Computer Labs: Make

2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

October 15, 2010

Compilation Dependencies

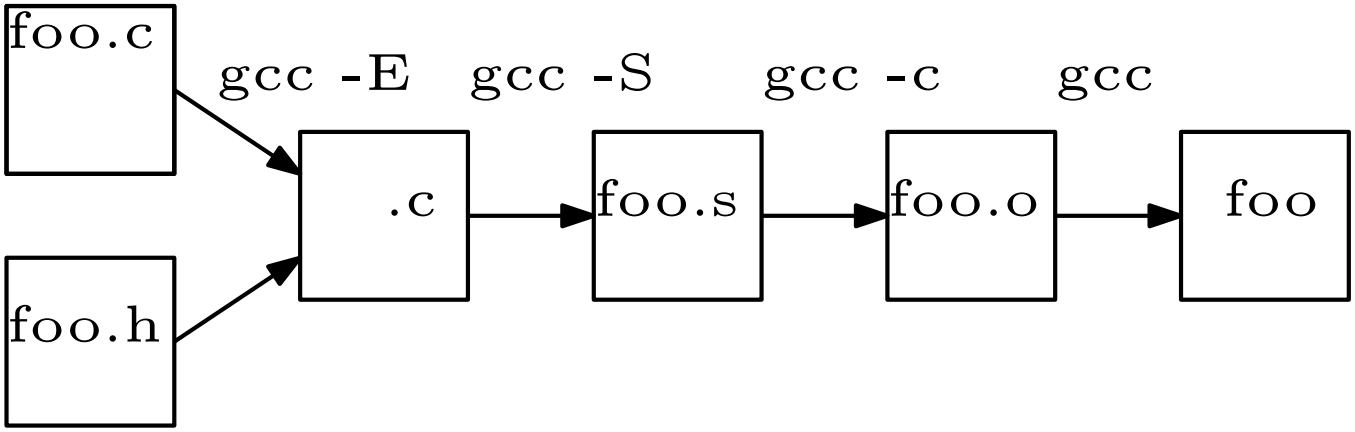
- Even relatively small programs comprise several source and header files
- Medium to large projects may be comprised of hundreds or even thousands source and header files.

Advantages

- Allows for easier structuring of the code
- Makes the code easier to manage
- Facilitates multi-programmer development easier
- May make compilation faster

Problem Realizing the last advantage may not be as easy as it appears.

Compilation of a C Program



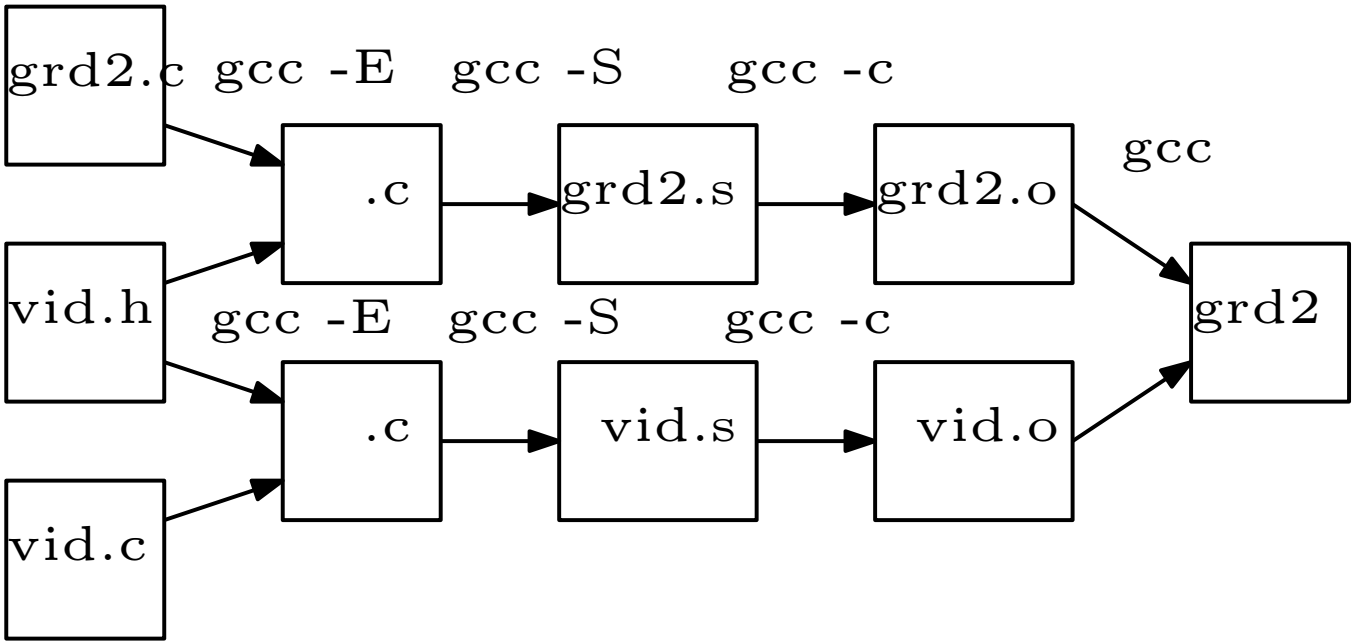
Preprocessing Stage The C pre-processor processes its directives in the source and header files: mostly text substitution

Compiling Stage The C source code is converted to an assembly file by the compiler

Assembling Stage The assembly code is converted to relocatable object code, which is stored in a `.o` file

Linking Stage The object code file is linked with libraries that contain functions like `printf()`, generating an executable program.

Compilation with Several C Files



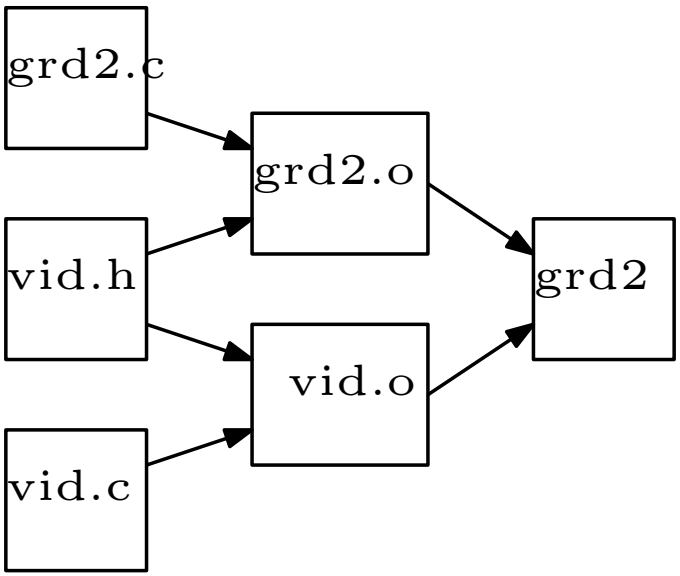
- Even in the case of Lab 2 makes sense to divide the source code into several source files
- Compilation of `grd2.o` and `vid.o` can be performed as two separate steps:

```
gcc -Wall -c grd2.c
gcc -Wall -c vid.c
```

- The executable program may be generated afterwards by linking the two object files with the C library:

```
gcc -Wall grd2.o vid.o -o grd2
```

Dependency Graphs



- Captures de dependency between files used in the generation of a program
- Change in a file in a dependency graph, requires generating all the files that depend on it, i.e. all the files in the paths from that file to the executable
- The `make` utility helps automating that process

Make and Makefiles

- Make generates the dependency from a makefile

```
grade2.exe: grade2.o video-graphics.o
        gcc -Wall grade2.o video-graphics.o -o grade2.exe

grade2.o: grade2.c utypes.h video-graphics.h
        gcc -Wall -c grade2.c

video-graphics.o: video-graphics.c utypes.h video-graphics
        gcc -Wall -c video-graphics.c
```

- The makefile specifies also how a file may be generated from the files on which it depends
- Make compares the date of the last modification of a file with that of the files that depend on it
  - If that date is more recent, it invokes the appropriate command to rebuild the files that depend on it
  - The whole process starts from the *bottom* of the dependency graph and progresses upward until it reaches the executable

The Makefile

- The makefile consists of a set of rules
- Each rule expresses a dependency in the dependency graph
  - I.e., the files on which a file in a dependency graph depends
  - How a given file may be (re)built from the files on which it depends
- The format of a make rule is as follows:

```
target: <list of files>
        <command line 0>
        ...
        <command line n>
```

where:

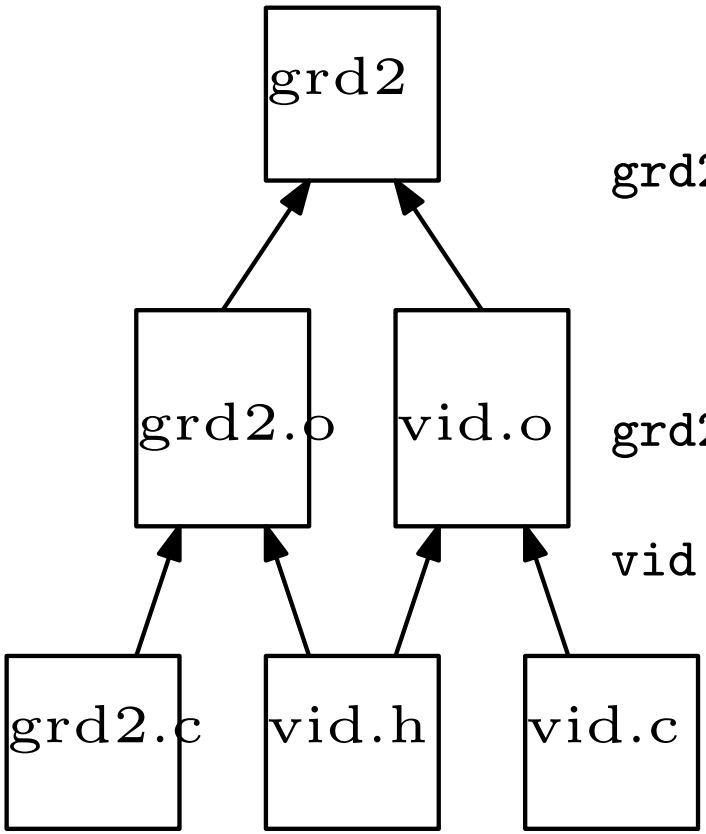
`target` is a node in the dependency file

`<list of files>` is a space separated list of the files on which the target depends. Also called **prerequisites**.

`command line i` command that must be executed to rebuild the target. Also called **recipe**

- Each command line **must start** with a **tab**

Example




```
grd2: grd2.o vid.o
        gcc -Wall grd2.o vid.o -o grd2

grd2.o: grd2.c vid.h
        gcc -Wall grd2.c -o grd2.o
vid.o: vid.c vid.h
        gcc -Wall vid.c -o vid.o
```

Invoking Make

`make`

1. Searches for files with names `makefile` or `Makefile` in current directory, in that order
   - And reads the one it finds first
2. Processes the first rule in the file read
   - Usually, this rule has the executable as the target

`make <target>` Processes the rule for the specified target

`make -f <makefile filename>` Reads in the makefile with the specified name

GNU `make` Variables or Macros

- `make` supports the definition of variables, or macros
- `make` variable definition is similar to `#define` directives of the C preprocessor:
  - A variable is defined once, and may be used at different points
  - `make` replaces a “variable” name by the text used in its definition
- Common use of `make` variables include:
  - Names of tools such as compiler, assembler or linker
  - Names of options to use with those utilities, including directories to be searched for
  - Lists of filenames to be used in targets
- The use of variables or macros makes it easier to manage and port a makefile

GNU `make` Variables or Macros: Examples

```
CC = gcc
CFLAGS = -Wall
OBJS = grade2.o video-graphics.o
HDRS = utypes.h video-graphics.h
EXEC = grade2.exe

$(EXEC): $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) -o $(EXEC)

grade2.o: grade2.c $(HDRS)
        $(CC) $(CFLAGS) -c grade2.c

video-graphics.o: video-graphics.c $(HDRS)
        $(CC) $(CFLAGS) -c video-graphics.c
```

Special/Automatic `make` Variables

`CC` The C compiler filename.

`CFLAGS` Special options that are added to built-in C rule

`$@` Full name of the current target

`$^` Prerequisites

`$?` Prerequisites that are newer than the target

`$<` The first prerequisite

Special/Automatic `make` Variables: Example

```
CC = gcc
CFLAGS = -Wall
OBJS = grade2.o video-graphics.o
HDRS = utypes.h video-graphics.h
EXEC = grade2.exe

$(EXEC): $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) -o $@

grade2.o: grade2.c $(HDRS)
        $(CC) $(CFLAGS) -c $<

video-graphics.o: video-graphics.c $(HDRS)
        $(CC) $(CFLAGS) -c $<
```

Predefined/Implicit Rules

- `make` has some built-in rules that simplify the writing of makefiles
- These rules depend on the language. For the C language:

  `n.o` is made automatically from `n.c` with a recipe of the form `$(CC) $(CPPFLAGS) $(CFLAGS) -c`
- Other relevant rules supporte by Gnu `make` are rules for assembling and linking

Pattern Rules

- Pattern rules use the character `%` on the target for pattern matching

```
CC = gcc
CFLAGS = -Wall
OBJS = grade2.o video-graphics.o
XHDRS = utypes.h
EXEC = grade2.exe

$(EXEC): $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) -o $@

%.o: %.c %.h $(XHDRS)
        $(CC) $(CFLAGS) -c $<
```

Phony Targets

- A phony target is a target that is not the name of a file
  - It is just a name for a recipe to be executed
  - Thus its prerequisites are empty
- If a file with the name of a phony target ever exists, the application of implicit rules will prevent the recipe from executing
- The `.PHONY` target tells `make` that the corresponding rule should be handled specially

```
.PHONY clean
clean:
        rm *.o *~
```

Further Reading

- Ben Yoshino, Make - a tutorial
- Byron Weber Becker A GNU Make Tutorial
- GNU 'make' – the ultimate reference for GNU's `make`