cis20.2-spring2008-sklar, unit I, lab 2

instructions

- This lab covers the use of RCS and make.

part I: using RCS

Here you will create a simple test program and an RCS directory to exercise the features of RCS. Refer to the class notes from Feb 4.

1. Start by creating a simple program, like **hello world**, in C++. Call your source file **hello.cc**.
2. Now create an **RCS** subdirectory in the working directory where you created your C++ file. Then check your C++ file into RCS by using the "check in" (ci) command:

   ```
   unix-prompt$ mkdir RCS
   unix-prompt$ ci hello.cc
   ```

3. Look at the file in your RCS directory:

   ```
   unix-prompt$ more RCS/hello.cc,v
   ```

   You'll see the special codes that RCS adds to track versions of the file.
4. Now check your file out of RCS using the "check out" (co) command:

   ```
   unix-prompt$ co hello.cc
   ```

   Note: do this from the same working directory where you did the "check in" command.
5. Look at the contents of both your working directory and your RCS directory:

   ```
   unix-prompt$ ls -l * RCS/*
   ```

   Check the file permissions on **hello.cc** in your working directory and **hello.c,v** in your RCS directory. Both files should have "read only" (`-r--r--r--`) permissions.
6. Now compile your program:

   ```
   unix-prompt$ g++ hello.cc -o hello
   ```

   and test it, just to make sure you still remember how to write a program in C++.

part II: using make

Here you will walk through a number of steps using the Unix **make** utility, starting by creating a simple **makefile**. Then you'll try it out, and then add more complexity to the makefile, following the notes from class on Feb 4. Stepping through the features of **make**, one at a time, testing after you add each step, should help you learn how this powerful utility works.

1. Create a file called **makefile** and put one target in it:

   ```
   hello:
        g++ hello.cc -o hello
   ```

   Don't forget to start the second line — the rule — with a tab character.

2. Test your makefile by running **make**:

   ```
   unix-prompt$ make
   ```

   Note that you should create your makefile and execute the make command from our working directory (not your RCS directory).

   When you run make, you might get a message like this:

   ```
   makefile:2: *** missing separator.  Stop.
   ```

   You'll get this if you forgot to put a **tab** at the beginning of the rule line (line 2).

   Or you might get a message like this:

   ```
   make: 'hello' is up to date.
   ```

   You'll get this if you had previously compiled your program and the executable ("hello") is still in the directory. If you get this message, you can test the makefile by removing the executable (`rm hello`) and running make again. (Don't remove the source file! just the executable)

3. Modify your make file to separate the compile and link steps, as well as establishing dependencies for the components:

   ```
   hello: hello.o
        g++ hello.o -o hello

   hello.o: hello.cc
        g++ -c hello.cc -o hello.o
   ```

   Don't forget to put a **tab** character at the beginning of each *rule* line.

   Try to compile your program again, by executing **make**, as above.

4. Now modify your makefile to use *default rules*:

   ```
   .SUFFIXES:
   .SUFFIXES: .o .cc

   .cc.o:
        g++ -c $*.cc -o $*.o

   hello: hello.o
        g++ hello.o -o hello

   hello.o: hello.cc
   ```

   And again, try your makefile. You may need to remove the object file (**hello.o**) and executable file (**hello**) in order to force make to recompile your file and avoid getting the "up to date" message.

   Another way to force make to recompile your file is to change its "last modified" date. Make decides what to compile by looking to see if a dependency is newer than its target; if it is, then make will follow the target's rule(s) to rebuild the target from the newer dependency.

   You can edit your source file (**hello.cc**) which will change the "last modified" date, or you can use the Unix **touch** command:

   ```
   unix-prompt$ touch hello.cc
   ```

   which simply changes the "last modified" date to the exact date and time when you execute the touch command.

   Then when you run **make**, the program will be recompiled and rebuilt.

part III: using RCS with make

Here we combine what you've learned about RCS and make. The two utilities can be used together to make your life easier and provide useful versioning information inside your program.

1. First, modify your makefile to include a default rule that tells the utility that if it cannot find a dependency or a target that builds that dependency, then it should look to see if the dependency exists in the RCS directory; if it does, then check it out to resolve the dependency.

```
.SUFFIXES:
.SUFFIXES: .o .cc

.cc.o:
     g++ -c $*.cc -o $*.o

.DEFAULT:
     co RCS/$@,v

hello: hello.o
     g++ hello.o -o hello

hello.o: hello.cc
```

2. Before trying this, check your source code file into RCS (if you have modified it since you checked it in before) or run **rcsclean** if you haven't modified it. This will remove your source code file from your working directory, giving then the opportunity to exercise the new default rule we just added. Now test your makefile:

```
unix-prompt$ rcsclean
unix-prompt$ make
```

   It should check **hello.cc** out of RCS and then compile and build it.

3. Finally, check out **hello.cc** with a lock:

```
unix-prompt$ co -l hello.cc
```

   and edit it to include a string constant set to the RCS variable `$Id$`, like this:

```
#include <iostream>
#include <string>
using namespace std;

static string const version = "$Id$";

int main() {
  cout << "hello world!\n";
  cout << "program version: " << version << endl;
}
```

   Run **make** on this version of your program, and test it. The output will look like this:

```
hello world!
program version: $Id$
```

Now check the file back in. Since you had checked out the file with a lock and have edited it, you will be asked to enter a log message describing what you did to the file.

Then run make.

Because of your default rule, make will now check the file out (again) and it will substitute for the RCS `$Id$` variable defined above a string containing versioning information. Run your program again, and you should get something like this:

```
hello world!
program version: $Id: hello.cc,v 1.2 2008/02/06 06:10:58 sklar Exp $
```

If you look at the file, you'll see that the 5th line contains a value something like the above, instead of just `$Id$`. Now, every time you edit the file, check it into RCS and back out again, the value of your C++ `version` variable will be updaated with the current version of the RCS variable `$Id$`.

Pretty cool, huh?

4. One more thing you can do to your makefile is make use of constants. This is frequently done with commands (like **g++**) to make it easy to change the compiler or other commands you are using, especially for configuring your makefile to work under different operating systems. An example is below:

```
CC = g++
RCS = RCS

.SUFFIXES:
.SUFFIXES: .o .cc

.cc.o:
     $(CC) -c $*.cc -o $*.o

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

hello: hello.o
     $(CC) hello.o -o hello

hello.o: hello.cc
```

5. And one more thing you can do is create a **clean** target in your makefile, by adding the lines below to the end of what's listed above:

```
clean:
     rcsclean
 rm hello.o hello
```

You can invoke this target by doing:

```
unix-prompt$ make clean
```

6. If you are using version control, it is a good idea to check your makefile into RCS. Check it in (`ci makefile`) for version control, but then you have to manually check it out (`co makefile`) in order to use it.