Lecture 7: Make Macros

Kenneth M. Anderson
Software Methods and Tools
CSCI 3308 - Fall Semester, 2004

Today's Lecture

- Brief review of make
- Explore make "macros" in more detail
  - Note: when you see "macro" think "variable"
- Brooks' Corner: The Mythical Man-Month
- but first... a quick look at Ant (a build management tool for Java programs)

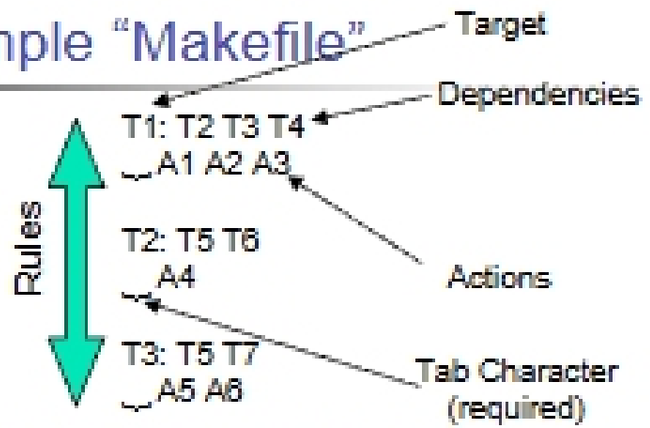
Unix Build Management

- In Unix environments, a common build management tool is "make"
  - Make provides very powerful capabilities via three types of specification styles
    - declarative
    - imperative
    - relational
  - These styles are combined into one specification: "the make file"

Make Specification Language

- Hybrid Declarative/Imperative/Relational
  - Dependencies are Relational
    - Make specifies dependencies between artifacts
  - Rules are Declarative
    - Make specifies rules for creating new artifacts
  - Actions are Imperative
    - Make specifies actions to carry out rules

Example "Makefile"



If a dependency changes, a rule's actions are executed to (re)create a rule's target

Make "Macros" - think "Variables"

- Make has variables known as "macros"
  - They are similar to shell variables with a few differences
    - Macros hold a string value
    - Macros are defined using an equal sign

```
INSTALLDIR = /home/faculty/kane/tmp/
```

    - And is used by preceding its name with a dollar sign

```
$(INSTALLDIR)/program : program
    cp program $(INSTALLDIR)
```

    - The parentheses **are required**, otherwise make assumes that a macro name is just one letter long
      - $INSTALLDIR is interpreted by **make** as $(I)NSTALLDIR

Macro Substitution

- Make variables perform strict textual replacement so the following two rules are equivalent
- (Do not do this in practice!):

```
program: output.o
   g++ output.o -o program
FOO = o
pr$(FOO)gram: $(FOO)utput.$(FOO)
   g++ $(FOO)utput.$(FOO) -$(FOO) pr$(FOO)gram
```


Using a '$' sign

- Since the dollar sign has special meaning…
  - it indicates the use of a macro
- …you need to "escape" it with a 2nd dollar sign, if you want it passed to the shell as part of an action
  - Note: make strips one of the dollar signs before invoking a shell to process the action
- Example: 'chapter$' is passed to `egrep` below

```
TableOfContents: book.txt
   egrep chapter$$ book.txt > TableOfContents
```

Increased Abstraction

- Macros increase the level of abstraction in a Makefile

```
program: main.o input.o output.o
    g++ main.o input.o output.o -o program
```

- is equivalent to

```
EXECUTABLE = program
OBJECTS    = main.o input.o output.o
$(EXECUTABLE): $(OBJECTS)
    g++ $(OBJECTS) -o $(EXECUTABLE)
```

- They can also save keystrokes

Increased Abstraction, cont.

- Why is this increase in abstraction important?
  - What benefit does abstraction typically provide?
- Definition of Abstraction
  - Identify the important aspect of a phenomenon and ignore the details

Increased Abstraction, cont.

- Allows the user of an abstraction to be independent of the hidden details
  - This allows the details to change without a user knowing about it (or caring)
- In makefiles, abstraction lets rules be defined that can be applied to many different situations

```
$(EXECUTABLE): $(OBJECTS)
    g++ $(OBJECTS) -o $(EXECUTABLE)
```

- The above rule can be applied to almost any C++ or C program

Definition and Use of Make Macros

- A shell script is executed from top to bottom. As such, a shell variable cannot be used before it is defined.
- Makefiles, on the other hand, are not executed top to bottom. Execution follows dependencies which can be anywhere in the file
  - As such, there is no concept of one rule coming before or after another rule
  - Therefore, all rules and macros are read entirely before the make algorithm is executed