Physics 2660: Fundamentals of Scientific Computing

Lecture 3
Instructor: Prof. Chris Neu (chris.neu@virginia.edu)
Reminders

• Weekly readings and homeworks will be posted on the class wiki site in general w/o any official notification to you – please check the links frequently

• Two textbooks – do you have them both?

• Enroll in piazza!

• Prepare for labs!
  – Make sure the readings are complete
  – Review the class notes from lecture

• Do your homework
  – None assigned this week – catching up a bit now

• In the coming 2-3 weeks do not fall behind – this is a crucial time
News

• Prof. Neu’s office hours this week are canceled
  – Join me instead in attending a special colloquium here in room 204 at 3:30pm – **Renee Horton**

  ![Special Colloquium](image)

  **Renee Horton**
  NASA
  "The Art of Metal Joining and How It’s Used NASA Michoud Assembly Facility"

  ABSTRACT:

  Metal joining is a controlled process used to fuse metals. There are several techniques of metal joining of which friction stir welding is one of the more basic forms. Friction stir welding is an innovative weld process that continues to grow in use, in the commercial, defense, and space sectors. It produces high quality and high strength welds in aluminum alloys. The process consists of a rotating weld pin tool that plasticizes material through friction. The plasticized material is welded by applying a high weld forge force through the weld pin tool against the material during pin tool rotation. Self-reacting friction stir welding (SR-FSW) is one variation of the FSW process developed at the National Aeronautics and Space Administration (NASA) for use in the fabrication of propellant tanks and other areas used on the Space Launch System (SLS)

  NASA’s SLS is an advanced, heavy-lift launch vehicle which will provide an entirely new capability for science and human exploration beyond Earth’s orbit. The SLS will give the nation a safe, affordable and sustainable means of reaching beyond our current limits and open new doors of discovery from the unique vantage point of space.

  – Examples: [https://www.youtube.com/watch?v=niVsJPFlg1Y](https://www.youtube.com/watch?v=niVsJPFlg1Y)  
    [https://www.youtube.com/watch?v=03FioPzhEBM](https://www.youtube.com/watch?v=03FioPzhEBM)

• TA office hours proceed as usual, in Room 022-C
Announcement: Repeat from Last Time

- From now on, compile your code using the `–Wall ("minus W all")` flag:

```
g++ -Wall -o example example.cpp
```

- The `–Wall` flag tells the compiler to report to you ALL warnings the compiler wants to issue (by default these are suppressed).

- **Types of warnings:**
  - use of un-initialized variables
  - definition of unused variables
  - syntax problems

- **Warnings are not good!!**

- **We will start penalizing in homework grading for compiler warnings**
Outline

• From last time:
  – Navigating the Linux filesystem
  – Basic Linux commands
  – Intro to C

• Today:
  – C program structure
  – Intro to a basic C program
  – Defining simple variables and doing arithmetic
  – Formatted input/output via printf() and scanf()
  – How variables’ values are stored on a computer
  – More on formatted I/O
  – Operators
The Structure of a C Program:

A C or C++ program is built up out of units called “functions”. Each program must have at least one function, called “main”. The large-scale structure of a C program consists of a statement like the one below, defining what this “main” function should do.

In the example below, the “main” function doesn't have any arguments, but we'll see later that it can do so.

```c
int main() {
    The body of the program goes here.
    return(0);
}
```

The body of the function is delimited by braces. The function returns a value. In the case of “main”, the operating system uses this value to determine whether the program completed successfully. By convention, non-zero means an error occurred.
The C Language: An Example

A Simple C Program:

```c
#include <stdio.h>
#include <math.h>
int main() {
    int a = 2;
    int b = 2;
    int c;
    double d;
    printf ("Hello, world!\n");
    c = a*b;
    printf ("The value of c is %d\n", c);
    d = sqrt(a);
    printf("The square root of %d is %f\n",a,d);
    return(0);
}
```

- Commonly-used statements can be stored in external “header” files and re-used in other programs.
- Each part of your program is a function. At the highest level is a special function named “main”.
- Every statement must end with a semicolon.
Defining and Assigning Variables

Variables, Assignment and Types:

C and C++ are strongly typed languages. This means that every variable you use must be defined to hold a specific type of data.

Some frequently used numeric types are shown below.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>An integer</td>
</tr>
<tr>
<td>float</td>
<td>A real (floating-point) number</td>
</tr>
<tr>
<td>double</td>
<td>A “double precision” floating-point number</td>
</tr>
</tbody>
</table>

These lines define the types of these variables.

```c
int a;
int b = 25;
float c;
float d;
double e;
```

These lines assign values to some variables.

```c
a = 50;
c = 3.14;
d = c;
```

- Variables can optionally be assigned a value when they're defined.
- Assigning the value of “c” to the variable “d”.


## Arithmetic Operators:

C and C++ include the following arithmetic operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>a+b</td>
</tr>
<tr>
<td>-</td>
<td>a-b</td>
</tr>
<tr>
<td>*</td>
<td>a*b</td>
</tr>
<tr>
<td>/</td>
<td>a/b</td>
</tr>
<tr>
<td>%</td>
<td>a%b</td>
</tr>
</tbody>
</table>

That's all! Other mathematical operations (like square roots and trigonometric functions) are available as functions that live in the standard math library, *libm*. We'll use these extensively, soon.

```c
a = 50;
b = 2*a - 10;
c = 3*b/(a-1);
```

Note that parentheses can be used in the expected way.
Using Functions:

The core functionality of the C language can be extended by adding functions. Many of these are available in the Standard C Library, or other libraries. You'll also be writing your own functions.

Here's how you could use the function “printf” (which prints out text) in a program:

```c
#include <stdio.h>
int main() {
    printf("Hello World\n");
    return (0);
}
```

“\n” is the special character “newline” (also known as “linefeed”).

The header file we included, stdio.h, defines the calling syntax of printf. The compiler checks that the function is used properly and stops with an error message if there is a usage mistake.

(Note that the code above is a complete C program.)
Preprocessor Directives

Preprocessor directives:

By default, most modern C “compilers” actually do more than just translate source code into machine code. For example, before compiling your code, they typically run a preprocessor program. The preprocessor program scans the code, looking for special instructions.

```
#include <stdio.h>
#include <math.h>
define PI 3.14
```

Some preprocessor directives.

Preprocessor directives begin with a pound sign (#). These statements are not part of the C/C++ language, per se. They form a small separate language of their own. We'll introduce the parts of it you need as we go along.

```
#include <stdio.h>  
directs the preprocessor to include the file called stdio.h into the text of your program. This is a header file. These are
```
used to define the meaning of statements you use that are not
intrinsic part of the C/C++ language, standard functions from the C
library, functions from your own code base, etc...
Variable Definition in C
Variable Types in C

- All variables in C must be declared as a particular type

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>short</td>
<td>A “small” integer</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>A “medium” integer</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>A “large” integer</td>
</tr>
<tr>
<td></td>
<td>unsigned short</td>
<td>Positive-definite versions of the types above.</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
<td></td>
</tr>
<tr>
<td>Floating-point numbers</td>
<td>float</td>
<td>A real (floating-point) number</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>A “double precision” floating-point number</td>
</tr>
<tr>
<td></td>
<td>long double</td>
<td>Even higher precision.</td>
</tr>
<tr>
<td>Characters</td>
<td>char</td>
<td>A character of text.</td>
</tr>
</tbody>
</table>

- Note: a “string” of characters in C is an array of single chars
Defining Variables in C

- In C, variables should be defined at the beginning of the function in which they are being used:

```c
#include <stdio.h>
int main() { // start of main function
    int an_int = 100; // start of variable declarations
    float a_float = 0.1;
    a_float = a_float * 100.5; // start of program statements
    an_int = an_int / 7;
    printf("%d %f\n", an_int, a_float);
    return(0);
}
```

Note format specifiers for integer and floating-point numbers. We'll talk more about this later.
Defining Variables: Defaults?

• What if I do not initialize (give an initial value to) a defined variable?

  What does the following code print?

  ```c
  #include <stdio.h>
  int main() {
    float a;
    a = a * 100.0;
    printf("%f\n", a);
    return(0);
  }
  ```

• In C, this could be 0.0, could be anything, WILL BE GARBAGE.
  – C does not initialize variables for you!
  – It reuses memory space when defining a new variable though – and this memory space is not “flushed” before the assignment: stale contents

• It is best to always initialize your defined variables to something appropriate
Casting Variables

- **Casting** is the conversion from one variable type to another
  - can either increase or decrease the precision of a stored value

Example:

```plaintext
float a = 101.1;
int i = 0;
i = a;
a = i + 1;
```

- Downward cast, setting `i` equal to 101 (lower precision). A cast to int always truncates!
- Upward cast, setting `a` to a value of 102.0 (higher precision).

These are known as **implicit casts** – the compiler knows how to increase/decrease precision for you!

Implicit casts are generally not a good idea – anything done for you by the compiler is generally not ideal.
Avoid Implicit Casts

Upward casting usually proceeds without complaint, but automatic or implicit downward, resolution-reducing casts, can generate a compiler warning:

```cpp
10: float a=101.1;
11: int i = 0;
12: i = a; // Implicit downward cast, giving i a value of 101 (lower precision).
```

```
~/demo> g++ cast.cpp
cast.cpp: In function `int main()':
cast.cpp:12: warning: assignment to `int' from `float'
```

Try to **avoid implicit casts**. Good programming style uses **explicit casts**, where data are consciously managed by the programmer.
Explicit Casts

- The preferred way to do this is the following:

```c
10: float a=101.1;
11: int i = 0;
12: i = (int) a;
```

Virtues:
- no warning messages
- you are more aware of the precision of the stored variables / values
Variables vs. Preprocessor Definitions

- You can define oft-used constants (think the speed of light, etc.) using preprocessor definitions
- Use the `#define` command

Comparison:

```c
#include <stdio.h>
#define RADIUS_OF_EARTH 6378.1 //km
#define PI 3.14159

int main() {  
    printf(  
        "The circumference of Earth = %f\n",  
        2.0*PI*RADIUS_OF_EARTH);  
    return(0);  
}
```

```c
#include <stdio.h>

int main(){
    double radiusEarth = 6378.1;
    double pi = 3.14159;
    printf(  
        "The circumference of the Earth = %f\n",  
        2.0*pi*radiusEarth);  
    return(0);  
}
```

With preprocessor definitions, the compiler sees all instances of `PI` and replaces with `3.14159`, etc…
Preprocessor Definitions vs Constants

• The virtue of using preprocessor definitions is speed – can speed up your program at run time, since fewer vars are stored in memory, meaning fewer accesses.

• But there are dangers:
  – universal replace might go poorly: What if you had a function called SAPInterestCalc()?
  – other preprocessor definitions are hidden from you – hard to see if yours clobbers another say in a header file

• Instead, one can use const:

```c
#include <stdio.h>

// Define constant values. Compiler will protect these:
const float RADIUS_OF_EARTH = 6378.1; // in km
const float PI = 3.14159;

int main() {
    printf("The circumference of Earth = %f\n",
        2.0*PI*RADIUS_OF_EARTH);
    return 0;
}
```

Using const is the best coding practice – compiler won’t let it be altered!
Storing Variables
Bits and Bytes

• How is data stored on a computer?
  – Data is stored in **bits** – think of them as values that can have either the value 0 or 1, or switches
  – One value is called a bit
  – bits are grouped together in groups of 8, called a **byte**
  – 8 bits are convenient for binary arithmetic – using the binary number system (1s and 0s only)

  – Examples: subscripts reflect base system
    • $1_d = 1_b = 1^*2_0$
    • $5_d = 101_b = 1^*2^2 + 0^*2^1 + 1^*2^0$
    • $12_d = 1100_b$
    • $21_d = 10101_b$
    • $182_d = 10110110_b$
Familiar Storage Sizes

- Byte: $10^0$
- Kilobyte: $10^3$
- Megabyte: $10^6$
- Gigabyte: $10^9$
- Terabyte: $10^{12}$
- Petabyte: $10^{15}$
Storing Variables

- Recall last time we introduced the most-common C variable types:
  - int
  - float and double
  - char

- When you run your program, an area in your computer’s memory is set up to store the value for each of the variables you define.

- Different variable types require a different amount of space.

- Hence, one must be careful to match variable types with the actual use one envisions … cannot put a decimal number in an int, for instance

```c
int main()
{
  double velocity = 0.0;
  float x = 0.0;
  int number = 0;
  char a = ‘a’;
  float y = 0.0;
  ...
}
```
• For an integer we have up to 4 bytes = 32 bits of storage

• So the variable “num” defined as this:

```
int num = 90937759;
```

is stored this way:

```
0 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
```

• Comments:
  – One bit is reserved for the sign +/- (leading bit):
    • leading bit = 1
    • leading bit = 0
  – So max value we can store is \(2^{31} - 1 = 2,147,483,647\)

Implication:
If your OS uses only 32-bit words for memory addresses, the max memory address is 4,294,967,296… meaning up to 4.3 GB of memory can be handled…

**hence 32-bit systems cannot have much more than 4 GB of memory.**
A variable declaration determines how its data are physically stored in memory.

In general the details of this storage differ from machine type to machine type, OS to OS, and programming language to programming language.

All data are ultimately stored as binary patterns, but the format differs depending on the variable's type.

Here's how one compiler, on one computer, stores the value “4” when it's an int, float or char:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>i = 4;</td>
<td>000000100 000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 bytes</td>
</tr>
<tr>
<td>float</td>
<td>f = 4;</td>
<td>000000000 000000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 bytes</td>
</tr>
<tr>
<td>char</td>
<td>c = '4';</td>
<td>00110100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 byte</td>
</tr>
</tbody>
</table>
Variables: How to know their size

The “sizeof” statement can be used to find out the number of bytes used by a variable or a data type.

<table>
<thead>
<tr>
<th></th>
<th>returns</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof(int)</td>
<td>4</td>
<td>4 bytes used to store an integer</td>
</tr>
<tr>
<td>sizeof(double)</td>
<td>8</td>
<td>8 bytes used to store a double</td>
</tr>
<tr>
<td>sizeof(char)</td>
<td>1</td>
<td>1 byte used to store a char</td>
</tr>
<tr>
<td>sizeof(5/2)</td>
<td>4</td>
<td>It's an integer</td>
</tr>
<tr>
<td>sizeof(5/2.0)</td>
<td>8</td>
<td>It's a double</td>
</tr>
</tbody>
</table>

In general, you'll get different results for the same data type on different computers. The sizes vary depending on operating system, compiler and computer architecture.

Note the last two lines: compound types. 5/2 is rightfully evaluated as a decimal number but has two int’s as args – hence it is assigned the size of an integer. So don’t try to divide integers and expect the right answer – unless you define things properly!

And the last one – the size of the result is given by the highest precision argument.
Using `sizeof()`

```c
#include <stdio.h>
int main()
{
    float b;
    // return size of a variable or data type in bytes
    printf("sizeof(int) = %d bytes\n",sizeof(int));
    printf("sizeof(b) = %d bytes\n",sizeof(b));
    printf("sizeof(5/2.0) = %d bytes\n",sizeof(5/2.0));
    printf("sizeof((int)(5/2.0)) = %d bytes\n",sizeof((int)(5/2.0)));
    return(0);
}
```

- `sizeof(int) = 4 bytes`
- `sizeof(b) = 4 bytes`
- `sizeof(5/2.0) = 8 bytes`
- `sizeof((int)(5/2.0)) = 4 bytes`
Formatted Input / Output

I/O
I/O Format Specifications

- Recall from lab00, you used `printf()` to print values to screen
- Similar function `scanf()` is used to read in values from the user:

```
Here are some more examples:

printf("%d\n", im_an_int);  // print an integer
scanf("%f\n", &a_float);    // read a float
```

- `%` is the control key indicating some I/O formatted data
- followed by an I/O specifier:

```
| Some common format specifiers: |
|-------------------------------|--------------------------------|
| i,d,ld,li                     | Integer data or long integer data. |
| f,lf                         | Floating-point number in decimal notation ("float" or "double"). |
| e,E                          | Floating-point numbers in Scientific Notation, like "6.02e+23". You can choose upper or lower case by picking "e" or "E". |
| g,G                          | Floating-point numbers, using either Scientific Notation or regular notation, whichever is shorter. |
| c,s                          | Single characters, or strings of characters. |
```

"i" and "d" are similar but very different. Use "d" for integers until we cover in more detail.
I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

```c
printf("%f", 5/2)
printf("%d", 5/2)
printf("%f", (float)(5/2))
printf("%f", 5/2.0)
printf("%d", 5/2.0)
printf("%d", (int)(5/2.0))
```

Similar care must be taken with scanf statements.
I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

```
printf("%f",5/2) → 0.000000
printf("%d",5/2)          
printf("%f",(float)(5/2)) 
printf("%f",5/2.0)        
printf("%d",5/2.0)        
printf("%d",(int)(5/2.0)) 
```

OOPS (integer data!)

Similar care must be taken with scanf statements.
I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

\begin{verbatim}
printf("%f",5/2) \rightarrow 0.000000       OOPS  (integer data!)
printf("%d",5/2) \rightarrow 2           OK
printf("%f",(float)(5/2))                
printf("%f",5/2.0)                      
printf("%d",5/2.0)                      
printf("%d",(int)(5/2.0))               
\end{verbatim}

Similar care must be taken with scanf statements.
I/O format specifiers are *important*. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or `printf` will *misinterpret* the data in translating it for output.

```c
printf("%f", 5/2) → 0.000000
printf("%d", 5/2) → 2
printf("%f", (float)(5/2)) → 2.000000
printf("%f", 5.2)  
printf("%d", 5.2)  
printf("%d", (int)(5/2.0))
```

OOPS (integer data!)
OK
OK

Similar care must be taken with `scanf` statements.
I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

```c
printf("%f",5/2) → 0.000000 OOPS (integer data!)  
printf("%d",5/2) → 2 OK  
printf("%f",(float)(5/2)) → 2.000000 OK  
printf("%f",5/2.0) → 2.500000 OK  
printf("%d",5/2.0)  
printf("%d",(int)(5/2.0))  
```

Similar care must be taken with scanf statements.
I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;%f&quot;,5/2)</code> → 0.000000</td>
<td></td>
<td>OOPS (integer data!)</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;,5/2)</code> → 2</td>
<td></td>
<td>OK</td>
</tr>
<tr>
<td><code>printf(&quot;%f&quot;,(float)(5/2))</code> → 2.000000</td>
<td></td>
<td>OK</td>
</tr>
<tr>
<td><code>printf(&quot;%f&quot;,5/2.0)</code> → 2.500000</td>
<td></td>
<td>OK</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;,5/2.0)</code> → 0</td>
<td></td>
<td>OOPS (double float data!)</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;,(int)(5/2.0))</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Similar care must be taken with scanf statements.
Format Mismatches

I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;%f&quot;, 5/2)</code> → 0.000000</td>
<td>OOPS (integer data!)</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;, 5/2)</code> → 2</td>
<td>OK</td>
</tr>
<tr>
<td><code>printf(&quot;%f&quot;, (float)(5/2))</code> → 2.000000</td>
<td>OK</td>
</tr>
<tr>
<td><code>printf(&quot;%f&quot;, 5/2.0)</code> → 2.500000</td>
<td>OK</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;, 5/2.0)</code> → 0</td>
<td>OOPS (double float data!)</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;, (int)(5/2.0))</code> → 2</td>
<td>OK</td>
</tr>
</tbody>
</table>

Similar care must be taken with scanf statements.
Controlling the Appearance of Output: Display

In general, the structure of a format specifier is:

```
%[parameter][flags][width][.precision][length]type
```

All elements except “%” and the type are optional.

**Examples:**

```c
int ia=12, ib=13;
float fx = 123.456;
printf("%10d %10d\n", ia, ib);
printf("%.8f\n", fx);
printf("%-d %-d\n", ia, ib);
```

<table>
<thead>
<tr>
<th>Ints printed in 10 columns w/ spaces between.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float printed in 8 columns, 4 numbers after decimal.</td>
</tr>
<tr>
<td>Int printed left justified.</td>
</tr>
</tbody>
</table>

By default, data are right-justified. The flag “-” causes them to be left-justified.
Controlling the Appearance of Output: Escape Characters

Some sequences of characters beginning with a backslash have a special meaning when used in printf's format string. These are sometimes called “escape sequences”.

Here's a list of commonly-used escape sequences. Among other things, these control the cursor on your monitor before/between/after characters are printed.

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Add a new line</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed (new page)</td>
</tr>
<tr>
<td>\b</td>
<td>Move back one character</td>
</tr>
<tr>
<td>\r</td>
<td>Go to beginning of line</td>
</tr>
<tr>
<td>\t</td>
<td>Go to next tab stop</td>
</tr>
<tr>
<td>\a</td>
<td>Ring the bell</td>
</tr>
<tr>
<td>\</td>
<td>Print the character \</td>
</tr>
<tr>
<td>&quot;</td>
<td>Print the character “</td>
</tr>
</tbody>
</table>

Some usage examples:

```c
printf("This is a line.\nThis is another line\n");
printf("This is a double-quote: \"\n");
```
output v. input

• Essentially all of the lessons about controlling output can be applied to controlling / designing input to your program

• printf() and scanf() are not very different
  – printf() – prints output to screen
  – scanf() – accepts input from the keyboard

• These are examples of interactive I/O
Interactive v. Passive I/O

• What about passive I/O?
  – Passive I/O = use of files

  – One can write output to a file:
    • fopen(), fclose(), and fprintf()

  – One can also accept input from a file:
    • fopen(), fclose(), and fscanf()

  – Some experience in this week’s lab03
We’ll pick up from here next time.

And of course do not forget lab on Thursday.

See you then!