Physics 2660: Fundamentals of Scientific Computing

Lecture 7
Instructor: Prof. Chris Neu (chris.neu@virginia.edu)
Reminder

• Welcome back from Spring Break!

• HW05 due Saturday 18 Mar electronically by 11:59pm
• Grading of HW03, HW04 coming!
• Register for piazza!
  – Only 52 out of 70 so far
• You will need the second textbook, Lyons, in about 2 weeks.

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

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

• Mid-term exam will be ***Tuesday 21 March*** at 7pm in this room
  – One week from now!

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

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

• Last time:
  – Loops
    • Count-controlled loops: for()More on loops
    • Conditioned controlled loops: while(), do
  – Recursive functions
  – Scope of variables
  – Commenting code
  – Static variables and external functions

• Today:
  – Pointers
  – Pointers to Functions
  – Arrays in C
  – Passing arguments to main()
  – Review
Functions: Two Types

- Functions in C can be classified into two types:
  - **pass-by-value**: (what we have done so far)
    - the initial values of the arguments are not allowed to change globally
    - they can change interior to the function, but those changes are never carried out on the global variable
    - Copies of the input vars are made but not used beyond the function
  - **pass-by-reference**: 
    - arguments can be values or memory locations
    - output can be “returned” through the parameter list
    - allows multiple results to be accessible to rest of program
Functions in C can be classified into two types:

- **pass-by-value**: *(what we have done so far)*
  - the initial values of the arguments are not allowed to change globally
  - they can change internally to the function
  - never change the global variable
  - Copies of the input vars are made but not used beyond the function

- **pass-by-reference**:
  - arguments can be values or memory locations
  - output can be “returned” through the parameter list
  - allows multiple results to be accessible to rest of program

Pass-by-reference functions requires a discussion of **pointers** to variables in memory....
Pointers
Pointers: What are they?

- Pointers are like variables that provide direct access to the memory location for a specific variable.
- One can then manipulate the contents of the memory location – and hence manipulate the contents of the variable from any location in the code.
  - allows global access and manipulation.
- Pointers give them user more power and more freedom!
- However, paraphrased from many (Voltaire, Churchill, FDR, Uncle Ben in Spiderman)

"With great power and freedom comes great responsibility."
Recall: Variable Storage

Another Look at Variable Storage:

- The values of variables are stored in memory.
- Different types of variable take up different amounts of memory.
- The values of variables are stored as ones and zeros (bits) arranged in groups of eight (bytes).

How does the computer find a particular variable's data in memory? Each variable has an address, expressed as the number of bytes from some starting location.
As we've seen before, you can use the “sizeof” operator to find out how much storage space a variable uses in memory.

You can use the “&” (“address of”) operator to see the memory address of a particular variable's storage area:

```c
#include <stdio.h>
int main () {
    int number = 123456;
    printf("Size of memory storage: \%d\n", sizeof(number));
    printf("Memory address of storage: \%p\n", &number);
}
```

The “%p” format descriptor can be used for printing memory addresses.

Size of memory storage: 4
Memory address of storage: 0xbfd2ab5c
```c
#include <stdio.h>
int test(int n);
int main () {
    int number = 123456;
    printf("Memory address of main storage: \%p\n", &number);
    test(number);
}
int test(int number) {
    printf("Memory address of test storage: \%p\n", &number);
}
```

Memory address of main storage: 0xbfe82bbc
Memory address of test storage: 0xbfe82ba0
Memory Addresses: Storage in Functions

Storage of Local Variables:

The variables inside a function, even those passed to function when we invoke it, are local to that function. A variable named “number” in the function “test” isn't the same as the variable named “number” in “main”.

```c
void test(int number)
{
    int number;
    ...
    ...
    ...
}
```

When the function is invoked, the value in `number` in “main” is copied into a new storage area for holding the “test” function's variable called `number`.
Pointers

A pointer is a special kind of variable that holds the memory address of another variable.

A pointer is defined by prefixing a variable name with an asterisk (*), the indirection operator:

```c
int main() {
    int number = 5;
    int *nptr;
    nptr = &number;
    return(0);
}
```

An int variable

A “pointer to int” variable

Use the “address of” (&) operator to get the address of `number` and store it in `nptr`. 
Storing Pointers in Memory

```c
int main() {
    double number = 3.1415;
    double *nptr;

    nptr = &number;

    return(0);
}
```

In this example, the “double” variable `number` stores the value 3.1415. The “pointer to double” variable `nptr` stores the address of `number`. 
You can get the data stored at a given memory location by using the "*" ("indirection") operator:

```c
int main() {
    double number = 3.1415;
    double *nptr;

    nptr = &number;

    printf("The value is %.1f\n", *nptr);

    return(0);
}
```

Since `nptr` is a "pointer to double", the result will be treated as "double" data.

The value is 3.1415
We can also use the indirection operator on the **left** side of an assignment statement, to **set** the value stored at a given memory address:

```c
int main() {
    double number = 3.1415;
    double *nptr;
    nptr = &number;
    *nptr = 6.02;
    printf("number is \%lf\n", number);
    return(0);
}
```

Here we change the value of `number` indirectly, by sticking a value into that variable's memory address.

**number is 6.02**
An Example: `scanf`

```c
int main () {
    int x, y;

    printf ("Enter x, y: ");
    scanf("%d %d", &x, &y);
}
```

`&x` and `&y` are the memory addresses of the variables `x` and `y`. These addresses are copied into variables inside `scanf`.

After `scanf` reads data from the keyboard, it sticks that data into the memory addresses given by `&x` and `&y`.

If `scanf` didn't know the addresses of these variables, it couldn't modify their contents.
You can use indirection in your own functions. Here's an example:

```c
int main() {
    float x = 2;
    float y = 5;
    float area;
    getarea(x, y, &area);
    printf("the area = %f units\n", area);
    return 0;
}
```

Here we pass `getarea` the **address** of the variable `area`.

```c
void getarea(float x, float y, float *aptr) {
    *aptr = x * y;
    return;
}
```

Here we tell our function to expect a **pointer** containing the address of a "float".

Deposit the calculated area into memory at the **address** of variable `area` in "main".
Advantages: Passing Pointers As Args To Functions

• Through the use of pointers passed to functions, we can manipulate an arbitrary number of variables, rather than just return one result.

```c
void c_and_a(float r, float *a, float *c) {
    *a = PI*r*r;
    *c = 2*PI*r;
}
```
Pointers: Be Careful!

**Pointer Errors: Null (zero) Pointers:**

If we accidentally leave off an ampersand when calling `scanf`, we'll usually get a **segmentation fault** error.

```c
int main () {
    double x;

    printf("Enter the value for x:");
    scanf("%lf", x);
}
```

The value in `x` is probably zero, so `scanf` interprets this to mean that it should stick the value of `x` into the memory address "0x00000000".

This is a low-lying part of memory that belongs to the operating system, and your program doesn't have permission to write there. That's what the "segmentation fault" error is telling you.
Pointers: Be Careful!

**Pointer Errors: Mis-casting:**

```c
int main () {
    int number = 4;
    double *nptr;
    nptr = &number;
    printf ("%lf\n", *nptr);
}
```

```c
int main () {
    int number = 4;
    double *nptr;
    nptr=(double *)&number;
    printf ("%lf\n", *nptr);
}
```

**error: cannot convert ‘int *’ to ‘double *’ in assignment**

0.00000000
Pointers to Variables: A Review

- Pointers allow direct access to memory locations where variables are stored in 1’s and 0’s
  - address via &
  - content via *

- Passing pointers into functions:
  - In function prototype
    - use pointer-to-values as argument(s)
  - In call to function
    - call with pointer-to-address as argument(s)
  - In function definition,
    - use pointer-to-values as argument(s) as done in prototype
    - use pointer-to-values in body of function

```c
#include <stdio.h>

void circle_stuff(double radius, double *a_ptr, double *c_ptr);

void main()
{
    double radius = 3.0;
    double area = 0.0;
    double circumference = 0.0;

    circle_stuff(radius, &area, &circumference);
    printf("area=%8.3f
circumference=%8.3f\n", area, circumference);
}

void circle_stuff(double radius, double *a_ptr, double *c_ptr){
    *a_ptr = 3.1416*radius*radius;
    *c_ptr = 2.0*3.1416*circumference;
}
```
Pointers to Functions and Trapezoidal Rule for Integration
How are Functions Stored in Memory?

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.
Pointers to Functions: Example

```c
#include <stdio.h>
#include "sqrtn.hpp"

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

int main(){
    double x;
    printf("enter a number \n");
    scanf("%lf", &x);
    print_func(sqrtn, x);
}

void print_func(double (*f) (double x), double val)
{
    printf("func(%lf) = %lf\n", val, f(val));
}
```

- This argument is a pointer to a function.
- "print_func" takes a function name as an argument.
- Any function that returns a double and takes a double argument can be plugged in here.
- Now "f" points to the function we named.
In a function definition, a function pointer appears like this:

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

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”.
Monte Carlo integration is one method one can use to integrate some complicated function, especially useful in multiple dimensions.

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 $A \leq x \leq B$.

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, $\Delta x$, of the subdivision.
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.
Pointers to Functions:
Example – Numerical Integration

We can estimate the area by summing the subdivisions.

\[
\int_{A}^{B} f(x) \approx \Delta x \left( \frac{f(A) + f(A + \Delta x)}{2} + \frac{f(A + \Delta x) + f(A + 2\Delta x)}{2} + \ldots + \frac{f(A + (n-1)\Delta x) + f(B)}{2} \right) \\
\int_{A}^{B} f(x) \approx \Delta x \left( \frac{f(A) + f(B)}{2} + \sum_{i=1}^{n-1} f(A + i\Delta x) \right)
\]
Implementing the Trapezoid Rule:

```c
#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));
}
```
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.
Until now, we've begun our programs like this:

```c
int main()
```

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

```c
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`. 
Program called “args”:

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

The first argument is always the program name.

Remember that this is a string, not a number.
The Keyboard Enters Strings Not Numbers!

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

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

Also available: `atol` (arg to long), `atoff` (arg to float). Feel free to complain about the lousy names.
Mid-term Exam
Information on Mid Term Exam

• Your mid-term is coming!
• Tuesday 21 March in this room at 7pm
  – You will have 60 min to finish
  – Bring a number 2 pencil
  – Regular lecture and labs will of course meet that week

• Format:
  – multiple choice
  – matching
  – short answer

• Let’s review some of the things we have learned so far
• Some examples follow
Review of C:
What We Have Learned So Far
Variables

- Have *types*.
- Commonly-used types are:
  - `int`, for integers (1, 2, 3, ...),
  - `double`, for floating-point numbers (3.14, 6.02, 9.8),
  - `char`, for characters and character strings ('a', 'Hello World!')
- Are usually defined at *top* of functions.
- May be *pointers* that hold the address of another variable.
• Use `printf` to write to the screen.
  • Requires `#include <stdio.h>`
  • Uses format descriptors for variables.
  • Commonly-used format descriptors are:
    • `%d`, for int
    • `%lf`, for double
    • `%s`, for character strings
  • You can use “\n” to insert carriage returns.

• Use `scanf` to read from the keyboard.
  • Use an `&` in front of variables, except for character strings.
• Referred to by “file pointers”: FILE *infile;

• Open files with fopen:
  • Pick “r” for reading, “w” for writing.

• Write to file with fprintf.

• Read from file with fscanf.

• Close file with fclose.
Loops

- **for loops:**
  - Repeat something a fixed number of times.
  - Should use integers for counters.

- **while loops:**
  - Keep repeating as long as a given statement is true.
  - Do the test before starting, so number of repeats may be zero.

- **do loops:**
  - Keep repeating as long as a given statement is true.
  - Do the test after the first pass, so number of repeats is always at least one.
• if statements:
  • Only do something if a given statement is true.

• if/else statements:
  • Pick between two options, depending on whether a given statement is true or false.

• if/else if/else statements:
  • Choose based on the first true expression you find.

• switch statements:
  • Choose between several options, based on the value of an integer or a character.
Functions

- Always have at least one, called “main”.
- Functions return one value to the caller.
- Functions take arguments.
- Functions have a syntax defined by a prototype statement.
Pointers

- Pointers allow direct access to memory locations where variables are stored in 1’s and 0’s
  - address via 
  - content via *

- Passing pointers to functions:
  - In function prototype
    - use pointer-to-values as argument(s)
  - In call to function
    - call with pointer-to-address as argument(s)
  - In function definition,
    - use pointer-to-values as argument(s) as done in prototype
    - use pointer-to-values in body of function

```c
#include <stdio.h>

void circle_stuff(double radius, double *a_ptr, double *c_ptr);

void main(){
  double radius = 3.0;
  double area = 0.0;
  double circumference = 0.0;

  circle_stuff(radius, &area, &circumference);
  printf("area=%8.3f
    circumference=%8.3f\n", area, circumference);
}

void circle_stuff(double radius, double *a_ptr, double *c_ptr){
  *a_ptr = 3.1416*radius*radius;
  *c_ptr = 2.0*3.1416*circumference;
}
```
Questions About the Basics

• What is an operating system?

• What are the main hardware parts of a computer?

• What is a compiler?

• Etc.
### Questions About the Linux Shell

<table>
<thead>
<tr>
<th>Question</th>
<th>Command(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write out or choose the GNU/Linux shell command that does the following:</td>
<td></td>
</tr>
<tr>
<td>1) lists files in your directory</td>
<td><code>ls</code></td>
</tr>
<tr>
<td>2) lists files in your directory, with sizes shown</td>
<td><code>ls -l</code></td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td><code>ls -al</code></td>
</tr>
<tr>
<td>3) renames the file <em>my.dat</em> to <em>your.dat</em></td>
<td><code>mv my.dat your.dat</code></td>
</tr>
</tbody>
</table>
| 4) places file *a.txt* in a subdirectory called *sub*                   | a) `mv a1.txt a2.txt`  
|                                                                          | b) `mv a.txt`       |
|                                                                          | c) `rename a.txt sub/a.txt` | 
|                                                                          | d) `mv a.txt sub`  |
## Questions About the C Language

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Choose the best answer.</strong></td>
<td></td>
</tr>
</tbody>
</table>
| 1) Define an integer variable, \( i \): | a) \( \text{int } i; \)  
  b) \( \text{int } &i; \)  
  c) \( \text{integer } i; \)  
  d) \( \text{int } *i; \)  |
| 2) Define a floating point variable with value=3.14 whose value cannot be changed: | a) \#define PI=3.14;  
  b) \( \text{const double } PI=3.14; \)  
  c) \#define PI 3.14  
  d) \( \text{static float } PI=3.14; \)  |
| 3) The statement to read a double value into the variable named \( \text{discount} \) is: | a) scanf("%lf", &discount);  
  b) scanf("%d", &discount);  
  c) scanf(discount);  
  d) scanf("%lf", &discount);  |
| 4) Print the double variable \( q \) in scientific or floating point notation, whichever is more compact: | a) printf("%ef", q);  
  b) printf("%e", q);  
  c) printf("%g", q);  
  d) printf("%lf", q);  |
Questions About the C Language

<table>
<thead>
<tr>
<th>Choose the best answer.</th>
<th>1) Using the file pointer, <code>input_file</code>, open the file <code>results.dat</code> for read mode.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a) <code>openf(&quot;results.dat&quot;,&quot;r&quot;,input_file);</code></td>
</tr>
<tr>
<td></td>
<td>b) <code>open(input_file,&quot;results.dat&quot;,&quot;r&quot;);</code></td>
</tr>
<tr>
<td></td>
<td>c) <code>fopen(input_file, &quot;results.dat&quot;, &quot;r&quot; );</code></td>
</tr>
<tr>
<td></td>
<td>d) <code>input_file = fopen(&quot;results.dat&quot;, &quot;r&quot; );</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2) Which code snippet reads an integer from the program's command line?</th>
<th>int main(int argc, char *argv[]){</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) <code>int i = argv[1];</code></td>
<td>...</td>
</tr>
<tr>
<td>b) <code>int i = atoi(argv[1]);</code></td>
<td>}</td>
</tr>
<tr>
<td>c) <code>int i = atoi(argv[0]);</code></td>
<td></td>
</tr>
<tr>
<td>d) <code>int i = (int)argv[1]);</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3) Function pointer type that can point to <code>sqrt()</code> function in the C math library:</th>
<th>a) <code>double (*f) (double x)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>b) <code>double *f(double x)</code></td>
<td></td>
</tr>
<tr>
<td>c) <code>double &amp;f(double x)</code></td>
<td></td>
</tr>
<tr>
<td>d) <code>double *f(double x)</code></td>
<td></td>
</tr>
</tbody>
</table>
### Questions About the C Language

<table>
<thead>
<tr>
<th>Choose the best answer.</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) What is the only function all C programs must contain?</td>
<td>a) <code>start()</code></td>
</tr>
<tr>
<td></td>
<td>b) <code>system()</code></td>
</tr>
<tr>
<td></td>
<td>c) <code>main()</code></td>
</tr>
<tr>
<td></td>
<td>d) <code>program()</code></td>
</tr>
<tr>
<td>2) Which of the following is the correct operator to compare two numerical variables?</td>
<td>a) <code>:=</code></td>
</tr>
<tr>
<td></td>
<td>b) <code>=</code></td>
</tr>
<tr>
<td></td>
<td>c) <code>equal</code></td>
</tr>
<tr>
<td></td>
<td>d) <code>==</code></td>
</tr>
<tr>
<td>3) How many times is a “do while” loop guaranteed to loop?</td>
<td>a) <code>0</code></td>
</tr>
<tr>
<td></td>
<td>b) <code>Infinitely</code></td>
</tr>
<tr>
<td></td>
<td>c) <code>1</code></td>
</tr>
<tr>
<td></td>
<td>d) <code>Variable</code></td>
</tr>
<tr>
<td>4) Evaluate:</td>
<td>a) <code>True</code></td>
</tr>
<tr>
<td><img src="expression" alt="Expression" /></td>
<td>b) <code>False</code></td>
</tr>
<tr>
<td></td>
<td>c) <code>Unevaluatable</code></td>
</tr>
</tbody>
</table>

```c
!(1 && !(0 || 1)).
```
1) Given the variables:

```c
int sum;
int maxint;
```

Write a statement that tests to see if `sum` is equal to 1000 and also that `maxint` is between 10 and 50, inclusive.

If the condition is satisfied, print the text "OK".

```c
if ( sum == 1000 &&
    maxint >= 10 &&
    maxint <= 50 ) {
    printf ("OK\n");
}

or

if ( sum == 1000 &&
    maxint >= 10 &&
    maxint <= 50 )
    printf ("OK\n");
```
We’ll pick up from here next time.

See you then.