Physics 2660: Fundamentals of Scientific Computing

Lecture 9
Instructor: Prof. Chris Neu (chris.neu@virginia.edu)
Reminder

• HW07 will be assigned today
• Grading of HW04, HW05, HW06 coming!
• Register for piazza!
  – Only 59 out of 70 so far
• You will need the second textbook, Lyons, starting now

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

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

• Last time:
  – Arrays in C
  – Strings as character arrays
  – Passing arguments to main()

• Today:
  – Structures
  – Making reusable code
Exam Review
Exam Results

- Average: 84.7%
- Low: 72.3%
- High: 100%

- Numbers 1-30 went well
  - These were the assessment, verbatim
- People did less well on sections 2,3,4
Structures!
Let's say we wanted to store some census information about each of the fifty states.

There are several interesting facts about each state, but we can only store one fact in each variable. So we might choose to store the data in a bunch of parallel arrays, like this:

```c
int population[50];
double income[50];
double area[50];
double birthrate[50];
double deathrate[50];
```

This will work, but it's a little awkward. It would be nicer if we could bundle together all of the facts about a given state into one package.
In addition to the regular variable types like “int” and “double”, C lets us define our own **custom-made types** for variables, and pack multiple pieces of data into them.

For example, we could define a 50-element array called “state” that would hold all of our census data:

```c
struct {
    int population;
    double income;
    double area;
    double birthrate;
    double deathrate;
} state[50];
```

“state” is of type “struct” (a **data structure**), and each element of the array will contain several pieces of related data.
You can refer to a particular piece of data in a struct by using the dot operator ("."): 

```c
struct {
    int population;
    double income; // Average/person/year.
    double area;   // In sq. miles.
    double birthrate; // Per year.
    double deathrate; // Per year.
} state[50];

state[0].population = 1234567;
state[0].income = 40280.0;
state[0].birthrate = 1280.5;
state[0].deathrate = 1280.1;
```
What if we wanted to use the same data structure for other variables? Say, for example, we wanted to store census data for a group of 100 countries. We could just re-type the struct definition:

```c
struct {
    int population;
    double income; // Average/person/year.
    double area;   // In sq. miles.
    double birthrate; // Per year.
    double deathrate; // Per year.
} state[50];

struct {
    int population;
    double income; // Average/person/year.
    double area;   // In sq. miles.
    double birthrate; // Per year.
    double deathrate; // Per year.
} country[100];
```
Using Multiple Equivalent Structures

What if we wanted to use the same data structure for other variables? Say, for example, we wanted to store census data for a group of 100 countries. We could just re-type the struct definition:

```c
struct {
    int population;
    double income; // Average/person/year.
    double area; // In sq. miles.
    double birthrate; // Per year.
    double deathrate; // Per year.
} state[5];

struct {
    int population;
    double income; // Average/person/year.
    double area; // In sq. miles.
    double birthrate; // Per year.
    double deathrate; // Per year.
} country[100];
```

Tedious!
Instead of re-typing the struct, we could use `typedef` to define an alias for this struct:

```c
typedef struct {
    int population;
    double income; // Average/person/year.
    double area;   // In sq. miles.
    double birthrate; // Per year.
    double deathrate; // Per year.
} census;
```

```c
census state[50];
census country[100];
```

With `typedef` we’ve created a new variable type called “census” and now we can use this to define variables, just like “int” or “double”. 
More Examples of typedef

You don't need to use struct to use typedef. You can use typedef to define aliases for any variable type:

```c
//Define aliases for some types:
typedef double funds;
typedef double weight;
typedef int days;

//Use these aliases to define some variables:
funds bank_balance;
weight fish_per_month[12];
days til_christmas;
```

This may make it easier for you to re-define your variables later on. Say, for example, that you've made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you'll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.

This may make it easier for you to re-define your variables later on. Say, for example, that you’ve made so much money that you now need to use a “long double” to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the typedef statement that defines “funds”.
Pointers to Structures

We can have pointers to structs just like pointers to any other type of variable:

```c
typedef struct {
    int population;
    double income; // Average/person/year.
    double area;    // In sq. miles.
    double birthrate; // Per year.
    double deathrate; // Per year.
} census;

census states[50];
census *sptr;
sptr = states;

printf ("Pop = %d\n", states[0].population);
printf ("Pop = %d\n", (*sptr).population);
printf ("Pop = %d\n", sptr->population);
```

When using pointers, C gives us two ways to get data from a struct.
Pointers to Structures: The "->" Operator

typedef struct {
...
} census;

census states[50];

for (i=0; i<50; i++) {
    clear_data( &states[i] );
}
...

void clear_data( census *s ) {
    s->population = 0;
    s->income = 0;
    s->area = 0;
    s->birthrate = 0;
    s->deathrate = 0;
}

Here's another example showing how the "->" operator can be used to refer to variables within a structure.
```c
typedef struct{
    double re, im;
} Complex;

double magnitude(Complex z) {
    return sqrt( z.re*z.re + z.im*z.im );
}

void conjugate(Complex *z) {
    z->im = -1.0*z->im;
}

int main() {
    Complex q;
    q.re = 12.;
    q.im = 23.;
    conjugate(&q);
    printf("q*=%f, %f; |q|=%f \n",
            q.re, q.im, magnitude(q));
}
```

Function to find the magnitude.

Function to convert number to its complex conjugate.

Define q as “complex”.

Aside: Passing Copies or Pointers to Functions

Here are two different ways we could write the “magnitude” function:

Pass a copy of the complex structure to the function. Do calculation from the copied data.

```c
double magnitude(complex z) {
    return sqrt(z.re*z.re + z.im*z.im);
}
```

Pass a pointer to the complex structure to the function. Do calculation from the original data.

```c
double magnitude(complex *z) {
    return sqrt(z->re*z->re + z->im*z->im);
}
```

Generally more efficient: less data to move around.
Making Pieces of Code that Are Reusable
Reusable Code

• Many tasks in computer programming for scientific purposes are confronted over and over again
  – imagine calculating the sine of some angle or the distance between two points in some linear algebra project
  – imagine every time you needed that value, having to write a piece of new code to do the calculation

• There is point in reinventing the wheel every time

• Utilities such as these – and many others we can think of – are best coded once and reused over and over again
  – reduction of chance for bugs
  – easier debugging
  – portability of code to other users
Using `#include` to Re-use Code:

File “sqrtn.cpp”:

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

File “main.cpp”:

```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.
Reusable Code: Option I

Using `#include` to Re-use Code:

File “sqrtn.cpp”:

```cpp
double sqrtn(double x) {
    double guess = x/2.0;
    while (true) {
        return guess;
    }
}
```

File “main.cpp”:

```cpp
#include <iostream>
#include <cmath>
#include <stdio.h>

int main() {
    double x;
    printf("enter a number \n");
    scanf("%lf", &x);
    printf("sqrt(%lf) = %lf\n", x, sqrt(x));
    return 0;
}
```

Drawback of this Option:

You have to compile the extra code every time!
Creating Object Files:

File “sqrtn.cpp”:

```cpp
#include <math.h>
double sqrtn(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.
Reusable Code: Option II

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”:

```cpp
double sqrtn(double x);
```

File “main.cpp”:

```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,
           sqrt(x));
}
```

We can then compile our main program by typing:

```
g++ -o main main.cpp sqrtn.o
```
Code Libraries
What is a Code Library?

• A code library is a collection of pre-compiled functions that one can use as needed plugging into newly developed pieces of code
  – Typically contain oft-used utilities that are convenient to just simply re-use rather than re-code

• We have been using code libraries all the time!
  – functions contained in stdio.h, stdlib.h, math.h, etc.

• We will learn how to make our own code library here!
To build a library, first make object files (as we have done before):

```c
    g++ -O -Wall -c hist.cpp
    g++ -O -Wall -c random.cpp
```

Next combine the object files into a library:

```c
    ar -csr libp2660.a hist.o random.o
```

The `archive` command is used to create your library, the syntax we will use is:

```c
    ar -csr lib<name>.a file1.o file2.o ...
```

You can list the contents of your library with a command like:

```c
    ar -t libp2660.a
```

...
Example Using Code from a Library

```c
// Header files for your library
#include "random.hpp"
#include "hist.hpp"

int main(){
    hl myHist;
    // Set range for histogram's x-axis
    hlinit( &myHist, 0, 100);
    for (int i=0; i<1000; i++) {
        // Fill the histogram w/ 100 data points
        // from the function randn:
        hlfill(&myHist, randn(50,10));
    }
    // Plot the histogram to the screen
    hlplot(&myHist, "");
    return 0;
}
```
To use your library with a program:

1) make sure your program includes header files defining the functions you use

2) tell the linker how to find your library

Let's say your program file is called `test_hist.cpp`. You would build the program as follows:

```
g++ -O -Wall test_hist.cpp -o test_hist -L. -lp254
```

- `-L` specifies a new directory to search for library files (here we add `"."`, the current directory, to the library search path)

- `-l` (small “L”) gives the name of a library (libp254.a) to search for object files needed to complete your program. Note that the “lib”/“.a” prefix/suffix is omitted from the command
In general the header files and libraries will not be located in your current working directory, so for more complex programs the build command could be of the form:

```
g++ -O -Wall \ 
-I<include_dir1> -I<include_dir2> \ test_hist.cpp 
-o test_hist \ 
-L<lib_dir1> -L<lib_dir2> \ 
-l<lib1> -l<lib2> -l<lib3>
```
Example: Structs and Functions
Example: The Gravity Problem

The Problem:
We want to read in the position vectors, initial velocities, and masses of a bunch of objects. Then, using this data, we want to calculate the gravitational force on each object, due to the others. (For the first part, we'll ignore the initial velocities.)

Here's one of our objects. It has mass “m”, and it's located at position $X$. The calculated force on it is $F$. 
Example: The Gravity Problem

Adding the Forces:

To find the total force on one mass, we just add the force vectors due to each of the other forces.

\[ \vec{F} = \vec{F}_b + \vec{F}_c + \vec{F}_d \]
Example: The Gravity Problem

Finding Distance and Direction:

We'll need to know the distance and direction to each other object.

This is the vector from a to b:

\[ \vec{r} = \vec{X}_b - \vec{X}_a \]

The magnitude of this vector gives us the distance:

\[ r = |\vec{r}| \]

Once we know these, we can make a unit vector pointing from a to b:

\[ \vec{u} = \vec{r} / r \]
Example: The Gravity Problem

**Calculating a Single Force:**

Newton tells us that the magnitude of the gravitational force between two objects is:

\[ F = G \frac{m_a m_b}{r^2} \]

The force will point toward the other object, so the force vector will just be:

\[ \vec{F} = F \hat{u} \]
Example: The Gravity Problem

Data Structure:

To solve this problem programmatically, we'll first need a data structure to store information about each body:

```c
typedef struct{
    double s_vec[3];    // space(position) vector
    double v_vec[3];    // velocity vector
    double f_vec[3];    // force vector
    double mass;
} body;

const int MAX_BODIES = 100;
body bodies[MAX_BODIES];    // array of bodies
```
Reading Data from a File:

```c
int read_data(char* file, body *bodies){
    int num=0; // number of entries read from file
    int status;
    FILE *file_p = fopen(file,"r");

    while(num<MAX_BODIES) {
        status=fscanf(file_p,"%lf %lf %lf %lf %lf %lf %lf %lf",
                       &bodies[num].s_vec[0],
                       &bodies[num].s_vec[1],
                       &bodies[num].s_vec[2],
                       &bodies[num].v_vec[0],
                       &bodies[num].v_vec[1],
                       &bodies[num].v_vec[2],
                       &bodies[num].mass);
        if (status==EOF) break;
        num++;
    }
    return num;
}
```
Example: The Gravity Problem

You will also need to write functions for:
- calculating the distance between objects
- the difference between objects’ positions, using vectors
- calculating the total force on each object
- calculating the center of mass for the array of objects
Data Visualization: Histograms
A histogram shows the distribution of data by dividing it up into **discrete intervals** and counting the data points that fall within each interval.
Elements of a Histogram
Elements of a Histogram

- range
- number of bins
- number of total entries
- number of entries in each bin
- number outside range
Reduction Data:

Histograms can reduce large quantities of data into a manageable summary.

Here we've divided the range from 0 to 100 into 50 bins.

We then generated 1000 random numbers and counted how many fell into each bin.

So, with just 50 numbers (the counts in each bin) we have a summary that shows us the distribution of our 1000 random numbers. We could go on and generate millions of numbers, and we'd still be able to summarize them in the same histogram, using only 50 counters.
Histograms in C as a Structure

Here's a structure we could use to store histogram data. At the heart of it is an array of counts.

```c
#define HBINS 50

typedef struct {
    double h_array[HBINS];
    double xmin, xmax;
    int entries;
    int under_flow, over_flow;
    double sumx, sumx2;
} h1;
```

- These histograms will have a fixed number of bins.
- Array of bin counts.
- min/max of x range.
- Total # of counts.
- Counts outside range.
- Sums of values and squares of values, for calculating statistics.
Operations With / On Histograms

- **create / initialize**: Set range for histogram, etc.
- **reset**: Clear bin contents to 0.
- **fill**: Add a data value to the histogram.
- **dump**: Print contents of histogram.
- **plot**: Graphically display the histogram.
Histogram Operations: Function Prototypes

/* initialize hist. Set min/max limits for the histogram */
void hlinit(hl *hist, double xmin, double xmax);

/* add a data point to a histogram */
void hlfill(hl *hist, double x);

/* dumps hist to screen (filename="") or to a file "filename". Returns 0 for success, 1 for error */
int hldump(hl *hist, char *filename);

/* calculate and return statistics for a histogram
   input: hl *hist
   output: int *entries, double *mean, double *std_dev */
void hlstats(hl *hist, int *entries, double *mean, double *std_dev);

/* plot a histogram to the screen (filename="") or a graphics file "filename" */
void hlplot(hl *hist, char *filename);

We will use these functions in lab this week!
Reset and initialize:

```c
void hlreset(hl *hist){
    int i;
    hist->entries=0;
    hist->sumx=0;
    hist->sumx2=0;
    hist->over_flow=0;
    hist->under_flow=0;
    for (i=0; i<HBINS; i++) hist->h_array[i]=0;
}

void hlinit(hl *hist, double min, double max){
    hist->xmax = max;
    hist->xmin = min;
    hlreset(hist);    // clear all storage variables
}
```
### Filling:

```c
void h1fill(h1 *hist, double x) {
    int bin = 0;
    double binSize, lowedge;

    if (x < hist->min) hist->under_flow++;
    else if (x >= hist->max) hist->over_flow++;
    else {
        binSize = (hist->max - hist->min) / HBINS;
        lowedge = hist->min; // low edge of 1st bin
        while (fabs(x - lowedge) > binSize) {
            bin++;
            lowedge += binSize; // move to next bin
        }
        hist->h_array[bin]++; // increment the appropriate bin
    }
    hist->entries++;
    hist->sumx += x;
    hist->sumx2 += x*x;
}
```
How to Make a Histogram

• Biggest mistake made by young scientists in my experience:
  – They have taken some nice data, they maybe have some model that they are testing, done a good job
  – They want to convey the message from their experiment through some visualization of their data, a plot
  – And then they make a plot like this:
How to Make a Histogram
How to Make a Histogram
How to Make a Histogram
How to Make a Histogram
How to Make a Histogram
How to Make a Histogram

![Histogram Graph]

- **Measurements**
- **Data**
- **Theoretical Model**

**Y-Axis:** Mass

**X-Axis:** Measurements
How to Make a Histogram
How to Make a Histogram

[Diagram showing a histogram with measurements on the y-axis and mass [kg] on the x-axis. The diagram includes data points and a theoretical model line.]
We’ll pick up from there next time.

Have a good rest of the week and see you at labs on Thursday!