Lecture 7: Make Automatic Variables

Kenneth M. Anderson

Software Methods and Tools

CSCI 3308 - Fall Semester, 2003

Today's Lecture

- Explore the topic of make's automatic variables in detail
- Brooks' Corner: The Surgical Team

Automatic Variables

- Make has a special feature called *automatic variables*
- Automatic variables can only be used within the actions of a make rule
  - The value of an automatic variable depends on the target and dependencies of the rule in which it occurs

Automatic Variables, cont.

- $@ - The target of the rule.
- $< - The first dependency.
- $^ - All of the dependencies.
- $? - All of the dependencies that are newer than the target
- $* - The stem of a pattern matching rule.
  - e.g. If you are building input.o from input.c the stem is "input"
  - This only works with pattern matching rules (the topic of Lecture 8)

Automatic Variables, cont.

- Below is a makefile that shows each rule with two actions: the first is a standard action, the second is the same action using automatic variables

```
program: input.o output.o
    g++ input.o output.o -o program
    g++ $^ -o $@
input.o: input.c defs.h
    g++ -c input.c -o input.o
    g++ -c $< -o $@
output.o: output.c defs.h
    g++ -c output.c -o output.o
    g++ -c $< -o $@
```

Use of $?

- **The variable $? can be useful for updating tar files**
  - **for example, where only those files that have changed need to be replaced.**

```
lab5.tar: README makefile lab5.cpp
   tar rf lab5.tar $?
```

The View Path

- Make applies special meaning to another variable, VPATH, also know as the view path.
  - While VPATH is not an automatic variable, it does interact with them (as we shall see shortly)
- VPATH consists of a list of directories, just like the `path` variable of the shell
  - If make cannot find a dependency in the current directory, it looks in the view path.
  - Note: just because a file is found in the view path does not mean that the shell can find it when executing commands (see next slide)

View Path Example

```
VPATH = $(HOME)/csci3308/src/lab05
lab05.o: lab05.c
   g++ -c lab05.c -o lab05.o
```

% make

```
g++ -c lab05.c -o lab05.o
g++: lab05.c: No such file or directory
```

- **Why does the action fail?**
  - **Assume we invoked the command in a build directory**

View Path Example, cont.

- To solve the problem, we can use automatic variables
  - When a file is found in the view path, automatic variables are set to contain the full path of the file
- Therefore, our action line needs to make use of automatic variables to reference files in an action
  - The full path of the file will be passed to the shell, which will then be able to find the file

View Path Example, continued

```
VPATH = $(HOME)/csci3308/src/lab05
lab05.o: lab05.c
   g++ -c $< -o $@
% make
g++ -c /home/.../src/lab05/lab05.c -o lab05.o
```

- Here, the action references lab05.c correctly. Note: lab05.o is created in the current directory, even though the source code is located elsewhere (which is similar to how we compiled gnuchess in lab 1)

Accessing File Information

- Using automatic variables, a file's name and directory can be extracted

```
 VPATH = $(HOME)/csci3308/src/lab05
 lab05.o: lab05.c
    echo found file $(<F)
    echo in directory $(<D)
% make
 found file lab05.c
 in directory /home/.../src/lab05
```

Brooks' Corner: The Surgical Team (Chapter 3)

- Or
  - How should a development team be arranged?
- The problem
  - Good programmers are much better than poor programmers
    - typically 10 times better in productivity
    - typically 5 times better in terms of program elegance
  - but we often do not have access to these "super programmers"

The dilemma of team size

- Consider the following example
  - 200-person project with 25 managers
    - where the managers are also experienced software developers
  - Previous slide argues for firing the 175 workers and use the 25 managers as the development team!
    - However, this is still bigger than “the ideal” small team size of 10 people (general consensus)
  - However, the original team of 200 was too small to tackle very large systems
    - OS/360 had over 1000 people working on it; consumed 5000 person-years of design, construction, and documentation!

Two needs to be reconciled

- For efficiency and conceptual integrity
  - a small team is preferred
- To tackle large systems
  - considerable resources are needed
- One solution
  - Harlan Mill’s Surgical Team approach
    - One person performs the work
      - all others perform support tasks
  - This is only one approach, there are many!

The Proposed Team

- The surgeon
  - The chief programmer
- The co-pilot
  - Like the surgeon but less experienced
- The administrator
  - Relieves the surgeon of administrative tasks
- The editor
  - Proof-edits documentation
- Two secretaries
  - Support admin and editor
- The program clerk
  - Probably obsolete today
- The toolsmith
  - Supports the work of the surgeon
- The tester
- The language lawyer

How is this different?

- Normally, work is divided equally
  - Now, only surgeon and copilot divide the work
- Normally, each person has equal say
  - Now, the surgeon is the absolute authority
- Note communication paths are reduced
  - Normally 10 people => 45 paths
  - Surgical Team => at most 13 (See Fig. 3-1.)

How does this scale?

- Reconsider the 200 person team
  - Communication paths => 19,900!
- Create 20, ten-person surgical teams
- Now, only 20 surgeons must work together
  - 20 people => 190 paths
    - Two orders of magnitude less!
- Key problem is ensuring conceptual integrity of the design

The Modern Surgical Team

- The surgical team, as conceived by Mills and described by Brooks, is not widely used today
- On Internet time, the chief programmer approach is impractical
- Now, it is more important that there be one to three designers, or software architects, that guide the design of the system
  - with many people implementing the system
  - This is true of the Microsoft approach
    - The program manager is responsible for the feature set