Loops

Students were asked to work out (without running the code) the output of this program:

```c
int main(void) {
    int n = 5;
    int i;
    for(i=0;i<2;i++) {
        printf("computing %d minus %d ...", i, n);
        n = i-n;
        printf("n is %d\n", n);
    }
    return EXIT_SUCCESS;
}
```

**Answer:** Here is what Mary got on her machine (but get them to work this out logically):

```
[fletcher]mcryan: ./a.out
computing 0 minus 5 ...n is -5
computing 1 minus -5 ...n is 6
```
Programming

Students were asked to write a program which accepts a number \( n \), complain if the number is negative, and otherwise output \( n! \).

**scanf comment:** We told them in lab3 about the return value from scanf. But we didn’t mention the EOF (−1) return value on end of input. In the lab, some will have tested for \( == 1 \), while others will have tested for \( != 0 \). The latter fails on end of input. So you could mention this.

**Answer:** Here is a program to solve this (note I should have really specified “integer”, not “number”). It is possible some student will have written an \( n! \) function in their solutions (the Monday 12th and Tuesday 13th lectures deal with functions, though we won’t cover recursion yet).

In discussions with them, please emphasize the examination of the result of the `scanf` to detect whether an int was entered, or alternatively some non-integer input; remember to mention that if the user types something like 4.8, this will result in success with \( %d \) (reading just the 4).

This particular version (changed from previous years) illustrates one style of validation and error handling: validate early, bail out on error, and leave the main code uncluttered. I prefer this to wrapping in conditionals and leaving the return to the end. You could also discuss different styles.

```c
/* Tutorial 4, question b (program) */

#include <stdio.h>
#include <stdlib.h>

int main(void) { 
    int n, flag;
    int i, fac;
    printf("Please input a non-negative integer: ");
    flag = scanf("%d", &n);
    if ( flag != 1 || n < 0 ) {
        printf("That wasn’t a non-negative integer.\n");
        return EXIT_FAILURE;
    }
    /* main code */
    fac = 1;
    i = 2;
    while(i <= n) {
        fac = i*fac;
        i = i+1;
    }
    printf("The factorial of %d is %d.\n", n, fac);
    return EXIT_SUCCESS;
}
```
Functions

Students were asked to consider the following code:

```c
int i = 3;

int triple( int a ) {
    a = a*3;
    return a;
}

int main(void) {
    triple(i);
    printf("i, triple(i): %d, %d", i, triple(i));
}
```

They should determine what gets printed on the screen?

**answer:**

`i, triple(i): 3, 9`

Possibly you might wonder whether the action of the first statement in `main()` might have initially increased `i` to have the value 9, before getting to the `printf`. However, the action of the `triple(i);` statement is only the following:

- Takes the variable `i`, looks at its value, which is 3.
- Makes a call to `triple`, copying the value 3 into a newly-created variable `a`, which is the local variable to `triple`.
- Runs the steps of the `triple` function, and returns the updated value of `a` (which is 9) to the outside environment.
- Back in `main`, the environment receives the value 9, but this is *not* stored anywhere (because the statement was not of the form `j = triple(i)` or similar).
- Overall, this means that the effect of the initial call to `triple` on the `main` environment has been absolutely nothing.

Even though `i` is actually a *global variable* with scope throughout all functions, this is irrelevant - because the parameter passing into the function is done on a *call by value* basis; only the value is passed into the function. Note that because `i` is a global variable (declared above the `main` function), it can be modified by other functions *if it is referred to by its own name*. For example, if the interior of `triple` was described in terms of `i`, the changes made by that function *would persist* after the function call had terminated.