Reminder

• Midterm exam – tonight!

• HW06 due Saturday 25 Mar electronically by 11:59pm
• Grading of HW04, HW05 coming!
• Register for piazza!
  – Only 56 out of 70 so far
• You will need the second textbook, Lyons, in about 1 week.

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

• TA office hours
  – In Room 022-C
    • Mondays 7-9:30pm
    • Tuesdays 4:30-6:30pm
    • Wednesdays 7-9:30pm
News and Announcements

• Mid-term exam ***TONIGHT*** at 7pm in this room
  – Five hours from now!

  – Will cover everything we will have covered through last Tuesday’s lecture

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

  – Yes, not all your homeworks have been returned graded – and this is not ideal
    • BUT this is not a significant interference with the exam
    • Exam questions designed to focus facts and syntax and logical examples moreso than individuals writing code
    • So the homeworks, though valuable, are less valuable than the lectures, text and labs for tonight
Review and Outline

• Last time:
  – Pointers
  – Pointers to Functions
  – Review

• Today:
  – Arrays in C
  – Strings as character arrays
  – Passing arguments to main()
  – Structures
  – Making reusable code

• Tonight:
  – Exam 1
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

```c
#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\n",dot);
    for (i=0; i<3; i++) {
        sum[i] = x1[i] + x2[i];
    }
    printf ("Sum: %lf %lf %lf\n", 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.

```
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.

\[
\text{value} = x1[i] + x2[i];
\]

\[
x[i] = M\_PI*\text{area};
\]

It’s important to remember that the values of array indices start with zero, and that they end at \(N-1\).

\[
\text{for (i=0; i<3; i++)} \{ \\
    \text{dot} += x1[i] * x2[i]; \\
\}
\]
Storage of Arrays in Memory

The elements of an array are stored in contiguous memory locations.

```c
int x[5];
```
Storage of Arrays in Memory: Danger

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];

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\n", 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".
Names of arrays and pointers

```c
int x[5];
int *top;
int *two;
top = x;
two = &x[2];
```

The name of an array can be used just like a pointer that points to the beginning of the array.

Here we define another pointer ("top"), and make it point to the beginning of the array, too.

As usual, we can get the address of individual elements using the & operator.
Incrementing a Pointer

```
int x[5];
int *top;
top = x;
top++;
```

When you use the "++" operator on a pointer, it moves the address of the pointer **forward** by an amount equal to the size of the type of variable ("int", in this case).

Here, the pointer is initially pointing at the top of the array, so "++" moves it to the **next array element**.

This is our first look at "pointer arithmetic".
Array Names ARE Pointers Already!

We can pass an array to a function by just giving the function a pointer to the beginning of the array:

```c
int main () {
    const int SIZE=100;
    double d_ary[SIZE];

    reset_data( d_ary, SIZE );
    ....
}
```

By giving the name, we pass the address of the top of the array.

```c
void reset_data(double *data, int n) {
    int i;
    for (i=0; i<n; i++){
        *data = 0;
        data++;
    }
}
```

Array names are just pointers.

Set the data at this address to zero.

Jump to the next array element..
Array Names ARE Pointers Already -- proof

The two functions below do **exactly the same thing**, they just say it in different ways. You can freely refer to array elements by either using **pointers** or **array** (square bracket) notation.

**Pointer notation:**

```c
void clear_data(double *data, int n) {
    int i;
    for (i=0; i<n; i++){
        *data = 0;
        data++;
    }
}
```

- **Use de-referencing (** * ** operator)**
  - and pointer arithmetic.

**Array notation:**

```c
void clear_data(double *data, int n) {
    int i;
    for (i=0; i<n; i++){
        data[i] = 0;
    }
}
```

- **Use array notation to access array elements.**
Pointers can be manipulated with all types of integer operations. The following (and more) are all valid:

```c
for (i=0; i<n ; i++){
    *data = 0;
    data++;
}
for (i=0; i<n ; i++){
    *data = 0;
    data--;
}
for (i=0; i<n/2 ; i++){
    *data = 0;
    data+=2;
}
```

- **Step forward.**
- **Step backward.**
- **Step over every 2nd element**
Multidimensional Arrays

A 2-dimensional array may be defined by specifying two indices:

```c
int main(){
    const int nrow = 20;
    const int ncol = 20;
    double matrix[nrow][ncol];
}
```

Defines a 20x20 array.

```c
int i,j;
for (i=0; i<nrow; i++) {
    for (j=0; j<ncol; j++) {
        matrix[i][j] = (double)i * (double)j;
    }
}
```

Higher-dimensional arrays can be defined by just adding more indices.
A 2-dimensional array may be defined by specifying two indices:

```c
int main(){
    const int nrow = 20;
    const int ncol = 20;
    double arr[nrow][ncol];
    int i, j;
    for (i = 0; i < nrow; i++)
        for (j = 0; j < ncol; j++)
            arr[i][j] = ...
}
```

Higher-dimensional arrays can be defined by just adding more indices.

Do remember though:

When you define an array of any size, the footprint of that array’s size is dedicated in the computer’s memory.

It is very easy to exhaust a computer’s memory by defining arrays that are too large.
Storage of 2D Arrays

type array[NR][NC];

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0][0]</td>
<td>[0][1]</td>
</tr>
<tr>
<td>[1][0]</td>
<td>[1][1]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[NR-1][0]</td>
<td>[NR-1][1]</td>
</tr>
</tbody>
</table>

In C, arrays are stored with “row-first” in memory. You can think of a 2-D array as an NCOLUMN array repeated NROW times.
Memory and 2D Arrays

2-D Arrays in Memory:

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0][0]</td>
<td>[0][1]</td>
</tr>
<tr>
<td>[1][0]</td>
<td>[1][1]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[NR-1][0]</td>
<td>[NR-1][1]</td>
</tr>
</tbody>
</table>

It is convenient to think of a 2-D array as a matrix like the one drawn above. However, all data must be stored in a linear manner in memory:

```
[0][0] [0][1] ... [0][NC-1]
[1][0] [1][1] ... [1][NC-1]
... ... ... ...
[NR][0] [NR][1] ... [NC-1][NC-1]
```
2D Arrays and Pointer Arithmetic: I

```
int array[NR][NC];
int *array_p = array;
array_p++;
```

Equivalent to:
```
array[n][m+1];
```
Incrementing by Row:

```c
int array[NR][NC];
int *array_p = array;
array_p += NC;
```

Equivalent to:

```c
array[n+1][m];
```
Once we know how 2-D arrays are stored in memory, we can use pointer arithmetic to point to any array element we want:

```c
// point to array [0][5]:
array_p2 = array_p + 5;

// point to array [1][3]:
array_p2 = array_p + (1*NC) + 3;

// point to last element:
array_p2 = array_p + (NR-1)*NC + NC-1;

// or, equivalently:
array_p2 = array_p + NR*NC - 1;
```
Using 1D Notation for 2D Arrays

1-D Notation for 2-D Arrays:

C doesn't know the dimensions of the array that a pointer is pointing at, so we can act as though we're pointing at a 1-D array even if we originally defined a 2-D array. We just enclose the total offset in square brackets:

```c
int a[NR][NC];
int *array_p = a;

// point to array [0][5]:
array_p2 = array_p[5];

// point to array [1][3]:
array_p2 = array_p[(1*NC) + 3];

// point to last element:
array_p2 = array_p[(NR-1)*NC + NC-1];
// or:
array_p2 = array_p[NR*NC - 1];
```
C doesn't know the dimensionality of the array being pointed to. We only need to care about the **total number of elements**. It doesn't matter whether the array is [30], [2][15] or [2][3][5]. Each has 30 elements, and the function below could be used to clear each of them.

```c
int main () {
...
    int *array_p = array;  // point to array [0][0]
    iclear(array_p, NROW*NCOL);
...
}

// This function clears any size/dimension // integer array:
void iclear(int *pntr, int size){
    int i;
    for (i=0; i<size; i++){
        *(pntr+i) = 0;
    }
}
```

**Total number of elements.**
Arrays of Characters
Arrays of Characters

Single characters are nice but …many in succession can be more meaningful to us.

Array of characters are called a string.

```c
#include <stdio.h>

int main () {
    char string1[20] = "this is a test.";
    char string2[20] = {'t','h','i','s',' ','i','t','s',' ','a','i','t','s',' ','t','e','s','t'};

    printf ("%s\n",string1);
    printf ("%s\n",string2);
}
```

As you can see, strings can either be initialized by giving individual characters in curly brackets, as you’d initialize any other type of array, or you can use the more natural way of doing it: Just write the string and enclose it in quotes.
In C, when you give the name of an array, it's equivalent to a pointer. For example, consider the following code:

```c
void printit(double a[], int s);
int main() {
    double a[50];
    double *aptr;
    ...
    printit(a, 50);
    printit(aptr, 50);
}
```

The prototype for "printit" could just as well have said "double *a" instead of "a[]". The two are equivalent.

This explains another of the mysteries of scanf: Why don't we need put an ampersand in front of the names of character strings? It's because these variables are already pointers.
Character strings are just arrays of characters. The `strlen` function (defined in `string.h`) returns the length of a string.

```c
#include <string.h>
...
char name[15] = "fred";
char day[] = "Tuesday";

printf("%d %d %s\n",
    sizeof(name),
    strlen(name),
    name);

printf("%d %d %s\n",
    sizeof(day),
    strlen(day),
    day);
```

Define a character string of up to 15 characters.

Let the compiler figure out the size.

What is going on?!
• **American Standard Code for Information Interchange (ASCII):**
  – ~first new widely adopted code for latter communication since Morse code
  – Ea. character represented by 8 bits
  – 128 characters supported
    • 26x2 letters of the english alphabet
    • 10 single-digit numerals
    • 33 symbols, incl. “space”
    • 33 control characters
  – Facilitates communication between terminal and computer
  – Other standards are used today (eg., UTF-8, others) but the spirit is the same
String Termination and Arrays

Null-Terminated Strings:

```c
char day[] = "Tuesday";
```

Each character takes up one byte (8 bits) in memory. A character string is just an array of characters.

But, as we've seen, C doesn't know how long an array is. When we make a statement like:

```c
printf("%s", day);
```

how does printf find the end of the string? We haven't told it the string's length explicitly.

<table>
<thead>
<tr>
<th>day</th>
<th>01010100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>01110101</td>
</tr>
<tr>
<td>u</td>
<td>01100101</td>
</tr>
<tr>
<td>e</td>
<td>01110011</td>
</tr>
<tr>
<td>s</td>
<td>01100100</td>
</tr>
<tr>
<td>d</td>
<td>01100001</td>
</tr>
<tr>
<td>a</td>
<td>01111001</td>
</tr>
<tr>
<td>y</td>
<td>00000000</td>
</tr>
</tbody>
</table>

The answer is that, in C, strings are "null-terminated".

By this we mean that a special character ("NUL") appears as the last character in the string. Because of this, functions like printf can find the end of the string by looking for the NUL.

This means that the array needs to have room for one more character than the text we're putting into it.
String Utilities: `strcmp`

Comparing Strings with `strcmp`:
The `strcmp` function (also defined in `string.h`) compares two strings:

```
int strcmp(char* S1, char *S2);
```

Returns:
- 0 if S1 = S2
- >0 if S1 > S2
- <0 if S1 < S2

```
#include <string.h>
int main () {
    char string1="abcde";
    char string2="fghij";
    if (!strcmp(string1,string2) ) {
        printf ("They match.\n");
    } else {
        printf ("They don't match.\n");
    }
}
```
Passing arguments to `main(...)`
Passing arguments to `main(...)`

- We have done this many times already....

When you run a program like `cp`, you are passing arguments at the command line. For example:

```
Command       Parameter 1       Parameter 2
```

```
cp myfile.txt yourfile.txt
```

C supports a simple interface for providing data to your program via the command line.

If a program needs few parameters to control its behavior, this is a nice alternative to using `scanf` or reading data files to get options.
Special parameters in C: `argc` and `argv`  

Until now, we've begun our programs like this:

```
int main()
```

But, just like other functions, the “main” function can take arguments. In particular, we could begin our program like this:

```
int main( int argc, char *argv[ ] )
```

If we do so, the operating system will use these arguments to pass information from the command line to our program.

`argc` is the “argument count”, the number of arguments the operating system is giving us, and `argv` is the “argument vector”, which is an array of character strings.

This may seem confusing at first, but we'll see how it works through examples.
Here's an example showing how `argc` and `argv` can be used:

```c
int main(int argc, char *argv[]){
    int i;
    for (i=0; i<argc; i++)
        printf("%d %s\n", i, argv[i]);
    return 0;
}
```

`argc` tells you how many arguments are passed into the program.

All arguments are read into memory as text strings (even if they are numbers). These strings are accessed via `argv`. 
Program called “args”:

```c
int main(int argc, char *argv[]){
    int i;
    for (i=0; i<argc; i++)
        printf("%d %s\n", i, argv[i]);
    return 0;
}
```

The first argument is always the program name.

Remember that this is a string, not a number.
The keyboard enters strings not numbers!

Stdlib.h offers functions that can translate strings into numbers:

```c
#include <stdlib.h>
int main(int argc, char *argv[]){
    int i;
    double f;
    if (argc < 3) return 1;
    i = atoi(argv[1]);
    f = atof(argv[2]);
}
```

Also available: `atol` (arg to long), `atoff` (arg to float). Feel free to complain about the lousy names.
We’ll pick up from here next time.

See you tonight!