

Physics 2660

Lecture 3: C – Part 2

Today

- Program flow control statements (~25% of C's vocabulary):
 - Conditionals: if / else / switch / case
 - Loops: for / while / do
break / cont
- Some comments leading into statistics

Reminders:

- Don't forget to follow the **weekly reading assignment** posted on the class web page. Aside from this notice, there generally will not be weekly reminders.
- Preparation for labs:
 - Complete the weekly **reading** or any **Prelab** assignment
 - **Review notes** from Tuesday lecture
- The next **2-3 weeks** are very important! We will finish covering a large portion of the C language. Everything builds on previous work. Don't fall behind or it will be very difficult to catch up.

Announcement:

From now on, you should add a new flag (-Wall) to the g++ command when you compile your programs:

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

The -Wall flag will turn on a number of **compiler warnings**. It will warn you about the use of undefined variables, tell you if you've defined some variables but left them unused, catch syntax errors, *etc...*

Your programs will now be expected to compile without warnings.

Part 1: Conditional Statements



4

Up until now, we've mostly dealt with programs that follow a single path from start to finish. Now we'll look at ways to control the execution of our programs.

The Flow of Execution:

The main part of the program:

```
#include <stdio.h>
int main() {
    double x1 = 43.5;
    double x2 = 52.7;
    double d;
```

```
    d = distance(x1, x2);
```

```
    return(0);
}
```

It's useful to think of your program's execution as something that **flows from the top of "main"** to the bottom of "main", possibly taking lots of twists and turns along the way

A function called by "main":

```
double distance(
    double x1,
    double x2) {
    return(x2-x1);
}
```

In this example, the computer reads down through "main", briefly **jumps out** into the function "distance", then returns to "main" and finishes.

5

In this example, the computer follows just one path through the program. It's a twisty path, but there's still only one way to get from the top of the program to the bottom.

“if” Statement Syntax:

A simple “if” statement can be written in two different ways. Here’s the more general way to write one:

Syntax:

```
if (CONDITION) {  
    BLOCK of statements  
}
```

Must be true (!=0) or false (==0).

No semicolon.

Example:

```
if (a > 1) {  
    printf("Hello There!\n");  
    b = a * 2;  
    printf("b is: %d\n",b);  
}
```

Alternatively, if you only have one line in your block of statements, you can omit the curly brackets and write it like this:

Syntax:

```
if (CONDITION)  
    statement;
```

Example:

```
if (a > 1)  
    printf("Hello There!\n");
```

6

With “if” statements, we can make the computer execute different parts of our code, depending on the result of a test.

Relational and Logical 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.

Precedence

!	Logical NOT. Invert a test or true/false value	! a	1
<	Less than	a<b	2
>	Greater than	a>b	2
<=	Less or equal	a<=b	2
>=	Greater or equal	a>=b	2
==	Equality	a==b	3
!=	Inequality	a!=b	3
&&	Logical AND	(a==b) && (c==d)	4
	Logical OR	(a<=b) (c>b)	5

Remember:
Use parentheses to
prevent precedence
perplexity.

7

We saw these operators last week, along with C's other operators. Note that there are precedence rules that determine the order in which these operators will be executed. As we said last week, it's best not to rely too heavily on the precedence rules, since this can make your code confusing. To avoid mistakes, use parentheses to clarify things.

Using Return Values in Tests:

C library functions often return values useful as conditional tests.

For example:

```
// fopen returns a non-NULL pointer if successful
FILE* inFile;
inFile = fopen("grades.dat","r"); // open grades.dat

if (inFile==NULL) {
    // exit program if file not found
    printf("Error: grades.dat not found!!\n");
    return(1);
}
```

NULL is a preprocessor
macro, defined in stdio.h.

The "if" statement could alternatively be written like this:

```
if (!inFile) {
    printf("Error...
}
```


Returning Zero for Success:

It's common practice for functions returning an integer status value (instead of returning data) to return **zero** for **"Success"**, and non-zero to indicate an error. You'll often see code that takes advantage of this convention when making tests. For example:

```
if ( function(param1,param2) ) {  
    printf("Error !!\n");  
    return 1;  
}
```

If "function" returns a non-zero (i.e., "true") value, it means that something has gone wrong.

This isn't true for all functions in the Standard C Library. Check the documentation if you're not sure about a particular function.

Comparing Floating-Point Numbers with “==”:

The == operator compares two numbers and returns “true” if they are the same. This works fine for integers, but you **shouldn't** use it for **floating-point** numbers. This example shows why:

```
int main(){
    double a=12345678.;
    double loga2 = log(a*a);
    double b=sqrt(exp(loga2));
    printf("b=%20.10lf    a=%20.10lf\n",b,a);
    return(0);
}
```

$$b = \sqrt{e^{\ln(a^2)}} = \sqrt{a^2} = a$$

Output:

b=12345678.0000000224 a=12345678.0000000000

Clearly, “b” is not equal to “a” due to the **limited precision** of the calculations.

10

In the purple box, you see that “b” should be equal to “a”. Each of the long series of mathematical operations on “a” (square root, log, square) results in some roundoff error, since we can't keep infinitely many decimal places. By the time we've done them all, the result is slightly different from the original value. This means that it's difficult to compare floating point numbers. We can't just use the comparison operator “==”, since the two numbers aren't, strictly speaking, equal.

Comparing Floating-Point Data with “<” or “>”:

Instead of ==, use inequalities to see if the **difference** between the floating-point numbers is **less than some threshold** (chosen by you).

```
int main(){
    const double SMALL=1e-6;
    double a=12345678.;
    double loga2 = log(a*a);
    double b=sqrt(exp(loga2));

    if (fabs(a-b) < SMALL)
        printf("a=b\n");
    else
        printf("a!=b\n");
    return(0);
}
```

$$b = \sqrt{e^{\ln(a^2)}} = \sqrt{a^2} = a$$

The **fabs()** function returns the absolute value of a floating-point number. It is part of the Standard C Library.

Notice that we've also introduced an “**else**” statement, after our “**if**”. We'll talk about that next.

11

We can set the value of **SMALL** to be anything we want. We should choose it in a way that makes sense for the problem at hand. Do we care about differences of one part in a million? One part in a billion? The programmer gets to decide.

“if/else if/else” Statements:

Sometimes a simple “if” statement isn't enough. You may want to choose between **two or more** different blocks of code, based on some test.

In that case, you can add an “else” clause to your “if” statement.

The second block of code will only be executed when CONDITION is **false**.

```
if (CONDITION) {  
    BLOCK of statements  
} else {  
    BLOCK of statements  
}
```

For more complicated cases, you can add multiple “else if” clauses.

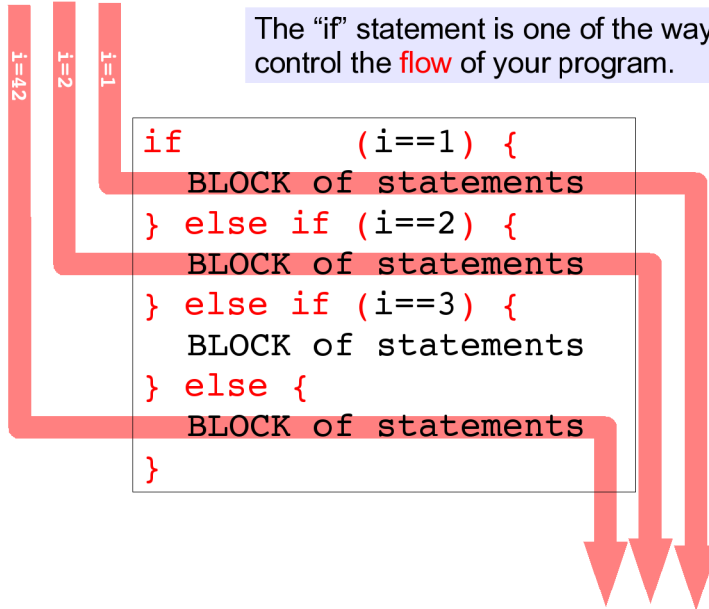
Only one of these blocks (the **first** one whose CONDITION is “true”) will be executed. The others will be ignored.

The **else** block, **if present**, will be executed if **none** of the CONDITIONS are met.

```
if (CONDITION) {  
    BLOCK of statements  
} else if (CONDITION) {  
    BLOCK of statements  
} else if (CONDITION) {  
    BLOCK of statements  
} else {  
    BLOCK of statements  
}
```

Program Flow with “if” Statements:

The “if” statement is one of the ways you can control the **flow** of your program.



Switch Statements:

It's common for programs to look at a **value** and use it to decide which **one of many alternative blocks** of code to execute.

For convenience, C provides another construct for this special case: the **switch** statement.

break means "jump out of the switch statement and continue with the rest of the program".

If there's a **default** case, it matches any value.

EXPRESSION must evaluate to an **int** or **char** value.

These are **int** or **char** too.

```
switch (EXPRESSION) {  
  ① case VALUE1: ←  
      BLOCK of statements  
      break;  
  ② case VALUE2:  
      BLOCK of statements  
      break;  
  ③ case VALUE3:  
      BLOCK of statements  
      break;  
      default:  
      BLOCK of statements  
}
```

14

We could do this using an "if/else if/else" statement, but that can get very long and complicated if we have many alternatives.

Controlling Flow with “break”:

Using **break** in a switch statement gives us a lot of control over the flow of our program.

```
switch (letter) {  
    case 'A':  
    case 'a':  
        printf("A is for Apple\n");  
        break;  
    case 'B':  
    case 'b':  
        printf("B is for Bear\n");  
        break;  
    default:  
        printf("Unknown letter\n");  
}
```

No **break**.

letter = 'A' or 'a'

If we **omit a break**, the program will continue to work its way through the switch statement, possibly matching other cases.

This lets us do the same thing for several different cases, working like an **OR** statement: **'A' || 'a'**.

```
case 'A':  
    printf("Capitol ");  
case 'a':  
    printf("A is for Apple\n");  
break;
```

No **break**.

We can also make one case a **superset** of another. Here, when letter='A', the program will print:

Capitol A is for Apple

To clarify the way “switch” statements work: the computer jumps to the first “case” statement that matches the value of “letter” (in this example), and then continues executing the program from there. It ignores any subsequent “case” statements. This is why the bottom example works.

If none of the “case” statements match, the program jumps to “default” if it exists, or skips the rest of the switch statement altogether if it doesn't.

Switch versus If:

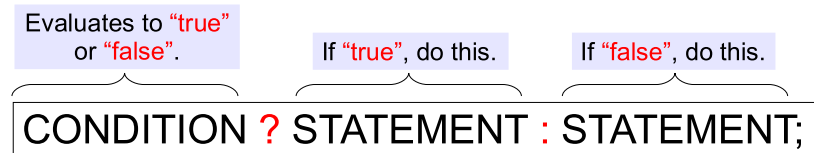
```
switch (letter) {  
    case 'A':  
    case 'a':  
    case '1':  
        printf("A\n");  
        break;  
    case 'B':  
    case 'b':  
    case '2':  
        printf("B\n");  
        break;  
}
```

Everything that can be accomplished with **switch** can also be done with a sufficiently complicated set of **if** statements. The two snippets shown here do the same thing, but notice how much more readable the switch statement is.

```
if (letter=='A' || letter=='a' || letter=='1'){  
    printf("A\n");  
} else if (letter=='B' || letter=='b' || letter=='2'){  
    printf("B\n");  
}
```


The ?: Operator:

As a shorter alternative to the regular if statement, C offers the special operator "?:". This is a **ternary** operator, meaning that it takes **three arguments**. The syntax is as follows:



Example:

```
(a==b) ? printf("it's true\n") : printf("it's false\n");
```

This is just equivalent to the if statement shown at the right:



```
if (a==b)
    printf("it's true\n");
else
    printf("it's false\n");
```

Assignment versus Comparison:

Be careful not to confuse the = (assignment) operator with the == (equality comparison) operator. This is one of the most common C typos.

The code below produces unexpected results. Why?

```
int a=0;
int b=1;
if (a=b)
    printf("they are equal\n");
else
    printf("they are not equal\n");
```

Prints: "they
are equal" !

The programmer
should have used
"a==b" !



```
int a=0;
int b=0;
if (a=b)
    printf("they are equal\n");
else
    printf("they are not equal\n");
```

Prints: "they
are not equal" !

The value returned by the assignment operation "a=b" is just the left-hand side of the assignment ("a", in this case) after the operation has completed (after "a" has been set equal to "b").

if (a=b) is equivalent to a=b;
if (a)

18

The trick is that operators (like "=") return values, just like functions. You could think of a statement like "a=b" as being equivalent to a function called "equals" that takes two arguments ("a" and "b"), sets the value of "a" to the value of "b", and returns the value of "a".

We'll talk about how to do things like this when we discuss pointers.

Using the Return Value of an Assignment:

Sometimes you may encounter code that **intentionally** uses an assignment statement as an **if** condition. See the following:

```
FILE* inFile;
if ( !inFile = fopen("grades.dat","r") ) {
    // exit program if file not found
    printf("Error: grades.dat not found!!\n");
    return(1);
}
```

If **fopen** fails, **inFile** will be **NULL**, a “false” value, and its logical inverse (**!inFile**) will be “true”.

It's better to avoid constructs like the one above, though. Instead, use something equivalent but more transparent, like this:

```
FILE* inFile;
inFile = fopen("grades.dat","r");
if ( !inFile ) {
```

Note that “g++ -Wall” will warn you when it sees you using **=** in a place where you might want to use **==**.

Boolean Data in C++:

In C++, there's a special data type that you can use for true/false data: "bool".

```
float x;  
bool validData = true;  
x = 3.14;  
validData = ( x > 1.0 && x < 10.0 );  
if ( validData )  
    printf("Valid data\n");  
else  
    printf("Invalid data\n");
```

true & false are reserved words in C++.

In C, we could do exactly the same thing using "int" instead of "bool". The bool type mainly serves to improve readability and clarity.

Part 2: Loops



Now we get to the thing that computers are best at:
doing the same thing over and over, many times.
This is what computers were invented for.

Types of Loops:

We use loops when we want to repeat a series of tasks several times. In most programming languages, there are two different types of control structures used to implement loops. Which one you use will depend on how your program knows that it's time to **stop repeating**:

- **Count-controlled Loops:**

We use these when we **know, beforehand**, how many times we want to repeat a series of tasks.

- **Condition-controlled Loops:**

These are used when we **don't know** how many repetitions will be needed, but we know that we want to stop when some well-defined thing happens.

The **for** Loop:

A “for” loop is a **count**-controlled loop. Its has the form shown below.

General form of a “for” loop:

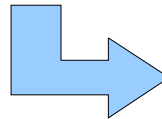
```
for ( initialize counter ; test condition ; counter update ) {  
    statement block;  
}
```

Example:

```
int i;  
for ( i = 0 ; i < 10 ; i++ ) {  
    printf("loop number %d\n", i);  
}
```

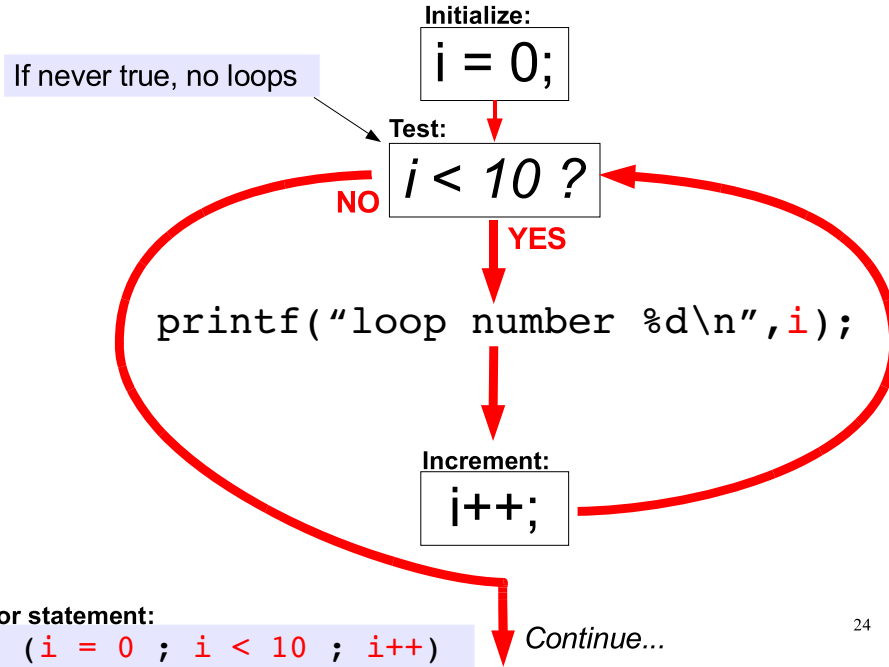
Output:

```
Loop number 0  
Loop number 1  
Loop number 2  
Loop number 3  
Loop number 4  
Loop number 5  
Loop number 6  
Loop number 7  
Loop number 8  
Loop number 923
```



The loop continues until the test condition is no longer true. See the following graphical representation.

How a **for** Loop Works:



Controlling **for** Loop Behavior:

```
for ( initialize counter ; test condition ; counter update ) {  
    statement block;  
}
```

The **for** statement is very flexible because:

- Any valid C expression can be used for initialization or update, and
- Any valid condition can be used for the test condition.

Here are some creative uses of the **for** statement:

```
for ( i=0 ; i < n ; i++ ) {
```

 Loop n times from $i=0$ to $i=n-1$

```
for ( i=0 ; i < m ; i+=2 ) {
```

 Loop $\sim m/2$ times $i = 0, 2, 4, \dots$

```
for ( i=100 ; i > 0 ; i-- ) {
```

 Loop 100 times, **decrementing** i

Compound statements are also allowed. (This may be a little **too** creative.)

```
for ( i=0, j=0; i<1000; i++, j=exp(i) )
```

Bad **for** Loop Usage:

```
int i;
for (i = 0 ; i < 10 ; i++) {
    printf("loop number %d\n", i);
    i = i+1;
}
```

How many iterations?

Note 1: It's extremely bad form to operate on the counter variable within the for loop. This leads to confusing code.

```
float a;
for (a = 0 ; a < 10 ; a+=0.5) {
    printf("counter= %f\n", a);
}
```

9.9999999 < 10.0

Do we loop 20 or 21 times?

Note 2: It's dangerous to use a float as your counter. Rounding errors may cause the loop run an unexpected number of times.

```
int i;
for (i = 0 ; i < 10 ; i++) {
    printf("loop number %d\n", i);
}
printf("completed %d loops", i);
```

In this case $i = 10$, a value not used in the loop.

Note 3: It's also bad practice to use a counter variable outside of the loop.



Good **for** Loop Usage:

```
int i;
for (i = 0 ; i < 10 ; i+=2) {
    printf("loop number %d\n", i);
}
```

Do all iterator math with the loop updater.

```
int i;
for (i = 0 ; i < 20 ; i++) {
    float a = i*0.5;
    printf("counter= %f\n", a);
}
```

Use integer iterators to avoid rounding errors with floats.

```
int const NLOOPS=10;
int i;
for (i = 0 ; i < NLOOPS ; i++) {
    printf("loop number %d\n", i);
}
printf("completed %d loops",
NLOOPS);
```

Use constants to define fixed NLOOPS, especially if you need to use the same value throughout your code.

Taking advantage of scoping rules in C++:

C++ allows us to limit the scope of variables. Variables can be defined so that they exist only within a loop. For example:

```
// define a variable i for use in a loop
for (int i = 0 ; i < 10 ; i++) {
    printf("loop number %d\n", i);
}

// i is no longer defined here

// define a new variable i for use in this loop
for (int i = 0 ; i < 1000 ; i+=100) {
    printf("i = %d\n", i);
}
```

This is a safer way to handle iterator variables, because prevents misuse of variables outside of their loops.


28

Notice that the statements “int i” in each “for” statement define a new variable, “i”, that only exists for the duration of that particular loop.

Using **break** to Exit a Loop:

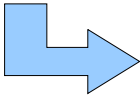
We can use a **break** statement to prematurely exit a loop:

```
#include <stdio.h>
int main ()
{
    int n;
    for (n=10; n>0; n--) {
        printf("%d, ", n);
        if (n==3) {
            printf("\ncountdown aborted!\n");
            break;
        }
    }
    return 0;
}
```



Output:

```
10, 9, 8, 7, 6, 5, 4, 3,
countdown aborted!
```



29

Imagine you're looking through a big stack of books, trying to find one with a particular title. You start from the top and look at the books one at a time until you find the one you want. Then you stop. You don't keep looking through the rest of the stack.

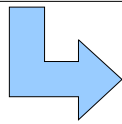
We can do the same thing in a “for” loop by using a “break” statement. When we find the thing we're looking for, we can immediately stop looping and go on with the rest of the program.

Using **continue** to Skip Iterations:

You can use a **continue** statement to skip the rest of the current loop, and go directly to the next iteration:

```
#include <stdio.h>
int main ()
{
    for (int n=10; n>0; n--) {
        if (n==5 || n==6) continue;
        printf("%d, ", n);
    }
    printf("GO!\n");
    return 0;
}
```

Note missing numbers



Output:
10, 9, 8, 7, 4, 3, 2, 1, GO!

Try to **minimize the use** of break/continue in all but the most obvious cases. Program flow that jumps around is more difficult to understand. This can sometimes be avoided with **conditional loops**.

Now imagine that you have a stack of books, some of which are paperback and some of which are hardback. You're looking for a particular title, and you remember that it's a hardback book. You'll go through the stack quickly, discarding the paperbacks without even looking at them and going on down the stack.

We can use a “continue” statement to do this kind of thing in a “for” loop. The “continue” causes the current iteration to stop, and the execution goes immediately back up to the top of the loop and starts the next iteration.

Conditional Loops:

Sometimes, you can't tell ahead of time how many times a loop must run. For Example:

- “Do something until a **convergence criterion is satisfied**.”
- “Do something until the **data are exhausted**.”

This is where conditional loops are useful.

Conditional loops come in two flavors:

- **Pre-test Loops** :

Check at the **start** of the loop to see if should be executed (again).

- These loops may possibly **never** be executed.

- **Post-test Loops** :

Check at the **end** of the loop to see if it is executed again.

- These are **ALWAYS** executed at least once.

Pre-test Loop:

The `while` loop is a pre-test loop:

Syntax:

```
while (condition) {  
    BLOCK of statements  
}
```

Example:

```
cash = get_paid();  
while (cash > 0) {  
    error = spend_money_on_snacks(cash, 0.75);  
    if (!error) cash = cash - 0.75;  
    else break;  
}
```

Do we have cash?

On successful
snack acquisition,
debit cash.

Use non-zero
status code to flag
error on snack
purchase.

We have too little
cash to continue.
Break loop here

While loops continue until their “condition” is no longer true. In the example above, the loop stops when cash is ≤ 0 .

Note that if the condition is not true initially (for example, if “cash” is zero before we start the loop), then the statements in the loop will never be executed (not even once).

Post-test Loop:

The `do` loop is a post-test loop:

Syntax:

```
do {  
    BLOCK of statements  
} while (CONDITION);
```

This loop will always be executed at least **once**.

Here we call two functions with no return values.

Example:

```
do {  
    goto_class();  
    do_homework();  
} while ( !semester_over() );
```

When the `semester_over()` function returns a **TRUE** value we can break the loop.

With a “do” loop, the statements within the loop are always executed at least once (the first time through the loop). The loop then continues to execute until the “condition” is false.

Nested Loops:

It is very common to **nest** loops in programs, by placing one loop inside of another:

```
const int NUMDAYS = 7;
const int NUMWEEKS = 14;
int day, week;
for (week=0 ; week<NUMWEEKS ; week++)
{
    gotoMovie(); // Done 14 times.
    for (day=0 ; day<NUMDAYS ; day++) {
        eat(); // Done 98 times
        study();
        sleep();
    }
}
```

Often limits on loops are set with constants or #define statements that are prominently visible at the top of the file. This allows you to easily change the behavior of your program.

Also note how we use indentation to clearly mark code blocks. This is very important, even for simple programs, and it will make your code much easier to debug and help you avoid mistakes while writing.

Part 3: The Scope of Variables



We mentioned earlier that C++ lets you define variables that only exist within a particular loop. C lets you define variables in a couple of different “scopes”, or parts of the program in which the variable is visible.

Global Scope:

All variables in C/C++ exist within a certain **scope** or context.

Scope refers to where variables may be accessed in your program.

Variables defined **outside of function blocks** are said to be in **Global Scope**. They may be accessed by any function below their definition statement.

```
int globalInt;  
void printInt();  
int main () {  
    globalInt = 1;  
    printInt();  
    return 0;  
}  
  
void printInt() {  
    printf("%d\n", globalInt);  
}
```

The global variable **globalInt** is defined outside of "main" or any other function.

globalInt is visible in all code listed after its definition.

Local versus Global Scope:

```
int number;
void printInt();
void printFloat();

int main (){
    number = 1;
    printInt();
    printFloat();
    return 0;
}

void printInt() {
    printf("%d\n", number);
}

void printFloat() {
    float number = 5.0;
    printf("%f\n", number);
}
```

A global integer variable named "number" is defined here.

A local variable (defined within a function) with the same name will override the global definition, but only within that function. The global variable is unaffected.

In the scope of this function, "number" is a float and has no relation to the global variable.

37

Only one variable of a given name can be declared in each scope:

```
int main(){
    int a;
    float a; // not allowed!
}
```

Using the Smallest Scope:

In general you should **avoid** using global variables.

If many functions can change a value it's very difficult to keep track of what's going on. It's much better to pass data to functions explicitly rather than to define data globally.

For clearer code, always restrict variables to the **smallest possible scope!**

Part 4: Introduction to Probability and Statistics



Probability and statistics are very important to science, and computers are very good at the kinds of calculations these disciplines require. Statistical calculations often involve repeating the same operations many times. As we saw in the section on loops, this becomes trivial when computers are available.

(The pictures are Jacob Bernoulli, Abraham DeMoivre, and Joseph Fourier.)

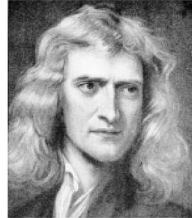
Inductive vs. Inductive Reasoning:

$$F = \frac{G m M}{r^2}$$



**Deductive
reasoning** →

“After one second, a falling body will be travelling at 9.8 m/s (near the surface of the earth).”



**Inductive
reasoning** →

$$F = \frac{G m M}{r^2}$$

Height	V_0
1.0	9.8
2.0	19.6
3.0	29.4
4.0	39.2
...	...

40

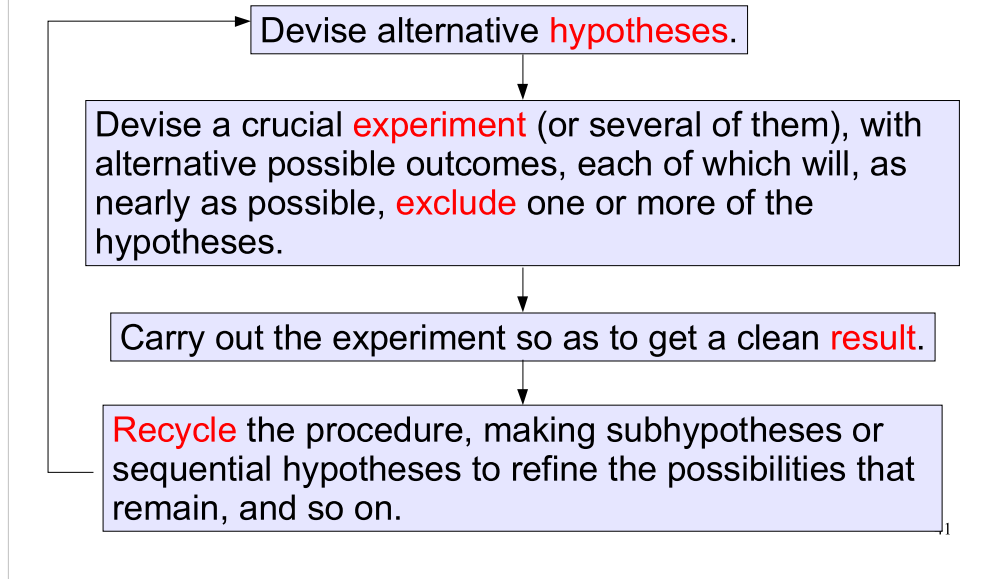
When we're in school, we tend to think of science as the process shown at the top of this picture: Taking a generalization (like the Law of Gravitation), plugging in some numbers, doing some algebra and maybe applying some other generalizations (like $F=ma$) to arrive at a specific answer to a specific question. This process of deriving a specific answer from a general principle is called “deductive reasoning”.

In reality, there's a second, more important and more difficult, component to science, called “inductive reasoning”. This is the process of taking some specific data, acquired through experiments, and producing a generalization (a hypothesis) that is consistent with those data. This is how science progresses.

Probability and statistics provide us with powerful tools for analyzing our data, and for testing the quality of our hypotheses.

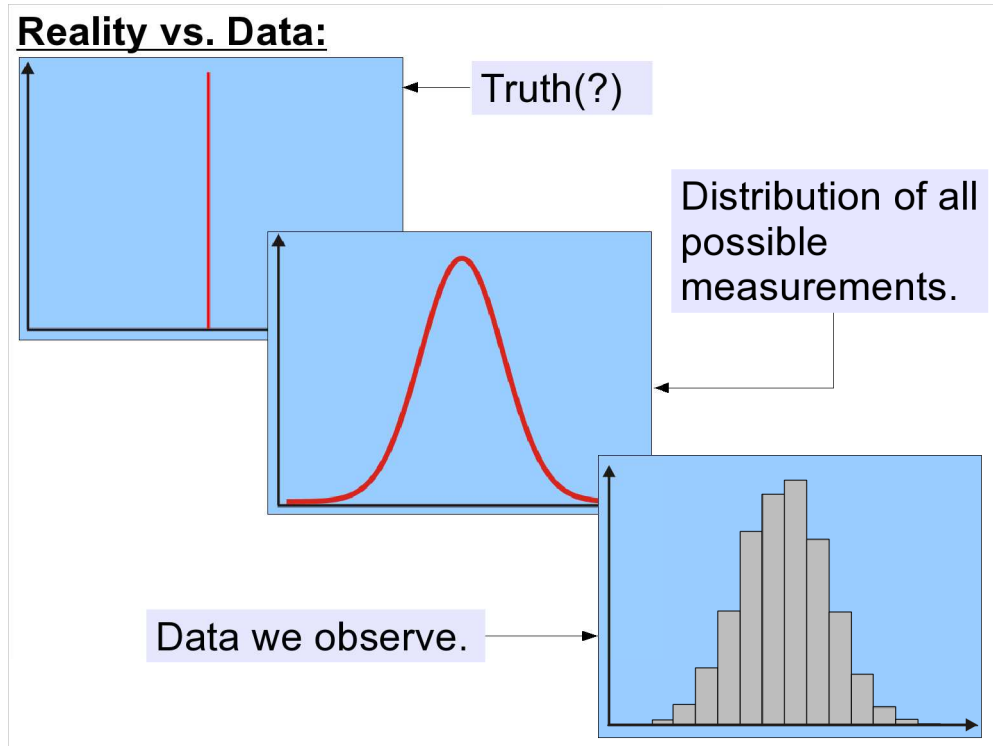
“Strong Inference”:

In 1964, John R. Platt wrote about a form of the Scientific Method that he called “Strong Inference”. It goes as follows:



Platt's article is here:

http://pages.cs.wisc.edu/~markhill/science64_strong_inference.pdf



When we do an experiment, we're always at least at two removes from the thing we're trying to see. We assume that there's some underlying truth that has a single, well-defined value (like the speed of light in a vacuum), but when we measure it, we see a range of different values. This is partly because of imperfections in our instruments, but there may also be physical limits to the precision of our measurements.

If we could make infinitely many measurements, we see that the values we observe fall into some distribution like the middle picture. This curve gives the probability that any given observation will have a particular value.

Using statistics, we can take our finite set of observations and use them to make educated guesses about the shape of this underlying probability distribution. But that's as close to the "truth" as we can get. From there, we just have to assume that the mean of the probability distribution is equal to the "true" value we're trying to measure.

Systematic vs. Random Uncertainties:

We can group the uncertainties in our data into two categories:

Systematic Uncertainties:

These are the result of things that systematically shift the values we measure by a **constant** amount (or factor). Once these uncertainties are identified, it's usually possible to correct for them.

For example, maybe the meter stick you used was really only 0.99 meters long. We can correct our data to account for this.

Random Uncertainties:

These are the result of things that shift each measurement by a different amount, at **random**. For example, electronic noise may introduce random uncertainties into your data.

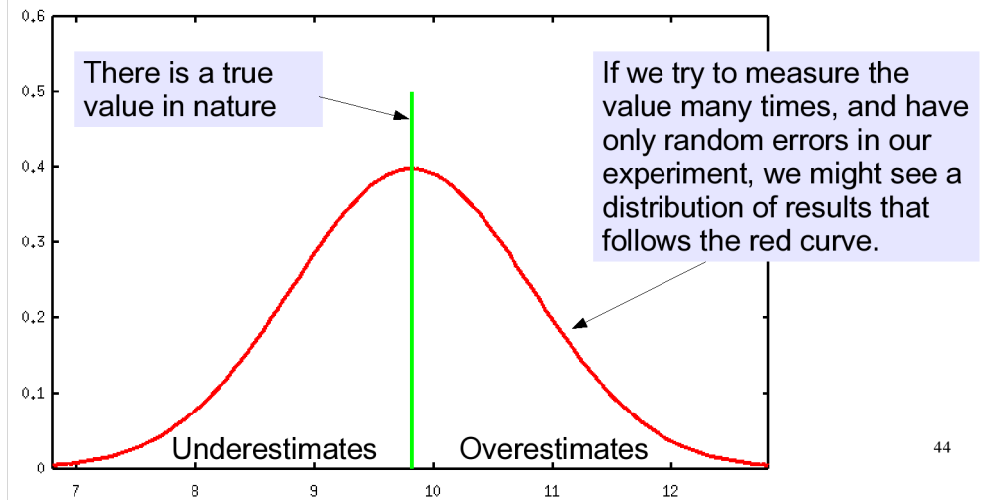
Since each measurement is shifted by a different amount, correcting for this sort of uncertainty is not so straightforward. Usually, to extract the underlying "truth" from our data we need to use the mathematics of **statistics**.

43

These are the things that define the shape of that underlying probability distribution. We'll concentrate on the random uncertainties.

Random Errors:

In Physics, it's common to refer to the uncertainty in a measurement as the "error". Remember, though, that this term doesn't imply that you've done anything wrong. Random errors are an inherent part of doing experiments, and it's important to understand them.

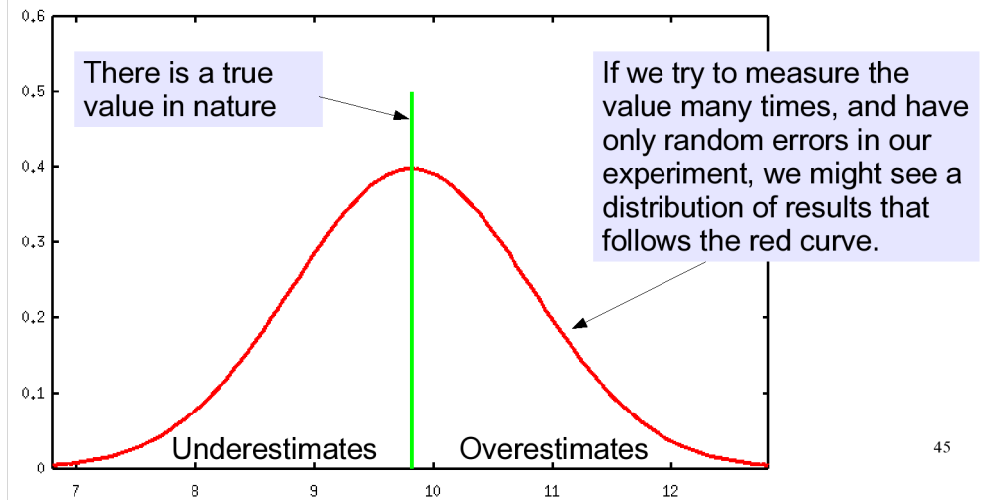


Consider a parameter we wish to measure (i.e. the acceleration due to gravity).

Probability Distribution:

The red curve maps out a **probability distribution**. It describes how random processes affect our observations as we repeat the experiment.

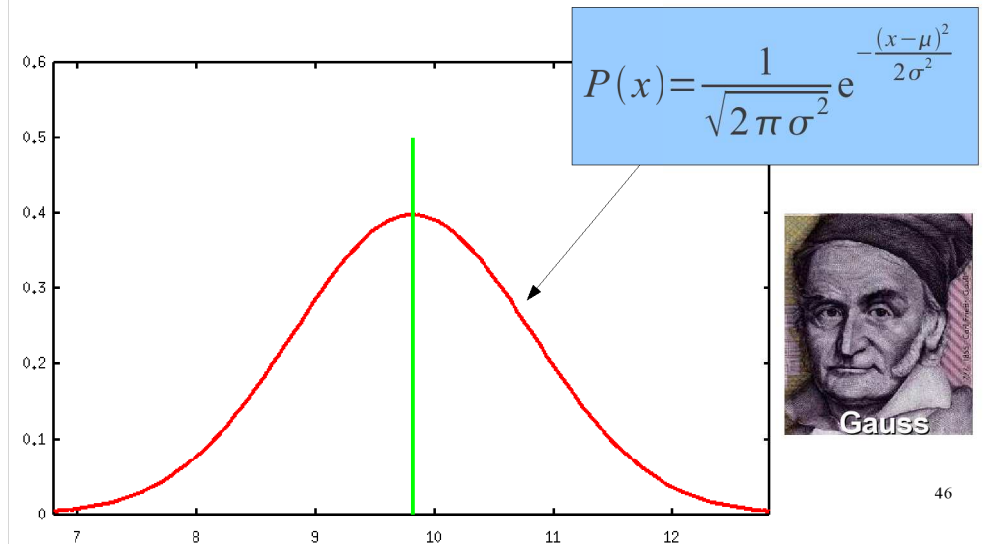
Lots of measurements cluster around the peak, few around the tails.



Specifically, if we normalize the area under the curve to 1, this distribution shows the probability that a given measurement will have a value between x and $x+dx$.

The Gaussian Distribution:

Measures of random processes, **in the limit of many samples** often tend to produce a characteristic **Gaussian or Normal** distribution. (The classic “Bell Curve”.)

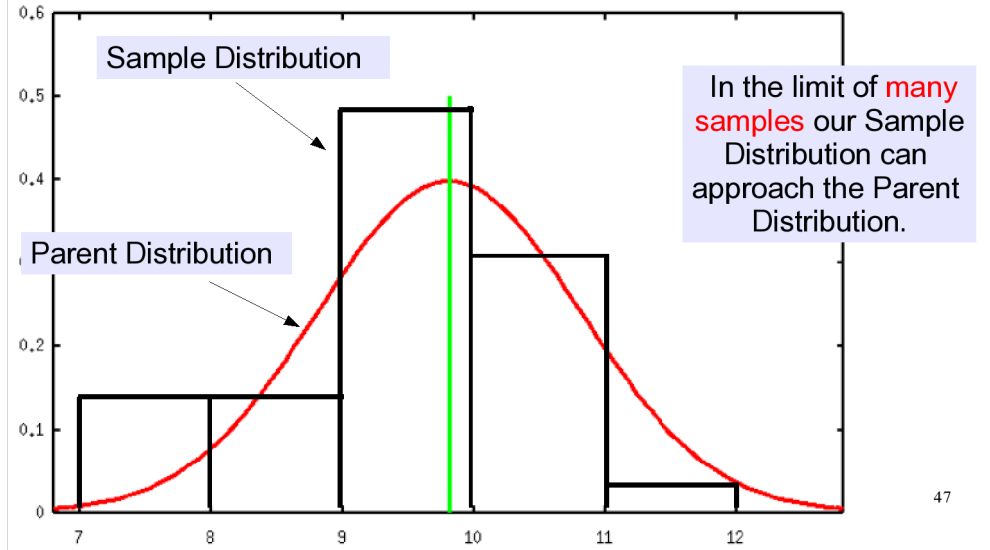


In fact the Central Limit Theorem tells us that a sufficiently large linear combination of random variables always approaches a Normal distribution.

This was a source of amazement to early statisticians, who saw this curve popping up everywhere: astronomical data, actuarial tables, agricultural data. The Central Limit Theorem explained why this was so.

Sample Distribution:

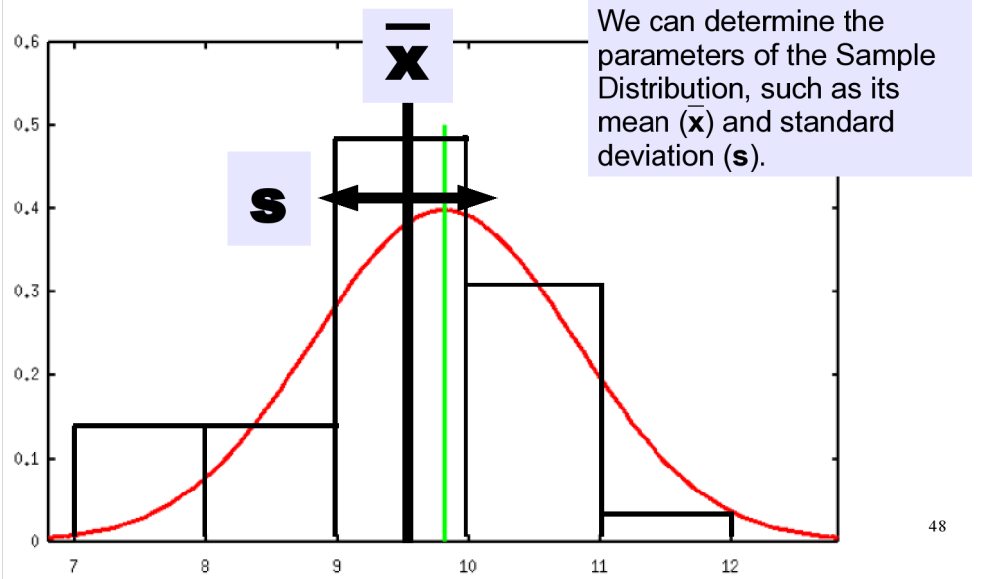
The red curve represents the probability of observing a particular value. This is the **Parent Distribution** from which our measurements will be drawn. Our experiment draws measurements, at random, from this. These measurements form our **Sample Distribution**.



Our certainty in the true value of the underlying quantity is limited by what we know about how the true value gives rise to the parent probability distribution.

What is Measurable:

Our **best estimates** of the mean and variance of the Parent Distribution come from the mean and variance of the Sample Distribution. This is something we can study with appropriate programs.



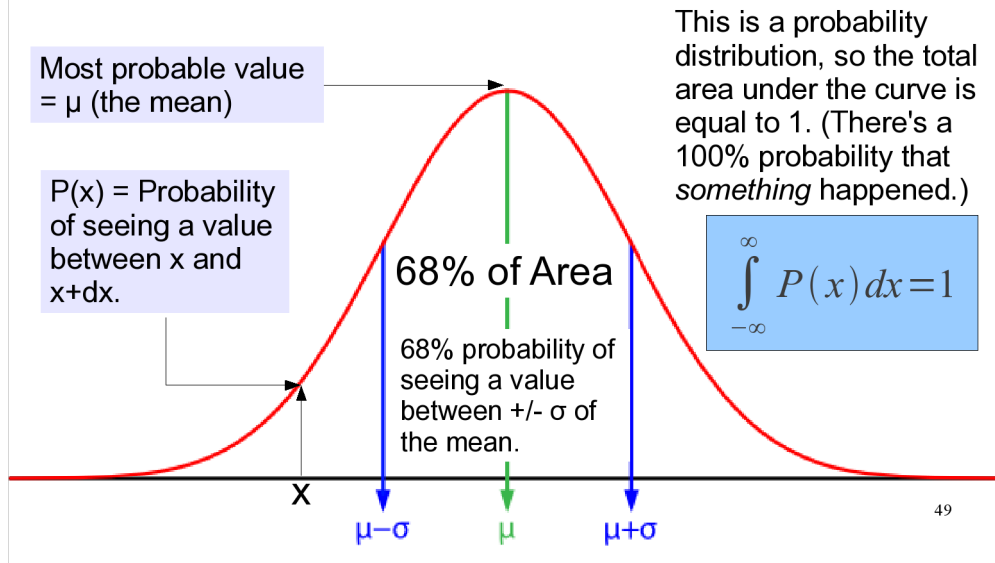
As we'll see, the “variance” is defined to be just the square of the standard deviation.

We can use the data in our Sample Distribution to make defensible statements about the Parent Distribution, such as “We are 95% certain that the mean of the Parent Distribution lies between 8.1 and 11.2”.

Notice that we're not even talking about the underlying “truth” any more. All we can really talk about are estimates of the parameters of the underlying Parent Distribution, based on the measurements in our Sample Distribution. Does the mean of Parent Distribution equal the True Value? Is there really a True Value, separate from this mean? Does it matter?

Properties of the Normal (Gaussian) Distribution:

To clearly distinguish between parent and sample distributions, we use μ and σ for the Parent Distribution and \bar{x} and s for the Sample Distribution.

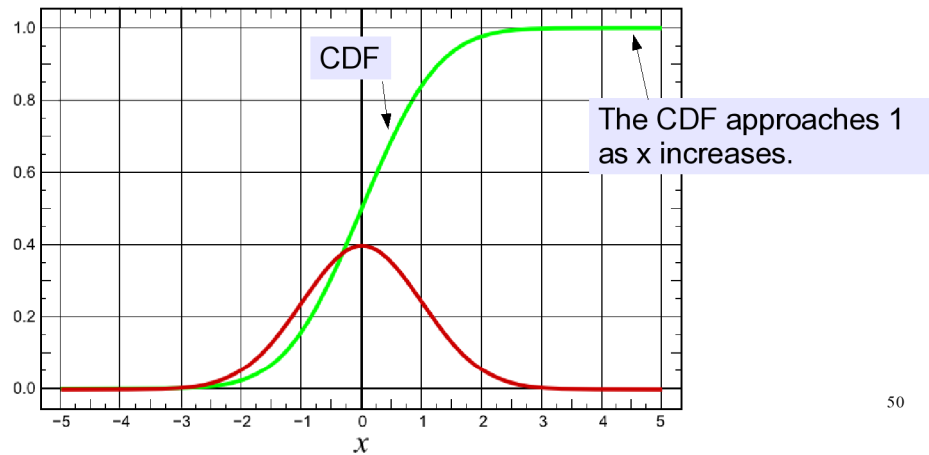


As you can see, there's a ~68% probability that any given measurement will yield a value that lies within \pm sigma of the mean. There's an ~95% probability that the value will lie within \pm 2 sigma.

Integral of the Normal Distribution:

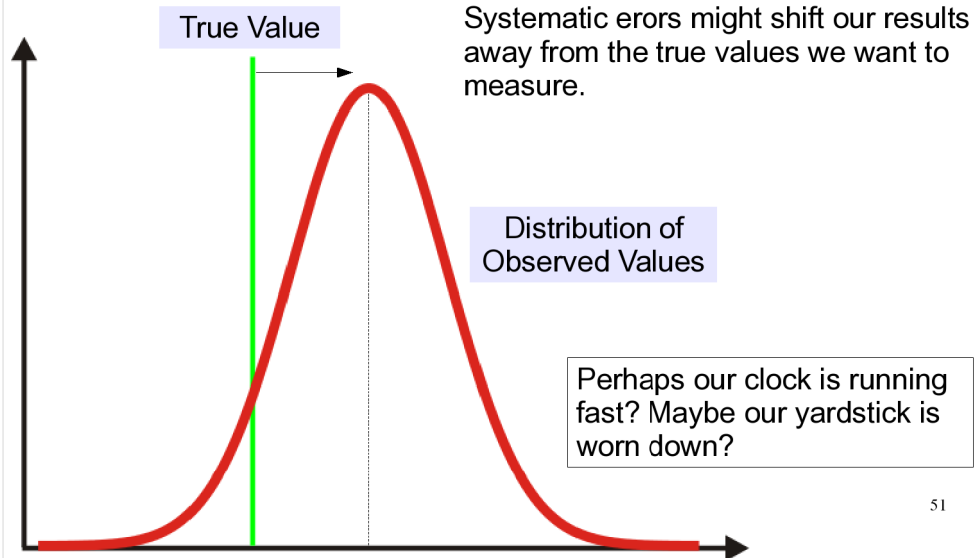
The Normal distribution is a **Probability Density Function** (PDF), so the total area under the curve must be equal to 1. One way to see this is to look at the integral of the Normal distribution. The integral of a PDF is called a **Cumulative Distribution Function** (CDF).

The CDF tells us the probability of observing a value **less than or equal to x** .



Systematic Errors:

Systematic errors (aka “**biases**”) must be corrected with expert knowledge of your apparatus and procedures. Statistics will not improve a mis-calibrated detector or a flawed design for your experiment!

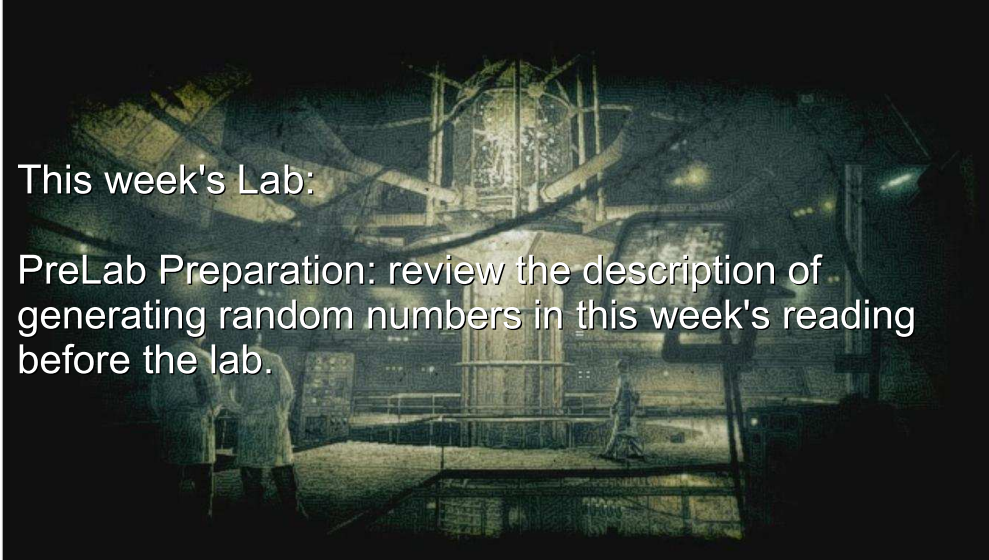


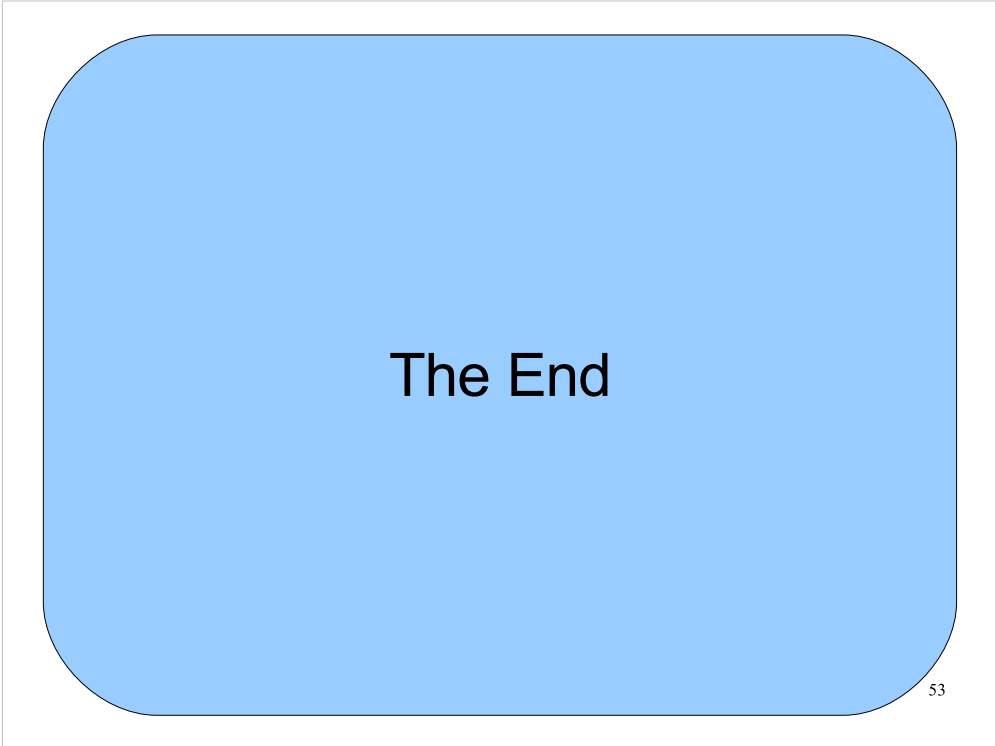
Next Time:

- More on passing data to functions
- Solving problems with the help of random numbers

This week's Lab:

PreLab Preparation: review the description of generating random numbers in this week's reading before the lab.





Thanks!