Reminder

• HW05 due Thursday 1 March electronically by noon

• HW grades are starting to appear!

• Office hours: all held in our computer lab, room 022-C of this bldg
  – Me: After lecture 3:30-4:30 every Tuesday – today CANCELED 😞
    • I get the feeling this time is bad for people...
    • Email me, I am happy to meet other times!
  – TAs:
    • Mondays: 3-5pm and 6-8pm
    • Wednesdays: 5-9pm
Review and Outline

• Last time:
  – Comments
  – 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:
  – Followup on conditionals
  – Followup on loops
  – Scope
  – Static vars
  – Pointers
  – Recursion
  – Random numbers
Tip: Running in the background and ‘&’
A few followups from our discussion on conditional statements...
Be careful not to confuse the = (assignment) operator with the == (equality comparison) operator. This is one of the most common C typos.

The code below produces unexpected results. Why?

```c
int a=0;
int b=1;
if (a=b)
  printf("they are equal\n");
else
  printf("they are not equal\n");
```

The programmer should have used "a==b"!

```c
int a=0;
int b=0;
if (a=b)
  printf("they are equal\n");
else
  printf("they are not equal\n");
```

The value returned by the assignment operation "a=b" is just the left-hand side of the assignment ("a", in this case) after the operation has completed (after "a" has been set equal to "b").

```
if (a=b) is equivalent to
```
```
  a=b;
  if (a)
```
DANGER: “=” versus “==”

Be careful not to confuse the = (assignment) operator with the == (equality comparison) operator. This is one of the most common C typos.

The code below produces unexpected results. Why?

```c
int a=0;
int b=1;
if (a=b)  
    printf("the answer is ");
else
    printf("the answer is ");
```  

```c
int a=0;
int b=0;
if (a=b)
    printf("the answer is ");
else
    printf("they are not equal\n");
```  

**Take-away message:**

Be careful to use “==” in conditionals and not “=”.

```c
if (a=b) is equivalent to
```  

```c
a=b;
if (a)
```
Return Values and Tests

- Values are returned for all conditional checks:
  - if false, the value is zero ….. the value of the expression $(5 < 3)$ is 0
  - if true, the value is not zero ….. the value of the expression $(5 > 3)$ is 1

- We can use these characteristics in our code:

```c
// fopen returns a non-NULL pointer if successful
FILE* inFile;
inFile = fopen("grades.dat","r"); // open grades.dat

if (inFile==NULL) {
    // exit program if file not found
    printf("Error: grades.dat not found!!\n");
    return(1);
}

The “if” statement could alternatively be written like this:

```c
if (!inFile) {
    printf("Error...
");
}
Return Values and Tests: From Functions

- One can call a function that performs some test

**Returning Zero for Success:**

It's common practice for functions returning a integer status value (instead of returning data) to return zero for “Success”, and non-zero to indicate an error. You'll often see code that takes advantage of this convention when making tests. For example:

```c
if ( function(param1,param2) ) {
    printf("Error \!\!\n");
    return 1;
}
```

If “function” returns a non-zero (i.e., “true”) value, it means that something has gone wrong.

This isn't true for all functions in the Standard C Library. Check the documentation if you're not sure about a particular function.
DANGER: Take Care with Floating-Point Numbers
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – Lets say $A = (10000000000.0 \div 3.0) \times 3.0$
Quiz time:
- Let’s say \( A = (10000000000.0 / 3.0) \times 3.0 \)
- What is \( A \)?
Quiz time:
- Lets say $A = (10000000000.0 / 3.0) \times 3.0$
- What is $A$?
- Well, we all know the calculation should yield $10000000000.0$
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – Lets say \( A = (10000000000.0 / 3.0) \times 3.0 \)
  – What is \( A \)?
  – Well, we all know the calculation should yield 10000000000.0
  – The computer would evaluate things in this manner:

```c
float A = (10000000000.0/3.0)*3.0;
if (A==10000000000.0 ){
    printf("A=%lf\n",A);
} else
    printf("A is not 10000000000.0 \n");
```
• Quiz time:
  – Lets say $A = (10000000000.0 / 3.0) \times 3.0$
  – What is $A$?
  – Well, we all know the calculation should yield 10000000000.0
  – The computer would evaluate things in this manner:

    ```c
    float A = (10000000000.0/3.0)*3.0;
    if (A==10000000000.0 ){
        printf(“A=%lf\n”,A);
    } else
        printf(“A is not 10000000000.0 \n”);
    ```
  – What will be printed?
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – Let's say \( A = (10000000000.0 / 3.0) \times 3.0 \)
  – What is \( A \)?
  – Well, we all know the calculation should yield 10000000000.0

  – The computer would evaluate things in this manner:

    ```c
    float A = (10000000000.0/3.0)*3.0;
    if (A==10000000000.0 ){
        printf("A=%lf\n",A);
    } else
    printf("A is not 10000000000.0 \n");
    ```

  – What will be printed?

    A is not 10000000000.0

    What?
Floats and Comparisons

The `==` operator compares two numbers and returns “true” if they are the same. This works fine for integers, but you shouldn't use it for floating-point numbers. This example shows why:

```c
int main(){
    double a=12345678.;
    double loga2 = log(a*a);
    double b=sqrt(exp(loga2));
    printf("b=%20.10lf a=%20.10lf\n",b,a);
    return(0);
}
```

Clearly, “b” is not equal to “a” due to the limited precision of the calculations.
Comparing Floating-Point Data with “<” or “>”:

Instead of `==`, use inequalities to see if the difference between the floating-point numbers is less than some threshold (chosen by you).

```c
int main()
{
    const double SMALL = 1e-6;
    double a = 12345678.;
    double loga2 = log(a*a);
    double b = sqrt((exp(loga2)));

    if (fabs(a-b) < SMALL)
        printf("a=b\n");
    else
        printf("a!=b\n");
    return(0);
}
```

The `fabs()` function returns the absolute value of a floating-point number. It is part of the Standard C Library.

Notice that we've also introduced an "else" statement, after our "if". We'll talk about that next.
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!
for() loop: Important Parts

• 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:

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

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!
for() loops and continue statements

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.
Example: Good for() loop usage

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

Do all iterator math with the loop updater.

```
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.

```
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;
}
```

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

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

```c
9.9999999 < 10.0
Do we loop 20 or 21 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.
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()} \\
\\nSyntax:

\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.
- Use non-zero status code to flag error on snack purchase.
- We have too little cash to continue. Break loop here.
Post-test Conditional Loops: `do/while()`

**Syntax:**
```
do {
    BLOCK of statements
} while (CONDITION);
```

This loop will always be executed at least once.

**Example:**
```
do {
    goto_class();
    do_homework();
} while ( !semester_over() );
```

When the `semester_over()` function returns a TRUE value we can break the loop.
The Scope of Variables
Scope

• 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();
    printFloat();
    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

Not allowed:

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

A global integer variable has been declared.

```c
int number;
```
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
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
test();
```

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

```c
int a=0;
a++;
printf("a = %d\n",a);
```
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;  // These statements have no effect
    mins = 0;  // 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 to the function
    - never change the global variables
    - Copies of the input variables 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 the rest of the program

Pass-by-reference functions requires a discussion of **pointers** to variables in memory....
Pointers
Pointers: What are they?

• Pointers are like variables that provide direct access to the memory location for a specific variable.
• One can then manipulate the contents of the memory location – and hence manipulate the contents of the variable from any location in the code – allows global access and manipulation.
• Pointers give them user more power and more freedom!
• However, paraphrased from many (Voltaire, Churchill, FDR, Uncle Ben in Spiderman)

"With great power and freedom comes great responsibility."
Recall: Variable Storage

Another Look at Variable Storage:

- The values of variables are stored in memory.
- Different types of variables take up different amounts of memory.
- The values of variables are stored as ones and zeros (bits) arranged in groups of eight (bytes).

How does the computer find a particular variable’s data in memory? Each variable has an address, expressed as the number of bytes from some starting location.
As we’ve seen before, you can use the “sizeof” operator to find out how much storage space a variable uses in memory.

You can use the “&” (“address of”) operator to see the memory address of a particular variable's storage area:

```c
#include <stdio.h>
int main () {
    int number = 123456;
    printf("Size of memory storage: %d\n", sizeof(number));

    printf("Memory address of storage: %p\n", &number);
}
```

The “%p” format descriptor can be used for printing memory addresses.

Size of memory storage: 4
Memory address of storage: 0xbfd2ab5c
Memory Addresses: Storage in Functions

```c
#include <stdio.h>
int test(int n);
int main () {
    int number = 123456;
    printf("Memory address of main storage: %p\n", &number);
    test(number);
}
int test(int number) {
    printf("Memory address of test storage: %p\n", &number);
}
```

Memory address of main storage: 0xbfe82bbc
Memory address of test storage: 0xbfe82ba02
Memory Addresses: Storage in Functions

Storage of Local Variables:

The variables inside a function, even those passed to function when we invoke it, are local to that function. A variable named “number” in the function “test” isn’t the same as the variable named “number” in “main”.

```
test(number);
```

When the function is invoked, the value in `number` in “main” is copied into a new storage area for holding the “test” function’s variable called `number`.
A pointer is a special kind of variable that holds the memory address of another variable.

A pointer is defined by prefixing a variable name with an asterisk (*), the *indirection operator*:

```c
int main() {
    int number = 5;
    int *nptr;
    nptr = &number;
    return(0);
}
```

An int variable

A “pointer to int” variable

Use the “address of” (&) operator to get the address of `number` and store it in `nptr`. 
Storing Pointers in Memory

```c
int main() {
    double number = 3.1415;
    double *nptr;
    nptr = &number;
    return(0);
}
```

In this example, the “double” variable `number` stores the value 3.1415. The “pointer to double” variable `nptr` stores the address of `number`.```
You can get the data stored at a given memory location by using the "*" ("indirection") operator:

```c
int main() {
    double number = 3.1415;
    double *nptr;

    nptr = &number;

    printf("The value is %lf\n", *nptr);

    return(0);
}
```

Since `nptr` is a “pointer to double”, the result will be treated as “double” data. The value is 3.1415
We can also use the indirection operator on the left side of an assignment statement, to set the value stored at a given memory address:

```c
int main() {
    double number = 3.1415;
    double *nptr;
    nptr = &number;
    *nptr = 6.02;
    printf("number is %lf\n", number);
    return(0);
}
```

Here we change the value of `number` indirectly, by sticking a value into that variable's memory address.

number is 6.02
An Example: `scanf`

```c
int main () {
    int x, y;

    printf ("Enter x, y:");
    scanf("%d %d", &x, &y);
}
```

`&x` and `&y` are the memory addresses of the variables `x` and `y`. These addresses are copied into variables inside `scanf`.

After `scanf` reads data from the keyboard, it sticks that data into the **memory addresses** given by `&x` and `&y`.

If `scanf` didn't know the addresses of these variables, it couldn't modify their contents.
You can use indirection in your own functions. Here's an example:

```c
int main() {
    float x = 2;
    float y = 5;
    float area;
    getarea(x, y, &area);
    printf("the area = %f units\n", area);
    return 0;
}
```

Here we pass `getarea` the address of the variable `area`.

```c
void getarea(float x, float y, float *aptr) {
    *aptr = x * y;
    return;
}
```

Here we tell our function to expect a pointer containing the address of a “float”.

Deposit the calculated area into memory at the address of variable `area` in “main”. 
Advantages: Passing Pointers As Args To Functions

- Through the use of pointers passed to functions, we can manipulate an arbitrary number of variables, rather than just return one result.

```c
void c_and_a(float r,
           float *a, 
           float *c) {
    *a = PI*r*r;
    *c = 2*PI*r;
}
```
Pointers: Be Careful!

**Pointer Errors: Null (zero) Pointers:**

If we accidentally leave off an ampersand when calling `scanf`, we'll usually get a **segmentation fault** error.

```c
int main () {
    double x;
    printf("Enter the value for x:");
    scanf("%lf", x);
}
```

The value in `x` is probably zero, so `scanf` interprets this to mean that it should stick the value of `x` into the memory address "0x00000000".

This is a low-lying part of memory that belongs to the operating system, and your program doesn't have permission to write there. That's what the "segmentation fault" error is telling you.
Pointers: Be Careful!

```c
int main () {
    int number = 4;
    double *nptr;
    nptr = &number;
    printf ("%lf\n", *nptr);
}
```

```c
int main () {
    int number = 4;
    double *nptr;
    nptr=(double *)&number;
    printf ("%lf\n", *nptr);
}
```

Error: cannot convert ‘int *’ to ‘double *’ in assignment

0.00000000
Recursive Functions
Recursion: An Example

**Factorials:**

The value of N factorial can be written like this:

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

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

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

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) { // Terminating condition
        return (long)1;
    } else
        return (long)n * fact(n-1); // Recursive function call
}
```

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.
We’ll pick up from here next time.

See you then.