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]mcrayn: ./a.out
computing 0 minus 5 ... n is -5
computing 1 minus -5 ... n is 6
```

The next question is rather obsolete: this year I changed Lab2 `averagewithwhile.c` so that it uses the return value of `scanf`, rather than having the unfortunate forward dependency to `char` and the even more awkward 'skip over white space' directive.

So actually they already know this from Lab2 ... except that I didn’t there mention the possibility of returning -1, so you could go into that. In the lab, some of them checked the return value for `!= 1`, in which case their program also worked on eof; and some checked for `== 0`, in which case it typically loops on eof. So you could discuss this little point. Rest of comment is as in previous years:

**scanf comment:** Both of the next two questions are about loops, but students also need to know a bit more about `scanf` to solve these questions. The standard use in lectures and labs is just to write the function call with `scanf` as a command, ignoring any result:

```c
int a;
scanf("%d", &a);
```

(note the ‘magic’ ampersand that is getting explained this week). The above should read in an integer to the variable `a`.

However, despite this typical use, it is important to know that `scanf` returns an integer, which is the number of items it successfully read. Thus `scanf("%d", &a)` will return 1 if it read something into `a`. It will return 0 if the user inputs a non-integer, or -1 on error or end of input.
Programming

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

**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;
}
```

**Programming with scanf**

This question is similar to some of the lab 3 tasks. Students are asked to write a program which takes in a series of integers from the screen and calculates the sum of these integers. The user indicates that (s)he is finished by inputting any non-integer input.

**Note:** Remind the students about the value returned by `scanf` and how it relates to the number of successfully-read “placeholders” (`%`).

**Answers:** There are two possible solutions, the first one being more straightforward, the second being shorter (but involving more subtle use of C concepts). Here are the programs (with some explanatory comments).
/* Tutorial 4, question c (program) */
/* First way of answering the Question */

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

int main(void) {
    int n, flg;
    int sum = 0;
    printf("Please input a series of integers separated by whitespace.\n");
    printf("Indicate you are finished using any non-integer value.\n");
    /* This way requires less intelligence about C, but has annoying
       repetition */
    flg = scanf("%d", &n);
    while (flg == 1) {
        sum = sum+n;
        flg = scanf("%d", &n);
    }
    printf("The sum of the integers is %d.\n", sum);
    return EXIT_SUCCESS;
}

Here is the second way of answering the question:

/* Tutorial 4, question c (program) */
/* Second way of answering the Question */

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

int main(void) {
    int n;
    int sum = 0;
    printf("Please input a series of integers separated by whitespace.\n");
    printf("Indicate you are finished using any non-integer value.\n");
    /* This way is cleaner, but perhaps conceptually a bit harder */
    /* The expression in our 'while' below assigns as well as tests.
       * This can be hard to understand/get right unless you are very
       * comfortable with C. If not, use the approach in tutw4c1.c */
    while ( scanf("%d", &n) == 1 ) {
        sum = sum+n;
    }
    printf("The sum of the integers is %d.\n", sum);
    return EXIT_SUCCESS;
}

Some students may have found/invented the do { } while ( ) construct, which in this case is
of intermediate slickness.
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:**

\[i, \text{triple}(i): \ 3, \ 9\]

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 = \text{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.