Reminder

- HW07 due **Saturday 24 March** electronically by midnight
- HW grades nearly caught up!

- Exam tonight!

- Join Piazza!
  - NB: It is easiest to join Piazza by first authenticating through NetBadge, accessing our UVaCollab site, and then connecting to Piazza from the link on the left-hand sidebar

- 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
Information on Mid Term Exam

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

• 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
Review and Outline

• Last time:
  – Review of pointers
  – Pointers to functions - experience in lab
  – Arrays – experience in lab
  – Review for exam
  – Passing arguments to main(..) – experience in lab

• Today:
  – More on arrays
  – Char arrays
  – More on passing arguments to main(..)
  – Random numbers
More on Arrays
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;

    for (int i = 0; i < nrow; i++) {
        for (int j = 0; j < ncol; j++) {
            // Array element
        }
    }
}
```

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.

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

\[
\begin{bmatrix}
[0][0] & [0][1] & \ldots & [0][NC-1] \\
[1][0] & [1][1] & \ldots & [1][NC-1] \\
\ldots & \ldots & \ldots & \ldots \\
[NR][0] & [NR][1] & \ldots & [NC-1][NC-1]
\end{bmatrix}
\]
2D Arrays and Pointer Arithmetic: I

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

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

Diagram:
```
[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: II

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_p[5] = array_p[0][5];

// point to array [1][3]:
array_p[3] = array_p[1*NR + 3];

// point to last element:
array_p[NC-1] = 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.

We now see that we've been using arrays all along, whenever we define a character string variable. Character strings in C are just arrays of characters:

```c
#include <stdio.h>

int main () {
    char string1[20] = "this is a test."
    char string2[20] = { 't', 'h', 'i', 's', ',', ' ', 'i', 's', ',', 'a', ' ', '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.
String Utilities: `strlen`

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>fred</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Tuesday</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

What is going on?!
Character Encoding

- **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.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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:

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

Returns:

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

```c
#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.
Special parameters in C: `argc` and `argv`

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`.
An Example

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.
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.
Random Numbers and Their Utility
Random Numbers

• Choose a random integer
Random Numbers

• Choose a random integer
  – What range? This is important!
    • There are ~60 people in this class
    • Let’s use [1,10]
Random Numbers

• Choose a random integer
  – What range? This is important!
    • There are ~70 people in this class
    • Let’s use [1,10]

  – What would a truly random distribution look like?
Random Numbers

• Choose a random integer
  – What range? This is important!
    • There are ~70 people in this class
    • Let’s use [1,10]

  – What would a truly random distribution look like?

  – What does ours look like?
Random Numbers

• Choose a random integer
  – What range? This is important!
    • There are ~70 people in this class
    • Let’s use [1,10]
  – What would a truly random distribution look like?
  – What does ours look like?
  – Are we truly “random”?
Generating Random Numbers

• Humans are not good random number generators
• Computers can do this for us
  – Standard C library functions
  – int \texttt{rand} (\texttt{void})
  – Don’t forget the seed!
  – You need \texttt{stdlib.h} to access this function and related

• But even computers are not perfect at this…
• Random numbers are essential for many computations:
  – Eg., imagine modeling a probabilistic process like nuclear decay

\[\text{CDF/MEMO/STATISTICS/PUBLIC/6850}\]

Detecting a Bad Random Number Generator
Joel Heinrich—University of Pennsylvania
January 22, 2004

1 Introduction

The main purpose of this note is to exhibit a defect in the Linux C and C++ standard library pseudo-random number generator \texttt{rand} and the Linux/UNIX system-library generator \texttt{random}. Since Linux \texttt{rand} and Linux/UNIX \texttt{random} use identical algorithms, we will use “\texttt{random}” to refer to both. The defect is uncovered when \texttt{random} fails a simple empirical test.

However, in the process of writing this note, a major bug in CLHEP’s \texttt{RandEngine} class (for Linux platforms) was discovered, which is described in section 4. This bug is unrelated to the failure of \texttt{random} mentioned above, since we call \texttt{random} directly, rather than using \texttt{RandEngine}, which is simply broken on Linux platforms.

Empirical tests of random number generators involve “goodness-of-fit” issues, and this note can also serve as a simple example of goodness-of-fit.
Generating a Random Number

• One can generate a random number in some range in which all the possibilities have the same probability:
  – The probability distribution function is sometimes called the “prior” in the vocabulary of statistics
    • Each value same probability = “flat prior”
    • Example:
      – Tossing a coin: Assign heads = 0, tails = 1. Each has probability 0.5.
      – Tossing a single fair dice: Each integer value [1,6] has same probability = ~0.166667

• A random number need not have a flat prior distribution!
  – Can have some other distribution…like a Gaussian, or some user-defined thing.
  – We will explore this in lab this week.
An Example: Integration Using Random Numbers

Integrating Over an Arbitrary Area:

Say we have some horrible function in two dimensions, like the one shown below, and we want to calculate the area enclosed within the curve.

We could arrive at a very rough estimate of the area like this:

Define a box as shown, where
- Y2 >= Max Y value of f(x,y) in the range X1 <= x <= X2
- Y1 <= Min Y value of f(x,y) in the range Y1 <= y <= Y2

Area of the box = (X2-X1)*(Y2-Y1) > Area enclosed by the function
An Example: Integration Using Random Numbers

Using Random Numbers to Estimate the Area:

If we generate random points within the box, we can use them to get a better estimate of the area:

\[ r = \frac{\text{Points inside shape}}{\text{Total points}} \]

As we generate more and more points we asymptotically approach the exact answer.

```c
n_in = 0;
for (i=0 ; i < num_trials < i++) {
    x = (double)rand() / RAND_MAX * (X2-X1) + X1;
    y = (double)rand() / RAND_MAX * (Y2-Y1) + Y1;
    if (in_fcn(x, y)) n_in++;
}

area = (X2-X1)*(Y2-Y1) * n_in / num_trials;
```
Monte Carlo Techniques: Integration

Programs that use random numbers are often referred to as “Monte Carlo” programs. The Monte Carlo method of integration isn't the best for every problem, but it's easy to implement. Even a chicken can do it:

Consider a circle in the dirt. Draw a box around the circle. Now allow a chicken to peck at will in and around your drawing.

By counting the total pecks and the pecks within the circle, we can estimate the circle's area.
Monte Carlo Integration

The Whale Example:

\[ \text{m = Number of points in whale shape} \]
\[ \text{M = Total number of points} \]

\[ \frac{m}{M} \text{ as } M \to \infty \to A_{\text{whale}} / A_{\text{box}} \]

\[ A_{\text{whale}} \approx A_{\text{box}} \times \frac{m}{M} \]

If we repeat this experiment with a different set of random points, our resulting estimate of the whale's area will be slightly different. By coming up with several estimates of the area, and looking at how much variation they display, we can get an idea of how accurate our estimate is.
Multiple Measurements

The Sample Mean:

Each experiment contains $M$ points. $N = \text{Number of experiments.}$

The mean (average) value will be:

$$\bar{A} = \frac{1}{N} \sum_{i=1}^{N} A_i$$
Advantages of Monte Carlo Integration:

• The Monte Carlo method can be trivially extended to higher dimensional problems.

• If it is at least possible to determine whether something is inside or outside of your area you can do the integral, even if the shape is beyond hope of integrating analytically.

• In general, it's more efficient to use other techniques to calculate integrals of 1D or 2D functions. But, for higher dimensional integrals, this technique is quite efficient compared to others.
We’ll pick up from here next time.

See you tonight!