Reminder

• HW04 due Thursday 2 Mar electronically by noon

• Are you registered for the course?

• Next week: Spring Break:
  – Hence no lecture, office hours or labs.

• Grading of HW01, HW02, HW03 coming!

• Register for piazza!
  – Only 46 out of 70 so far

• You will need the second textbook, Lyons, in about 2 weeks.

• My office hours:
  – 3:30-5pm Tuesdays in Room 022-C (our computer lab)

• TA office hours
  – In Room 022-C
    • Mondays 7-9:30pm
    • Tuesdays 4:30-6:30pm
    • Wednesdays 7-9:30pm
• Mid-term exam will be ***Tuesday 21 March*** at 7pm in this room
  – Three weeks from now!

  – Originally scheduled for 14 March. Reasons for change:
    • Want to ~finish off the introduction to C portion of the course and we have about 1.5 lectures left to do that
    • We’re not quite ready for a midterm exam in terms of amount of content
    • not the nicest thing to do to hold a midterm 2 days after return from Spring Break

  – Will cover everything we will have covered through 14 March lecture

  – Format: Mix of multiple choice, matching, short answer – no in-class coding, just need a **Number 2 pencil!**
Review and Outline

- Last time:
  - Operators
  - Functions
  - Conditional structures:
    - if, if/else, if/elseif/else
    - switch/case
  - Loops
    - Count-controlled loops: for()More on loops
    - Conditioned controlled loops: while(), do

- Today:
  - Review of loops
  - Recursive functions
  - Scope of variables
  - Commenting code
  - Static variables and external functions
  - Pointers
  - Pointers to Functions
  - Arrays in C
  - Passing arguments to main()
Loops!
Loops

• Loops allow computers to do what computers do best:
  – Execute repetitive boring tasks efficiently and accurately over and over again

• Two types:

  • **Count-controlled Loops:**
    We use these when we *know, beforehand*, how many times we want to repeat a series of tasks.

  • **Condition-controlled Loops:**
    These are used when we *don't know* how many repetitions will be needed, but we know that we want to stop when some well-defined thing happens.
Count-Controlled Loops!
A "for" loop is a count-controlled loop. Its has the form shown below.

General form of a "for" loop:
```c
for ( initialize counter ; test condition ; counter update ) {
    statement block;
}
```

Example:
```c
int i;
for ( i = 0 ; i < 10 ; i++ ) {
    printf("loop number %d\n", i);
}
```

Output:
```
Loop number 0
Loop number 1
Loop number 2
Loop number 3
Loop number 4
Loop number 5
Loop number 6
Loop number 7
Loop number 8
Loop number 9
```
How a `for()` loop works

The `for` statement:

```c
for (i = 0; i < 10; i++)
```

```c
printf("loop number %d\n", i);
```

- Initialize: `i = 0;`
- Test: `i < 10`?
- If never true, no loops
- Increment: `i++;

Diagram:
Every for() loop has four important parts:
- counter initialization
- test condition
- counter update
- body of execution

The for statement is very flexible because:
- Any valid C expression can be used for initialization or update, and
- Any valid condition can be used for the test condition.

Here are some creative uses of the for statement:

```c
for (i=0 ; i < n ; i++) {
    // Loop n times from i=0 to i=n-1
}
for (i=0 ; i < m ; i+=2) {
    // Loop ~m/2 times i = 0, 2, 4,...
}
for (i=100 ; i > 0 ; i--) {
    // Loop 100 times, decrementing i
}
```

Compound statements are also allowed. (This may be a little too creative.)

```c
for (i=0,j=0; i<1000; i++,j=exp(i))
```

Example: Good for() loop usage

```c
int i;
for (i = 0 ; i < 10 ; i+=2) {
    printf(“loop number %d\n”, i);
}
```

Do all iterator math with the loop updater.

```c
int i;
for (i = 0 ; i < 20 ; i++) {
    float a = i*0.5;
    printf(“counter= %f\n”, a);
}
```

Use integer iterators to avoid rounding errors with floats.

```c
const int NLOOPS=10;
int i;
for (i = 0 ; i < NLOOPS ; i++) {
    printf(“loop number %d\n”, i);
}
printf(“completed %d loops”, NLOOPS);
```

Use constants to define fixed NLOOPS, especially if you need to use the same value throughout your code.
Example: BAD for() loop usage

**Bad for Loop Usage:**

```c
int i;
for (i = 0 ; i < 10 ; i++) {
    printf("loop number \%d\n", i);
    i = i+1;
}  // How many iterations?
```

*Note 1:* It's extremely bad form to operate on the counter variable within the for loop. This leads to confusing code.

```c
float a;
for (a = 0 ; a < 10 ; a+=0.5) {
    printf("counter= \%f\n", a);
}  // 9.999999 < 10.0
```

*Note 2:* It's dangerous to use a float as your counter. Rounding errors may cause the loop run an unexpected number of times.

```c
int i;
for (i = 0 ; i < 10 ; i++) {
    printf("loop number \%d\n", i);
}  // printf("completed \%d loops", i);
```

*Note 3:* It's also bad practice to use a counter variable outside of the loop.

In this case `i = 10`, a value not used in the loop.
for() loops and break statements

We can use a break statement to prematurely exit a loop:

```c
#include <stdio.h>
int main ()
{
    int n;
    for (n=10; n>0; n--)
    {
        printf("%d, ", n);
        if (n==3)
        {
            printf("\ncountdown aborted!\n");
            break;
        }
    }
    return 0;
}
```

Output:
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
You can use a `continue` statement to skip the rest of the current loop, and go directly to the next iteration:

```c
#include <stdio.h>
int main ()
{
    for (int n=10; n>0; n--)
    {
        if (n==5 || n==6) continue;
        printf("%d, ", n);
    }
    printf("GO!\n");
    return 0;
}
```

**Output:**
```
10, 9, 8, 7, 4, 3, 2, 1, GO!
```

Try to minimize the use of break/continue in all but the most obvious cases. Program flow that jumps around is more difficult to understand. This can sometimes be avoided with conditional loops.
It is very common to nest loops in programs, by placing one loop inside of another:

```c
const int NUMDAYS = 7;
const int NUMWEEKS = 14;
int day, week;

for (week=0; week<NUMWEEKS; week++) {
    gotoMovie(); // Done 14 times.
    for (day=0; day<NUMDAYS; day++) {
        eat(); // Done 98 times
        study();
        sleep();
    }
}
```
Condition-Controlled Loops!
Sometimes, you can't tell ahead of time how many times a loop must run. For Example:

- “Do something until a convergence criterion is satisfied.”
- “Do something until the data are exhausted.”

This is where conditional loops are useful.

Conditional loops come in two flavors:

- **Pre-test Loops**: Check at the start of the loop to see if should be executed (again).
  - These loops may possibly never be executed.

- **Post-test Loops**: Check at the end of the loop to see if it is executed again.
  - These are ALWAYS executed at least once.
Pre-test Conditional Loops: \texttt{while()}

Syntax:

\begin{verbatim}
while (condition) {
    BLOCK of statements
}
\end{verbatim}

Example:

\begin{verbatim}
cash = get_paid();
while (cash > 0) {
    error = spend_money_on_snacks(cash, 0.75);
    if (!error) cash = cash - 0.75;
    else break;
}
\end{verbatim}

Do we have cash?

On successful snack acquisition, debit cash.

We have too little cash to continue. Break loop here.

Use non-zero status code to flag error on snack purchase.
Post-test Conditional Loops: `do/while()`

**Syntax:**

```plaintext
do {
    BLOCK of statements
} while (CONDITION);
```

This loop will always be executed at least once.

**Example:**

```plaintext
do {
    goto_class();
    do_homework();
} while (!semester_over());
```

Here we call two functions with no return values.

When the `semester_over()` function returns a TRUE value we can break the loop.
Recursive Functions
Recursion: An Example

Factorials:

The value of $N$ factorial can be written like this:

$$N! = N \times (N-1) \times (N-2) \times (N-3) \times (N-4) \times \ldots$$

But notice that this is just $N$ times $(N-1)$ factorial:

$$(N-1)! = (N-1) \times (N-2) \times (N-3) \times (N-4) \times \ldots$$

So, we could define the factorial function more compactly by writing it in terms of itself, like this:

$$N! = N \times (N-1)!$$

• So we really don’t need to calculate the product of $N$ numbers, just the product of 2 numbers $(N-1)$ times, each time one of the numbers being new, one being retained from the previous iteration – recursion
C supports the construction of recursive functions. Recursive functions are defined in terms of themselves. Notice that the factorial function, below ("fact") actually uses itself:

```c
long fact(int n) {
    if (n<=1)
        return (long)1;
    else
        return (long)n * fact(n-1);
}
```

Recursive algorithms are typically very short and are used when simple relationships may be defined between steps in a calculation or a data manipulation strategy.

All recursive functions must have a terminating condition, so the recursion has a limit.
The Scope of Variables
• When one defines a variable in C, it is accessible only in a well-defined extent within the program
• **Scope** refers to where variables can be accessed
  – Variables defined inside a function are available only in that function. These variables are of **local scope**
  – Variables can be defined outside all functions and accessed anywhere. These variables are of **global scope**.
Global vs. Local Scope

```c
int number;
void printInt();
void printFloat();

int main (){
    number = 1;
    printInt();
    printf("%d\n", number);
    printf("%f\n", number);
    return 0;
}

void printInt() {
    printf("%d\n", number);
}

void printFloat() {
    float number = 5.0;
    printf("%f\n", number);
}
```

A global integer variable named “number” is defined here.

A local variable (defined within a function) with the same name will override the global definition, but only within that function. The global variable is unaffected.

In the scope of this function, “number” is a float and has no relation to the global variable.
Global vs. Local Scope

```c
int main(){
    int a;
    float a; // not allowed!
    return(0);
}
```

Not allowed:
Guidelines for Scope and Variable Definitions

• In general, it is best to keep your variables limited to the *smallest scope possible*

• Having a globally-defined and accessible variable means it can be manipulated/changed ANYWHERE in your code – and this could lead to
  – confusion
  – bad results
  – heartache
Internal Comments and Documenting Code
/* Traditional C defines comments by using opening and closing comment markers as shown in this example. These comments may span multiple lines. */

// However, it is often better to write multiple line comments in this way to make the extended comments more clear in the text of your code

int a = 15; // C99 and C++ allow single line comments w/ the double slash
int b = 6; /* this type of comment is allowed, but not preferred */
Static Variables and Functions
Lost Information in Functions

- As described before, typically variables created in functions are only temporary – their contents are lost when a function completes.

```c
int main () {
    test();
    test();
    test();
}

void test () {
    int a=0;
    a++;
    printf ("a = %d\n",a);
}
```

By default, variables declared in function blocks are ephemeral. They are created when the function is entered, and destroyed when the function is exited.

No information about 'a' is saved between function calls. 'a' is recreated, starts out fresh, each time test() is called.
Static Variables in Functions

- We can get around this by declaring the variable in the function as `static`
Static Variables: Uses

• Static variables can be used to...

  – flag a function as having already been called

  – keep track of how many times a function has been called

  – keep track of values a variable attained in previous calls

  – keep track of previous parameters sent to a function
Functions: Limitations and Further Uses
Functions As We Know Them So Far

- The kinds of functions we have used so far
  - have some type
  - are given some arguments – which do not change in a global sense
  - return some value – which we then use
Functions As We Know Them So Far

```c
int main() {
    int hours = 1;
    int mins = 15;
    int secs = 0;
    howLong(hours, mins, secs);
    return 0;
}

void howLong(int hours, int mins, int secs) {
    printf("This class is %d seconds long\n", hours*3600 + mins*60 + secs);
    hours = 0;
    mins = 0;  // These statements have no effect on the variables in "main".
}
```

Local variables are **completely isolated** from variables in the calling function. They may be changed without affecting the original values.
Functions: More Functionality

• So far we have used functions in this manner:
  – Supply some arguments to a function, some values we need for a calculation
  – The function uses those arguments/values to perform some task
  – The result is:
    • stored in some variable of global scope OR
    • returned as the result of the function OR
    • the function returns success / failure and the code proceeds accordingly

• This behavior is somewhat limited!
  • single output is a limitation
  • cannot manipulate the arguments…this is often a less-than-desirable feature

• We need a more general function interface
Functions: Two Types

- Functions in C can be classified into two types:
  - **pass-by-value**: (what we have done so far)
    - the initial values of the arguments are not allowed to change globally
    - they can change interior to the function, but those changes are never carried out on the global variable
    - Copies of the input vars are made but not used beyond the function
  - **pass-by-reference:**
    - arguments can be values or memory locations
    - output can be “returned” through the parameter list
    - allows multiple results to be accessible to rest of program
Functions: Two Types

- Functions in C can be classified into two types:
  - **pass-by-value**: (what we have done so far)
    - the initial values of the arguments are not allowed to change globally
    - they can change internally, but those changes are never carried out on the global variable
    - Copies of the input vars are made but not used beyond the function
  - **pass-by-reference**: arguments can be values or memory locations
    - output can be “returned” through the parameter list
    - allows multiple results to be accessible to rest of program

Pass-by-reference functions requires a discussion of **pointers** to variables in memory….
We’ll pick up from here next time.

Thursday labs -- see you then.