COMP 2718: Software Development Tools: gcc and make

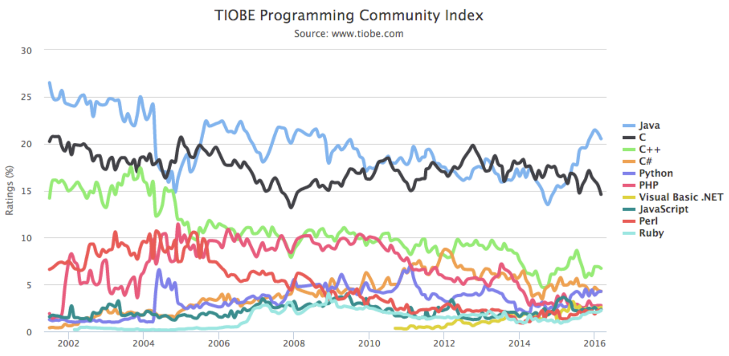
Slides adapted by Andrew Vardy (Memorial University)
Originally created by Marty Stepp, Jessica Miller, and Ruth Anderson (University of Washington)
http://www.cs.washington.edu/390a/

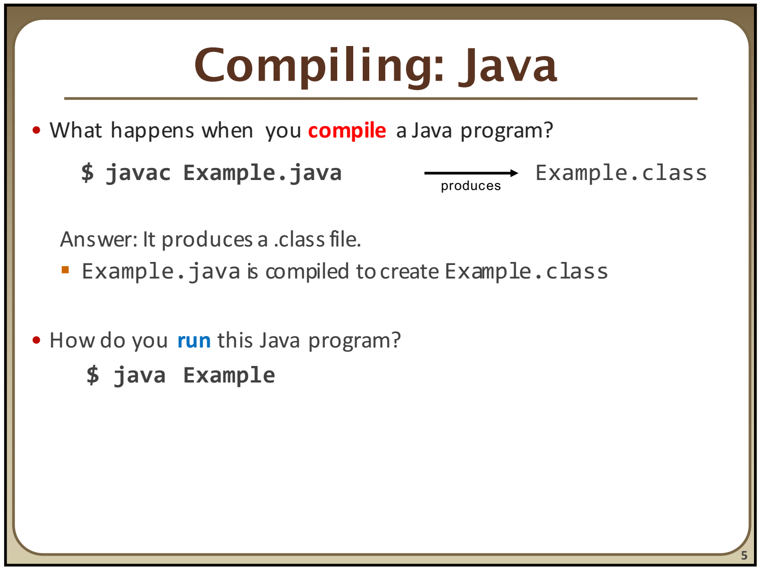
Outline

- We will look at the following software development tools:
  - The gcc compiler
  - The make build tool (usually used for C/C++)
  - (later) The ant built tool (usually used for Java)
- Along the way we'll introduce some additional concepts:
  - Compiling and linking
  - Object files
- We will not focus on the actual programming language concepts, but on the tools themselves. So don't worry if you don't know or remember either Java or C/C++.

C and Java

- We will discuss tools for software development, mostly with respect to C and Java
- C is one of the most well-used programming languages
  - C++ is the object-oriented successor to C
  - Yet C continues to be used, particularly for embedded systems
  - C/C++ are compiled languages. The compiler generates a machine language executable program.
- Java is also extremely popular and shares many of its object-oriented features with C++
  - Java programs are compiled to Java bytecode (not machine language)
  - Java bytecode runs on a Java Virtual Machine (JVM) which can be implemented on a wide array of platforms according to the Java slogan: "Write once, run anywhere"





Compiling: Java

- What happens when you **compile** a Java program?

```
$ javac Example.java      produces →   Example.class
```

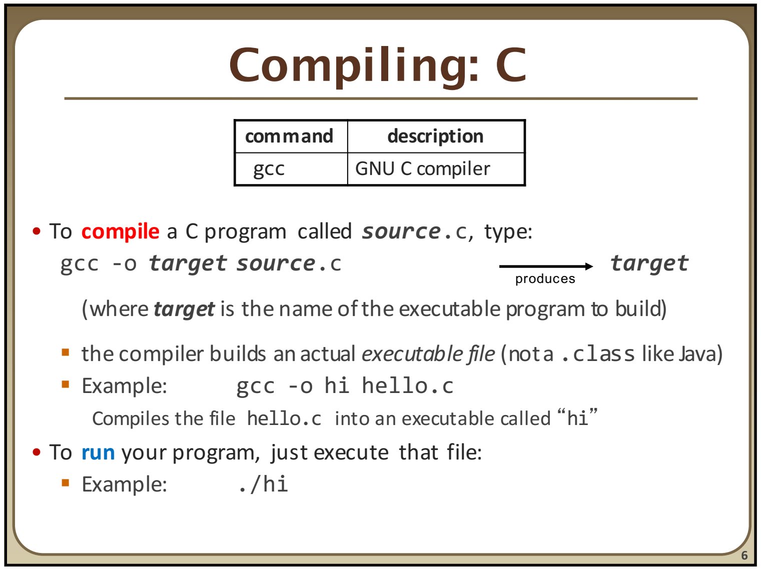
Answer: It produces a .class file.

  - `Example.java` is compiled to create `Example.class`

- How do you **run** this Java program?

```
$ java Example
```

5



Compiling: C

command	description
gcc	GNU C compiler

- To **compile** a C program called ***source***`.c`, type:

```
gcc -o target source.c      produces →   target
```

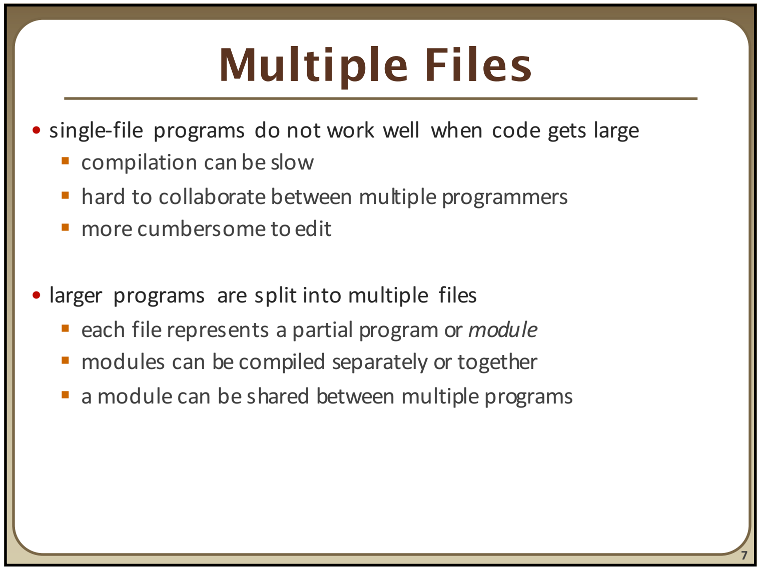
(where ***target*** is the name of the executable program to build)

  - the compiler builds an actual *executable file* (not a `.class` like Java)
  - Example: `gcc -o hi hello.c`

    Compiles the file `hello.c` into an executable called "hi"

- To **run** your program, just execute that file:
  - Example: `./hi`

6

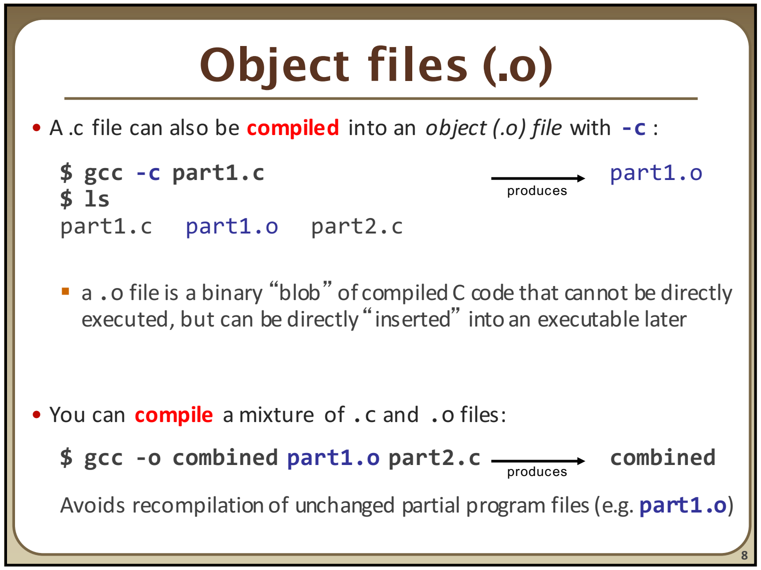


Multiple Files

- single-file programs do not work well when code gets large
  - compilation can be slow
  - hard to collaborate between multiple programmers
  - more cumbersome to edit

- larger programs are split into multiple files
  - each file represents a partial program or *module*
  - modules can be compiled separately or together
  - a module can be shared between multiple programs

7



Object files (.o)

- A .c file can also be **compiled** into an *object (.o) file* with `-c` :

```
$ gcc -c part1.c      produces →   part1.o
$ ls
part1.c   part1.o   part2.c
```

  - a `.o` file is a binary "blob" of compiled C code that cannot be directly executed, but can be directly "inserted" into an executable later

- You can **compile** a mixture of `.c` and `.o` files:

```
$ gcc -o combined part1.o part2.c   produces →   combined
```

Avoids recompilation of unchanged partial program files (e.g. **`part1.o`**)

8

Header files (.h)

- **header** : A C file whose only purpose is to be #included (#include is like java import statement)
  - generally a filename with the `.h` extension
  - holds shared variables, types, and function declarations
  - similar to a java interface: contains function *declarations* but *not implementations*
- key ideas:
  - every ***name*.c** intended to be a module (not a stand alone program) has a ***name*.h**
  - ***name*.h** declares all global functions/data of the module
  - other `.c` files that want to *use* the module will #include ***name*.h**

Compiling large programs

- Compiling *multi-file* programs repeatedly is cumbersome:

```
$ gcc -o myprogram file1.c file2.c file3.c
```

- Retyping commands like this is wasteful:
  - often the required compile command is much longer for larger projects
  - even if one file is changed (e.g. file3.c) all files need to be recompiled; again this becomes really time consuming for larger projects
- We'll look at this through the following ***Running Example***...

Running Example

```
------------------------------------------------------------------
--- hello.c
------------------------------------------------------------------
/* Classic "hello world" program, split into three .c files. */

#include "file2.h"
#include "file3.h"
#include <stdio.h>

int main(void) {
    print_hello();
    printf(" ");
    print_world();
    printf("\n");
}


------------------------------------------------------------------
--- file2.h
------------------------------------------------------------------
/* Print "hello". */
void print_hello(void);


------------------------------------------------------------------
--- file2.c
------------------------------------------------------------------
#include "file2.h"
#include <stdio.h>

void print_hello(void) {
    printf("hello");
}


------------------------------------------------------------------
--- file3.h
------------------------------------------------------------------
/* Print "world". */
void print_world(void);


------------------------------------------------------------------
--- file3.c
------------------------------------------------------------------
#include "file3.h"
#include <stdio.h>

void print_world(void) {
    printf("world");
}
```

Running Example

- We have three modules: hello, file2, file3
- Two of the modules have both .h and .c files: file2, file3
  - These are intended to be used externally
- The third module is the main program: hello
  - No .h file because this module is the main one, it pulls in file2 and file3 with `include` statements
- Simplest solution to compile is the following one-liner:
  - `gcc -o hello hello.c file2.c file3.c`
- Using this one-liner we would have to re-compile if any file changes
- No intelligence built into this process; easy for mistakes to be made (e.g. file2.h is changed, but we forget to re-compile)

Running Example

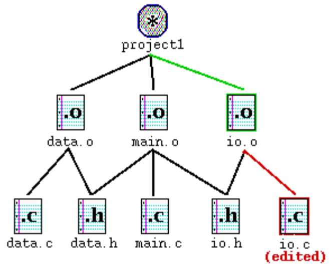
- The following solution is a bit better:
  - `gcc -c hello.c`
    - Generates hello.o
  - `gcc -c file2.c`
    - Generates file2.o
  - `gcc -c file3.c`
    - Generates file3.o
  - `gcc -o hello hello.o file2.o file3.o`
    - Generates hello executable
- Yes, this was more work to type, but the .o files can be separately re-compiled if necessary
- Still it would be good to encode all of this in a script or something; Even better if files could be made to compile only when necessary

make

- **make** : A utility for automatically compiling ("building") executables and libraries from source code.
  - a basic compilation manager
  - often used for C programs, but not language-specific
  - primitive, but still widely used due to familiarity, simplicity
  - similar programs: `ant`, `maven`, IDEs (Eclipse), ...
- **Makefile** : A script file that defines rules for what must be compiled and how to compile it.
  - Makefiles describe which files depend on which others, and how to create / compile / build / update each file in the system as needed.

Dependencies

- **dependency** : When a file relies on the contents of another.
  - can be displayed as a *dependency graph*
  - to build `main.o`, we need `data.h`, `main.c`, and `io.h`
  - if any of those files is updated, we must rebuild `main.o`
  - if `main.o` is updated, we must update `project1`



make Exercise

- **figlet** : program for displaying large ASCII text (like `banner`).
  - http://freecode.com/projects/figlet
- Download a piece of software and compile it with make:
  - download `.tar.gz` file
  - un-`tar` it
  - look at README file to see how to compile it
  - (sometimes) run `./configure`
    - for cross-platform programs; sets up make for our operating system
  - run `make` to compile the program
  - (optional) run sudo `make install` to install on your system
  - execute the program

Makefile rule syntax

```
target : source1 source2 ... sourceN
         command
         command
         ...
```

- ***source1*** through ***sourceN*** are the *dependencies* for building ***target***
- Make will execute the ***command**s* in order

Example:

```
myprogram : file1.c file2.c file3.c
        gcc -o myprogram file1.c file2.c file3.c
```

this is a tab THIS IS NOT spaces!!

- The ***command*** line must be indented by a single tab
  - not by spaces; ***NOT BY SPACES!*** SPACES WILL NOT WORK!

Running make

```
$ make target
```

- uses the file named `Makefile` in current directory
- Finds a rule in Makefile for building ***target*** and follows it
  - if the ***target*** file does not exist, or if it is older than any of its *sources*, its *commands* will be executed

- variations:

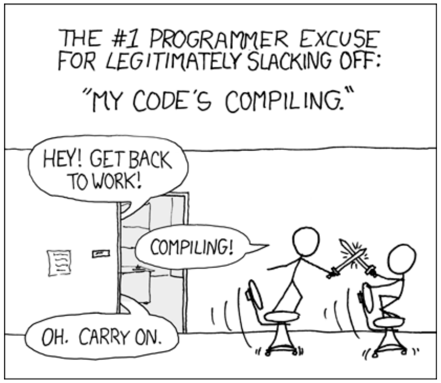
```
$ make
```

- builds the *first* target in the Makefile

```
$ make -f makefilename
$ make -f makefilename target
```

- uses a makefile other than `Makefile`

Making a Makefile



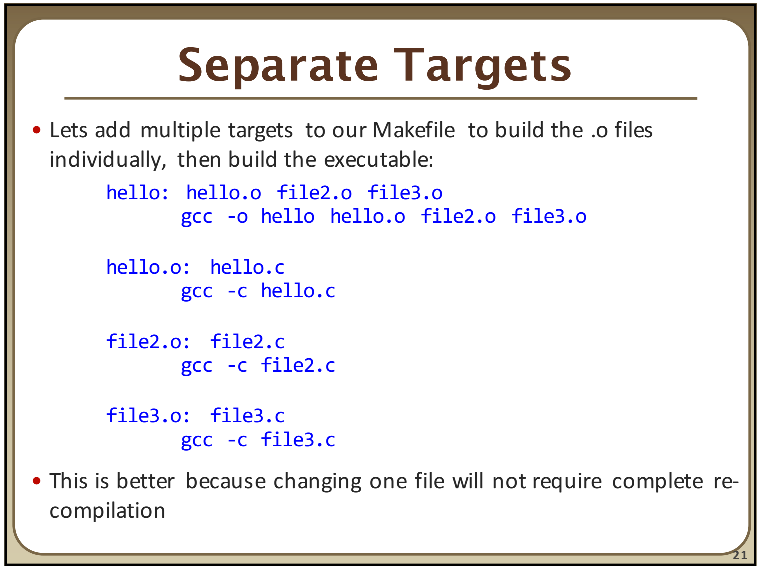
courtesy XKCD

Making a Makefile

- Here is a very simple makefile for our running example:

```
hello: hello.c file2.c file3.c
       gcc -o hello hello.c file2.c file3.c
```

- Its nice and short but notice that it is really just the same as our first gcc "one-liner", only wrapped in the syntax of a makefile although it does have some advantages:
  - Documents build instructions
  - Will re-build hello if any of the .c files change (if make is called)
- The disadvantage is that a change requires compilation of all files.



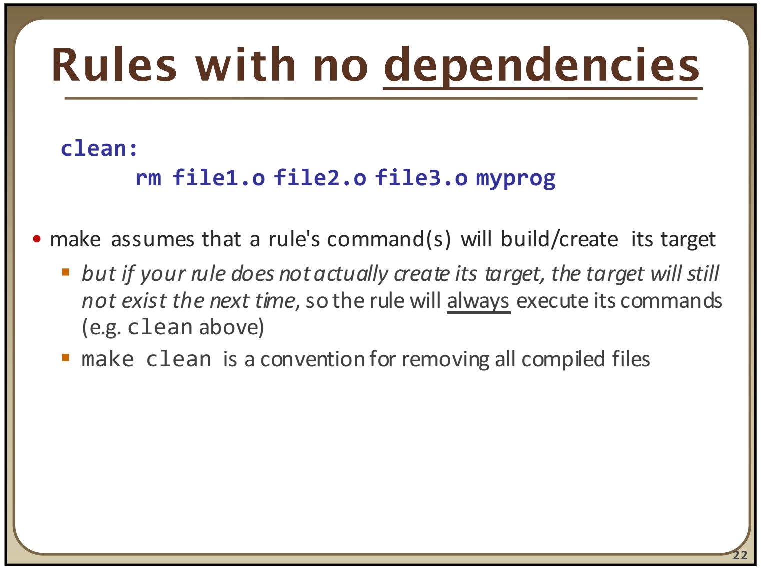
Separate Targets

- Lets add multiple targets to our Makefile to build the .o files individually, then build the executable:

```
hello: hello.o file2.o file3.o
       gcc -o hello hello.o file2.o file3.o

hello.o: hello.c
       gcc -c hello.c

file2.o: file2.c
       gcc -c file2.c

file3.o: file3.c
       gcc -c file3.c
```

- This is better because changing one file will not require complete re-compilation

21

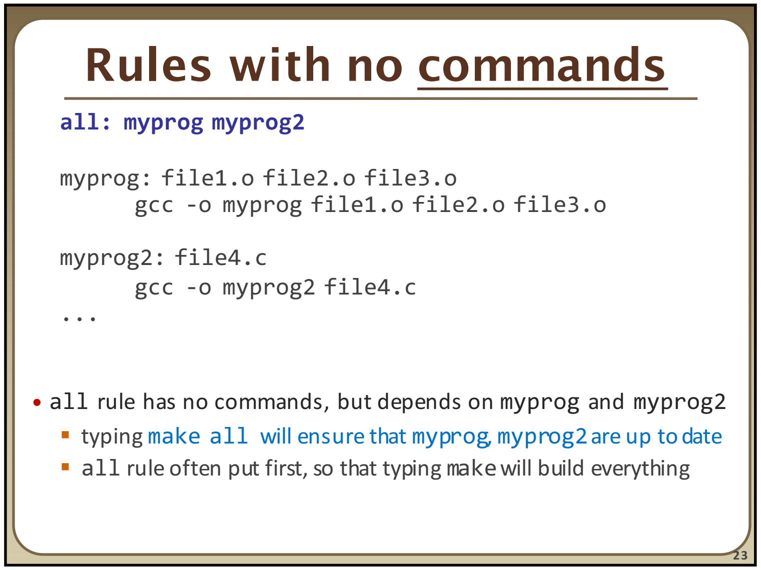


Rules with no dependencies

```
clean:
      rm file1.o file2.o file3.o myprog
```

- make assumes that a rule's command(s) will build/create its target
  - *but if your rule does not actually create its target, the target will still not exist the next time*, so the rule will always execute its commands (e.g. `clean` above)
  - `make clean` is a convention for removing all compiled files

22

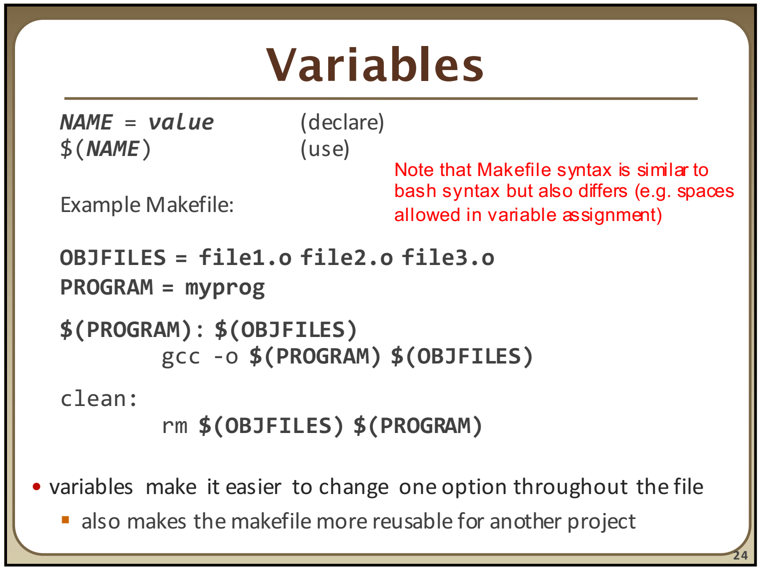


Rules with no commands

```
all: myprog myprog2

myprog: file1.o file2.o file3.o
      gcc -o myprog file1.o file2.o file3.o

myprog2: file4.c
      gcc -o myprog2 file4.c
...
```

- `all` rule has no commands, but depends on `myprog` and `myprog2`
  - typing `make all` will ensure that `myprog`, `myprog2` are up to date
  - `all` rule often put first, so that typing `make` will build everything

23



Variables

```
NAME = value          (declare)
$(NAME)               (use)
```

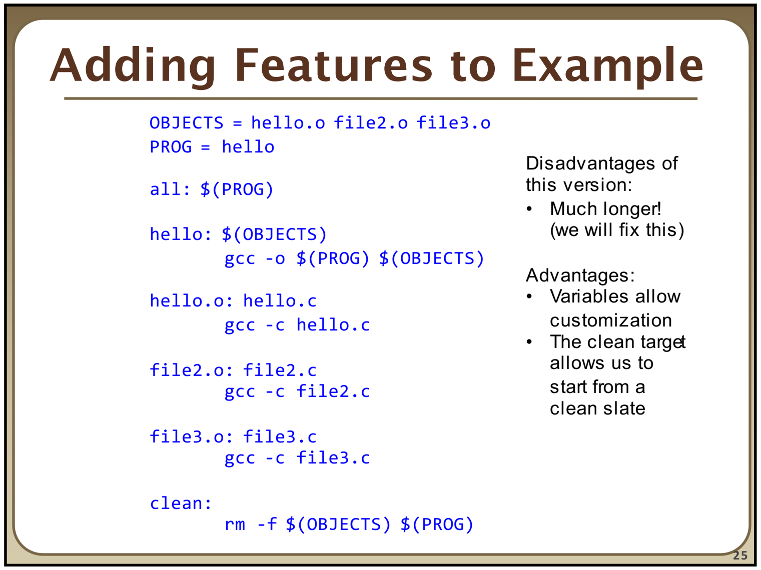
Example Makefile:

Note that Makefile syntax is similar to bash syntax but also differs (e.g. spaces allowed in variable assignment)

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog

$(PROGRAM): $(OBJFILES)
        gcc -o $(PROGRAM) $(OBJFILES)

clean:
        rm $(OBJFILES) $(PROGRAM)
```

- variables make it easier to change one option throughout the file
  - also makes the makefile more reusable for another project

24



Adding Features to Example

```
OBJECTS = hello.o file2.o file3.o
PROG = hello

all: $(PROG)

hello: $(OBJECTS)
        gcc -o $(PROG) $(OBJECTS)

hello.o: hello.c
        gcc -c hello.c

file2.o: file2.c
        gcc -c file2.c

file3.o: file3.c
        gcc -c file3.c

clean:
        rm -f $(OBJECTS) $(PROG)
```

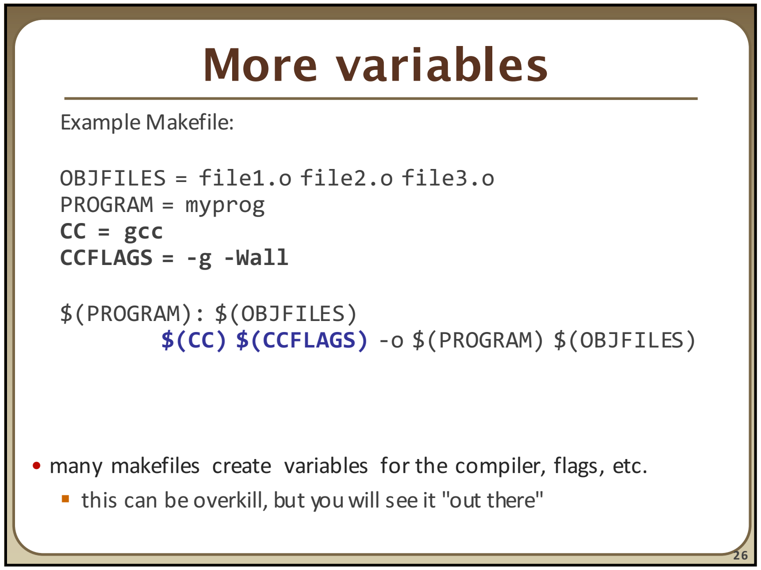
Disadvantages of this version:

- Much longer! (we will fix this)

Advantages:

- Variables allow customization
- The clean target allows us to start from a clean slate

25



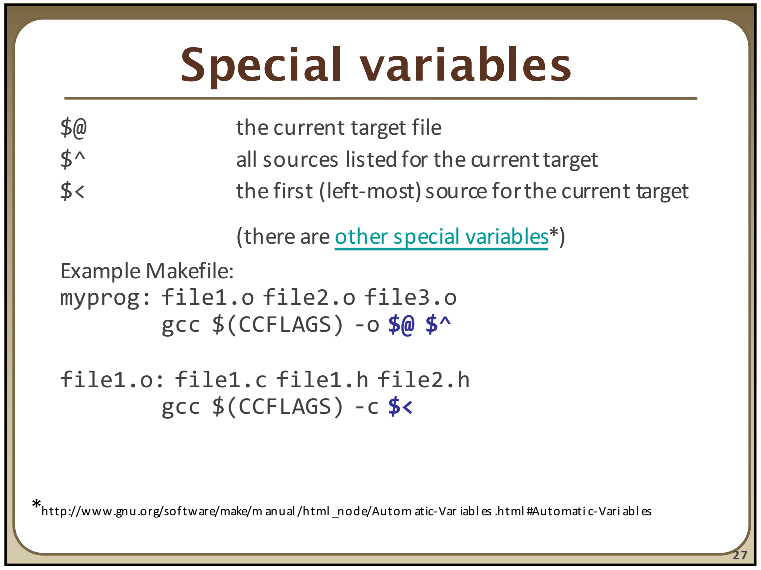
More variables

Example Makefile:

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog
CC = gcc
CCFLAGS = -g -Wall

$(PROGRAM): $(OBJFILES)
        $(CC) $(CCFLAGS) -o $(PROGRAM) $(OBJFILES)
```

- many makefiles create variables for the compiler, flags, etc.
  - this can be overkill, but you will see it "out there"

26



Special variables

$@	the current target file
$^	all sources listed for the current target
$<	the first (left-most) source for the current target

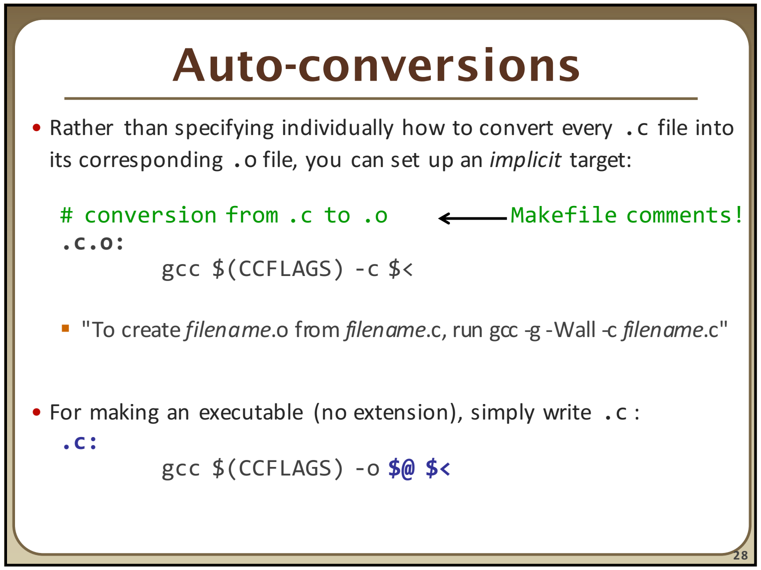
(there are other special variables*)

Example Makefile:

```
myprog: file1.o file2.o file3.o
        gcc $(CCFLAGS) -o $@ $^

file1.o: file1.c file1.h file2.h
        gcc $(CCFLAGS) -c $<
```

*http://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables

27



Auto-conversions

- Rather than specifying individually how to convert every .c file into its corresponding .o file, you can set up an *implicit* target:

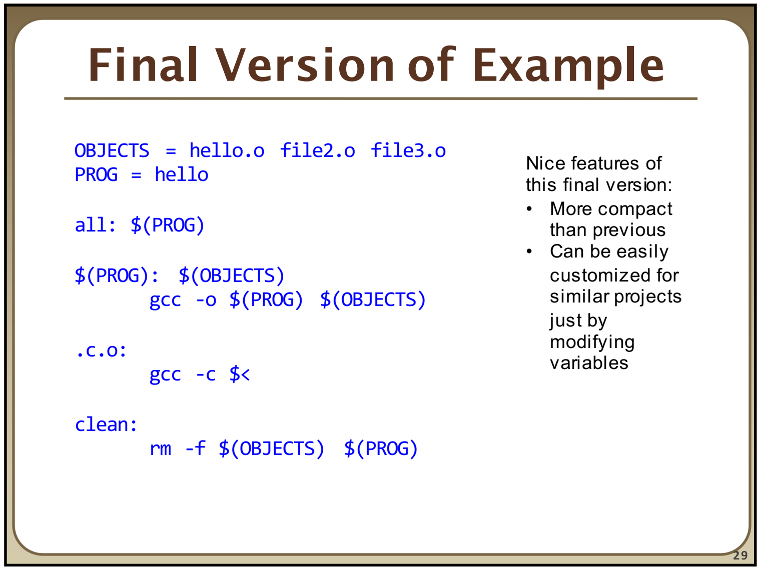
```
# conversion from .c to .o        ←——Makefile comments!
.c.o:
        gcc $(CCFLAGS) -c $<
```

  - "To create *filename*.o from *filename*.c, run gcc -g -Wall -c *filename*.c"

- For making an executable (no extension), simply write .c :

```
.c:
        gcc $(CCFLAGS) -o $@ $<
```

28



Final Version of Example

```
OBJECTS = hello.o file2.o file3.o
PROG = hello

all: $(PROG)

$(PROG): $(OBJECTS)
        gcc -o $(PROG) $(OBJECTS)

.c.o:
        gcc -c $<

clean:
        rm -f $(OBJECTS) $(PROG)
```

Nice features of this final version:

- More compact than previous
- Can be easily customized for similar projects just by modifying variables