Reminder

• HW06 due Thursday 15 March electronically by noon
• HW grades are starting to appear!

• Exam next week! More on this later...

• Join Piazza!

• Office hours: all held in our computer lab, room 022-C of this bldg
  – Me: After lecture 3:30-4:30 every Tuesday
  – TAs:
    • Mondays: 3-5pm and 6-8pm
    • Wednesdays: 5-9pm
Review and Outline

• Last time:
  – Followup on conditionals
  – Followup on loops
  – Scope
  – Static vars
  – Pointers – experience in lab
  – Recursion – experience in lab

• Today:
  – Review of pointers
  – Pointers to functions
  – Arrays
  – Passing arguments to main(..)
  – Making code reusable
  – Review for exam
  – Random numbers
Pointers
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`.
In this example, the “double” variable `number` stores the value 3.1415. The “pointer to double” variable `nptr` stores the address of `number`.
Pointers: The Indirection Operator ‘*’

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
Pointers: Changing Data in a Memory Location

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!

**Pointer Errors: Mis-casting:**

```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
Pointers to Functions and Trapezoidal Rule for Integration
Pointers to Variables: A Review

- Pointers allow direct access to memory locations where variables are stored in 1’s and 0’s
  - address via &
  - content via *

- Passing pointers into functions:
  - In function prototype
    - use pointer-to-values as argument(s)
  - In call to function
    - call with pointer-to-address as argument(s)
  - In function definition,
    - use pointer-to-values as argument(s) as done in prototype
    - use pointer-to-values in body of function

```c
#include <stdio.h>
void circle_stuff(double radius, double *a_ptr, double *c_ptr);

void main(){
  double radius = 3.0;
  double area = 0.0;
  double circumference = 0.0;
  circle_stuff(radius, &area, &circumference);
  printf("area=%8.3f
  circumference=%8.3f\n", area, circumference);
}

void circle_stuff(double radius, double *a_ptr, double *c_ptr){
  *a_ptr = 3.1416*radius*radius;
  *c_ptr = 2.0*3.1416*circumference;
}
```
How are Functions Stored in Memory?

Functions in Memory:

Like variables, each function in your program is stored in memory. The function’s memory location holds the machine-code instructions that implement the function.

Just as you can use pointers to refer to the location of a variable in memory, you can use pointers to functions to refer to the address of a function.

Function pointers allow you to pass functionality around your program just like data.
Pointers to Functions: Example

```c
#include <stdio.h>
#include "sqrtn.hpp"

void print_func(double (*f)(double x), double val);

int main()
{
    double x;
    printf("enter a number
\n");
    scanf("%lf", &x);
    printf("\n");
    print_func(sqrtn, x);
}

void print_func(double (*f)(double x), double val)
{
    printf("func(\%lf) = \%lf
\n", val, f(val));
}
```

This argument is a pointer to a function.

“print_func” takes a function name as an argument.

Any function that returns a double and takes a double argument can be plugged in here.

Now “f” points to the function we named.
Pointers to Functions: Pieces

In a function definition, a function pointer appears like this:

```c
void print_func(double (*f)(double x), double val);
```

1. The name of the function pointer is “f”. The name must be enclosed in parentheses.

2. This pointer points to a function that returns a “double”.

3. This pointer points to a function that takes one argument, of type “double”.

```c
double (*f)(double x)
```
Monte Carlo integration is one method one can use to integrate some complicated function, especially useful in multiplie dimensions.

Trapezoidal rule integration offers a very simple method to approximate the integral of a one-dimensional function.

Consider the integral of $f(x)$ over the range $A \leq x \leq B$.

The area of each of the subdivisions: 1, 2, 3, 4 may be roughly estimated as as the average of $f(x)$ in this subdivision times the width, $\Delta x$, of the subdivision.
Pointers to Functions:
Example – Numerical Integration

By summing the rectangular regions shown, we can estimate the integral of $f(x)$.

In the limit that $\Delta x$ is small enough so that $f(x)$ is essentially linear over $\Delta x$, this estimate is very accurate. If $f(x)$ is linear over $\Delta x$, then the shape of each sub area is trapezoidal and our box covers the same area.
Pointers to Functions:
Example – Numerical Integration

We can estimate the area by summing the subdivisions.

\[
\int_{A}^{B} f(x) \approx \Delta x \left[ \frac{f(A) + f(B)}{2} + \sum_{i=1}^{n-1} f(A + i \Delta x) \right]
\]
Implementing the Trapezoid Rule:

```c
#include <stdio.h>
#include <math.h>

double trap_rule(double (*f)(double), double min, double max, int steps)
{
    int i;
    double sum=0;
    double dx=(max-min)/steps;
    for (i=1; i<steps ; i++) sum += f(min + i*dx);
    return dx * ((f(min)+f(max))/2 + sum);
}

int main() {
    printf("Integral of sin(x) in [0:pi/2] = %f\n", 
           trap_rule(sin,0,M_PI/2,100));

    printf("Integral of exp(x) in [0:10] = %f\n", 
           trap_rule(exp,0,10.,200));
}
```

We now can integrate any 1D function we choose. Accuracy is controlled by the `steps` parameter.
Arrays in C
What is an Array?

• Very often we need to store and access many instances or sets of related data

• Example: coordinates in 3-space:
  – x=5, y=23, z=1
  – Can be represented by a vector: (5,23,1)
  – The points in space are of the same type – would like a “container” that holds them in a convenient format

• Arrays do this for us!
• Not just god for vectors in 3-space
  – Good for matrices
  – Good for collections of related items
  – Good generally for similar pieces of data
An example: A vector in 3-space

```
#include <stdio.h>
int main () {
    double x1=1.0, y1=2.0, z1=3.0;
    double x2=4.0, y2=5.0, z2=6.0;
    double sum1, sum2, sum3;
    double dot;

    dot = x1*x2 + y1*y2 + z1*z2;
    printf ("Dot-product: %lf\n",dot);

    sum1 = x1+x2;
    sum2 = y1+y2;
    sum3 = z1+z2;
    printf ("Sum: %lf %lf %lf\n",sum1,sum2,sum3);

    return(0);
}
```

Define a variable for each vector element.

Be careful of typos! It's easy to type "x1" instead of "x2".

Nothing ties the vector together as a single item. You have to keep track of all of the parts yourself.
An example: A vector in 3-space using an array

```c
#include <stdio.h>
int main () {
    double x1[3] = {1.0, 2.0, 3.0};
    double x2[3] = {4.0, 5.0, 6.0};
    double sum[3], dot=0;
    int i;
    for (i=0; i<3; i++) {
        dot += x1[i] * x2[i];
    }
    printf ("Dot-product: %lf
", dot);
    for (i=0; i<3; i++) {
        sum[i] = x1[i] + x2[i];
    }
    printf ("Sum: %lf %lf %lf
", 
            sum[0], sum[1], sum[2]);
    return(0);
}
```

Here's a better way:

Define each vector as an array of three doubles.

Note how arrays can be initialized.

Loop through all of the elements of the array.

Note that array indices go from zero to N-1, where N is the size of the array.
Defining Arrays

- The elements of an array can be of **any type** (but all elements of a given array must be of the same type).

- When defining an array, the number in **square brackets** says how many elements are in the array.

- Arrays can optionally be **initialized** when they're defined.

```plaintext
int population[50];
char name[25];
double x1[3] = {1.0, 2.0, 3.0};
```

Arrays take up memory. It's easy to write "double a[1000]", but remember that this takes as much memory as a thousand single variables. Keep this in mind when defining large arrays.
Using Arrays

- Array elements can be referred to by their indices.
- The index must be an integer.
- The index uniquely identifies a single array element.

```plaintext
value = x1[i] + x2[i];
x[i] = M_PI*area;

It's important to remember that the values of array indices start with zero, and that they end at N-1.
```

```plaintext
for (i=0; i<3; i++) {
    dot += x1[i] * x2[i];
}
```
The elements of an array are stored in contiguous memory locations.

```c
int x[5];
```
Array Boundary Checking:

Unlike some languages, C doesn't check your array indices to make sure they're within the bounds of the array.

For example:

```c
int x[3];
```

```c
x[128] = 100;
```

This is the most common source of run-time errors when using arrays.

The compiler will not check for these errors, and they won't become apparent until your program generates a "segmentation fault" error.

What data is stored in this location? Whatever it is, it's not part of the array "x", and it's probably not even owned by this program.
Other Array Errors:

What's wrong with the following code?:

```java
double days[7];
double months[12];

days[7] = 3.14;
```

The index of “days” goes from 0 to 6.

If the arrays “days” and “months” are stored next to each other in memory, it's possible that the value 3.14 gets written to the first element of the months array!

In this case, the operating system doesn’t care, because the program has the right to modify that memory.
Passing Arrays to Functions

When passing an array, we don't specify the size in the square brackets.

But we do need to tell the function what the size is. Arrays in C don't carry around any information about their sizes.

```c
void print_stuff(float a[], int size);

int main()
{
    const int max = 20;
    float an_array[max];
    print_stuff(an_array, max);
}

void print_stuff(float a[], int size)
{
    int i;
    for (i=0; i<size; i++)
    {
        printf (“%f
”, a[i]);
    }
}
```

We give the function the name of the array, and the size.

Inside the function, we can use the array just like we'd use it in “main”.
Mid-term Exam
Information on Mid Term Exam

- Your mid-term is coming!
- Tuesday 20 March in this room at 7pm
  - You will have 60 min to finish
  - Bring a number 2 pencil
  - Regular lecture and labs will of course meet that week

- Format:
  - multiple choice
  - matching
  - short answer

- All material in readings, lectures, labs, HWs is fair game
  - That being said I would not expect an emphasis in the exam on things that I did not emphasize in lecture, labs, HW

- Let’s review some of the things we have learned so far
- Some examples follow
Review of C:
What We Have Learned So Far
Variables

- Have types.
- Commonly-used types are:
  - `int`, for integers (1, 2, 3, ...),
  - `double`, for floating-point numbers (3.14, 6.02, 9.8),
  - `char`, for characters and character strings ('a', 'Hello World!')
- Are usually defined at top of functions.
- May be pointers that hold the address of another variable.
• Use `printf` to write to the screen.
  • Requires `#include <stdio.h>`
  • Uses format descriptors for variables.
  • Commonly-used format descriptors are:
    • `%d`, for int
    • `%lf`, for double
    • `%s`, for character strings
  • You can use "\n" to insert carriage returns.

• Use `scanf` to read from the keyboard.
  • Use an `&` in front of variables, except for character strings.
Files

- Referred to by “file pointers”: `FILE *infile;`
- Open files with `fopen`:
  - Pick “r” for reading, “w” for writing.
- Write to file with `fprintf`.
- Read from file with `fscanf`.
- Close file with `fclose`.
Loops

- **for loops:**
  - Repeat something a **fixed number** of times.
  - Should use **integers** for counters.

- **while loops:**
  - Keep repeating **as long as a given statement is true**.
  - Do the test **before starting**, so number of repeats may be zero.

- **do loops:**
  - Keep repeating **as long as a given statement is true**.
  - Do the test **after the first pass**, so number of repeats is always at least one.
Conditionals

• if statements:
  • Only do something if a given statement is true.

• if/else statements:
  • Pick between two options, depending on whether a given statement is true or false.

• if/else if/else statements:
  • Choose based on the first true expression you find.

• switch statements:
  • Choose between several options, based on the value of an integer or a character.
Functions

• Always have at least one, called “main”.
• Functions return one value to the caller.
• Functions take arguments.
• Functions have a syntax defined by a prototype statement.
Pointers

- Pointers allow direct access to memory locations where variables are stored in 1’s and 0’s
  - address via &
  - content via *

- Passing pointers to functions:
  - In function prototype
    - use pointer-to-values as argument(s)
  - In call to function
    - call with pointer-to-address as argument(s)
  - In function definition,
    - use pointer-to-values as argument(s) as done in prototype
    - use pointer-to-values in body of function

```c
#include <stdio.h>

void circle_stuff(double radius, double *a_ptr, double *c_ptr);

void main(){
  double radius = 3.0;
  double area = 0.0;
  double circumference = 0.0;

  circle_stuff(radius, &area, &circumference);
  printf("area=%8.3f
        circumference=%8.3f\n", area, circumference);
}

void circle_stuff(double radius, double *a_ptr, double *c_ptr){
  *a_ptr = 3.1416*radius*radius;
  *c_ptr = 2.0*3.1416*circumference;
}
```
Questions About the Basics

• What is an operating system?

• What are the main hardware parts of a computer?

• What is a compiler?

• Etc.
<table>
<thead>
<tr>
<th>Question</th>
<th>Command/Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) lists files in your directory</td>
<td><code>ls</code></td>
</tr>
<tr>
<td>2) lists files in your directory, with sizes shown</td>
<td><code>ls -l</code></td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td><code>ls -al</code></td>
</tr>
<tr>
<td>3) renames the file <code>my.dat</code> to <code>your.dat</code></td>
<td><code>mv my.dat your.dat</code></td>
</tr>
<tr>
<td>4) places file <code>a.txt</code> in a subdirectory called <code>sub</code></td>
<td>a) <code>mv a1.txt a2.txt</code></td>
</tr>
<tr>
<td></td>
<td>b) <code>mv a.txt</code></td>
</tr>
<tr>
<td></td>
<td>c) <code>rename a.txt sub/a.txt</code></td>
</tr>
<tr>
<td></td>
<td>d) <code>mv a.txt sub</code></td>
</tr>
</tbody>
</table>
### Questions About the C Language

<table>
<thead>
<tr>
<th>Choose the best answer.</th>
<th></th>
</tr>
</thead>
</table>
| **1) Define an integer variable, \( i \):** | a) `int i;`  
b) `int &i;`  
c) `integer i;`  
d) `int *i;`  |
| **2) Define a floating point variable with value=3.14 whose value cannot be changed:** | a) `#define PI=3.14;`  
b) `const double PI=3.14;`  
c) `#define PI 3.14`  
d) `static float PI=3.14;`  |
| **3) The statement to read a double value into the variable named `discount` is:** | a) `scanf("%lf", discount);`  
b) `scanf("%d", &discount);`  
c) `scanf(discount);`  
d) `scanf("%lf", &discount);`  |
| **4) Print the double variable \( q \) in scientific or floating point notation, whichever is more compact:** | a) `printf("%ef", q);`  
b) `printf("%e", q);`  
c) `printf("%g", q);`  
d) `printf("%lf", q);`  |
# Questions About the C Language

Choose the best answer.

1) Using the file pointer, `input_file`, open the file `results.dat` for read mode.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a) <code>openf(&quot;results.dat&quot;, &quot;r&quot;, input_file);</code></td>
<td>b) <code>open(input_file, &quot;results.dat&quot;, &quot;r&quot;);</code></td>
</tr>
<tr>
<td>c) <code>fopen(input_file, &quot;results.dat&quot;, &quot;r&quot;);</code></td>
<td>d) <code>input_file = fopen(&quot;results.dat&quot;, &quot;r&quot;);</code></td>
</tr>
</tbody>
</table>

2) Which code snippet reads an integer from the program's command line?

```c
int main(int argc, char *argv[]){
    ... 
    a) int i = argv[1];
    b) int i = atoi(argv[1]);
    c) int i = atoi(argv[0]);
    d) int i = (int)argv[1]);
```  

3) Function pointer type that can point to `sqrt()` function in the C math library:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a) <code>double (*f)(double x)</code></td>
<td>b) <code>double *f(double x)</code></td>
</tr>
<tr>
<td>c) <code>double &amp;f(double x)</code></td>
<td>d) <code>double *f(double x)</code></td>
</tr>
</tbody>
</table>
### Questions About the C Language

<table>
<thead>
<tr>
<th>Choose the best answer.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1) What is the only function all C programs must contain?</td>
<td></td>
</tr>
<tr>
<td>a) <code>start()</code></td>
<td></td>
</tr>
<tr>
<td>b) <code>system()</code></td>
<td></td>
</tr>
<tr>
<td>c) <code>main()</code></td>
<td></td>
</tr>
<tr>
<td>d) <code>program()</code></td>
<td></td>
</tr>
<tr>
<td>2) Which of the following is the correct operator to compare two numerical variables?</td>
<td></td>
</tr>
<tr>
<td>a) <code>:=</code></td>
<td></td>
</tr>
<tr>
<td>b) <code>=</code></td>
<td></td>
</tr>
<tr>
<td>c) <code>equal</code></td>
<td></td>
</tr>
<tr>
<td>d) <code>==</code></td>
<td></td>
</tr>
<tr>
<td>3) How many times is a “do while” loop guaranteed to loop?</td>
<td></td>
</tr>
<tr>
<td>a) 0</td>
<td></td>
</tr>
<tr>
<td>b) Infinitely</td>
<td></td>
</tr>
<tr>
<td>c) 1</td>
<td></td>
</tr>
<tr>
<td>d) Variable</td>
<td></td>
</tr>
<tr>
<td>4) Evaluate:</td>
<td></td>
</tr>
<tr>
<td>`!(1 &amp;&amp; !(0</td>
<td></td>
</tr>
<tr>
<td>a) True</td>
<td></td>
</tr>
<tr>
<td>b) False</td>
<td></td>
</tr>
<tr>
<td>c) Unevaluatable</td>
<td></td>
</tr>
</tbody>
</table>
1) Given the variables:

```c
int sum;
int maxint;
```

Write a statement that tests to see if \textit{sum} is equal to 1000 and also that \textit{maxint} is between 10 and 50, inclusive.

If the condition is satisfied, print the text "OK".

```c
if ( sum == 1000 &&
    maxint >= 10 &&
    maxint <= 50 ) {
    printf ("OK\n");
}
```

or

```c
if ( sum == 1000 &&
    maxint >= 10 &&
    maxint <= 50 )
    printf ("OK\n");
```
We’ll pick up from here next time.

See you then.