Physics 2660: Fundamentals of Scientific Computing

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

• Weekly readings will be assigned and available through the class wiki home page

• HW02 was assigned
  – Due Thursday 8 February
    • Electronic submission: by 12:00 noon EST

• This week: Lab03
  – Will be posted on Thursday

• Office hours: all held in our computer lab, room 022-C of this bldg
  – Me: After lecture 3:30-4:30 every Tuesday
  – TAs:
    • Mondays: 3-5pm and 6-8pm
    • Wednesdays: 5-9pm
Variable Definition in C, Storage in Memory
Variable Types in C

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

<table>
<thead>
<tr>
<th>Integers</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>short</td>
<td>A “small” integer</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>A “medium” integer</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>A “large” integer</td>
</tr>
<tr>
<td>unsigned short</td>
<td>unsigned short</td>
<td>Positive-definite versions of the types above.</td>
</tr>
<tr>
<td>unsigned</td>
<td>unsigned</td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td>unsigned long</td>
<td></td>
</tr>
</tbody>
</table>

| Floating-point numbers    | float    | A real (floating-point) number |
| double                    | double   | A “double precision” floating-point number |
| long double               | long double | Even higher precision. |

| Characters                | char     | A character of text. |

- 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.
Storing Variables

- 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

```
int main(){
    double velocity = 0.0;
    float x = 0.0;
    int number = 0;
    char a = 'a';
    float y = 0.0;
    ...
}
```
Storage of an integer: Example

• For an integer we have up to 4 bytes = 32 bits of storage

• So the variable “num” defined as this:

  ```
  int num = 909337759;
  ```

  is stored this way:

  ![Binary representation of num](image)

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

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.
Storage of a non-integer: Example

Floating Point Number Representation - Conversion Example

https://www.youtube.com/watch?v=n-XozGu1viM
Variable Storage: Examples

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>Type</th>
<th>Value</th>
<th>Binary Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 4;</td>
<td>4 bytes</td>
<td>000000100 000000000 000000000 000000000</td>
</tr>
<tr>
<td>float f = 4;</td>
<td>4 bytes</td>
<td>000000000 000000000 010000000</td>
</tr>
<tr>
<td>char c = '4';</td>
<td>1 byte</td>
<td>00110100</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>Results for g++ on Galileo:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof(int)</td>
<td>returns 4</td>
<td>4 bytes used to store an integer</td>
</tr>
<tr>
<td>sizeof(double)</td>
<td>returns 8</td>
<td>8 bytes used to store a double</td>
</tr>
<tr>
<td>sizeof(char)</td>
<td>returns 1</td>
<td>1 byte used to store a char</td>
</tr>
<tr>
<td>sizeof(5/2)</td>
<td>returns 4</td>
<td>It's an integer</td>
</tr>
<tr>
<td>sizeof(5/2.0)</td>
<td>returns 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.
#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
Defining Variables: Defaults?

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

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

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

In C, this could be 0.0, could be anything, MEANING IT HAS THE CHANCE TO 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:

```
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).
```

```shell
~/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:

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

Explicit downward cast \((i = 101)\).

The syntax for implicit casts is “\((type)\)variable”. For example:

```
i = (int) a;
g = (float)i;
h = (double)a;
```

• Virtues:
  – no warning messages
  – you are more aware of the precision of the stored variables / values
Preprocessor Directives

```c
#include <stdio.h>
int main() {
    float a;
    a = a * 100.0;
    printf("%f\n", a);
    return(0);
}
```
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 intrinsically part of the C/C++ language, standard functions from the C library, functions from your own code base, etc...
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() {
    double radiusEarth = 6378.1;
    double pi = 3.14159;
    printf("The circumference of the Earth = \%f\n", 2.0*PI*RADIUS_OF_EARTH);
    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!
Formatted Input / Output
I/O Format Specifications

• Recall from lab00, you used `printf()` and `scanf()` to print/read values to/from screen/keyboard:

```c
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:

<table>
<thead>
<tr>
<th>Some common format specifiers:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i, d, ld, li</code></td>
<td>Integer data or long integer data.</td>
</tr>
<tr>
<td><code>f, lf</code></td>
<td>Floating-point number in decimal notation (&quot;float&quot; or &quot;double&quot;).</td>
</tr>
<tr>
<td><code>e, E</code></td>
<td>Floating-point numbers in Scientific Notation, like &quot;6.02e+23&quot;. You can choose upper or lower case by picking &quot;e&quot; or &quot;E&quot;.</td>
</tr>
<tr>
<td><code>g, G</code></td>
<td>Floating-point numbers, using either Scientific Notation or regular notation, whichever is shorter.</td>
</tr>
<tr>
<td><code>c, s</code></td>
<td>Single characters, or strings of characters.</td>
</tr>
</tbody>
</table>

Read Brooks (Physics library reserves!)
page 33
re: the difference between “i” and “d”
and leading zeroes
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.

\begin{verbatim}
printf("%f", 5/2) \rightarrow 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))
\end{verbatim}

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.

```
printf("%f", 5/2) \to 0.000000
printf("%d", 5/2) \to 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!)

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
printf("%d", 5/2) → 2
printf("%f", (float)(5/2)) → 2.000000
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.

```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.

```
printf("%f",5/2) → 0.000000
printf("%d",5/2) → 2
printf("%f",(float)(5/2)) → 2.000000
printf("%f",5/2.0) → 2.500000
printf("%d",5/2.0) → 0
printf("%d",(int)(5/2.0))
```

OOPS (integer data!)

OK

OK

OK

OOPS (double float 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.

```
printf("%f",5/2) → 0.000000
printf("%d",5/2) → 2
printf("%f",(float)(5/2)) → 2.000000
printf("%f",5/2.0) → 2.500000
printf("%d",5/2.0) → 0
printf("%d",(int)(5/2.0)) → 2
```

OOPS (integer data!)

OK

OK

OK

OOPS (double float data!)

OK

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("%8.4f\n", fx);
printf("%-d %-d\n", ia, ib);
```

- ints printed in 10 columns w/ spaces between.
- float printed in 8 columns, 4 numbers after decimal.
- int printed left justified.

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>&quot;</td>
<td>Print the character &quot;</td>
</tr>
<tr>
<td>\</td>
<td>Print the character \</td>
</tr>
</tbody>
</table>

Some usage examples:

```
printf("This is a line.\nThis is another line\n");
printf("This is a double-quote: \"\n");
```
Operators
Operator Types

• What kind of operations do we need to be able to do in a computer program?

• Here are some:
  – Arithmetic (eg., +, -, *, / )
  – Assignment (eg., = )
  – Increment / decrement (eg., ++, --) used in counters
  – Logical / Conditional (eg., && , || )
  – Bitwise operations…
  – Pointer operations…
C and C++ support the following arithmetic operators. Note that some operators are binary (operating on two numbers) and some are unary (operating on one number).

**Binary Operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>a+b Addition</td>
</tr>
<tr>
<td>-</td>
<td>a-b Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>a*b Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>a/b Division</td>
</tr>
<tr>
<td>%</td>
<td>a%b Remainder (modulo)</td>
</tr>
</tbody>
</table>

**Unary Operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-a Arithmetic inverse</td>
</tr>
</tbody>
</table>
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>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder (modulo)</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.
### Assignment Operators

The simplest assignment operator is “=”, which is used to set the value of a variable equal to some expression (e.g., “a = b”).

C also offers an array of additional assignment operators that combine assignment with the various arithmetic functions:

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

Be careful when you're typing these. It's easy to type “+=” instead of “+=”!
Increment/Decrement Operators

The unary operators `++` and `--` add or subtract 1 from the operand:

**Usage:**

- **increment**: `a++` or `++a` → `a = a + 1`
- **decrement**: `a--` or `--a` → `a = a - 1`

Notice that these operators can be used either before or after the variable. Their action differs slightly, depending on which of these is chosen. Here are some examples:

```c
int a = 1;
a++;  // Set a to a+1 before moving to the next line.
++a;  // Set a to a+1 immediately upon entering this line.

x = a++ * 2;  // Set x = a*2, then set a = a+1.
x = ++a * 2;  // Set a = a+1, then set x = a*2.
```

It's best to avoid statements like the last two unless you have a good reason to use them.
Logical / Conditional Operators

These operators test or combine logical expressions. The answer to a test is either true (not 0) or false (0). Any non-zero value is considered true.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equality</td>
<td>a==b</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality</td>
<td>a!=b</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>a&lt;b</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>a&gt;b</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less or equal</td>
<td>a&lt;=b</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater or equal</td>
<td>a&gt;=b</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT. Invert a test or true/false value</td>
<td>!a</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>(a==b) &amp;&amp; (c==d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
These operators test or combine logical expressions. The answer to a test is either true (not 0) or false (0). Any non-zero value is considered true.

<table>
<thead>
<tr>
<th>Logical Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equality</td>
<td>a==b</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality</td>
<td>a!=b</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>a&lt;b</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>a&gt;b</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>a&lt;=b</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td>a&gt;=b</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT. Invert a test or true/false value</td>
<td>!a</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>(a==b) &amp;&amp; (c==d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operator Precedence: Which Comes First?

When we see a statement like this, we know that we should first multiply “b*c” and then add “a”.

\[ x = a + b \times c \]

By convention, multiplication & division precede addition & subtraction.

Here's a table showing the order of precedence of some common operators in C and C++:

<table>
<thead>
<tr>
<th>First</th>
<th>a++, a--, type casts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second</td>
<td>a*b, a/b, a%b</td>
</tr>
<tr>
<td>Third</td>
<td>a+b, a-b</td>
</tr>
<tr>
<td>Fourth</td>
<td>a&amp;&amp;b, a</td>
</tr>
</tbody>
</table>

Operations of equal precedence are evaluated from left to right.

To clarify statements, you can use parentheses as needed or split statements into several steps. Make your intentions clear and you’ll be much happier.
Bad Coding Example: I

Don't even think about writing code like this! What does this even do?

```
a=1;
b=2;
c=3;
d=4;
f= ++a + c*d/a++ + b;
```

Here's a better way to do the same thing:

```
a += 1;
f = (a+b) + c*d/a;
a += 1;
```

```
a = 2
f = (2+2) + 3*4/2 = 10
a = 3
```
Bad Coding Example: II

Another Bad Coding Example:

```c
float a=10.0;
float b=5.0
float c;

c = 1 / 2 * a * b;
```

What value does `c` have?

Precedence rules dictate:
1) \(1/2 = 0\) by integer division
2) \(0*10.0 = 0.0\)
3) \(0.00*5.0 = 0.0\)

So, `c` is zero!

Some possible fixes:

- \(c = 1/2.0 * a * b;\)
- \(c = 0.5 * a * b;\)
- \(c = a * b / 2;\)
- etc...

Of course, these assume that you didn't really mean to type:

```c
   c = 1/(2*a*b);
```
We’ll pick up from here next time.

Don’t forget homework and lab on Thursday! See you then.