1. **Shells** When you are logged on to a terminal window in the lab, the various commands that you enter ("ls", "gcc", "man", etc.) are processed by a command interpreter called the shell. This is actually more than a command interpreter — it is a full-fledged programming language with variables, loops, input/output, etc. (Shell languages are sometimes classified under the broader category of “scripting languages.”) By default the user accounts in the lab are set up to use a shell called “tcsh.”

The original shell command scripting language was named “sh” and was invented by Stephen Bourne at AT&T. (Bourne, by the way, was the president of the ACM between 2000 and 2002.) This was followed by the C shell, csh, invented at Berkeley by Bill Joy (who went on to become one of the co-founders of Sun Microsystems), the Korn shell, ksh, invented by David Korn at AT&T, the bash shell, a superset of Bourne’s sh (the name bash stands for “Bourne Again SHell”), and many others. The tcsh shell is an enhanced version of csh (the “t” in tcsh comes from the TENEX/TOPS-10 operating system for which it was first designed).

2. **Return values.** You may have wondered what that “return 0;” really means at the end of the C program examples we have looked at. In the Unix operating system, there is a mechanism by which a process can notify the system of its exit status by returning an int value. Usually, a return value of “0” indicates “normal completion.” Other integer values can be used to signal a variety of errors, warnings, or status indications; these are often under the control of the programmer. One feature of tcsh (and other shells) is the existence of special variables that remember certain recent events. In particular, the variable “$?” remembers the return code of the most recently completed process.

Compile the following C program:

```c
int main() {
    return 123; /* or any other value from 0 through 255 */
}
```

Now execute the following two commands (I am assuming your executable is named a.out, but if you named it something else then use that instead):

```bash
./a.out
echo $?
```

The command “echo $?” should print out the value 123. This is useful when executing commands by means of a script (as opposed to direct execution from the terminal window.)
3. **C arrays.** Write a C program that creates an array of ten integers, an array of ten doubles, an array of ten floats, and an array of ten characters. Print out a “map” of the memory locations for all of these values. Your output should be sorted by increasing memory location and should look like this (these are not necessarily correct values!):

<table>
<thead>
<tr>
<th>Location</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCAE1100</td>
<td>i[0]</td>
</tr>
<tr>
<td>BCAE1104</td>
<td>i[1]</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

If there are “gaps” in the memory addresses don’t worry about them.

4. **MIPS and Memory** Today you are going to write your first programs in MIPS assembly language. We will be doing simple integer arithmetic using temporary registers and user-defined memory. We will be using the following MIPS commands (which are explained in your book, but will be briefly explained at the beginning of the lab period):

- **lw** – load word. Used to move a word from memory into a register.
- **sw** – store word. Used to move from a register into memory.
- **add, sub** – add and subtract.
- **la** – load address. Used to put a memory address into a register
- **li** – load immediate. Used to put a constant into a register
- **syscall** – system call. Used for special operations like I/O; each syscall has a distinct code that must be loaded into register v0 prior to the call; there may also be zero or more arguments that must be placed into argument registers.

Write a program that computes something other than the example and prints the result. All I’m interested in is having you make small changes to the sample MIPS program that is provided for you on the Web page.