Physics 2660

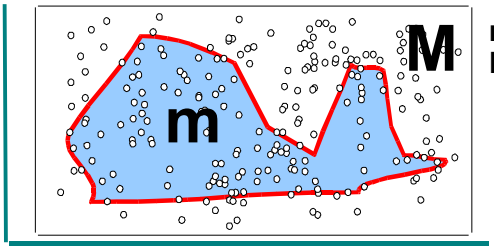
Lecture 5: C – Part 4

Today
- Estimating Errors in Monte Carlo Results
- Arrays
- Reusing code
- Passing arguments to main()
- Pointers to functions
- Basic numerical integration / differentiation techniques

Part 1: Estimating Errors

Let's take another look at Monte Carlo integration, focusing on the “Whale” example we saw in the last lecture.

The Whale Example:



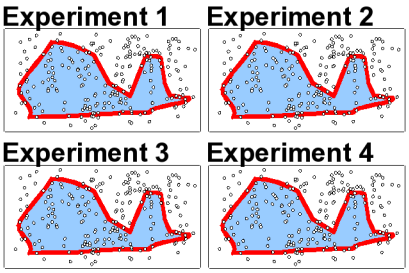
m = Number of points in whale shape
M = Total number of points

$$m/M \xrightarrow{\text{as } M \to \infty} A_{whale}/A_{box}$$

$$A_{whale} \approx A_{box} * m/M$$

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.

The Sample Mean:



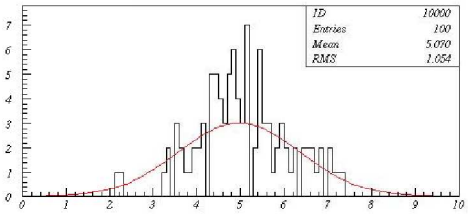
Each experiment contains M points.
N = Number of experiments.

Say we repeat the experiment N times. From our N experiments, we get a set of independent estimates of the area: $A_1, A_2, A_3 \ldots A_N$.

The mean (average) value will be:

$$\overline{A} = \frac{1}{N}\sum_{i=1}^{N} A_i$$

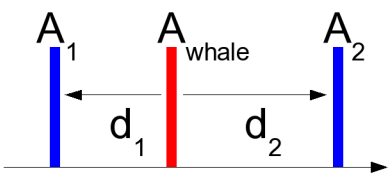
If we made a histogram of our estimates, it might look like this:



Click here for a demonstration.

We call this the “sample mean” because our experiments are a small sample of the infinite number of possible experiments that we could do. If we could do an infinite number of experiments, we'd find that the sample mean would approach an underlying value called the “population mean”.

The Standard Deviation:



Each of our A_i values will deviate from the true value (A_{whale}) by some amount, d_i.

Since these deviations may be positive or negative, the average deviation will tend toward zero.

How, then, can we come up with an estimate for a "typical" deviation? We can sum the squares of the d_i values, instead!

We define the "variance" (σ^2) as the average squared deviation from the true value:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} d_i^2 = \frac{1}{N}\sum_{i=1}^{N}(A_i - A_{whale})^2$$

Its square root, σ, is called the "Standard Deviation" or "Standard Error". It's also sometimes called the Root-Mean-Square (RMS) deviation, for obvious reasons.

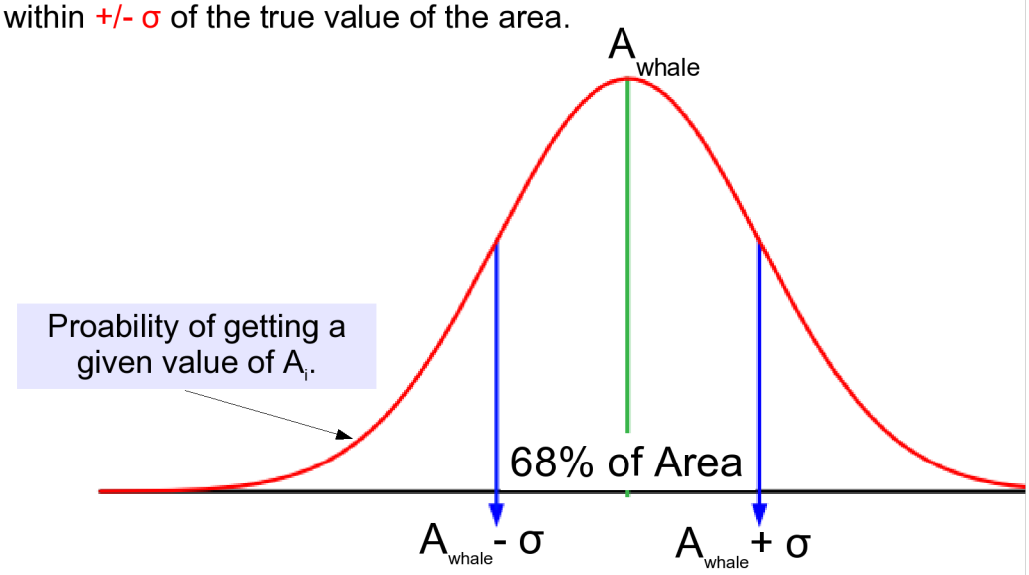
Note that there are other ways we could choose to quantify a "typical" deviation. For example, we could look at the average absolute value of d.

The standard deviation has some nice properties, though. In particular, it has a natural relationship to the gaussian (normal) distribution. For example, 2σ is the distance between the "points of inflection" (the place where the curvature goes from positive to negative) of the gaussian distribution.

More importantly, the sample standard deviation is usually the best estimate of the standard deviation of the parent population.

What the Standard Deviation Means:

The standard deviation tells us about the distribution of our A_i values. In particular, if the values follow a "normal" (gaussian) distribution, it tells us that we should expect that about 68% of our A_i values will fall within +/- σ of the true value of the area.



And it's a good bet that the Ai values will be distributed approximately like a gaussian. Why is this? Because of the "Central Limit Theorem", which says that any linear sum of random variables tends toward a gaussian, no matter what the distribution of the individual variables looks like.

The Central Limit Theorem is so important that it's called the "second fundamental theorem of probability". (The first is the Law of Large Numbers.) For more information, see: http://en.wikipedia.org/wiki/Central_limit_theorem

Note that this means you can construct a pretty good gaussian distribution just by adding together sufficiently many numbers pulled from a uniform distribution, without using things like the Box-Muller transform.

The Sample Standard Deviation:

The problem with computing σ is that we don't know the true value, A_{whale}. Our best approximation of this value is the average of all of our A_i measurements, $\overline{A}$.

But, it can be demonstrated that just plugging $\overline{A}$ in place of A_{whale} will systematically underestimate the value of sigma.

We correct for this by dividing by N-1 instead of N, and arrive at the following approximation:

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(A_i - \overline{A})^2 \simeq \sigma^2$$

We call s the Sample Standard Deviation.

Dividing by N-1 instead of N is known as “Bessel's Correction”. See the following Wikipedia article for more information: http://en.wikipedia.org/wiki/Bessel%27s_correction

You can see the problem intuitively by considering the case where you only have one measurement, A_1. Since this measurement will probably be different from the true value, A_{whale}, it makes perfect sense to calculate σ from it. We would just arrive at $\sigma = |A1-A_{whale}|$.

If we use $\overline{A}$ in place of A_{whale}, we run into trouble, though. If we only have one point, then $\overline{A} = A_1$, so $A_1-\overline{A}$ will always be zero, giving us zero as the estimate of our standard deviation. This isn't very realistic. If we divide by N-1 instead of N, then our value for s becomes “undefined”, instead (0/0). This is a more reasonable approximation of the truth.

Of course, for large values of N, there's not much difference between N and N-1.

Computing the Standard Deviation in a Program:

For purposes of writing computer programs, it's useful to rearrange our expression for s^2 a little. Here are our expressions for s^2 and the average, $\overline{A}$:

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(A_i - \overline{A})^2 \qquad \overline{A} = \frac{1}{N}\sum_{i=1}^{N} A_i$$

Substituting the expression for $\overline{A}$ into the first equation, we can collect terms and arrive at this:

$$s^2 = \frac{1}{N-1}\left[\sum_{i=1}^{N} A_i^2 - \frac{1}{N}\left(\sum_{i=1}^{N} A_i\right)^2\right]$$

Sum of squares — Sum of values

This is easy to code, because our program just needs to do one loop, and keep two sums: the sum of the values and the sum of their squares.

If we didn't do it this way, we'd have to loop through all of the data once, to calculate the average, and then loop through it again to use the average value in calculating s^2.

Note, however, that there's some evidence that the method you choose may have a noticeable effect on the accuracy of your result. See the following article for an interesting discussion of this:
http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation/

Standard Error of the Mean:

We said that our best guess of the true value, A_{whale}, is the mean of our sample values, $\overline{A}$. Now let's get back to the problem of quantifying just how good that guess is. Earlier, we defined the mean as:

$$\overline{A} = \frac{1}{N}\sum_{i=1}^{N} A_i = \frac{1}{N}(A_1 + A_2 + A_3 ...)$$

If we know the standard deviations of the A_i values, we can use them to calculate the standard deviation of the mean, through the regular propagation of errors process:

$$\sigma^2_{f(x,y,z...)} = \sigma^2_x(\frac{\partial f}{\partial x})^2 + \sigma^2_y(\frac{\partial f}{\partial y})^2 + \sigma^2_z(\frac{\partial f}{\partial z})^2 ...$$

Where we treat $\overline{A}$ as a function of the variables A_1, A_2, A_3...A_N.

Why is $\overline{A}$ our best guess for the value of A_{whale}? The answer lies in something called the “Law of Large Numbers”, which says that "the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed. “ For more information and proofs, see this Wikipedia article:
http://en.wikipedia.org/wiki/Law_of_large_numbers

How does propagation of errors work? See the following for more information about that:
http://en.wikipedia.org/wiki/Propagation_of_uncertainty

Standard Error of the Mean, cont'd:

Applying this to our expression for $\overline{A}$, we get:

$$\sigma_{\overline{A}}^2 = \sigma_1^2 \frac{1}{N^2} + \sigma_2^2 \frac{1}{N^2} + \sigma_3^2 \frac{1}{N^2} \ldots$$

What are the σ_i values? They're the estimates of how far each A_i value typically deviates from the true value. In our case, the values for σ_i will all be equal to σ, the standard deviation of our collection of A_i values. So, we can simplify the equation above and write:

$$\sigma_{\overline{A}}^2 = \frac{1}{N}\sigma^2 \quad \text{or} \quad \sigma_{\overline{A}} = \frac{\sigma}{\sqrt{N}}$$

Our uncertainty in $\overline{A}$ decreases like the $\sqrt{N}$ as N increases. For example, to get a 10-times better value for $\overline{A}$, we need to do 100 times as many measurements. Also, because we don't know the exact value of σ, we say that $\sigma_{\overline{A}} \approx s/\sqrt{N}$.

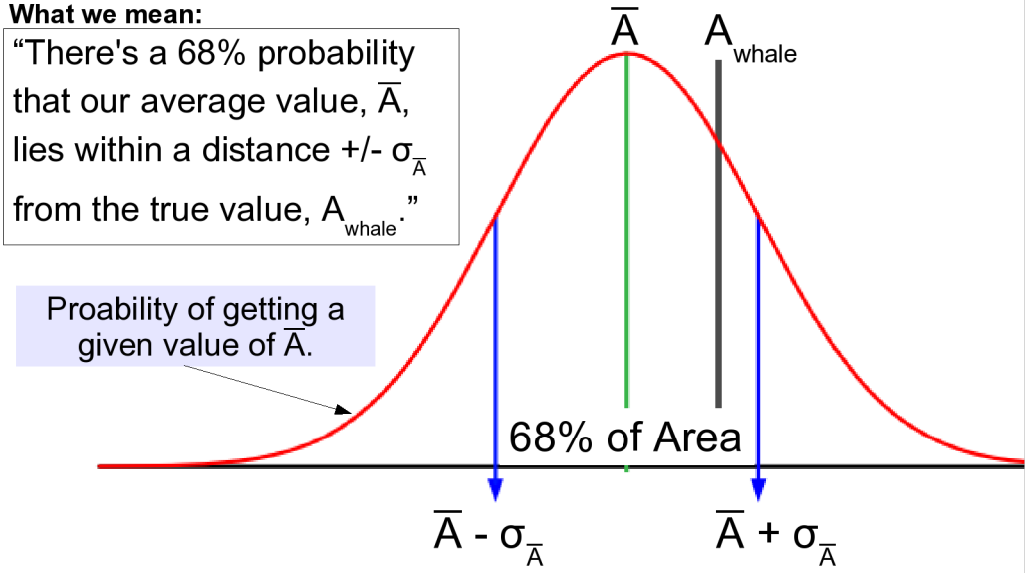
To know σ, we would have to know the true value of the area (A_{whale}).

Confidence Levels:

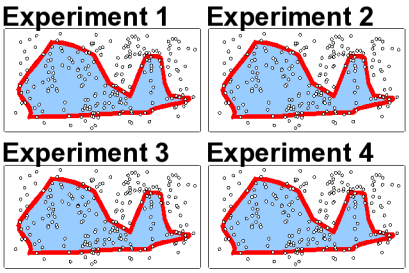
When we cite an experimental result, we also state the uncertainty in the result. So, in our whale-shape example, we might give our final estimate of the area as $\overline{A}$ +/- $\sigma_{\overline{A}}$ (for example "1.12 +/ 0 .01").

What we mean:

"There's a 68% probability that our average value, $\overline{A}$, lies within a distance +/- $\sigma_{\overline{A}}$ from the true value, A_{whale}."



Number of Points per Experiment:



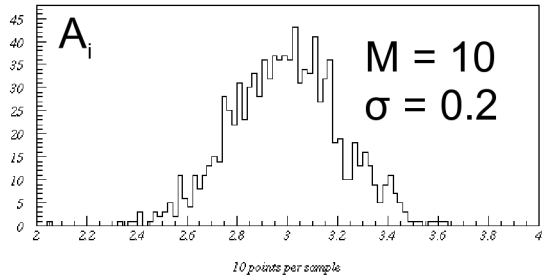
Each experiment contains M points.
N = Number of experiments.

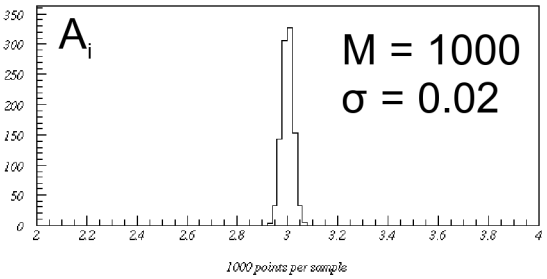
Fewer points will increase the uncertainty in each A_i value (σ). But, since

$$\sigma_{\bar{A}} = \sigma/\sqrt{N}$$

we can compensate for a factor of ten increase in σ by just increasing N (the number of experiments) by a factor of 100.

What about "M", the number of points in each experiment? So far, everything we've said has been independent of M.

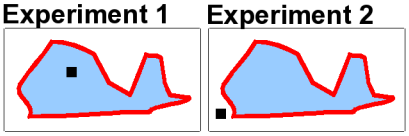




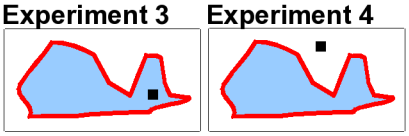
What we find is that only the total number of points matters, added up over all experiments. This is what you'd expect. If we, for example, decided to subdivide the points in each experiment into "1a", "1b", "2a", "2b",etc., so that we had twice as many experiments, we wouldn't expect that artificial division to affect our final result. The data is still the same, we've just categorized it differently.

Treating Each Point as an Experiment:

Experiment 1 Experiment 2



Experiment 3 Experiment 4



Each experiment contains M points.
N = Number of experiments.

What would happen if we set M to 1? Then the number of points within the whale for each experiment (m_i) would always either be 0 or 1.

Each experiment's estimate of A_{whale} is given by:

$$A_i = \frac{m_i}{M} A_{box}$$

So, our mean value for the area could be written:

$$\overline{A} = \frac{1}{N}\sum_{i=1}^{N} A_i = \frac{1}{N}\sum_{i=1}^{N} \frac{m_i}{M} A_{box} = \frac{A_{box}}{NM}\sum_{i=1}^{N} m_i$$

If M=1, then m_i must be either 0 or 1,

and:

$$\overline{A} = \frac{A_{box}}{N}\sum_{i=1}^{N} m_i = \frac{n}{N} A_{box}$$

where "n" is just the sum of all of the m_i values (in other words, the total number of points within the whale).

If you're uncomfortable with m_i being either zero or one, another way to look at it is to define p as the probability that a given point will land inside the whale, with p = n/N.

Standard Deviation when M=1:

So, if we now define the mean value of A like this:

$$\overline{A} = \frac{A_{box}}{N} \sum_{i=1}^{N} m_i$$

And (by analogy with what we've done before) the standard deviation of m like this:

$$s_m^2 = \frac{1}{N-1} \left[\sum_{i=1}^{N} m_i^2 - \frac{1}{N} (\sum_{i=1}^{N} m_i)^2 \right]$$

We arrive at the following expression for the standard error of the mean, in the case where we have only one experiment with many points:

$$\sigma_{\overline{A}} \simeq \frac{s_m}{\sqrt{N}}$$

Click Here for a Demonstration.

The demonstration shows the mean and standard error of the mean being continuously re-calculated as we add more and more points. As you can see, it gets narrower as N increases, and the peak settles into a stable position.

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.

Working with Vectors:

Here's how we've worked with vectors up until now:

```
#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);
}
```

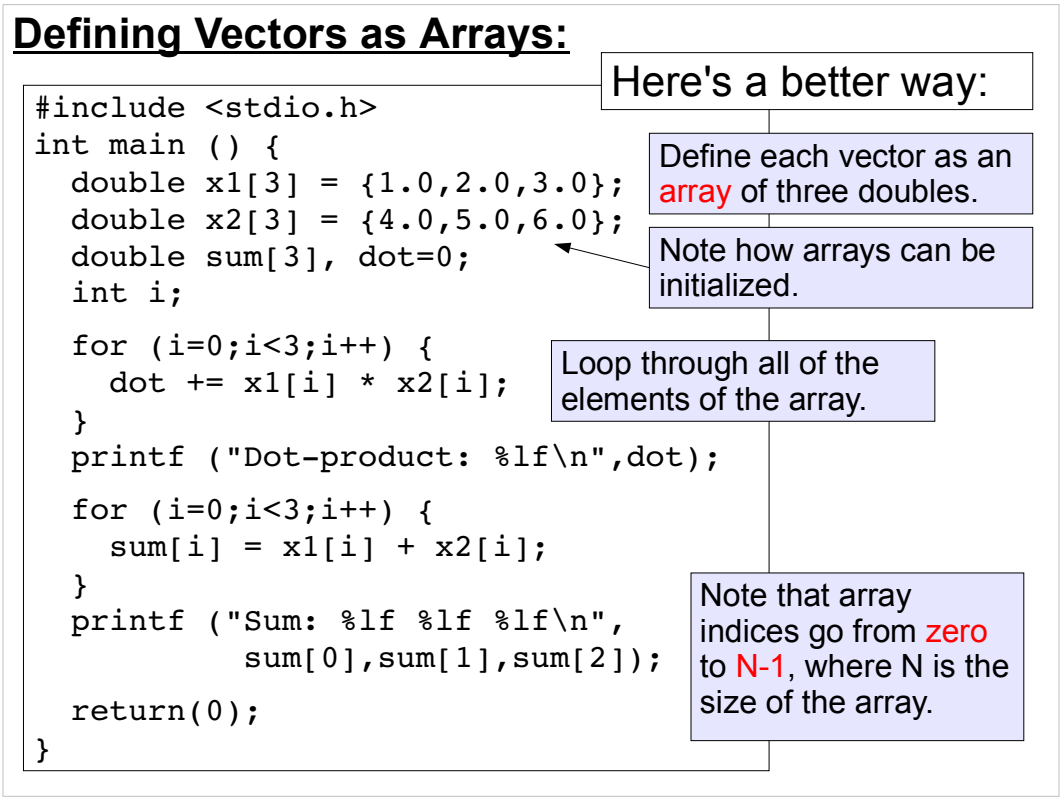
Define a variable for each vector element.

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.

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:




```
#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);
}
```

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.

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.

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 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 = x1[i] + x2[i];
```

```
x[i] = M_PI*area;
```

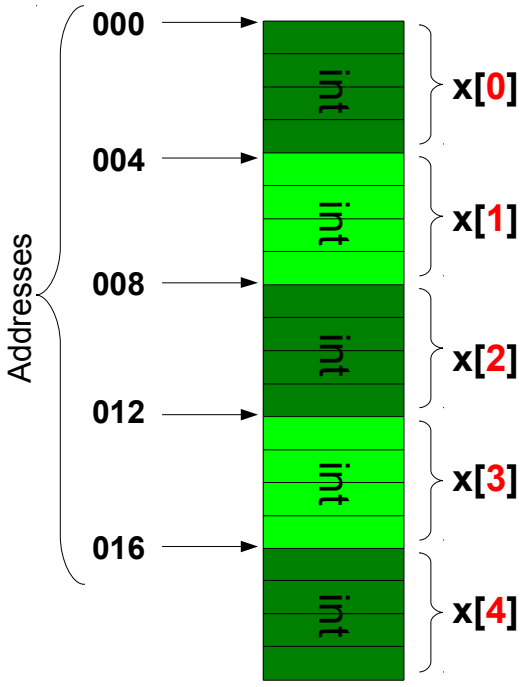
It's important to remember that the values of array indices start with zero, and that they end at N-1.

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

Array Storage:

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

```
int x[5];
```



When you give the program an array index, the program multipies the index times the size of each element to find the 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.

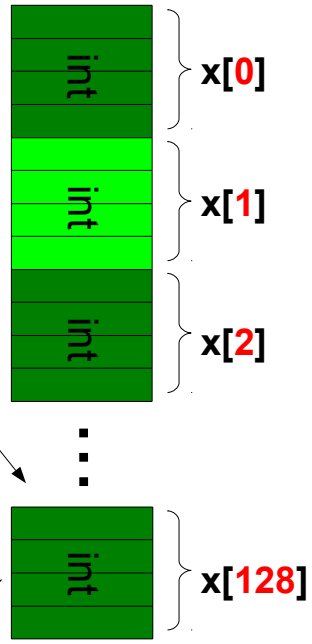
For example:

```
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.

Other Array Errors:

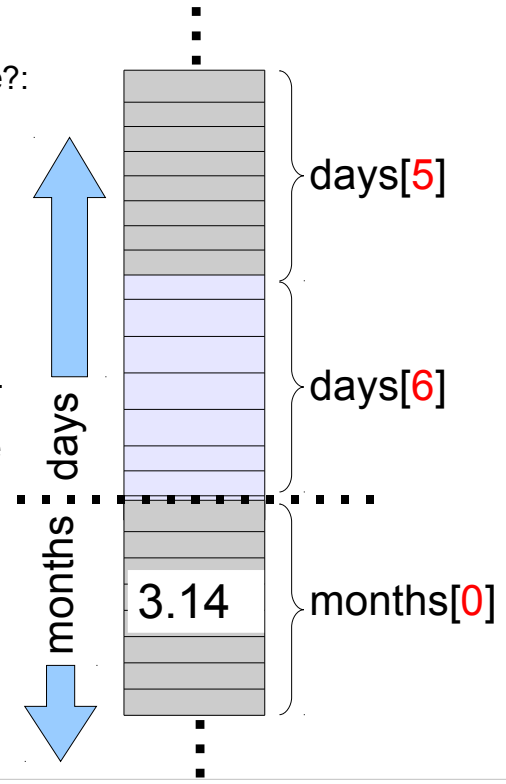
What's wrong with the following code?:

```
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.



Overwriting array boundaries can cause very confusing problems in your programs:

Segmentation faults are often the easiest to deal with if the fault is immediate on the access to the undefined array location

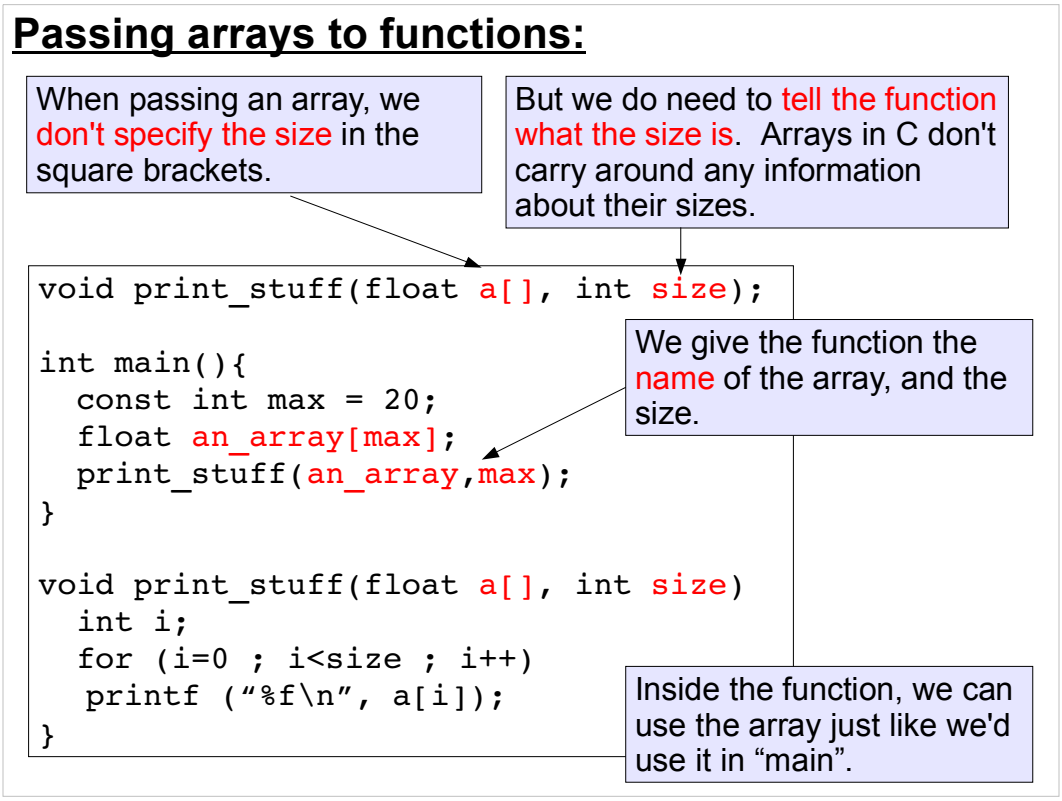
Overwriting unrelated data may allow your program to run without crashing, but your results may be bizarre

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!

Passing arrays to functions:



As we'll see "a [ ]" is just equivalent to a pointer.

Multidimensional Arrays:

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

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



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

Defines a 20x20 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.

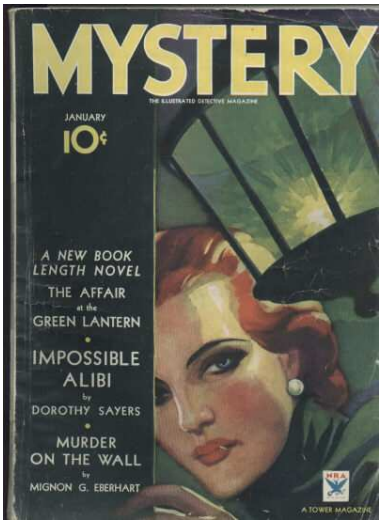
Character Strings as Arrays:

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:

```
#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.

Arrays and Pointers:



In C, when you give the name of an array, it's equivalent to a pointer. For example, consider the following code:

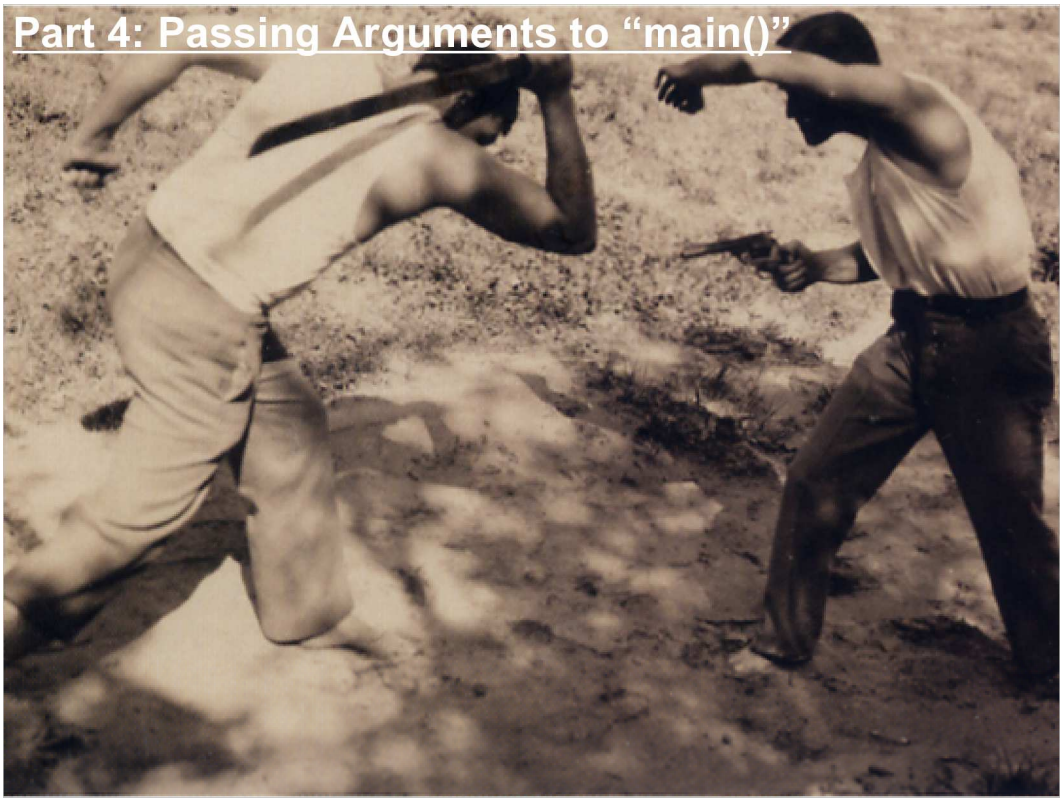
```
void printit(double a[],int s);
int main() {
  double a[50];
  double *aptr;
...
  printit (a,50);
  printit (aptr, 50);
}
```

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 to put an ampersand in front of the names of character strings? It's because these variables are already pointers.

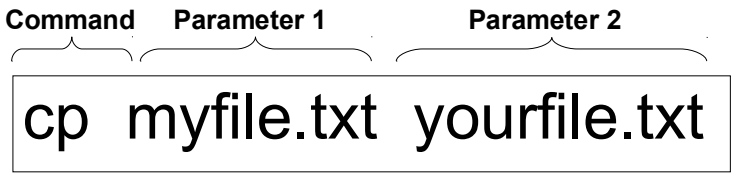
The name of an array variable is treated as a pointer pointing to the address of the beginning of the array.

Part 4: Passing Arguments to “main()”



Passing Arguments to “main”:

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



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.

The “argc” and “argv” Parameters:

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.

Using “argc” and “argv”:

Here's an example showing how argc and argv can be used:

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

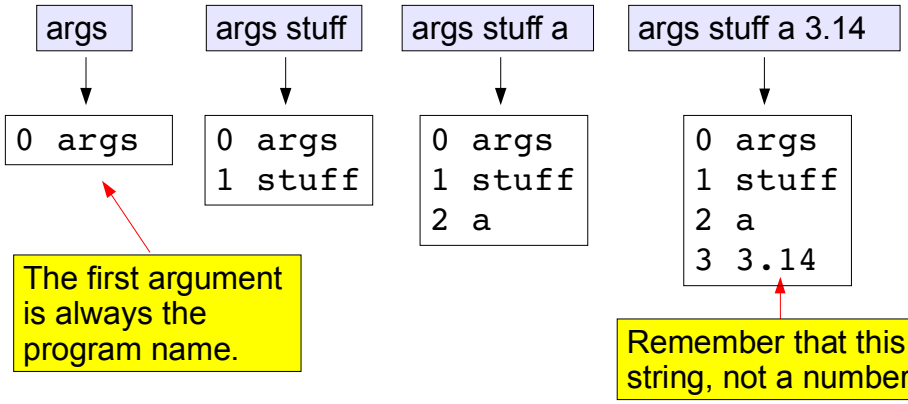
argc tells you how many arguments are passed into the program.

All arguments are read into memory as text strings (even if they are numbers). These strings are accessed via argv.

Argument Examples:

Program called "args":

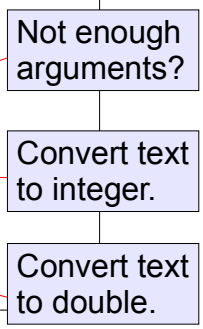
```
int main(int argc, char *argv[]){
  int i;
  for (i=0; i<argc; i++)
    printf("%d  %s\n", i, argv[i]);
  return 0;
}
```



Getting Numbers from argv:

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

```
#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.

Part 3: Reusing Code

There's no point in reinventing the same programming solution over and over again.

For example, deep down the printf function is horribly complicated, yet once it is coded, you simply have to remember a fairly simple interface to reuse the function again and again.

Ditto for things like sin(x), sqrt(x), etc... Do we really want to rewrite the code for these functions?

Perhaps as a challenge... but when you get down to doing real work, there's no point in repeating what you or someone else has already completed.

Using #include to Re-use Code:

File "sqrtn.cpp":

```
double sqtrn(double x) {
   double guess = x/2.0;
   while (fabs(guess*guess-x)>1e-6)
         guess = (guess + x/guess)/2;
   return guess;
}
```

File "main.cpp":

```
#include <stdio.h>
#include <math.h>
#include "sqrtn.cpp"
int main(){
  double x;
  printf("enter a number\n");
  scanf("%lf", &x);
  printf("sqrt(%lf) = %lf\n", x,
         sqrtn(x));
}
```

Note the use of quotes around the file name. Angle brackets (<>) are reserved for files in "standard" system directories. In general your personal includes must give the full directory path to the file, or be in the current directory.

The disadvantage of this approach is that every time you compile a program using your function sqrtn you must also compile the code for sqrtn.

If sqrtn is a large function (or if you include many such functions), this can unnecessarily increase the time it takes to build your programs.

Creating Object Files:

File "sqrtn.cpp":

```
#include <math.h>
double sqtrn(double x) {
   double guess = x/2.0;
   while (fabs(guess*guess-x)>1e-6)
         guess = (guess + x/guess)/2;
   return guess;
}
```

An object file is compiled code that hasn't been fully processed into a program. The above code isn't a complete program.

We can compile the code into an object module as follows, using the "-c" flag of g++:

```
g++ -O -Wall -c sqrtn.cpp
```

→ Creates the file sqrtn.o.

sqrtn.o contains the function's code translated into CPU instructions. The -c flag causes g++ to stop after compiling, without continuing to the "linking" step that produces a runnable program.

We'll talk later about how object files can be packed into "libraries".

Linking Object Files with Your Program:

If we have a pre-compiled object file, we only need to #include a header file containing the prototype for the function:

File "sqrtn.hpp":

```
double sqtrn(double x);
```

File "main.cpp":

```
#include <stdio.h>
#include <math.h>
#include "sqrtn.hpp"
int main(){
  double x;
  printf("enter a number\n");
  scanf("%lf", &x);
  printf("sqrt(%lf) = %lf\n", x,
         sqrtn(x));
}
```

We can then compile our main program by typing:

g++ -o main main.cpp sqrtn.o

The disadvantage of this approach is that every time you compile a program using your function sqrtn you must also compile the code for sqrtn.

If sqrtn is a large function (or if you include many such functions), this can unnecessarily increase the time it takes to build your programs.

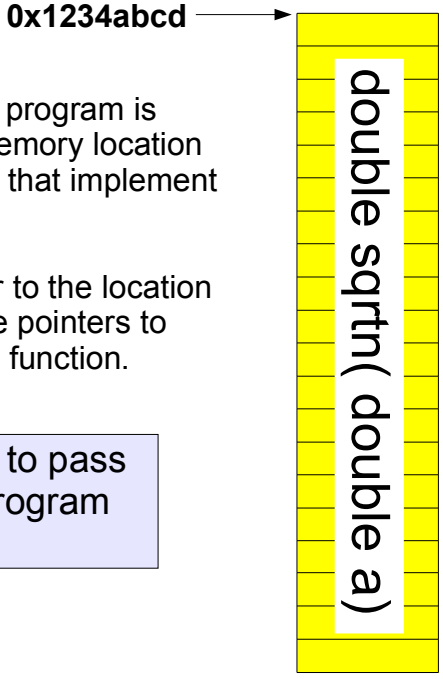
The disadvantage of object files (and libraries) is that they're not portable from one type of computer to another. I could conceivably take a cpp file from Linux to Windows to OS X and compile and run it in each place. If I copy an object file from Linux to Windows, it will be useless there.

Part 5: Pointers to Functions

Remember the quartic Mandelbrot set problem? We took an example that used the Z^2 version of the set, and we modified it by changing the iteration function to use Z^4 instead. Most of the rest of the program stayed the same. Wouldn't it have been nice to have a “generate_mandelbrot” function that just took a function as an argument and then did the right thing?

Well, as it turns out, you can do that kind of thing in C by using pointers to functions.

Functions in Memory:

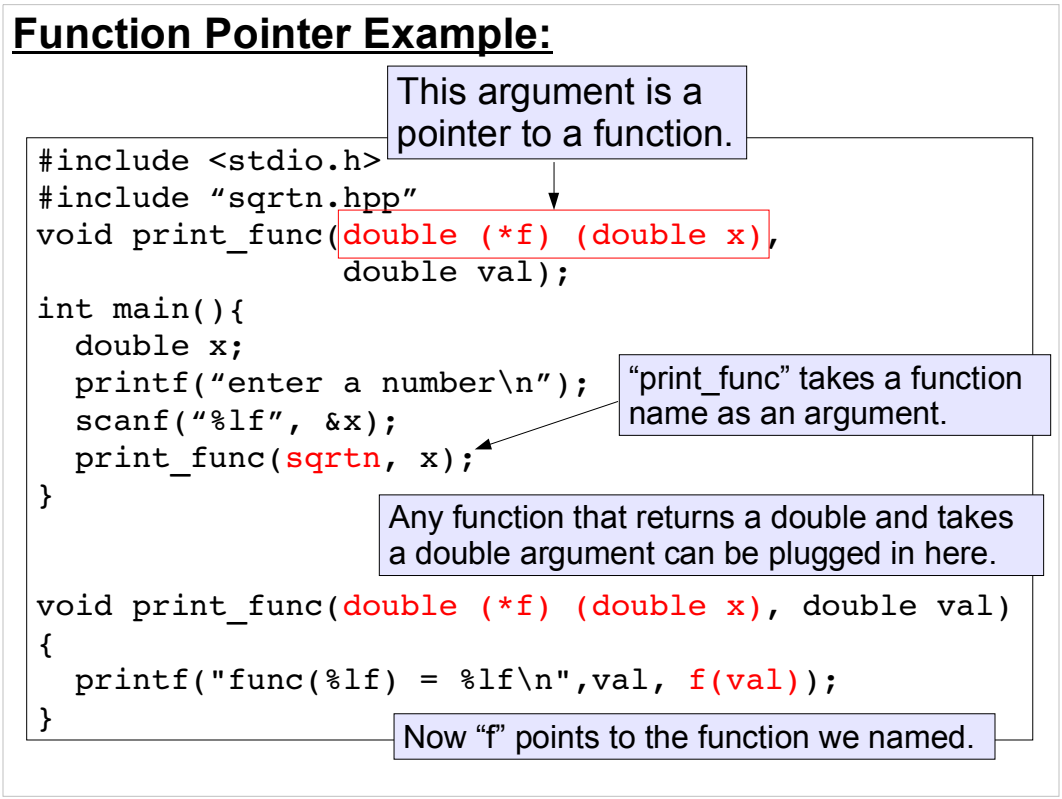


Like variables, each function in your program is stored in memory. The function's memory location holds the machine-code instructions that implement the function.

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.

Function Pointer Example:



print_func uses the name f to call the function

Q: What's going on?

A: f contains the location of the code for sqrtn in our example.
f also "knows" that sqrtn returns a double, so f(val) is interpreted as a double in the printf function.

Think of f(val) above as doing:
1) "run the code for a function at the address stored in f"
2) "pass that function a double" (val in this case)
3) "expect to receive a double returned from the function"

Anatomy of a Function Pointer:

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

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

```
double (*f) (double x)
```

2 1 3

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”.

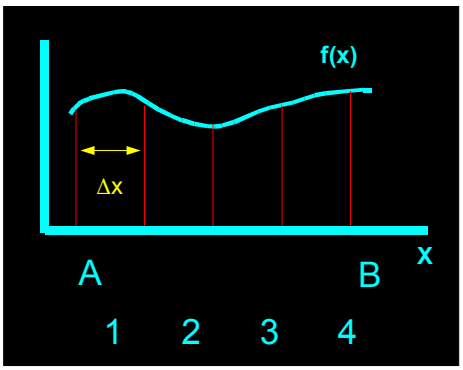
3. This pointer points to a function that takes one argument, of type “double”.

Part 6: Numerical Integration

Now let's look at a place where we can make good use of pointers to functions.

We've looked at Monte Carlo integration, but for some problems there are easier, quicker ways to do the integration.

The Trapezoidal Rule:

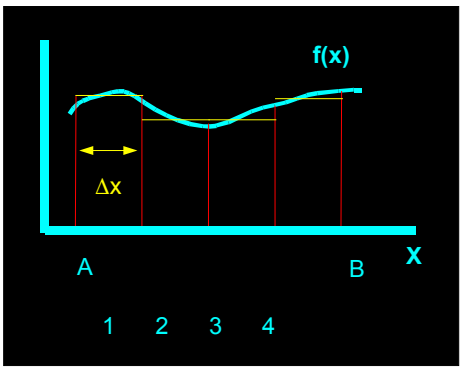


Trapezoidal rule integration offers a very simple method to approximate the integral of a one-dimensional function.

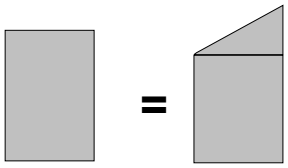
Consider the integral of f(x) over the range A<= x <= B.

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, Δx, of the subdivision.

The Trapezoidal Rule, cont'd:

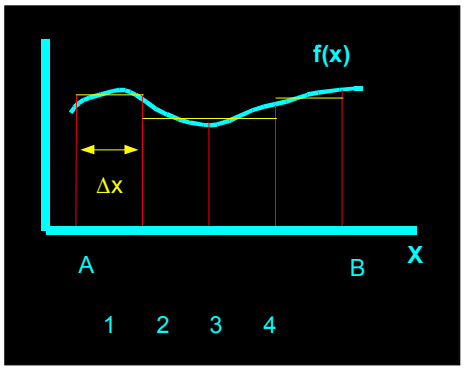


By summing the rectangular regions shown, we can estimate the integral of f(x)....



In the limit that Δx is small enough so that f(x) is essentially linear over Δx, this estimate is very accurate. If f(x) is linear over Δx, then the shape of each sub area is trapezoidal and our box covers the same area.

The Trapezoidal Rule, cont'd:



We can estimate the area by summing the subdivisions.

$$\int_A^B f(x) \simeq$$

$$\Delta x \frac{f(A)+f(A+\Delta x)}{2} + \Delta x \frac{f(A+\Delta x)+f(A+2\Delta x)}{2} + \ldots + \Delta x \frac{f(A+(n-1))+f(B)}{2}$$

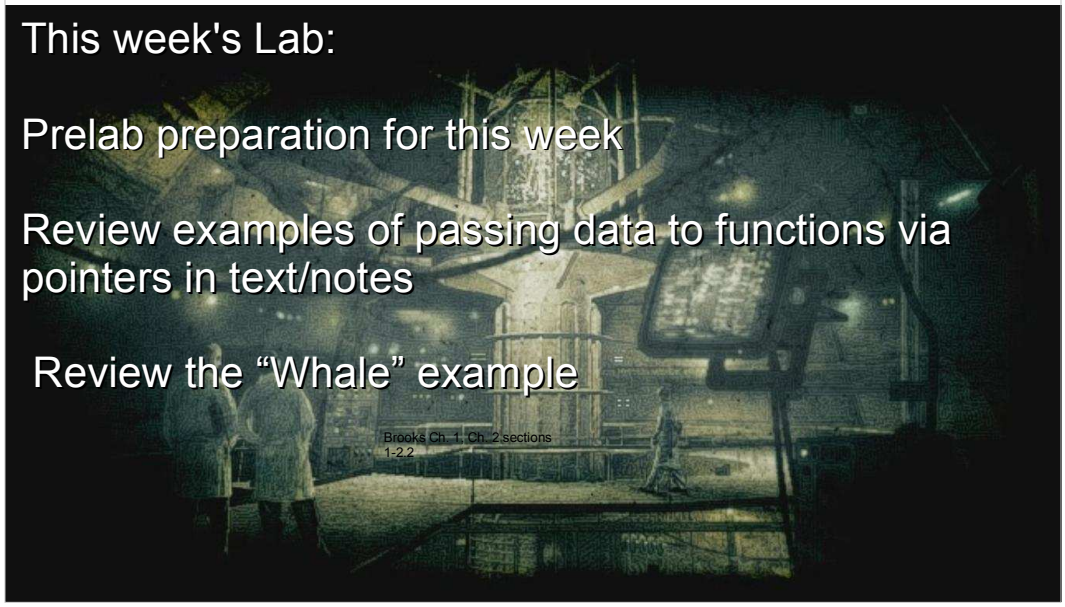
$$\int_A^B f(x) \simeq \Delta x \left[\frac{f(A)+f(B)}{2} + \sum_{i=1}^{n-1} f(A+i\Delta x) \right]$$

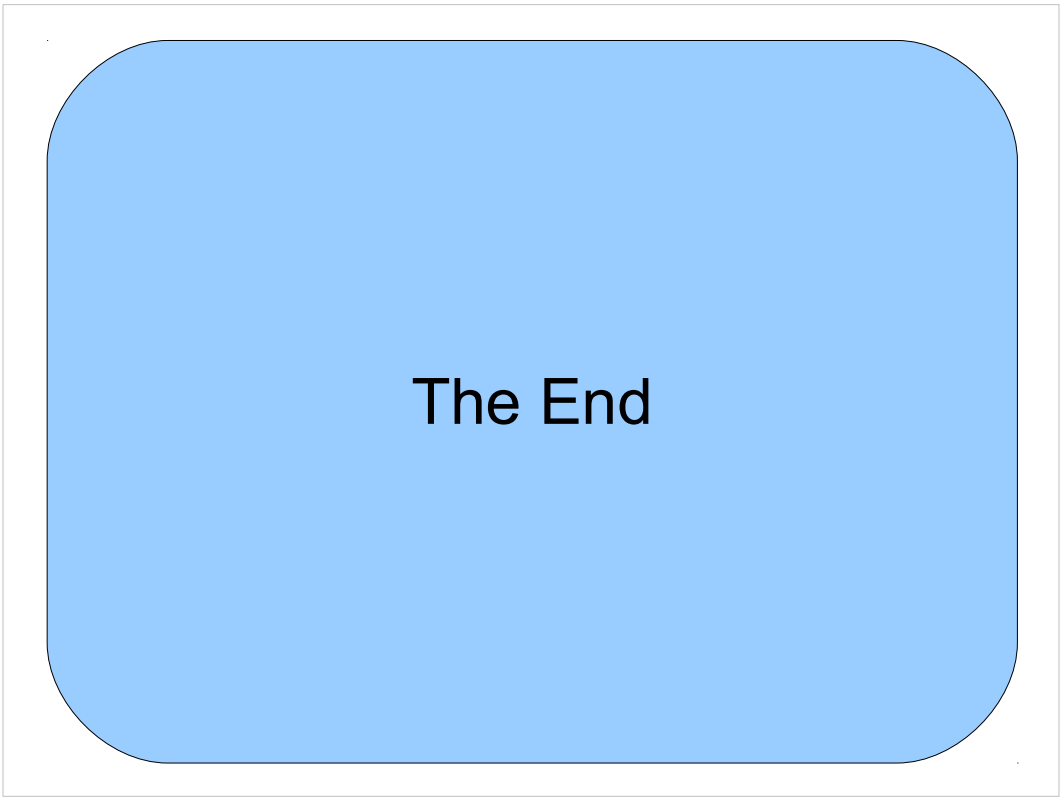
Implementing the Trapezoid Rule:

We now can integrate any 1D function we choose. Accuracy is controlled by the steps parameter

```
#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);
}
```

Next Time:


This week's Lab:
Prelab preparation for this week
Review examples of passing data to functions via pointers in text/notes
Review the “Whale” example
Brooks Ch_1_Ch_2 sections 1-2.2



Thanks!