cis20.2
design and implementation of software applications II
spring 2008
session # I.2
software project management

topics:

- source code management and version control
- makefiles

software development models.

- integrated development environment (IDE)
  - integrate code editor, compiler, build environment, debugger
  - graphical tool
  - single or multiple languages
  - VisualStudio, JCreator, Forte, ...
- Unix model
  - individual tools, command-line

source code management.

- problem: lots of people working on the same project
  - source code (C, Perl, ...)
  - documentation
  - specification (protocol specs)
- mostly on different areas
- different versions
  - *released* — maintenance only
  - *stable* — about to be released, production use
  - *development, beta*
- different hardware and OS versions

configuration management.

- version control system
- there are many popular tools:
  - CVS
  - RCS
  - SCCS
- collection of directories, one for each "module"
- release control
- version control
- there is a single master copy ("repository") and local (developer) copies

about rcs.

- it doesn't build a system (alone)
- it isn't project management (alone)
- all changes are isolated vs. single logical change
- it can help with bug fix tracking
- it can help with track change verification
- it doesn't test program (regression testing)
- it is not a work flow or process model

setting up a repository.

- create a directory for the repository:

```
unix$ mkdir RCS
```

which creates an RCS directory under your current working directory

adding a file to the repository.

- use the "check in" command:

```
unix$ ci movie.c
RCS/movie.c,v  <--  movie.c
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> this file manipulates the movie database
>> .
initial revision: 1.1
done
```

- you'll be asked to enter a description of the file you are adding to the repository
- you only have to do this the first time a file is checked in

what's in the directory now?

- the directory:

```
unix$ ls -lt RCS
total 8
-r--------   1 sklar sklar  4338 Jan 28 11:27 movie.c,v
```

- notice that the file is only read-only by owner

the RCS file...

```
head    1.1;
access;
symbols;
locks; strict;
comment @ * @;


1.1
date    2008.01.28.16.27.27;    author sklar;  state Exp;
branches;
next    ;


desc
@this file manipulates the movie database
@


1.1
log
@Initial revision
@
text
@/* movie.c */

#include <stdio.h>
etc
```


checking a file out of the repository.

- there are two modes:
  - read-only
  - read-write
- command for read-only:

```
unix$ co movie.c
RCS/movie.c,v  -->  movie.c
revision 1.1
done
```

- command for read-write:

```
unix$ co -l movie.c
RCS/movie.c,v  -->  movie.c
revision 1.1 (locked)
done
```


locking files.

- checking out a file in read-write mode is called checking it out *with a lock*
- this means that only the user who checked out the file can check it back in and unlock the file
- you can also lock a file that is already checked out:

```
unix$ rcs -l movie.c
```

- if the file is already locked by another user, you'll be asked if you want to break the lock
- this can be bad...

getting file information.

- the *rlog* command is used to get information about files in the repository

```
unix$ rlog movie.c

RCS file: RCS/movie.c,v
Working file: movie.c
head: 1.1
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
this file manipulates the movie database
----------------------------
revision 1.1
date: 2008/01/28 16:27:27;  author: sklar;  state: Exp;
Initial revision
=============================================================================
```

finding out about locks.

- you can use rlog to find out which files are locked
- to find out which files are locked:

```
unix$ rlog -R -L RCS/*
RCS/movie.c,v
```


checking changed files back in.

- once you make a change to a file (and test it), you should check the file back into the repository

```
unix$ ci movie.c
RCS/movie.c,v  <--  movie.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> added comments
>> .
done
```

- you'll be asked to enter a message describing the changes you made
- if the file is unchanged, RCS is smart enough not to increment the revision number:

```
unix$ ci movie.c
RCS/movie.c,v  <--  movie.c
file is unchanged; reverting to previous revision 1.1
done
```


keeping the working directory clean.

- use the *rcsclean* command
- this removes from the current working directory all files that are checked out in read-only mode but have not been changed since they were checked out

```
unix$ rcsclean
rm -f movie.h
```


finding differences.

- the *rcsdiff* command is used to show the differences between the version in your current working directory and the version that was last checked in to RCS

```
unix$ rcsdiff movie.c
===================================================================
RCS file: RCS/movie.c,v
retrieving revision 1.2
diff -r1.2 movie.c
4a5
>    this program was developed by prof sklar.
```

using with your makefile.

- it is handy to integrate RCS into your makefile
- add a `DEFAULT` rule that will check files out of RCS for the purpose of building your project:

```
.DEFAULT:
co $(RCS)/$@,v
```

- add this line just after the `SUFFIXES` line
- you can also add `rcsclean` to your `clean` rule:

```
clean:
rcsclean
rm *.o
```


ident

- you can record version information directly in your source code
- place a line like this:

```
static char const rcsid[] = "$Id$";
```

in the global declaration section of your source code files

- after you check the file in and check it out again, RCS will automatically expand the tag:

```
static char const rcsid[] =
  "$Id: movie.c,v 1.5 2008/01/28 16:55:09 sklar Exp $";
```

- now you can use the `rcsid` variable in your program
- you can also use the *ident* command to see the values:

```
unix$ ident movie.c
movie.c:
     $Id: movie.c,v 1.5 2008/01/28 16:55:09 sklar Exp $
```


revision tagging.

- each revision increases rightmost number by one: 1.1, 1.2, ...
- more than one period implies branches
- versions of file = RCS revisions
- use the *rcs* command to set revisions and branches
- do *man rcsfile* for more information
- there's also a script called *rcsfreeze* which is handy for these functions, but it is not a standard part of RCS (unfortunately)

what is make?

- utility typically used for building software packages that are comprised of many source files
- determines automatically which pieces need to be rebuilt
- uses an input file (usually called `makefile` or `Makefile`) which specifies *rules* and *dependencies* for building each piece
- you can use any name for the makefile and specify it on the command line:

```
unix-prompt# make
unix-prompt# make -f myfile.mk
```

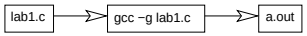
- first way (above) uses default (`makefile` or `Makefile`) as input file
- second way uses `myfile.mk` as input file

make tutorial (1)

- Let's begin by considering the simplest case of compiling a C program.
- Suppose that you have a C program called `lab1.c`.
- If you were going to compile this on the command line using the `gcc` compiler and send the output to the default file `a.out`, you'd execute the following command:

```
unix-prompt# gcc lab1.c
```

- This is illustrated in the figure below:

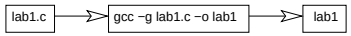


make tutorial (2)

- Now suppose you don't want to use the default output file name, but instead you want to name the output executable `lab1`.
- Then you would execute the following command:

```
unix-prompt# gcc lab1.c -o lab1
```

- This is illustrated in the figure below:



make tutorial (3)

- Next, suppose that you have a more complicated case — you have two C source files (`hw1.c` and `inv.c`), and you want to compile them and link them together to create one executable program called `hw1`.
- This is a multi-step process. First, you need to compile each source file into object code:

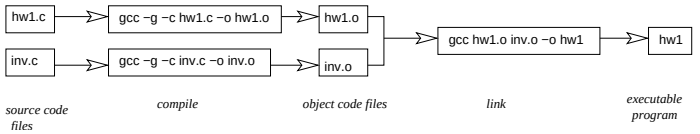
```
unix-prompt# gcc -c hw1.c -o hw1.o
unix-prompt# gcc -c inv.c -o inv.o
```

  The `-c` switch on the gcc command tells gcc to *compile only* and not link. So, after these two commands are executed, you have two object code files (`hw1.o` and `inv.o`). These files are not executable — they must be linked.
- We'll assume that `hw1.c` refers to components in `inv.c`, indicating that the files must be linked together. For example, `inv.c` contains the definition of a function called `printInventory()`, and `hw1.c` contains a call to that function. Thus `hw1.c` needs `inv.c` to resolve its "external references". Similarly, `hw1.c` contains a `main()` function, but `inv.c` doesn't; so the files must be linked together — or at least, `inv.c` needs to be linked to some file that contains a `main()`.

- The link step is as follows:

```
unix-prompt# gcc hw1.o inv.o -o hw1
```

- This is illustrated in the figure below:



make tutorial (4)

- The advantage of using a `makefile` is that it will keep track of which files need to be compiled when building an executable program that takes more than one source file.
- It is also easier to compile using make, because at the unix command line, all you have to type is:

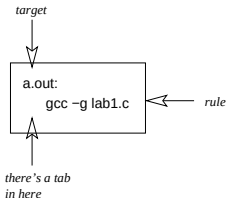
```
unix-prompt# make
```

- Note that sometimes you will also type a target as an argument to make, such as:

```
unix-prompt# make step1
```

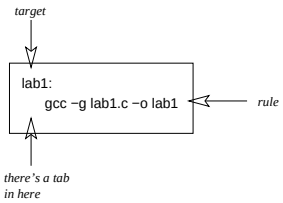

make tutorial (5)

- Just as above we started simple and added complexity, let's start with a very simple `makefile` and build from it. The figure below shows a simple example. The simple makefile has two components: a *target* and a *rule*.
- When you type `make` at the unix prompt, the make utility reads the `makefile` and executes the *rule* associated with the `a.out` *target*, namely it does exactly the same thing that is done on make tutorial page (1).



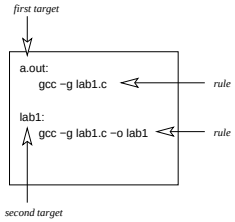
make tutorial (6)

- As above in going from "make tutorial pages (1) to (2), we add an output file name to the simple command, shown in the makefile in the figure below. The execution is the same.
- When you type `make` at the unix prompt, the make utility reads the `makefile` and executes the *rule* associated with the `lab1` *target*.



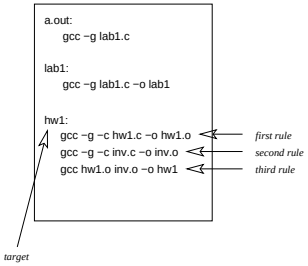
make tutorial (7)

- In both of the simple `makefile` examples above, there is only one *target*. If there were more than one target, then typing `make` with no arguments would cause the `make` utility to execute the rule associated with the first target that appears in the file.
- For example, in the figure below:
  - Typing `make`, would build target `a.out`.
  - Typing `make a.out`, would also build target `a.out`.
  - Typing `make lab1`, would build target `lab1`.



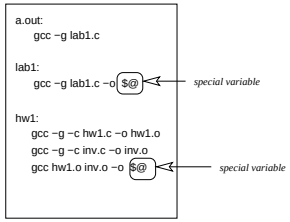
make tutorial (8)

- Okay, now let's move on to building a program that has two source files, as in make tutorial page (3).
- We add a target, `hw1`, to our makefile, as shown in the figure below. This new target has three rules, which get executed consecutively, just as if they were typed on the unix command line one after the other. To execute this target, type `make hw1` on the unix command line.



make tutorial (9)

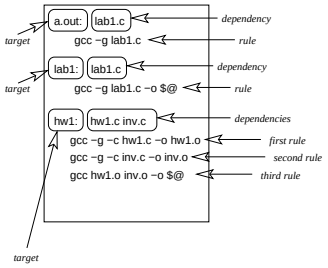
- Up to this point, we haven't really taken advantage of the power of the `make` utility, except to save ourselves some typing. Next, let's look at some of the features of `make`. We'll start by introducing some special variables.
- The variable `$@` is used inside a rule and it stands for the name of the target that the rule is associated with. For example, we could replace the `-o lab1` portion of the rule for the `lab1` target with `-o $@`. The meaning would be exactly the same. The figure below is the same makefile on make tutorial page (8), but uses the `$@` special variable.



- Note that we don't use this special variable with the first target's rule, because the name of the first target (`a.out`) is not speciifed in that target's rule.
- Also note that we don't use the special variable in the first two rules belonging to the `hw1` target, again because the name of the target (`hw1`) is not specified in either of these rules. Using the special variable does not change the way the makefile is executed.
- You would still, for example, type `make lab1` to build the second target.

make tutorial (10)

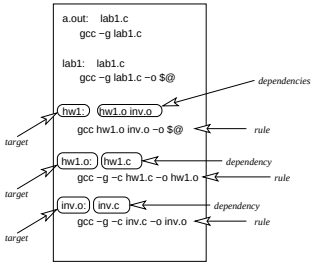
- Now let's talk about *dependencies*, which is one of the really nifty things about `make`. A dependency is something that tells `make` whether or not it needs to execute the rules associated with a target. Dependencies are listed on the same line as a target, after the colon (`:`) which follows the name of the target. Multiple dependencies are separated by spaces.
- In the figure below, our `makefile` includes dependencies for all three targets.



- For the first target (`a.out`), the dependency listed is `lab1.c`.
- When make executes to build this target, it compares the "last modified" date of the target file (if it exists) with the last modified date of its dependency. If the dependency is *newer* than its target, then the target's rule is executed. If a file bearing the same name as the target doesn't exist, then the target's rule is executed as well. The same goes for the second target. If the file `lab1` exists and it is older than its dependency, `lab1.c`, or if `lab1` doesn't exist, then the target's rule is executed.
- For the third target (`hw1`), there are two dependencies: `hw1.c` and `inv.c`. If either one of these files is newer than `hw1`, or if `hw1` doesn't exist, then the target's three rules are all executed.
- Adding dependencies doesn't change the way the `makefile` is executed. You would still type `make a.out` or `make lab1` or `make hw1` to build each of the three targets.

make tutorial (11)

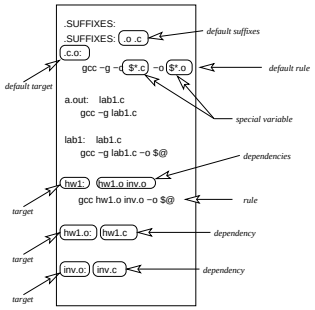
- Now, let's examine this third target a little more closely. To be more precise, the target itself, `hw1`, actually depends directly on `hw1.o` and `inv.o`, not the C source files.
- In addition, if you edit `hw1.c` but not `inv.c` since the last time you built the target, then you really only need to recompile `hw1.c`, not both `hw1.c` and `inv.c`.
- So let's split this up into its three constituent rules, as illustrated in the figure below.



- Doing this adds two targets to the makefile. This means that in addition to being able to type `make a.out` or `make lab1` or `make hw1` to build each of the three original targets, you could also build either of the two "intermediate" targets by typing `make hw1.o` or `make inv.o`.
- These are referred to as "intermediate" because building these two targets only results in updated object code, not executable programs.

make tutorial (12)

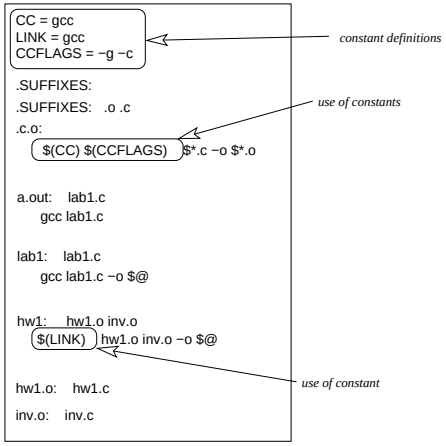
- You probably noticed in the figure on the previous page that the rules for the last two targets are very similar. Indeed, they are identical except for the file names. This is where *default rules* come in handy.
- In the figure below, the *rule* portions of the last two targets are removed and replaced by the single *default rule* at the top of the file.



- The default rule has two parts to it:
  1. The *SUFFIXES* define which file extensions have default targets associated with them.
  2. The *default target* is listed with its associated *default rule*. In this example, the *default target* gives a default rule for building any `.o` file out of a `.c` file.
- The default rule uses another special variable: `$*`. This variable stands for the *filename* portion of the dependency that invoked the rule. In other words, if `hw1.c` invoked the rule (because it was newer than its target `hw1.o`), then the special variable `$*` would take on the value `hw1`. Similarly, if `inv.c` invoked the rule (because it was newer than its target `inv.o`), then the special variable `$*` would take on the value `inv`.
- Note that these changes do not affect the way the `makefile` is executed. Default targets cannot be built directly by specifying them on the command line, so we still have 5 targets that can be built with this makefile; and each of these targets is specified in the same way as in the figure on make tutorial page (11).

make tutorial (13)

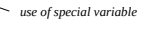
- Another feature of make is the ability to *user-defined constants*.
- The example shown in the figure below illustrates the use of three user-defined constants:
  - `CC` (which stands for the C Compiler)
  - `LINK` (which stands for the Linker)
  - `CCFLAGS` (which contains the flags to be used when the C Compiler is invoked)
- Note that the C Compiler and the Linker are actually the same program (`gcc`), but defining them separately provides the flexibility to use different programs for each if we wanted to do so.



make tutorial (14)

- The last feature of make that we have used is the special variable `$^`, which is used in a rule to stand for the list of a target's dependencies.
- This illustrated in the figure below.
- When make executes the target where the special variable is indicated, the value of `$^` is replaced with `hw1.o inv.o`, the list of dependencies which belong to that rule's target.

```
CC = gcc
LINK = gcc
CCFLAGS = −g −c

.SUFFIXES:
.SUFFIXES:  .o .c
.c.o:
     $(CC) $(CCFLAGS)    $*.c −o $*.o

a.out:   lab1.c
     gcc lab1.c

lab1:   lab1.c
     gcc lab1.c −o $@

hw1:    hw1.o inv.o
    $(LINK)  $^  −o $@

hw1.o:   hw1.c
inv.o:   inv.c
```



make: defining rules

- the syntax is:

```
<target> : <dependencies>
     <tab><command1>
     <tab><command2>
     ...
     <tab><commandN>
```

- there must be a `<tab>` at the beginning of each command line
- for example:

```
foo.o : foo.c defs.h       # rule for building foo.o
     cc -c -g foo.c
```


make: specifying targets

- you can specify a target on the command line:

```
unix-prompt# make -f myfile.mk install
```

- the default target is the first one in the makefile (i.e., if you don't specify a target on the command line)
- often you have the following targets:
  - `all`
  - `clean`
  - `install`

make: wildcards

- wildcard characters are `*`, `?` and `[...]` are the same as in the Bourne shell
- variables are also like in the Bourne shell (i.e., begin with `$`)
- but be careful because environment variables are imported into make
- there are a number of automatic variables:
  - `$@` = the file name of the rule target
  - `$?` = names of all dependencies that are newer than the target
  - `$^` = names of all dependencies
- you can also use `F` and `D` to get the file and directory (respectively) portions of full paths
- e.g., `$(@D)` and `$(@F)` return the directory and file names of the target

make: example

- example:

```
LIB  = $(HOME)/lib
INC  = $(HOME)/include
BIN  = $(HOME)/bin

RCS      = RCS
CC       = gcc
LINK     = gcc
CCFLAGS  = -c -g
```

- defines many variables
- which are refered to like this, e.g.: `$(CC)`
- notice use of `$(HOME)` which is read from the environment

make: implicit rules

- *implicit* rules can be used to define a general way of building one type of file from another
- for example

```
.SUFFIXES:
.SUFFIXES: .o .c

.c.o:
     $(CC) $(CCFLAGS) $*.c -o $*.o
```

- note use of variables

make: dependencies

- it is good practice to list include files as dependencies
- for example:

```
hw4sklarserver: hw4sklar.o util.o
     $(LINK) $(LDFLAGS) -o $@ $^

hw4sklar.o: hw4sklar.c hw4sklar.h
util.o: util.c util.h
```

- this will use the implicit rule to know how to build a `.o` file from a `.c` file