Makefile

Z. Cihan TAYŞİ

Outline

- Splitting Your C Program
- Compiling with Several Files
- Makefile
- Makefile Macros

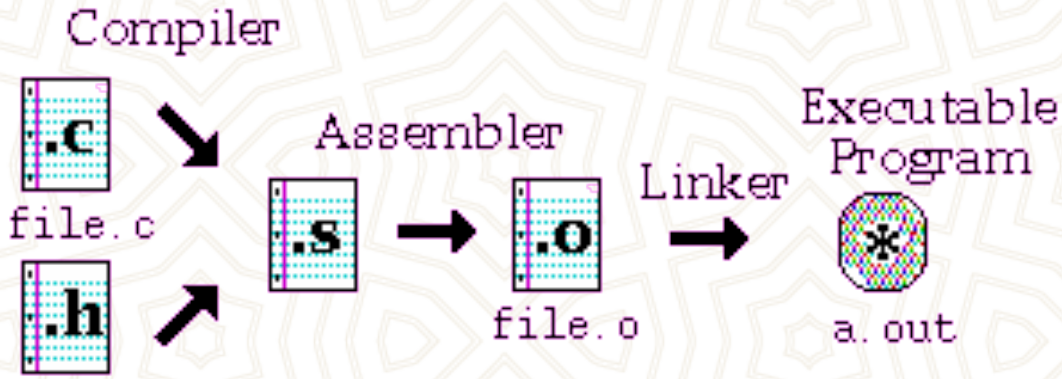
Splitting Your C Program

- At least one of the files must have a main() function.
- To use functions from another file, make a .h file with the function prototypes, and use #include to include those .h files within your .c files.
- Be sure no two files have functions with the same name in it. The compiler will get confused.
- Similarly, if you use global variables in your program, be sure no two files define the same global variables.

Splitting Your C Program

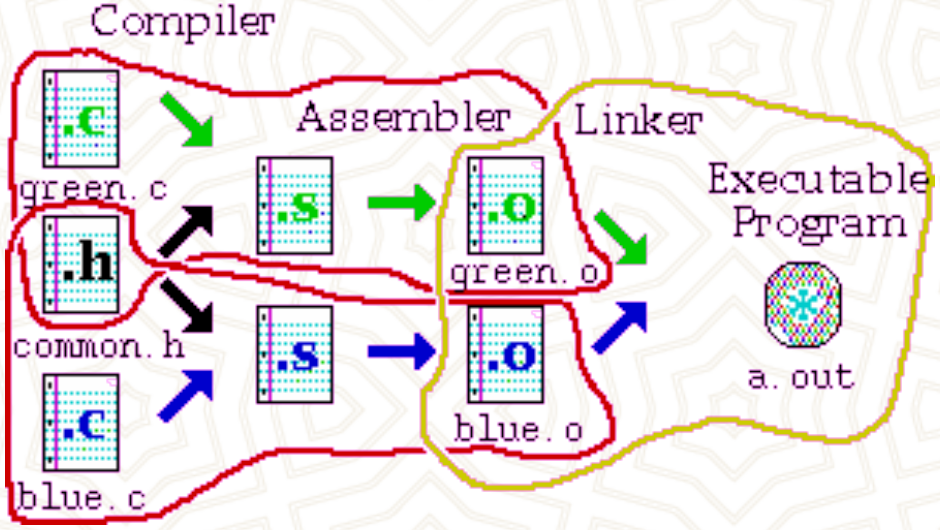
- If you use global variables, be sure only one of the files defines them, and declare them in your .h as follows:
  - **extern int globalvar**;
- When you define a variable, it looks like this:
  - **int globalvar;**
- When you declare a variable, it looks like this:
  - **extern int globalvar;**
- The main difference is that a variable definition creates the variable, while a declaration indicates that the variable is defined elsewhere. A definition implies a declaration.

Compiling with Several Files



- the command to perform this task is simply
  - gcc file.c
- there are 3 steps to obtain the final executable program
  - Compiler stage
  - Assembler stage
  - Linker stage

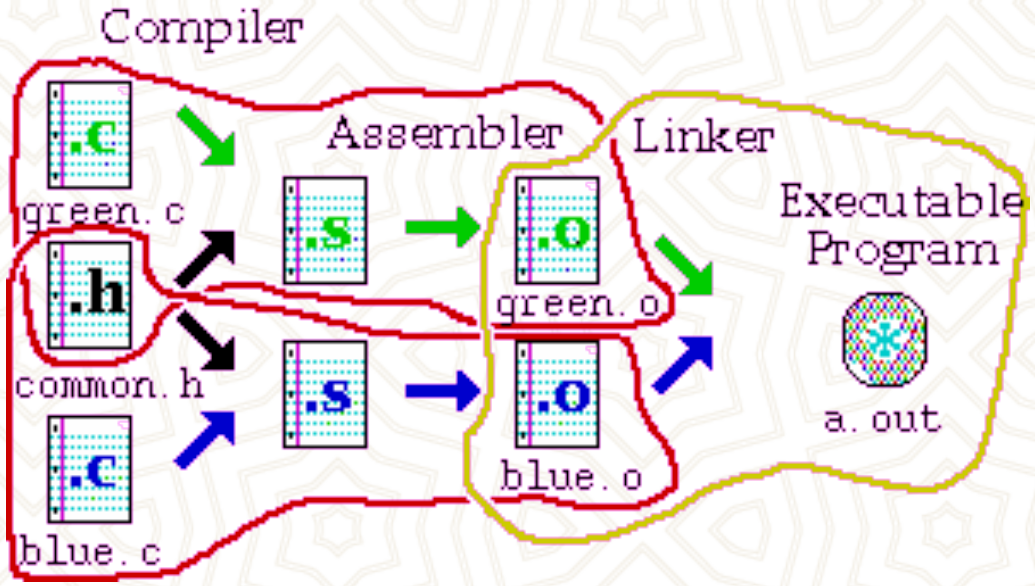
Compiling with Several Files



- You can use the -c option with gcc to create the corresponding object (.o) file from a .c file.
  - **gcc -c green.c**
- will not produce an a.out file, but the compiler will stop after the assembler stage, leaving you with a green.o file.

Compiling with Several Files

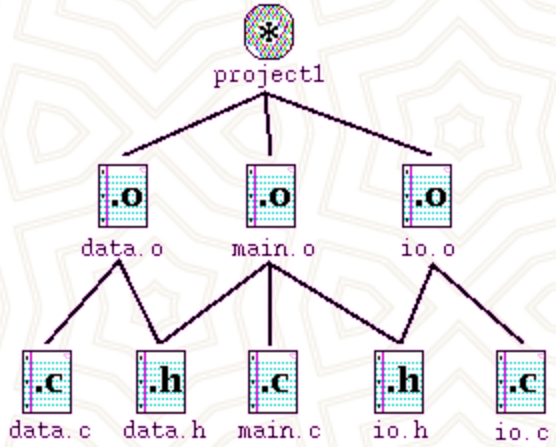
- The three different tasks required to produce the executable program are as follows:
- Compile green.o:
  - **gcc -c green.c**
- Compile blue.o:
  - **gcc -c blue.c**
- Link the parts together:
  - **gcc green.o blue.o**



The Make Command

- helps you to manage large programs or groups of programs
- keeps track of which portions of the entire program have been changed
- compiles only those parts of the program which have changed since the last compile.

How does make do it?

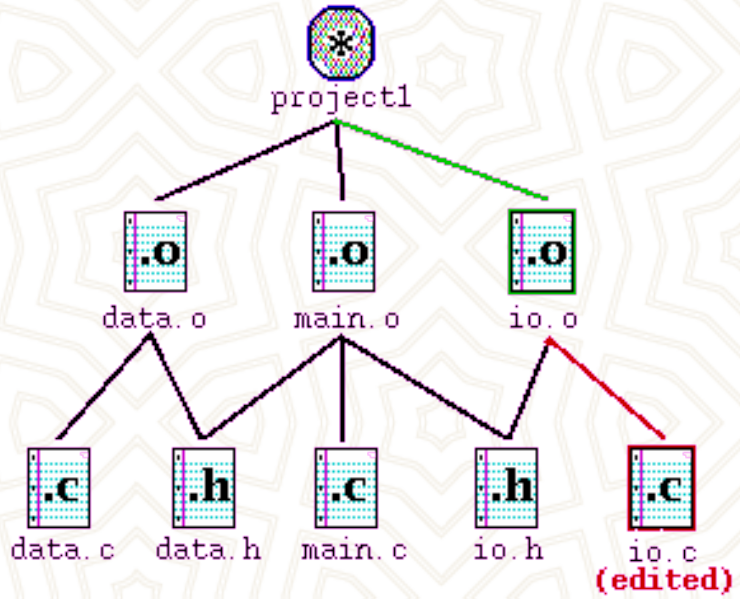


Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

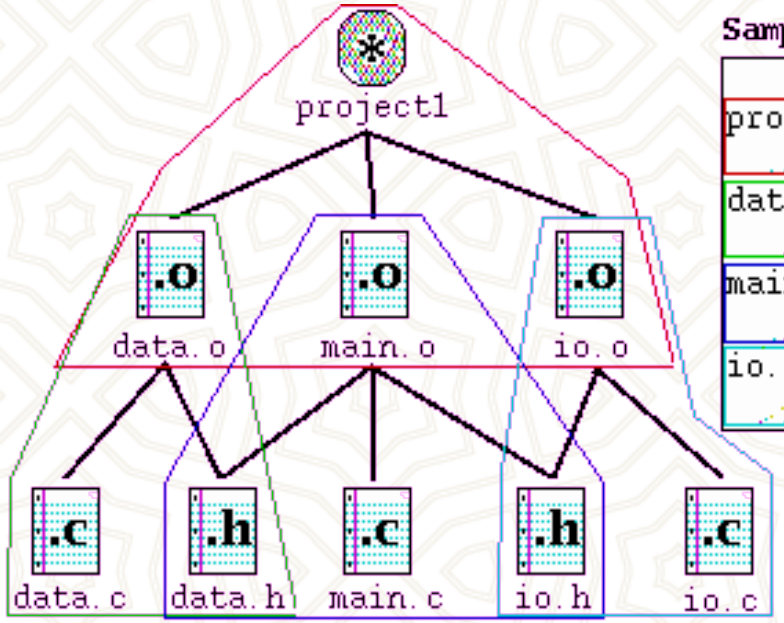
- The make program gets its **dependency** "graph" from a text file called **makefile** or **Makefile** which resides in the same directory as the source files
- make checks the modification times of the files, and whenever a file becomes "newer" than something that depends on it, it runs the compiler accordingly.

How Dependency Works



- Case : while you are testing the program, you realize that one function in io.c has a bug in it. You edit io.c to fix the bug.
- notice that io.o needs to be updated because io.c has changed. Similarly, because io.o has changed, project1 needs to be updated as well.

Translating The Dependency Graph



Sample Makefile

```
project1: data.o main.o io.o
	cc data.o main.o io.o -o project1
data.o: data.c data.h
	cc -c data.c
main.o: data.h io.h main.c
	cc -c main.c
io.o: io.h io.c
	cc -c io.c
```

- Each dependency shown in the graph is circled with a corresponding color in the Makefile, and each uses the following format:
- target : source file(s)
  - command (must be preceded by a tab)

Listing Dependencies

Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

- **Note** that in the Makefile shown above, the .h files are listed, but there are no references in their corresponding commands. This is because the .h files are referred within the corresponding .c files through the #include "file.h". If you do not explicitly include these in your Makefile, your program will not be updated if you make a change to your header (.h) files.

Using the Makefile with make

- Once you have created your Makefile and your corresponding source files, you are ready to use make.
  - If you have named your Makefile either Makefile or makefile, make will recognize it.
  - If you do not wish to call your Makefile one of these names, you can use make -f mymakefile.

Macros in make

Here is our sample *Makefile* again, using a macro.

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

- The make program allows you to use macros, which are similar to variables, to store names of files. The format is as follows:
  - OBJECTS = data.o io.o main.o
- Whenever you want to have make expand these macros out when it runs, type the following corresponding string $(OBJECTS)

Special macros

- In addition to those macros which you can create yourself, there are a few macros which are used internally by the make program. Here are some of those, listed below:

CC	Contains the current C compiler. Defaults to cc.
CFLAGS	Special options which are added to the built-in C rule.
$@	Full name of the current target.
$?	A list of files for current dependency which are out-of-date.
$<	The source file of the current (single) dependency.

References & more Reading

- References
  - http://www.eng.hawaii.edu/Tutor/Make/index.html
- More reading on **make** command
  - http://www.cs.duke.edu/~ola/courses/programming/Makefiles/Makefiles.html
  - http://www.hsrl.rutgers.edu/ug/make_help.html
  - http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.genprogc/doc/genprogc/make.htm