

➔ Assembly file “b.s” is generated

```
[root@localhost ~]$ gcc b.s -o xx
```

➔ Binary file “xx” is generated

Compilation with multiple object files

main.c

```
extern void hi() ;
main()
{
        hi() ;
}
```

hi.c

```
#include <stdio.h>
void hi()
{
          printf ("Linux World \n");
}
```

Method 1

```
[root@localhost ~]$ gcc  main.c  hi.c  -o  test
```

Method 2

```
[root@localhost ~]$ gcc -c main.c
[root@localhost ~]$ gcc -c hi.c
[root@localhost ~]$ gcc main.o hi.o -o test
```

Using libraries and header files

- "ar" command for creating a library
  - Usage : ar csr libraryname objectfile

main2.c

```
#include "hi.h"
main()
{
        hi() ;
}
```

hi.c

```
#include <stdio.h>
void hi()
{
          printf ("Linux World \n");
}
```

hi.h

```
void hi();
```

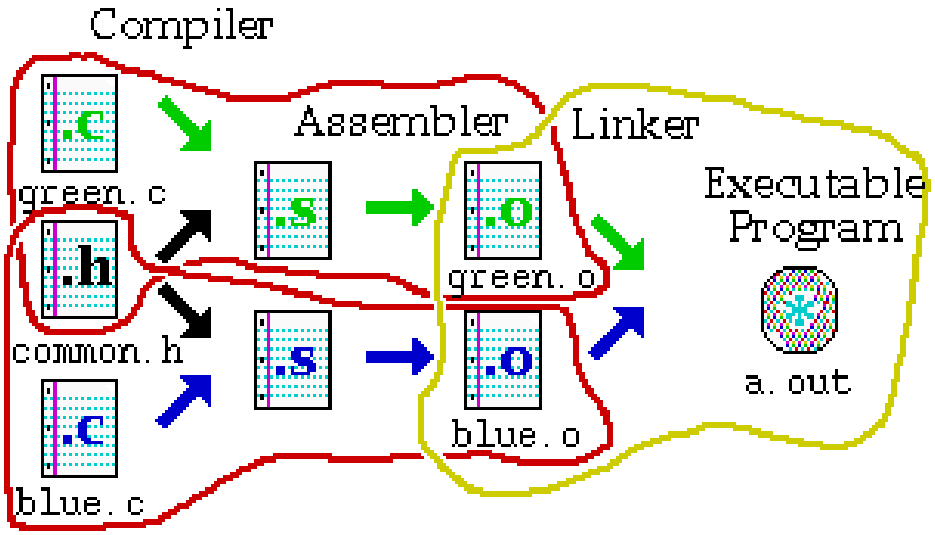
```
[root@localhost ~]$ gcc -c main2.c
[root@localhost ~]$ gcc -c hi.c
[root@localhost ~]$ ar csr libhi.a hi.o
[root@localhost ~]$ gcc main2.o -lhi -L .
```

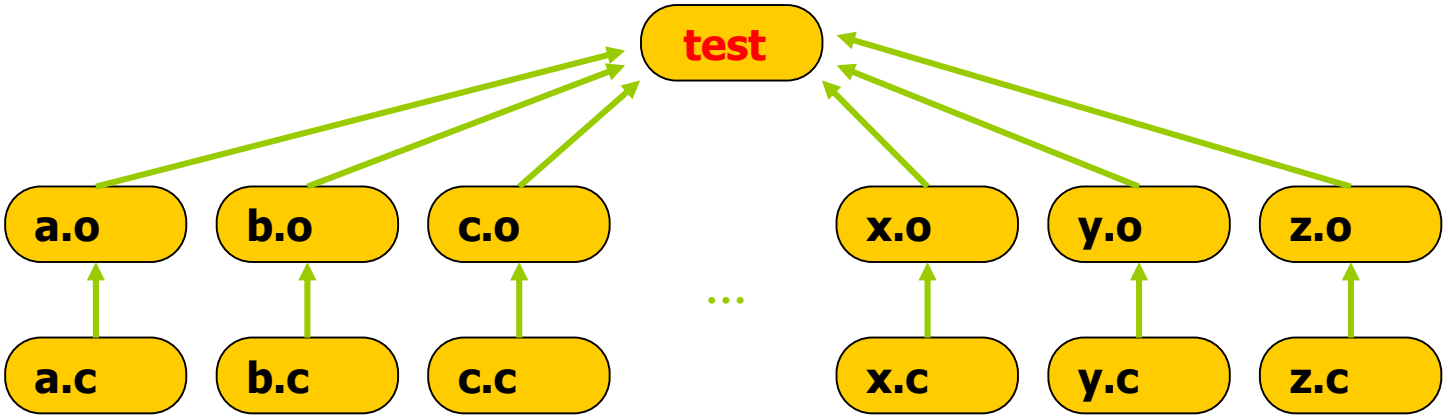
Library and header file are moved

```
[root@localhost ~]$ mkdir i; mv hi.h i
[root@localhost ~]$ mkdir t; mv libhi.a t
[root@localhost ~]$ gcc main2.o -Ii -Lt -lhi
```

Make

- Provide a way for separate compilation
- Describe the dependencies among the project files





Using makefiles

- Naming:
  - *Makefile* or *makefile* are standard
  - Other name can be also used
- Running make
  - make
  - make –f filename
    - If the name of your file is not “makefile” or “Makefile”
  - make target_name
    - If you want to make a target that is not the first one

Makefile content

- Rules: implicit, explicit
- Variables (macros)
- Directives (conditionals)
- # : comments everything till the end of the line
- ₩ : to separate on command line on two rows

Rule

```
target : dependencies
TAB commands      #shell commands
```

Example:

```
my_prog : eval.o main.o
      g++ -o my_prog eval.o main.o

eval.o : eval.c eval.h
      g++ -c eval.c
main.o : main.c eval.h
      g++ -c main.c

_____________________________
# -o to specify executable file name
# -c to compile only (no linking)
```

Variables

No variables

```
my_prog : eval.o main.o
        g++ -o my_prog eval.o main.o
eval.o : eval.c eval.h
        g++ -c -g eval.c
main.o : main.c eval.h
        g++ -c -g main.c
```

Using variables

```
C = g++
OBJS = eval.o main.o
HDRS = eval.h

my_prog : eval.o main.o
        $(C) -o my_prog $(OBJS)
eval.o : eval.c
        $(C) -c -g eval.c
main.o : main.c
        $(C) -c -g main.c
$(OBJS) : $(HDRS)
```

Defining variables on the command line:

Take precedence over variables defined in the makefile.

```
make C=cc
```

Automatic variables

- Automatic variables are used to refer to specific part of rule components

```
eval.o : eval.c eval.h
        g++ -c eval.c
```

`$@` – The name of the target of the rule (`eval.o`).
`$*` – The file name without extension (`eval`).
`$<` – The name of the first dependency (`eval.c`).
`$^` – The names of all the dependencies (`eval.c eval.h`).
`$?` – The names of all dependencies that are newer than the target

Normal way

```
my_prog : eval.o main.o
        $(C) -o my_prog $(OBJS)
eval.o : eval.c
        $(C) -c -g eval.c
main.o : main.c
        $(C) -c -g main.c
```

Using automatic variables

```
my_prog : eval.o main.o
        $(C) -o $@ $(OBJS)
eval.o : eval.c
        $(C) -c -g $*.c
main.o : main.c
        $(C) -c -g $<
```

Implicit rules

- Implicit rules are standard ways for making one type of file from another type.
- There are numerous rules making an .o file – from a.c file, a.p file, etc.
  - "make" applies the first rule it meets.
- If you have not defined a rule for a given object file, make will apply an implicit rule for it.

example1)

```
.c.o:
        gcc  -c  $<  $(CFLAGS)
```

example2)

```
.java.class:
        javac $(JFLAGS) $*.java
```

example3)

```
%_d.o : %.c
        gcc  -c  -g $< -o $@
```

Phony targets

- Targets that have no dependencies.
- Used only as names for commands that want to execute

example1)

```
clean:
		rm $(OBJS) $(TARGET)
```

➔ To invoke it : **make clean**

example1)

```
all: my_prog1  my_prog2
```

➔ To invoke it : **make all**

Conditionals (directives)

- Possible conditionals are:
  `if  ifeq  ifneq  ifdef  ifndef`
- All of them should be closed with `endif`
- Complex conditionals may use `elif` and `else`

Example:

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
        libs=$(libs_for_gcc)          #no tabs at the beginning
else
        libs=$(normal_libs)           #no tabs at the beginning
endif
```

Example

```
[root@localhost ~]$ gcc -c main.c
[root@localhost ~]$ gcc -c hi.c
[root@localhost ~]$ gcc main.o hi.o -o test
```

example1)

```
CC = gcc
OBJS = hi.o main.o

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

test1 : $(OBJS)
        gcc -o $@ $^

clean:
        rm -f $(OBJS) test1
```

```
[root@localhost ~]$ gcc -c main2.c
[root@localhost ~]$ gcc -c hi.c
[root@localhost ~]$ ar csr libhi.a hi.o
[root@localhost ~]$ gcc main2.o -lhi -L .
```

example2)

```
CC = gcc
AR = ar
ARFLAG = csr

main2.o : main2.c hi.h
        $(CC) -c $<

hi.o : hi.c
        $(CC) -c $<

libhi.a : hi.o
        $(AR) $(ARFLAG) $@ $<

test2 : main2.o libhi.a
        $(CC) -o $@ $< -L . -lhi

clean:
        rm -f main2.o hi.o libhi.a test2
```

Example

example2)

```
CC = gcc
AR = ar
ARFLAG = csr

main2.o : main2.c hi.h
        $(CC) -c $<

hi.o : hi.c
        $(CC) -c $<

libhi.a : hi.o
        $(AR) $(ARFLAG) $@ $<

test2 : main2.o libhi.a
        $(CC) -o $@ $< -L . -lhi

clean:
        rm -f main2.o hi.o libhi.a test2
```

example3)

```
CC = gcc
AR = ar
ARFLAG = csr

OBJS = main2.o hi.o

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

LIBOBJ = libhi.a
LPATH = -L .
LIBS = -lhi

lib%.a : %.o
        $(AR) $(ARFLAG) $@ $<

test2 : $(OBJS) $(LIBOBJ)
        $(CC) -o $@ $< $(LPATH) $(LIBS)

clean:
        rm -f $(OBJS) $(LIBOBJ) test2
```

Example with JAVA

example)

```
JFLAGS = -g
JC = javac
.SUFFIXES: .java .class
.java.class:
        $(JC) $(JFLAGS) $*.java

CLASSES = \
        Foo.java \
        Blah.java \
        Library.java \
        Main.java

default: classes

classes: $(CLASSES:.java=.class)

clean:
        $(RM) *.class
```

$(name:string1=string2)

➔ Replace “string1” with “string2”