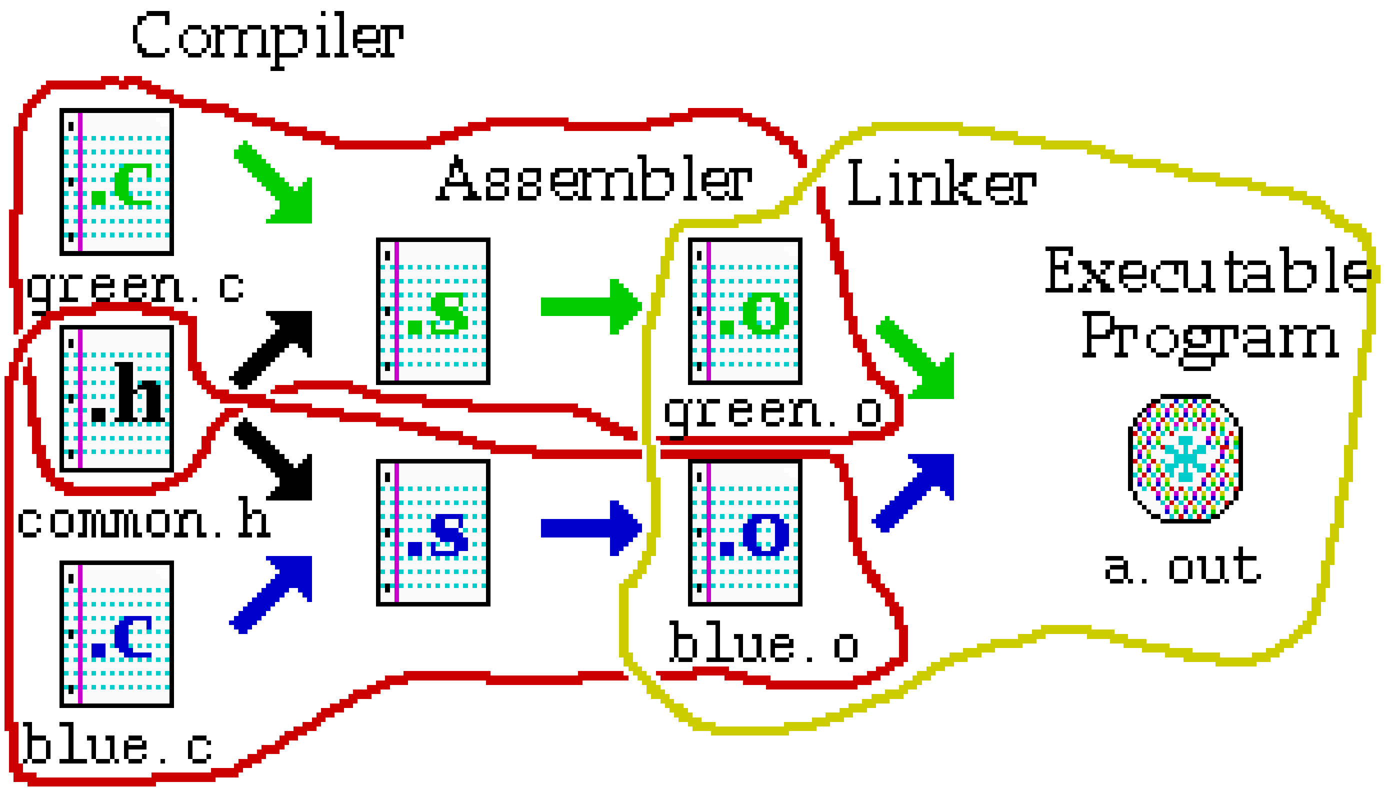
Makefiles

- Provide a way for separate compilation.
- Describe the dependencies among the project files.
- The `make` utility.



Using makefiles

<u>Naming:</u>

- *makefile* or *Makefile* are standard
- other name can be also used

<u>Running `make`</u>

`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

Sample makefile

➢ Makefiles main element is called a *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

The old way (no variables)	A new way (using variables)

The old way (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
```

A new way (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)
```

<u>Defining variables on the command line:</u>

Take precedence over variables defined in the makefile.

```
make C=cc
```

Implicit rules

- Implicit rules are standard ways for making one type of file from another type.
- There are numerous rules for 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.

Example:

Our makefile

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

The way `make` understands it

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

Defining implicit rules

```
%.o : %.c
	$(C) -c -g $<

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

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

<u>Avoiding implicit rules - empty commands</u>

`target: ;` #Implicit rules will not apply for this target.

Automatic variables

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

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

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

`$@` - The name of the target of the rule (`eval.o`).

`$<` - 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

`make` options

`make` options:

`-f` *filename* – when the makefile name is not standard

`-t` – (touch) mark the targets as up to date

`-q` – (question) are the targets up to date, exits with 0 if true

`-n` – print the commands to execute but do not execute them

/ `-t`, `-q`, and `-n`, cannot be used together /

`-s` – silent mode

`-k` – keep going – compile all the prerequisites even if not able to link them **!!**