Introduction to Computational Physics

Meeting 5: Arrays

Today:
- Vectors and Histograms.

Welcome back!

From now on, remember that you can try out these programs using your account on Galileo. For instructions on how to use Galileo, see:

http://galileo.phys.virginia.edu/compfac/courses/comp_intro/connecting.html

Today we'll be looking at arrays, which are lists of related variables. We'll be using them to make vectors and histograms.

Let's get started!

Last Week's Practice Problem:

How would you modify the "readfile.cpp" program so that it reads data from "writerdata.txt" and then writes two columns of data into a new file? Let's say the two columns are x and (y22 + y32).

readwrite.cpp

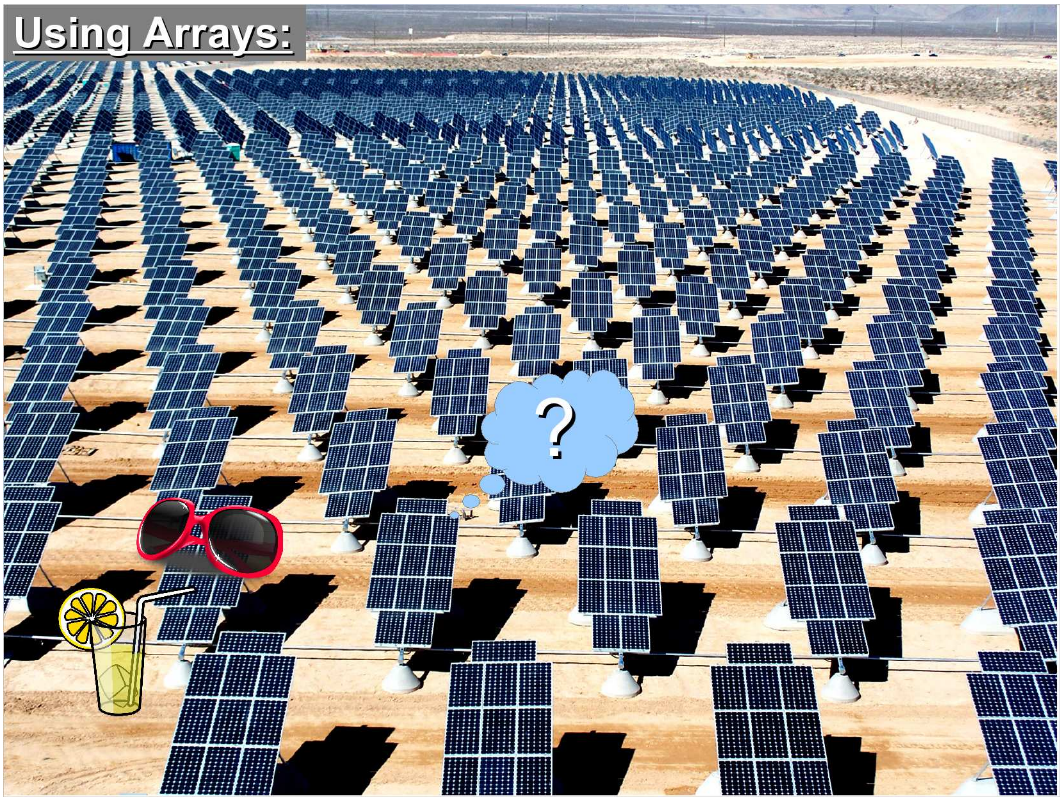
```
int status;
double x, y1, y2, y3, y4;

FILE *input = fopen("writerdata.txt","r");
FILE *output = fopen("newdata.txt","w");

while ( 1 ) {
  status = fscanf( input,
                   "%lf %lf %lf %lf %lf",
                   &x,&y1,&y2,&y3,&y4 );
  if ( status == EOF ) {
    break;
  }
  fprintf (output, "%lf %lf\n",
           x, y2*y2 + y3*y3);
}

fclose( input );
fclose( output );
```

Open a second file, for wrtiting into.

Write things into the file.

Close the file.

Sometimes programs need to have lots of files open at the same time. Remember: you don't need to use "input" and "output" as the names of your files handles. You can call them anything you like. If you need to have write to two files, you might refer to them as "out1" and "out2". Or, even better, you might use a name that tells you something about what's in the file, like "baddatapoints" and "gooddatapoints".



We often need to carry around sets of related data: the coordinates of a vector, for example. Until now, we've had no way to tell the computer that a group of variables was related.

Arrays let us do that.

Let's start out by looking at a program that doesn't use arrays, and then compare it to another program that does the same thing using arrays.

The following is the BAD example.....

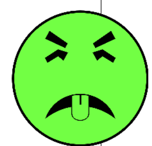
Working with Vectors:

If you had the vectors **v1** and **v2**, you might use them like this in C:

```
#include <stdio.h>
int main () {
  double x1=1.0, y1=2.0, z1=3.0; //v1
  double x2=4.0, y2=5.0, z2=6.0; //v2
  double sum1, sum2, sum3;
  double dot;

  dot = x1*x2 + y1*y2 + z1*z2;
  printf ("Dot-product of v1.v2: %lf\n",dot);

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

}
```

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.

Doing it this way will work, but it's fraught with peril. You have to keep track of the subscripts yourself, and it's really easy for typos to creep in.

Defining Vectors as Arrays:

Here's a better way:

```
#include <stdio.h>
int main () {
  double v1[3] = {1.0,2.0,3.0};
  double v2[3] = {4.0,5.0,6.0};
  double sum[3], dot=0;
  int i;

  for (i=0;i<3;i++) {
    dot += v1[i] * v2[i];
  }
  printf ("Dot-product: %lf\n",dot);

  for (i=0;i<3;i++) {
    sum[i] = v1[i] + v2[i];
  }
  printf ("Sum: %lf %lf %lf\n",
           sum[0],sum[1],sum[2]);

  return(0);
}
```

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.

With vectors, we can tie the components of the vector together, and carry the whole vector around in the program. The computer keeps track of the components, and makes sure they're in the right places.

Instead of manually typing x, y and z, we can just loop through the vector's three components with a “for” loop.

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];
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.

Think of indices as the subscripts we use in mathematics when we write expressions like X_i.

Arrays let us bundle together related data, like the elements of a vector or (as we'll see) the characters in a text string.

It's important to remember that each element of an array takes up just as much memory as a separate variable of that type. So, if we define a large array with thousands of elements, we may run into the limits of the computer's memory.

Working with Arrays:

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

```
value = v1[i] + v2[i];
```

```
result[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.

```
for (i=0;i<3;i++) {
    dot += x1[i] * x2[i];
}
```

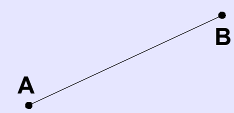
The examples above show basic array usage.

Try it!

```
#include <stdio.h>
#include <math.h>
int main () {
  double a[3], b[3], distance;
  int i;

  printf ("Enter the coordinates of point A:\n");
  for ( i=0; i<3; i++ ) {
    printf ("\tCoordinate %d: ", i);
    scanf("%lf", &a[i]);
  }

  printf ("Enter the coordinates of point B:\n");
  for ( i=0; i<3; i++ ) {
    printf ("\tCoordinate %d: ", i);
    scanf("%lf", &b[i]);
  }

  distance = 0;
  for ( i=0; i<3; i++ ) {
    distance += pow ( a[i] - b[i], 2 );
  }
  distance = sqrt( distance );

  printf("The distance between the points is %lf\n",
         distance);

}
```

This program finds the distance between two points in 3-d space.

"pow" raises numbers to a given power (2 here).



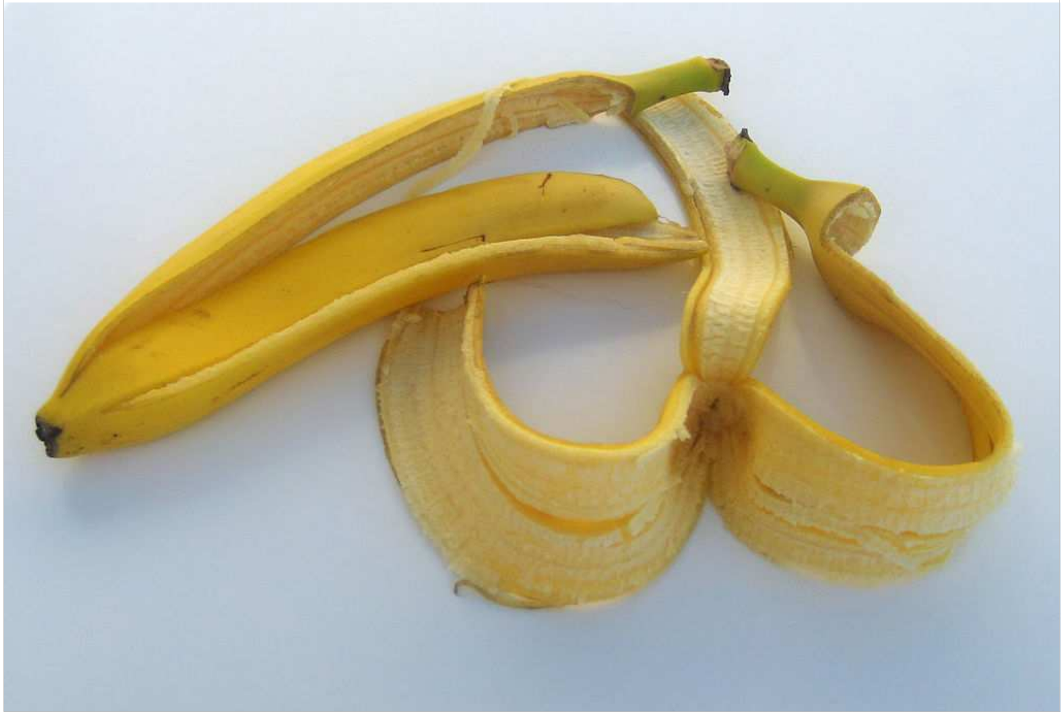
Here we calculate the distance between **A** and **B.**

The program above prompts the user for the coordinates of two points, A and B, and then computes the distance between those points.

A good way to test the program is to enter -1, 0, 0 for the first vector and 1,0,0 for the second. The resulting distance should be 2.

I've used the "pow" function here just to save space. Usually, when squaring a number, it's faster to use "x*x" instead of "pow(x,2)". Pow is useful for raising numbers to other powers, though. You can use any exponent, even fractional ones.

Some Things to Look Out For:



Arrays give us a lot of new abilities, but they also introduce a whole zoo full of potential pitfalls to beware of.

Bits and Bytes:

The data in your computer is all stored in bunches of microscopic switches. Each switch can only have two values, “1” or “0” (“on” or “off”). The amount of information stored by one switch is called a “bit”, and we often talk about flipping bits on or off.

These bits are usually grouped together in sets of eight. A group of eight bits is called a “byte”.

1	0	1	1	0	1	1	0

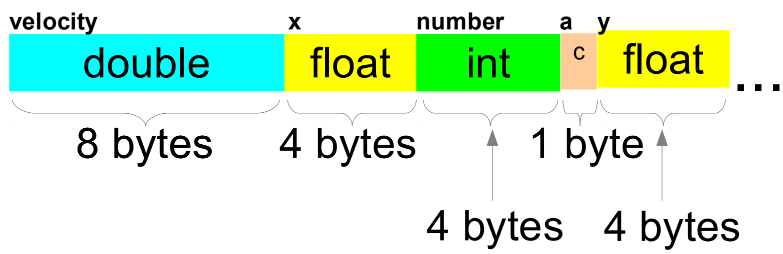
Why eight bits? First, because eight is a power of two (2^3), making it convenient for binary (base-2) arithmetic. (Just as 10, 100 or 1000 are convenient in base-10.) Second, because the very popular early Intel CPUs used data in 8-bit chunks.

To explain some of the mistakes you can make with vectors, let's start by looking again at the way your programs store numbers.

When you define a variable, the program allocates a chunk of memory where that variable's value will be stored as a binary number.

Storing Variables:

When your program runs, it sets up an area in the computer's memory for storing the value of each of your variables:



Different types of variables are given different amounts of space. Bad things can happen if you try to stick the wrong type of data into a variable.

What would happen if you tried to stick a “double” value the variable named “x”, above?

Answer: If you succeeded, the data would spill over into the adjoining variable (“number”) and corrupt it.

Similar, but more subtle things can occur with the elements of an array.

Storage Example:

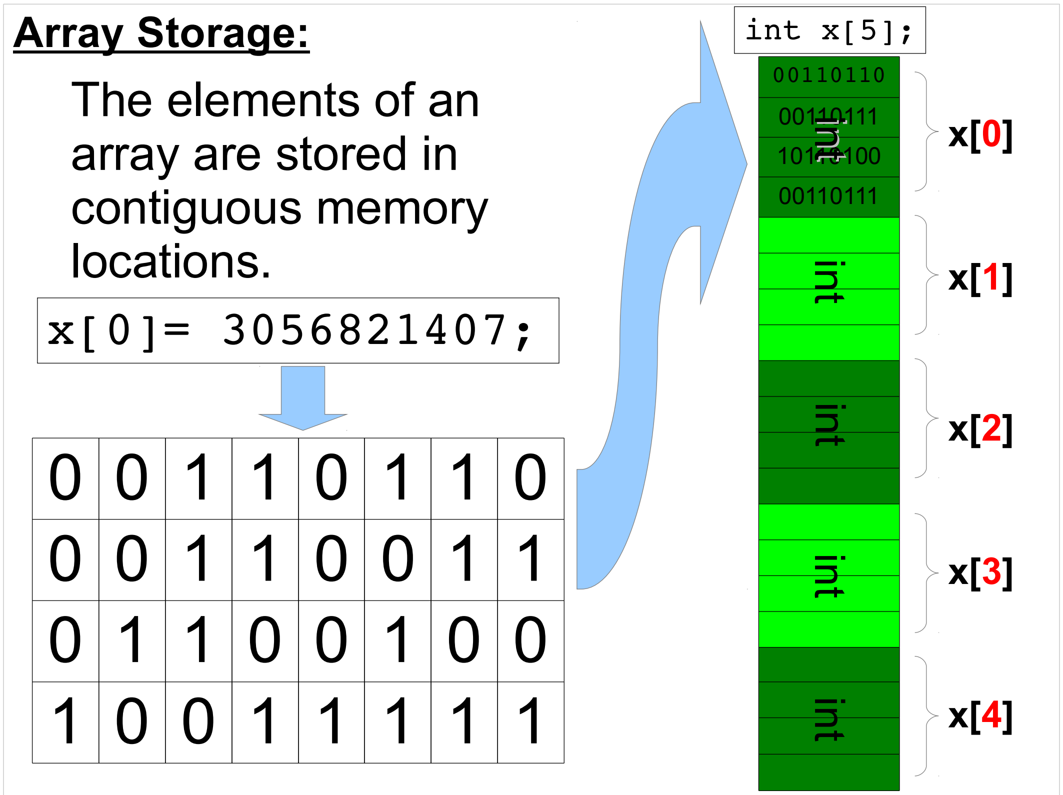
Here's how we might store an integer value, using 4 bytes of storage space. We have 32 bits of data, so we can store any number from zero up to 2^{32}-1 (which is 4,294,967,295).

1 byte
(8 bits)

1	0	1	1	0	1	1	0
0	0	1	1	0	0	1	1
0	1	1	0	0	1	0	0
1	0	0	1	1	1	1	1

4 bytes
(32 bits)

Although I've shown these bits arranged in a rectangle so you can see that it's four bytes, the bits are actually just stored one after the other.



When we define an array, our program allocates a contiguous chunk of memory that's big enough to hold all of the array's elements. (Five of them, in this example.)

The illustration shows what happens when we stick a number into element zero of the array “x”.

When you give the program an array index, the program multipies the index times the size of each element to find the memory address where a particular element lives.

Array Boundary Checking:

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

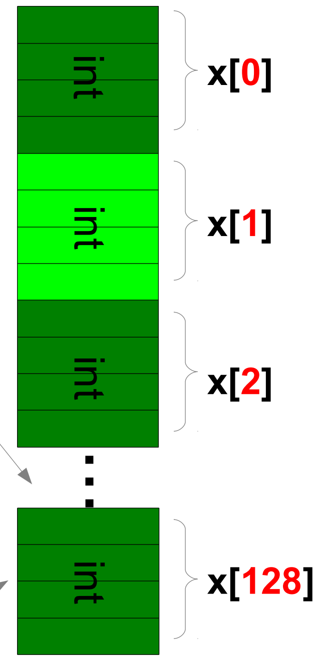
For example:

```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.



Remember: C computes the location (memory address) of an array element by multiplying the index times the size of each element.

Having your program crash with a "segmentation fault" is bad, but at least you know something went wrong, so you can try to fix it.

But what if you got no error message and the program just kept running with the wrong data in the wrong place? That can happen too....

Other Array Errors:

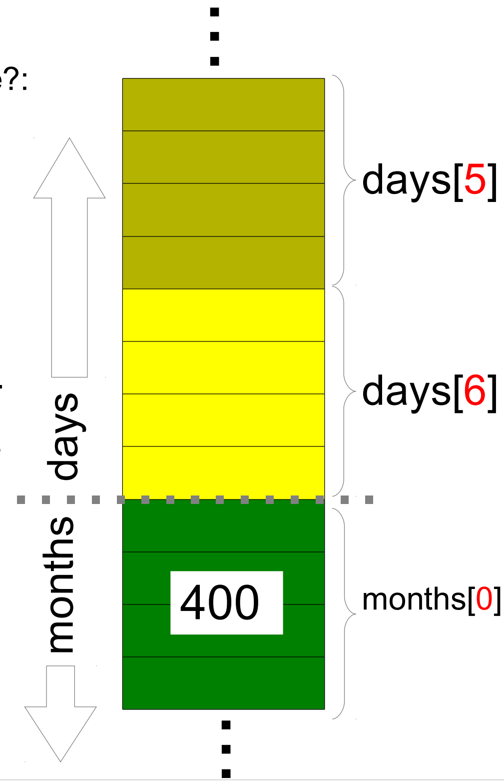
What's wrong with the following code?:

```
int days[7];
int months[12];

days[7] = 400;
```

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 400 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.



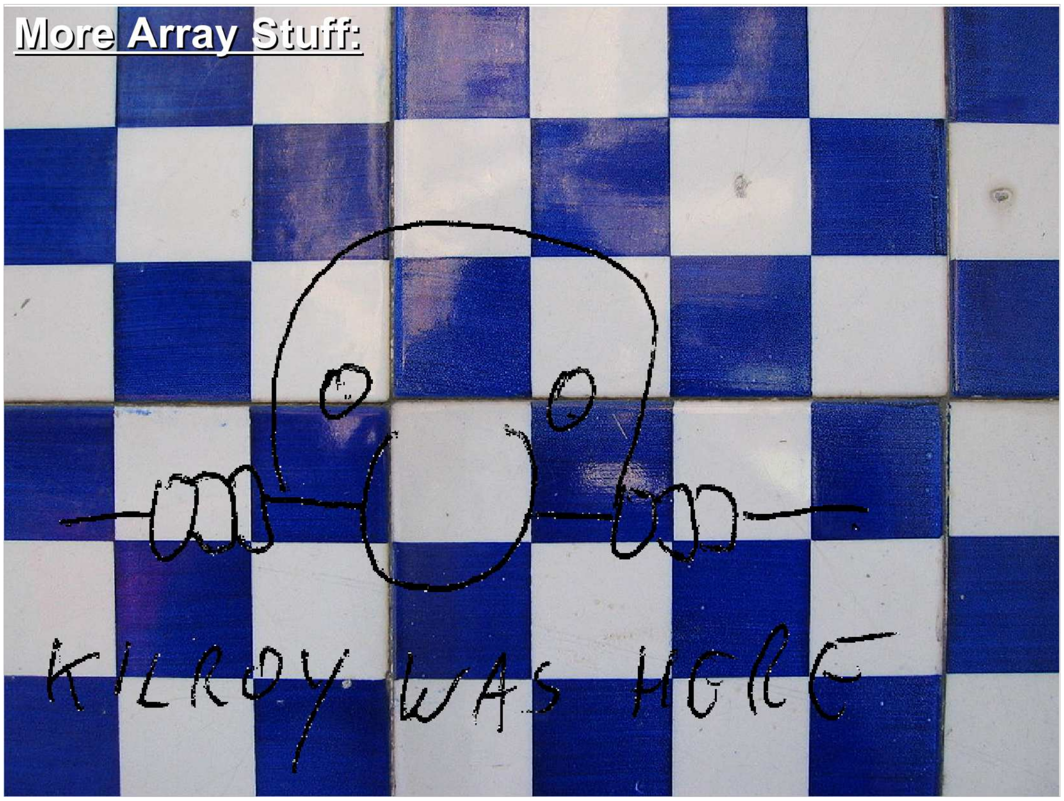
In the situation above, the program doesn't crash! It just keeps running with the wrong data in the wrong place. This is because the program is writing data into an array it really owns. As far as the operating system is concerned, the program is welcome to change this memory any way it wants to.

Accidentally writing over data in another of your own program's arrays might allow your program to run without crashing, but your results could be bizarre.

Alternatively, corrupt data may cause your program to crash in code that is far removed from where the array boundary error initially occurred.

(Think about a program w/ millions of lines of code. Ouch!)

Some tools are available to help debug these specific issues, but that's beyond our scope. Always try to write code carefully up front!



Here a couple of other things you can do with arrays. We'll come back and look at these in more detail on another day.

Multidimensional Arrays:

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

```
int main(){

  double matrix[20][30];
  int i,j;



  for (i=0; i<20; i++) {
    for (j=0; j<30; j++) {
      matrix[i][j] =
           (double)i * (double)j;
    }
  }
```

Defines a 20x30 array.

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

But again, remember that arrays take up just as much memory as the same number of individual variables. If you define a 100x100 array, you've taken up as much memory as 10,000 single variables. You can quickly run into memory limits with multi-dimensional arrays.

Character Strings as Arrays:

You may have noticed that all of our variables so far have contained numbers. We can also use variables to contain text. In programming, we usually refer to a chunk of text as a "character string". Character strings in C are just arrays of characters:

```
#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.

As you can see above, there's a special format specifer (%s) for strings. You can use this to write them out with printf or fprintf.

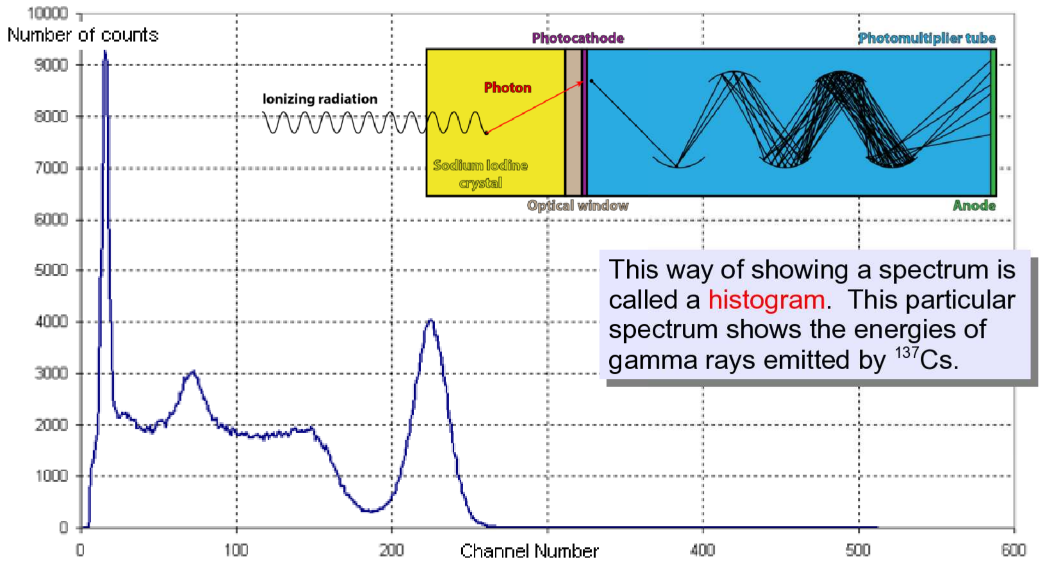
Don't try to use scanf to read strings, though. We'll talk about the right way to read strings on another day.

To Be Continued....

Stay tuned. There's more about matrices and strings to come in later meetings.

Making Histograms:

It's common to see spectra like this in all fields of science. A spectrum is just a way of showing how often something happens, or (equivalently) how likely it is to happen.

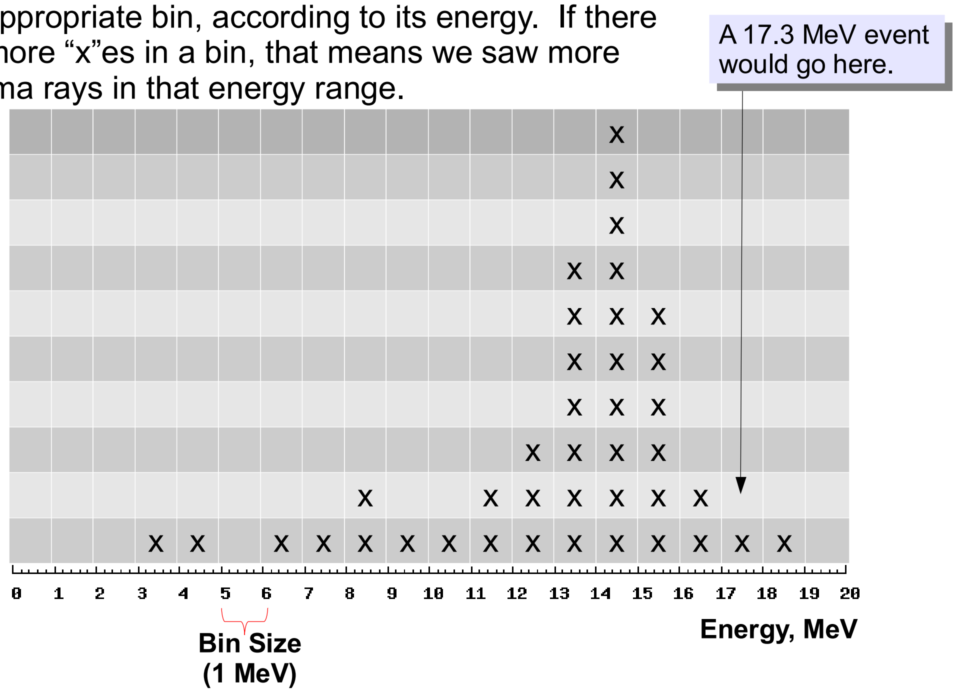


If you stay in Physics, you'll definitely generate a spectrum like this at least once during your undergraduate career.

This particular spectrum shows gamma ray energies, as seen by a Sodium Iodide (NaI) scintillation detector. The incoming gammas create flashes of light in the NaI. The amount of light is proportional to the amount of energy deposited in the crystal. This light is then detected by a photomultiplier tube (PMT) and converted into an electrical signal that we can measure.

Constructing the Histogram:

Whenever we see a new gamma ray, we put an "x" in the appropriate bin, according to its energy. If there are more "x"es in a bin, that means we saw more gamma rays in that energy range.



We could use any bin size and any range of energies. It's up to us to choose something appropriate.

Getting Some Spectrum Data:

I've made a data file for you that contains the energies of 100,000 different gamma rays. You can get it by typing this:

```
wget http://tinyurl.com/spectrum-dat
mv spectrum-dat spectrum.dat
```

If your computer doesn't have “wget”, type this instead:

```
curl -L -O  http://tinyurl.com/spectrum-dat
mv spectrum-dat spectrum.dat
```

spectrum.dat

```
35.130490
36.942571
36.627112
40.780935
34.569799
35.192141
...
```

If you look inside the file (with “nano”, for example), you'll just see a column of numbers. Each number is the measured energy, in MeV, of a particular gamma ray. All of the energies are between 0 and 50 MeV.

Let's make a histogram of these energies.

In a real experiment, you might need to convert the numbers you measure from, say, voltages to energies. In this case, let's just assume that all of those problems have been dealt with, and we're being presented with a file full of energy values and being asked to analyze them.

```c
int i, status, bin, overflow = 0;
double energy, binsize = 1.0; //MeV.
int counter[50];

for ( i=0; i<50; i++ ) {
  counter[i] = 0; // Reset all counters to zero.
}

FILE *input = fopen( "spectrum.dat", "r" );
while (1) {
  status = fscanf( input, "%lf", &energy );
  if ( status == EOF ) {
    break;
  }

  bin = energy/binsize;
  if ( bin < 0 || bin >= 50 ) {
    overflow++;
    continue; // Skip this line, and jump to the next.
  }

  counter[bin]++; // Increment the appropriate counter.
}

for ( i=0; i<50; i++ ) {
  printf ("%d %d\n", i, counter[i]);
}
printf ("Saw %d over/underflows\n", overflow);
```

We've chosen a bin size of 1.0 MeV.

This is important! We'll see why later.

“bin” is an integer, so if “energy/binsize” is, say, 4.3, bin will be equal to 4.

If bin is too big or small, skip this gamma.

Here I've introduced a new C statement: “continue”. “continue” is like “break”, but instead of quitting the loop, it just skips the rest of this iteration and immediately goes to the top of the loop again.

In this week's practice problem you'll see why it's important to set all of the array elements to zero before you start filling them.

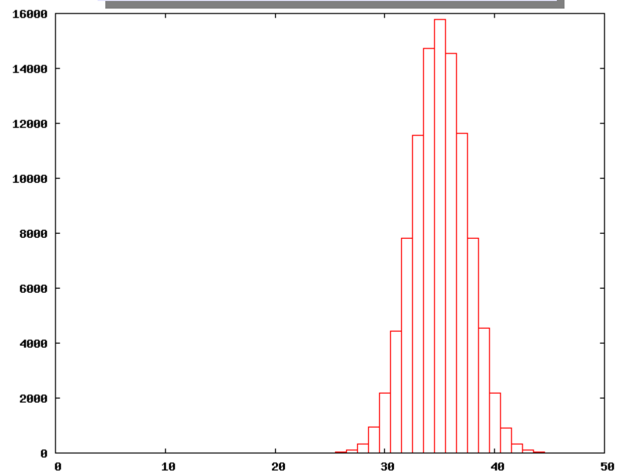
Why do we bother to check for overflows and underflows? In general, there might be oddball events in our data. (Maybe a high-energy cosmic ray zips through our detector, or maybe an electronic problem creates some negative numbers in our data file.) If we don't deal with these oddballs, we risk writing past the bounds of our array, and corrupting memory.

Running the Program:

When you run your program, you should see output like this.

```
24 1
25 15
26 45
27 181
28 600
29 1474
30 3188
31 6042
32 9623
33 13399
34 15437
35 15366
36 13296
37 9848
38 6047
39 3212
40 1409
41 560
42 192
43 46
44 11
45 6
46 1
47 0
48 0
49 0
Saw 0 over/underflows
```

Here's what it would look like if we graphed it:



Yay! We did it.

Practice Problem:

What if we didn't zero out out counters in the histogram program? Try writing, compiling and running this program to see what happens:

```
#include <stdio.h>
int main () {
  int data[10];
  int i;

  for ( i=0; i<10; i++ ) {
    printf ("%d %d\n", i, data[i]);
  }

}
```

Try running the program several times. What happens? Do you see why it's important to set counters to zero before using them?

The End

Thanks!