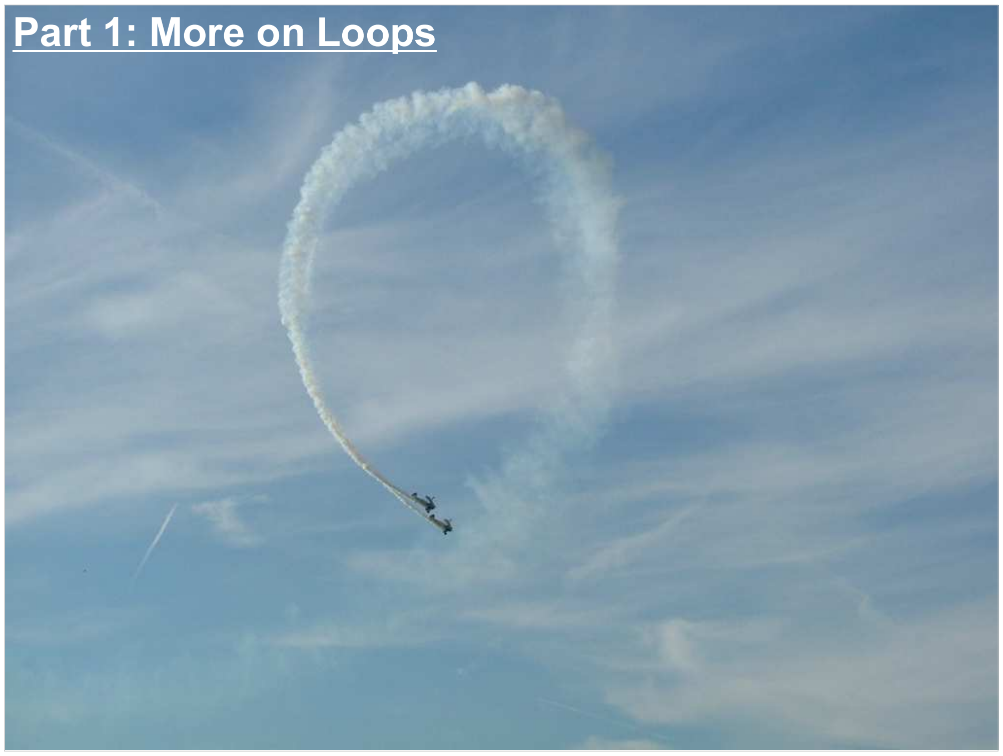
Physics 2660

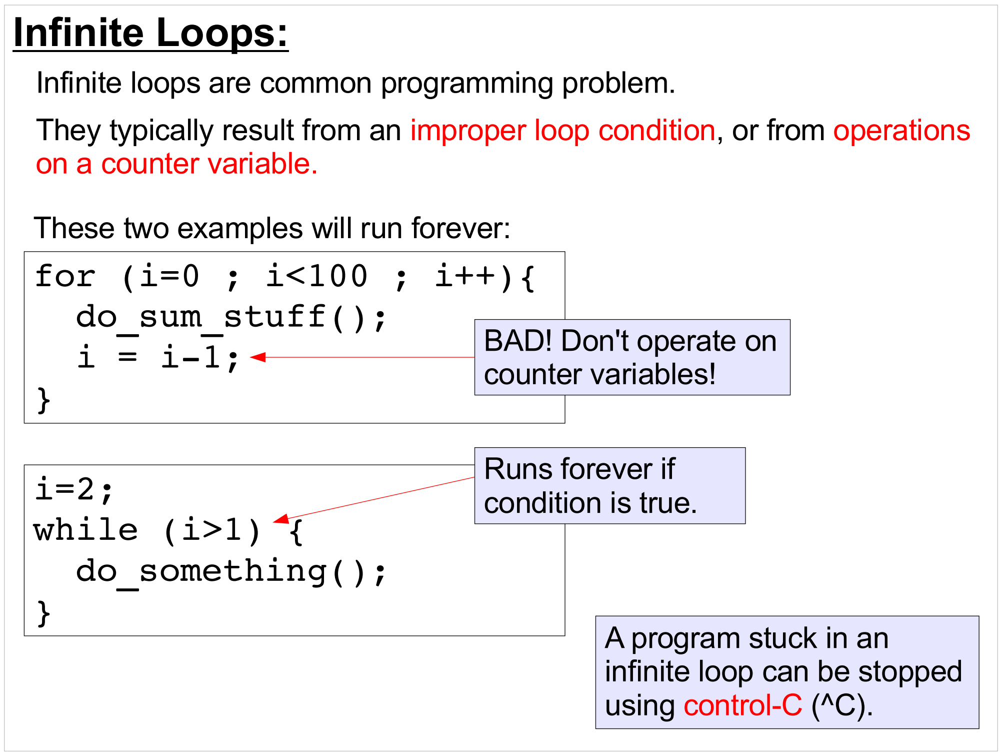
Lecture 4: C – Part 3

Today
- More on Loops
- Pointers
- More on functions
- Static Variables
- Performing integrations with random numbers

Part 1: More on Loops



As we've said before, loops are the reason computers exist. Any individual calculation can be done without too much effort by a human being. But when that calculation needs to be repeated hundreds, thousands or millions of times, that's when you need a computer.



Infinite Loops:

Infinite loops are common programming problem.

They typically result from an improper loop condition, or from operations on a counter variable.

These two examples will run forever:

```
for (i=0 ; i<100 ; i++){
  do_sum_stuff();
  i = i-1;
}
```

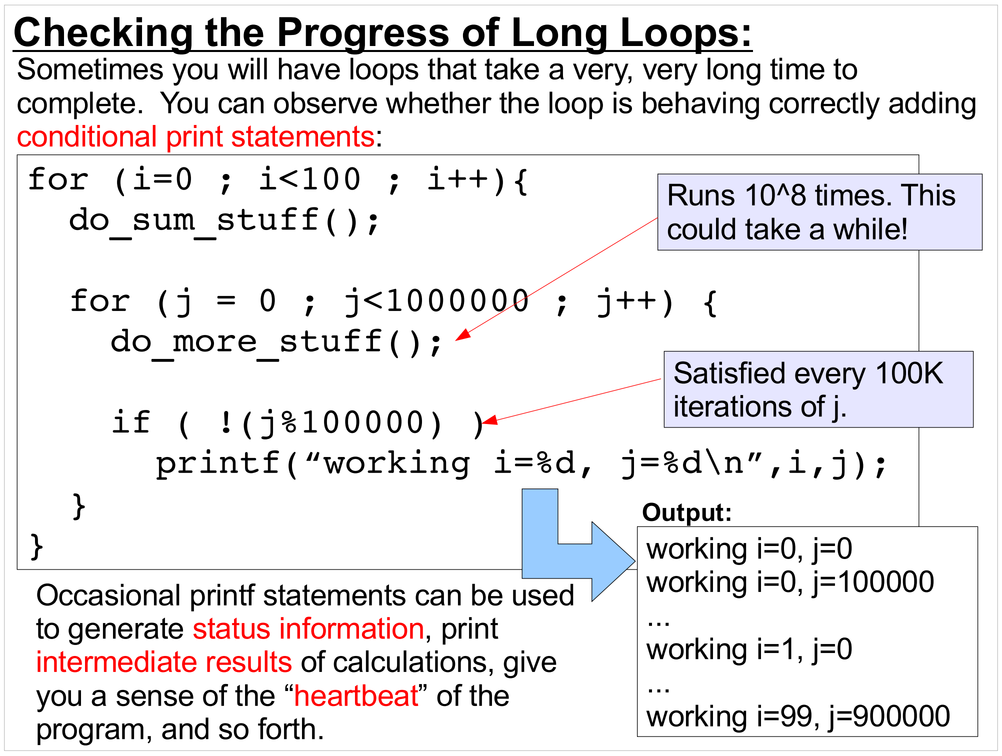
BAD! Don't operate on counter variables!

```
i=2;
while (i>1) {
  do_something();
}
```

Runs forever if condition is true.

A program stuck in an infinite loop can be stopped using control-C (^C).

Here are some examples of things that can go wrong when you try to write a loop into your program.



Checking the Progress of Long Loops:

Sometimes you will have loops that take a very, very long time to complete. You can observe whether the loop is behaving correctly adding conditional print statements:

```
for (i=0 ; i<100 ; i++){
  do_sum_stuff();

  for (j = 0 ; j<1000000 ; j++) {
    do_more_stuff();

    if ( !(j%100000) )
      printf("working i=%d, j=%d\n",i,j);
  }
}
```

Runs 10^8 times. This could take a while!

Satisfied every 100K iterations of j.

Output:

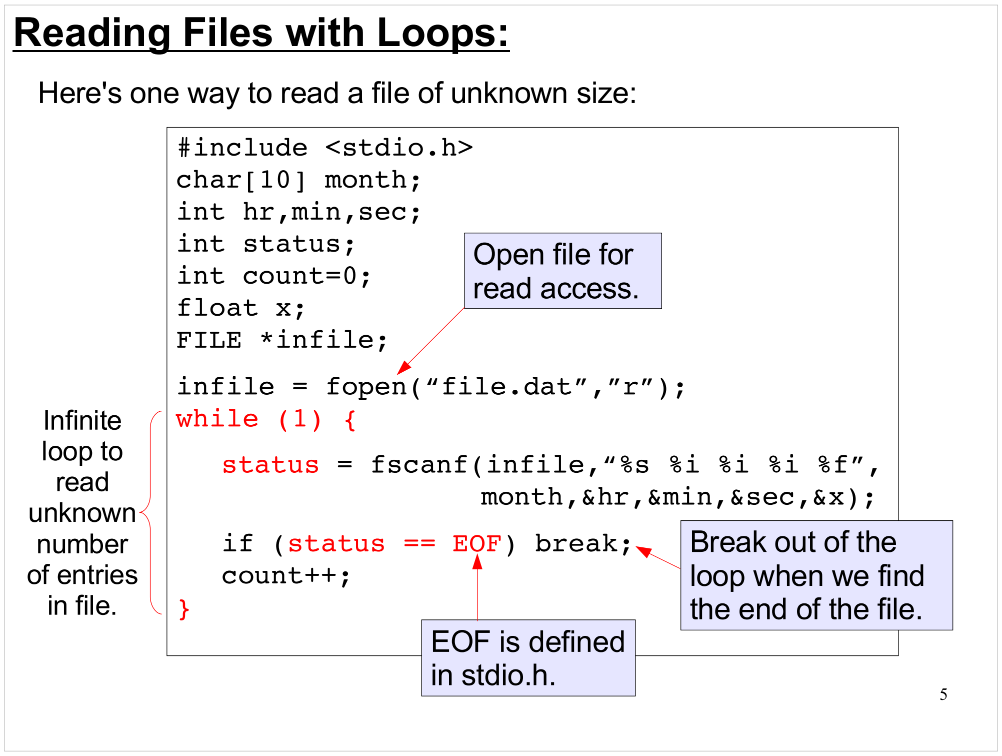
```
working i=0, j=0
working i=0, j=100000
...
working i=1, j=0
...
working i=99, j=900000
```

Occasional printf statements can be used to generate status information, print intermediate results of calculations, give you a sense of the "heartbeat" of the program, and so forth.

Let's parse what that "if" statement does:

"j%100000" uses the modulo operator (%) to get the remainder after dividing j by 100,000. Whenever j is a multiple of 100,000, "j%100000" will be zero.

The "!" means "not". If "A" is zero, then "!A" will be one, or "true" in a test condition. If "A" is non-zero, then "!A" will be "false". So, the "if" statement says "Do the following only when j is a multiple of 100,000."



Reading Files with Loops:

Here's one way to read a file of unknown size:

```
#include <stdio.h>
char[10] month;
int hr,min,sec;
int status;
int count=0;
float x;
FILE *infile;

infile = fopen(“file.dat”,”r”);
while (1) {

   status = fscanf(infile,“%s %i %i %i %f”,
                           month,&hr,&min,&sec,&x);

   if (status == EOF) break;
   count++;
}
```

Open file for read access.

Infinite loop to read unknown number of entries in file.

Break out of the loop when we find the end of the file.

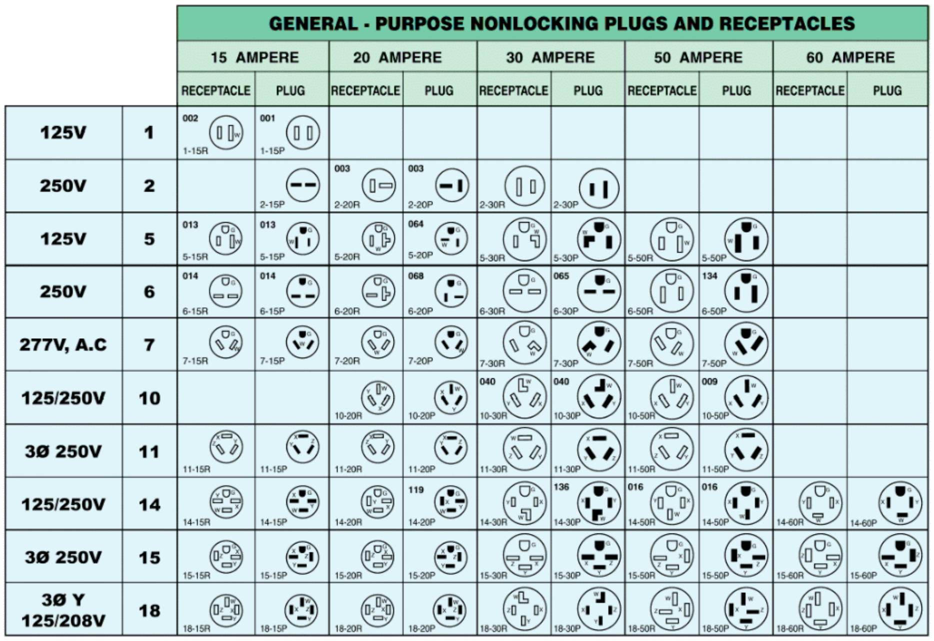
EOF is defined in stdio.h.

Here we've created an infinite loop on purpose. The loop will continue repeating until we use “break” to break out of it.

In this case, we check to see when scanf returns the value EOF (“End Of File”). Note that scanf will only return EOF the next time it's called after reading the last line. In other words, the scanf statement above will be called one more time than the number of lines in the file.

Scanf returns status = # of fields converted or EOF. In general, we could check to make sure the returned value equals the number of values we expect to see. In this case if (status !=5) we have a problem.

Part 2: More on Functions



		GENERAL - PURPOSE NONLOCKING PLUGS AND RECEPTACLES									
		15 AMPERE		20 AMPERE		30 AMPERE		50 AMPERE		60 AMPERE	
		RECEPTACLE	PLUG	RECEPTACLE	PLUG	RECEPTACLE	PLUG	RECEPTACLE	PLUG	RECEPTACLE	PLUG
125V	1	002 1-15R	001 1-15P								
250V	2		2-15P	003 2-20R	003 2-20P	2-30R	2-30P				
125V	5	013 5-15R	013 5-15P	5-20R	064 5-20P	5-30R	5-30P	5-50R	5-50P		
250V	6	014 6-15R	014 6-15P	6-20R	068 6-20P	6-30R	065 6-30P	6-50R	134 6-50P		
277V, A.C	7	7-15R	7-15P	7-20R	7-20P	7-30R	7-30P	7-50R	7-50P		
125/250V	10			10-20R	10-20P	040 10-30R	040 10-30P	10-50R	009 10-50P		
3Ø 250V	11	11-15R	11-15P	11-20R	11-20P	11-30R	11-30P	11-50R	11-50P		
125/250V	14	14-15R	14-15P	14-20R	119 14-20P	14-30R	136 14-30P	016 14-50R	016 14-50P	14-60R	14-60P
3Ø 250V	15	15-15R	15-15P	15-20R	15-20P	15-30R	15-30P	15-50R	15-50P	15-60R	15-60P
3Ø Y 125/208V	18	18-15R	18-15P	18-20R	18-20P	18-30R	18-30P	18-50R	18-50P	18-60R	18-60P

These are electrical plugs and receptacles defined by a standard called "NEMA". The idea behind this standard is that it shouldn't be physically possible to plug the wrong kind of device into the wrong kind of receptacle.

This is the way your function prototypes work. They define the number and type of the functions arguments, and the type of data it returns. The C ompiler can then check each time you use the function to make sure you're plugging in the right type and number of variables, and you're capturing the functions return value in an appropriate way.

Review of Function Interfaces:

General form for a function prototype:

$$\underbrace{\text{type}}_{\text{Type of data returned}}\ \underbrace{\text{func\_name}}_{\text{Function Name}}\ (\underbrace{\text{type1}}_{\text{Type of 1st argument}}\ \underbrace{\text{var1}}_{\text{1st argument}},\ \text{type2 var2, ...});$$

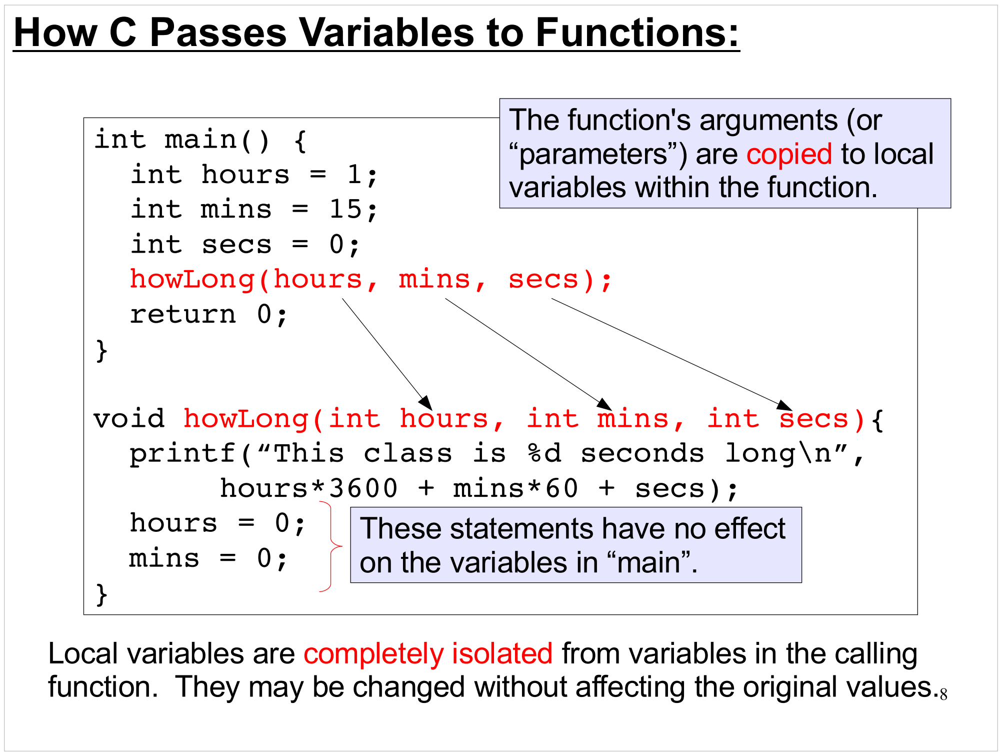
Any number of arguments, but only one return value.

A function may be used just like a variable of its return type.

```
printf( “Distance = %lf”, dist(x2,x1) );
```

A function may return nothing as it does here (type = void).

```
void howLong(int hours, int mins, int secs){
  printf(“This class is %d seconds long\n”,
       hours*3600 + mins*60 + secs);
}
```



How C Passes Variables to Functions:

```
int main() {
  int hours = 1;
  int mins = 15;
  int secs = 0;
  howLong(hours, mins, secs);
  return 0;
}

void howLong(int hours, int mins, int secs){
  printf("This class is %d seconds long\n",
       hours*3600 + mins*60 + secs);
  hours = 0;
  mins = 0;
}
```

The function's arguments (or "parameters") are copied to local variables within the function.

These statements have no effect on the variables in "main".

Local variables are completely isolated from variables in the calling function. They may be changed without affecting the original values.

C always copies parameter data from the calling function to the target function.

The target function has an independent copy of the data and can never change the value of a variable in the calling function.

Using an alternate syntax, C++ can operate on the original variable in a function. But we will instead concentrate on a workaround to the above restriction available in both C/C++

To understand this, we must first discuss pointers.

Part 3: Pointers

Pointers are a very powerful and dangerous feature of the C language. To understand them, we'll first need to talk some more about how variables are stored in memory.

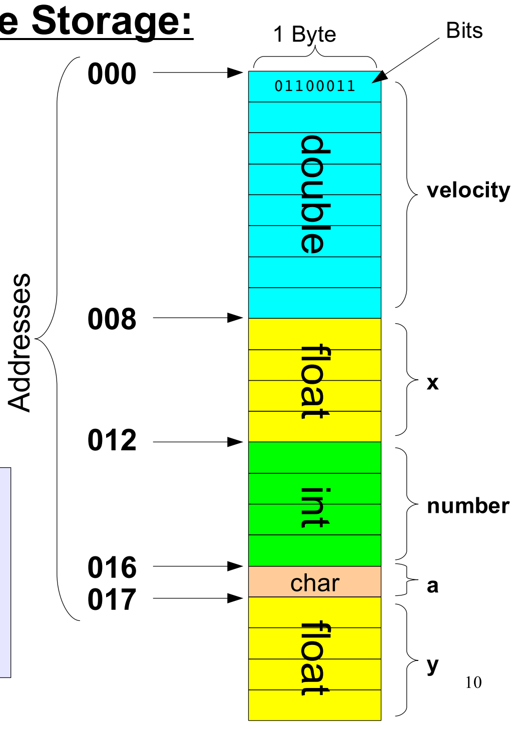
Another Look at Variable Storage:

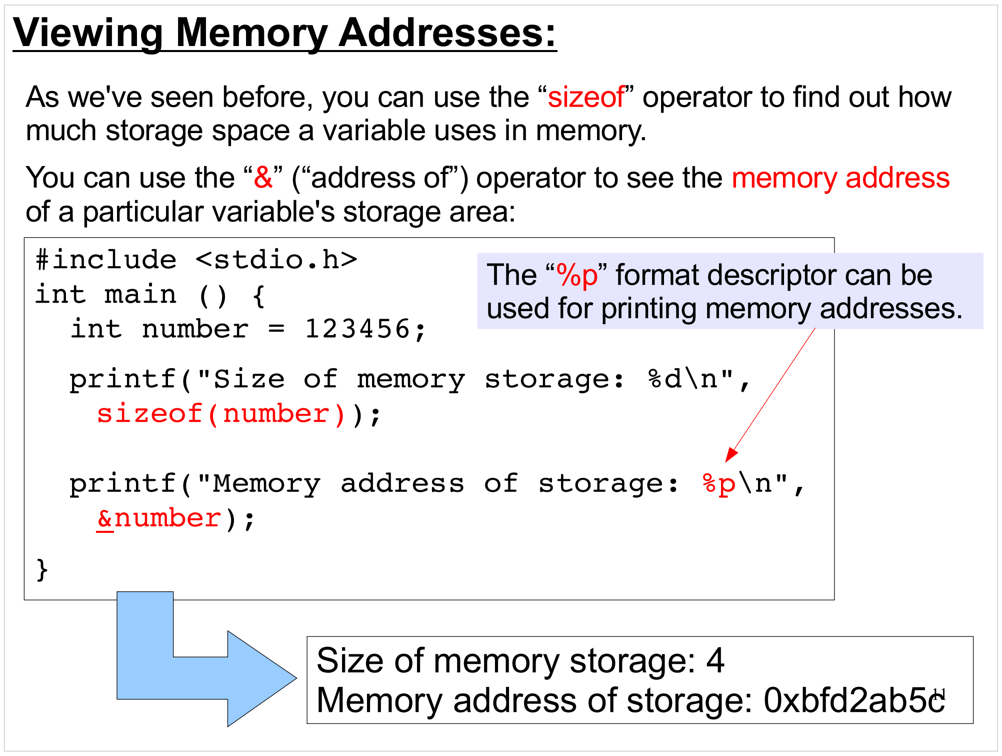
- The values of variables are stored in memory.

- 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.





Viewing Memory Addresses:

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);

}
```

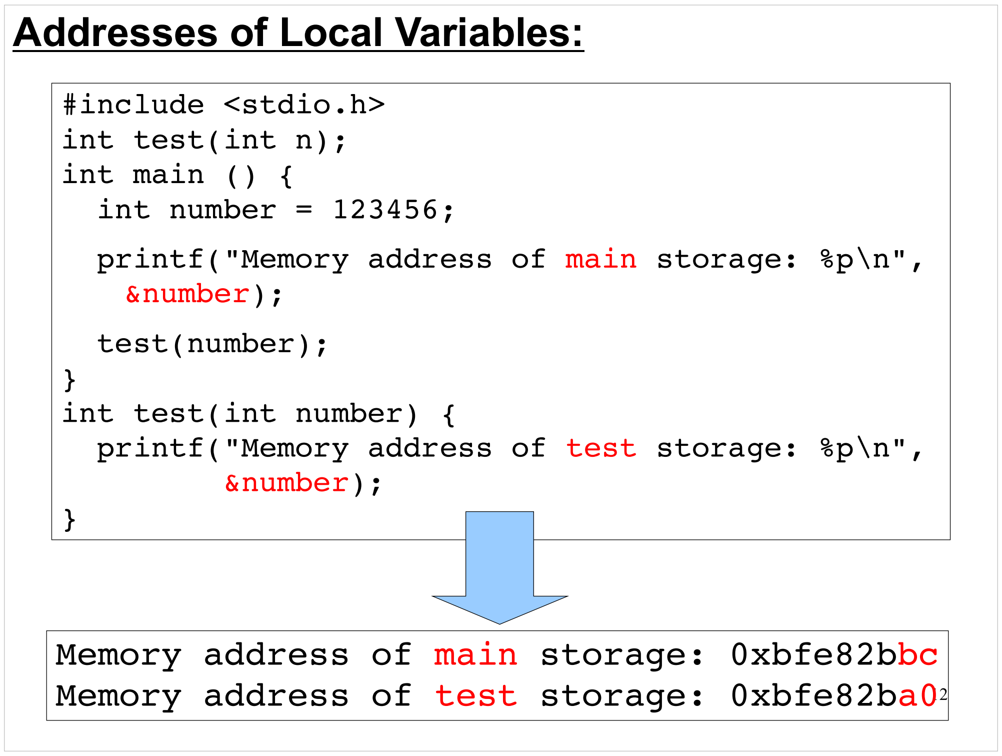
The “%p” format descriptor can be used for printing memory addresses.

```
Size of memory storage: 4
Memory address of storage: 0xbfd2ab5c
```

As we'll see “p” stands for “pointer”.

We usually express memory addresses in the form of a hexadecimal (base 16) number. This is a natural, compact way of writing binary numbers. 16 is just 2^4, so each 8-bit byte can be represented by two hexadecimal digits, like “5c”.

The memory address will be different every time you run the program. The program looks around for a chunk of available memory when it starts, and in general the chunk will be in a different place each time.

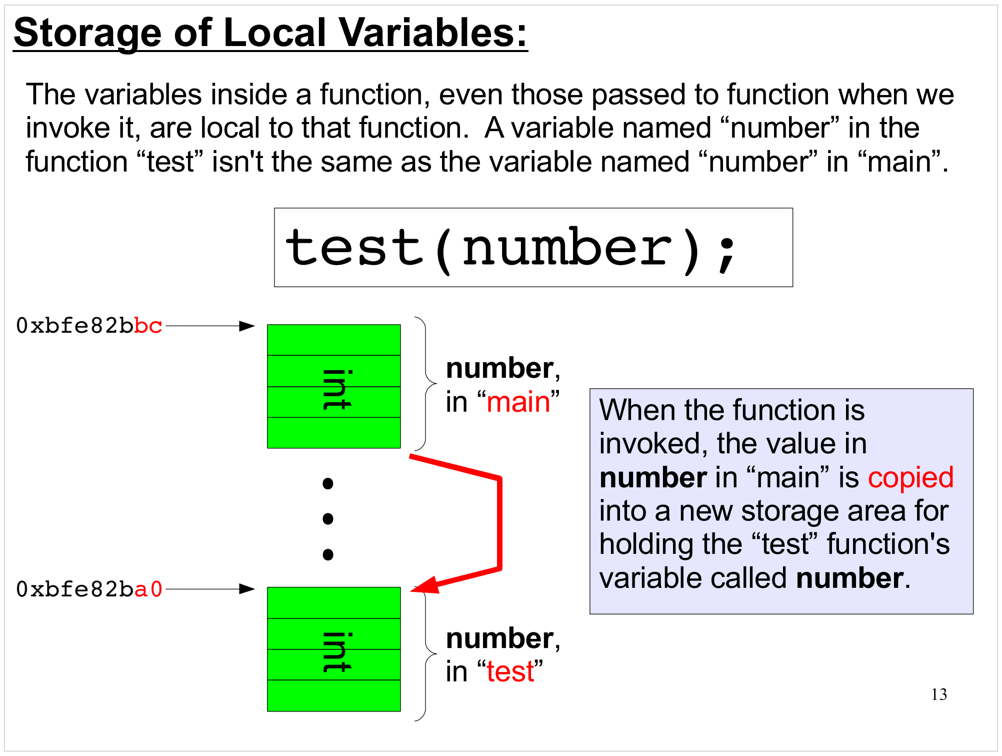


Addresses of Local Variables:

```
#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
```

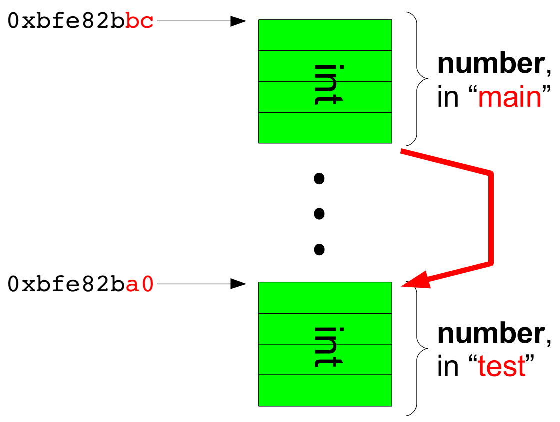
Now that we know how to look at memory addresses, we can confirm that the variable “number” in main really is different from the variable “number” in the function “test”. The two variables live in different locations in memory.



Storage of Local Variables:

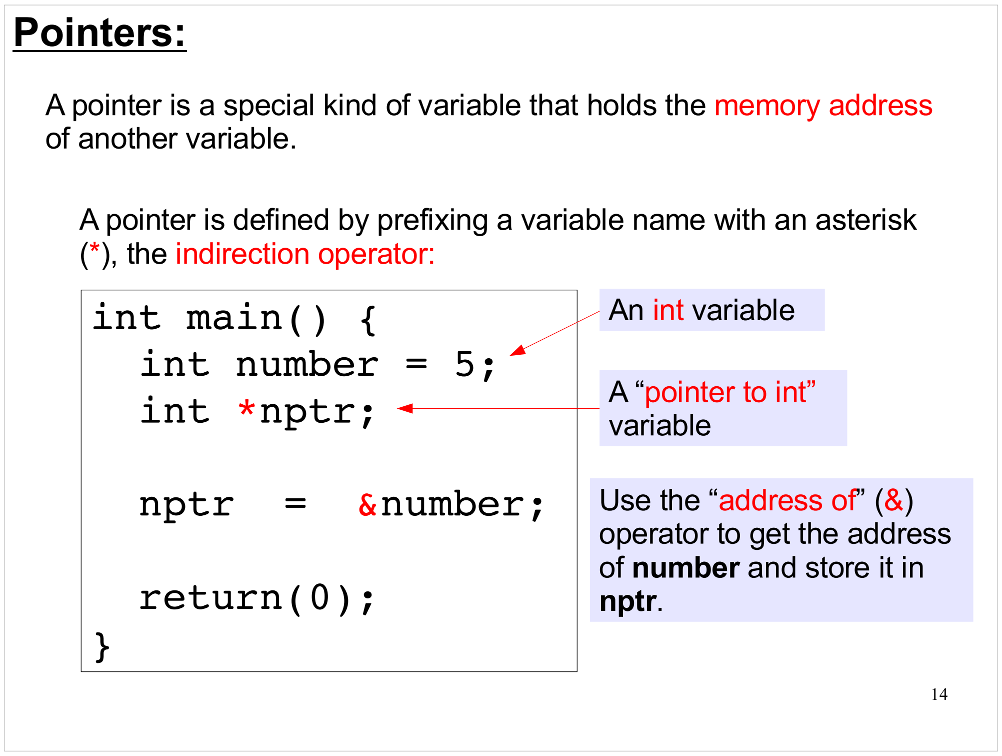
The variables inside a function, even those passed to function when we invoke it, are local to that function. A variable named "number" in the function "test" isn't the same as the variable named "number" in "main".

```
test(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**.

We say that, in C, function arguments are "passed by value".



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:

```
int main() {
  int number = 5;
  int *nptr;

  nptr  =  &number;

  return(0);
}
```

An int variable

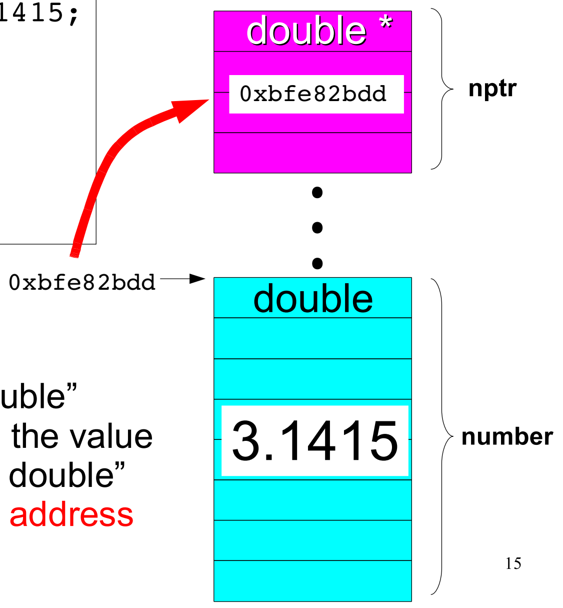
A "pointer to int" variable

Use the "address of" (&) operator to get the address of **number** and store it in **nptr**.

Note: the * is NOT part of the pointer's name. And, of course, don't confuse this with multiplication.

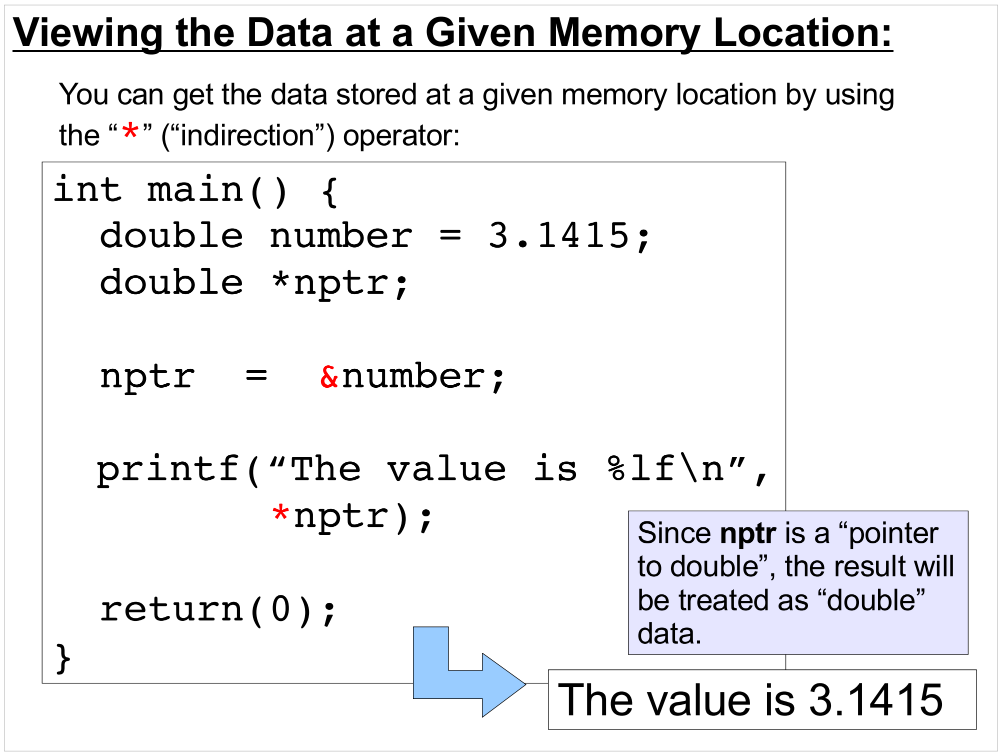
How Pointers are Stored in Memory:

```
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**.

Notice that the memory storage for a “double” variable is a different size than the storage area for a “pointer to double” (double *). Pointers are usually stored in 4 bytes (on 32-bit computers) or 8 bytes (on 64-bit computers).



Viewing the Data at a Given Memory Location:

You can get the data stored at a given memory location by using the "*" ("indirection") operator:

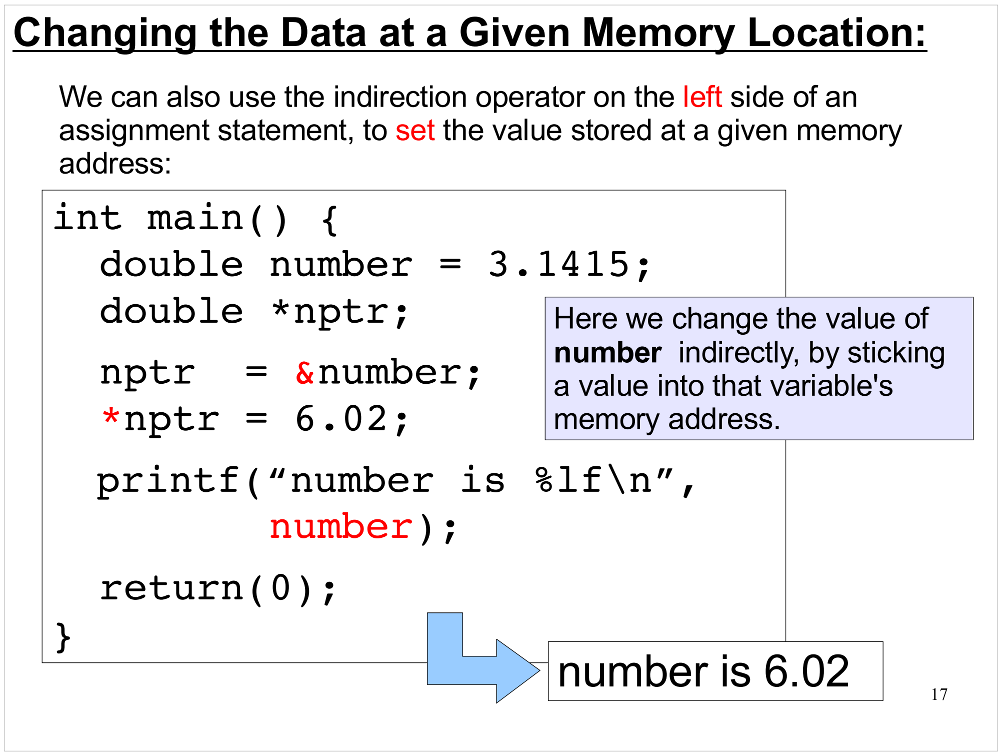
```
int main() {
  double number = 3.1415;
  double *nptr;

  nptr  =  &number;

  printf("The value is %lf\n",
          *nptr);

  return(0);
}
```

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

```
The value is 3.1415
```

The compiler interprets the indirection (or "dereferencing") operator (*) as follows:

"use the data in nptr to find the memory address it "points to" and fetch the data from that address"



Changing the Data at a Given Memory Location:

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:

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

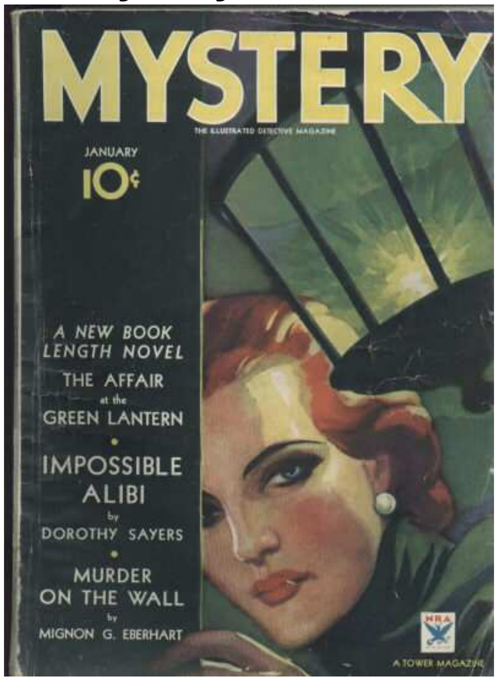
Here we change the value of **number** indirectly, by sticking a value into that variable's memory address.

```
number is 6.02
```

17

So, we can change a variable's value without explictly referring to the variable by name. All we need to know is its memory address.

The Mystery of “scanf”:

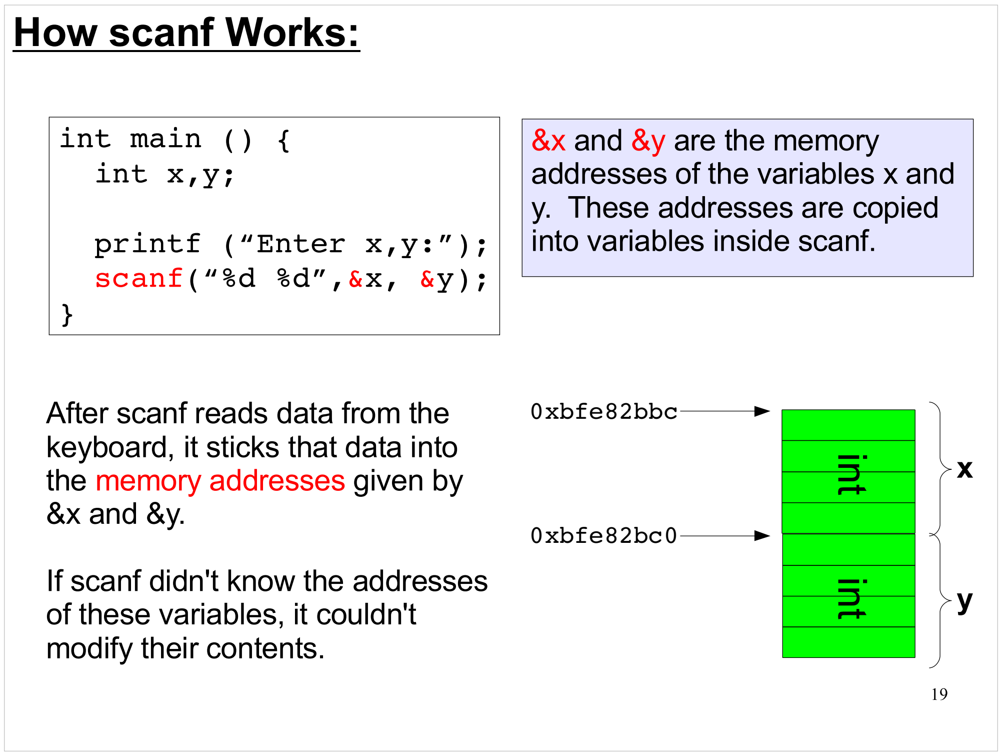

MYSTERY
THE ILLUSTRATED DETECTIVE MAGAZINE
JANUARY
10¢
A NEW BOOK LENGTH NOVEL
THE AFFAIR at the GREEN LANTERN
IMPOSSIBLE ALIBI by DOROTHY SAYERS
MURDER ON THE WALL by MIGNON G. EBERHART
NRA
A TOWER MAGAZINE

When we use scanf (or fscanf) we create some variables, and then scanf sets the values of those variables for us.

How can scanf modify the values of the variables we've created, if C always gives functions a copy of each variable we pass?

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

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

The answer is “indirection”!



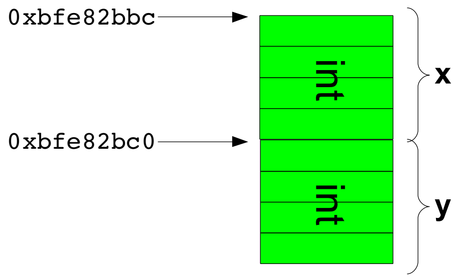
How scanf Works:

```
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.



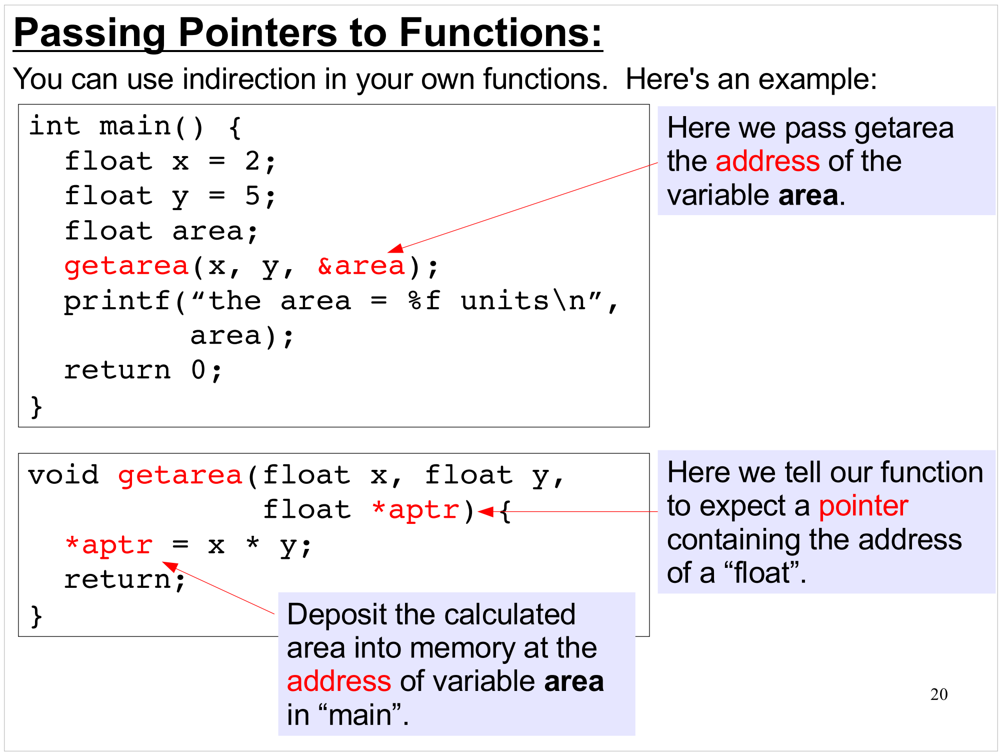
We stated earlier that C copies data from the calling function to local variables when another function is called.

This effectively makes it impossible for a function to change the data associated with a variable in its parameter list,

but...

if the data passed to a function is a pointer, we can dereference the pointer and change the data in the memory location referenced by that pointer

In this way we can change data in the main or calling function by accessing it in an indirect manner



Passing Pointers to Functions:

You can use indirection in your own functions. Here's an example:

```
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**.

```
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".

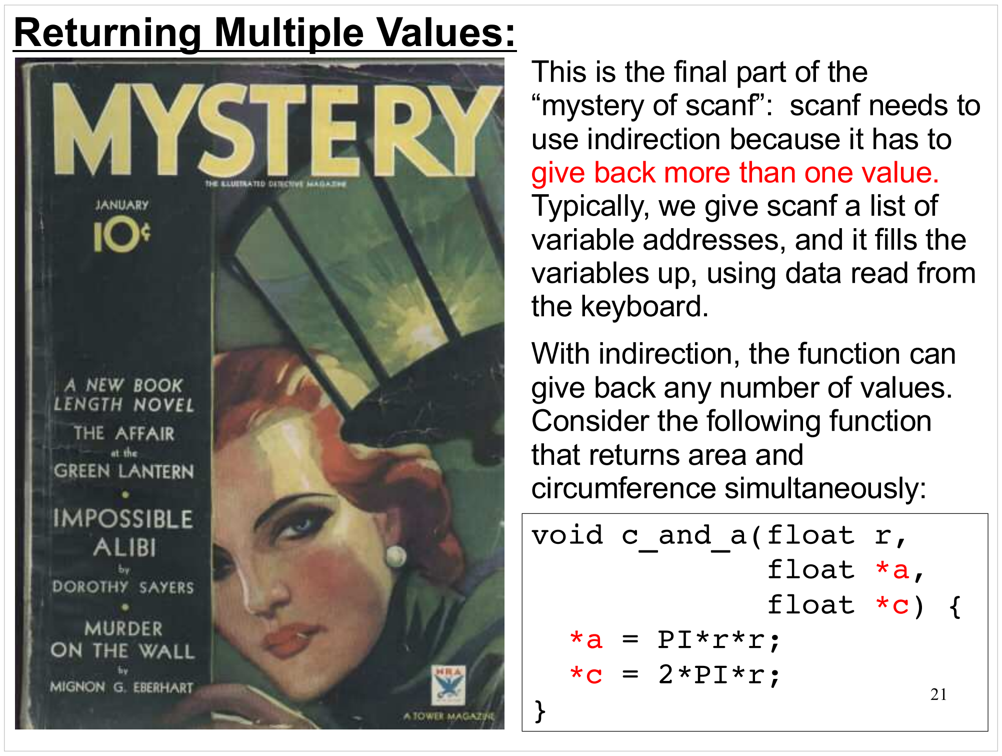
20

Notice that "getarea" doesn't have a return value. Instead of returning a value, as we've previously done with functions, "getarea" uses indirection to write its results into a specified memory location, where we can use them later.

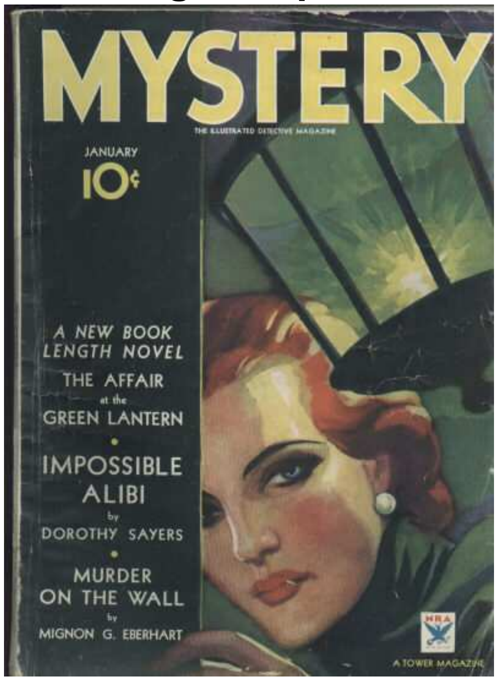
What's the advantage of this?

When a function gives its results through return, it can only give us one value. But, with this indirection technique, a function can send back as many different pieces of data as it wants to.

Consider the case of a function that calculates cross-products of three-dimensional vectors. The function would need to send back three values (the x,y, and z components of the cross-product vector). We can only give back one value with return, but we could pass the function the addresses of three variables, and have the function drop the components of the vector into those memory locations.



Returning Multiple Values:


MYSTERY
THE ILLUSTRATED DETECTIVE MAGAZINE
JANUARY
10¢
A NEW BOOK LENGTH NOVEL
THE AFFAIR at the GREEN LANTERN
IMPOSSIBLE ALIBI by DOROTHY SAYERS
MURDER ON THE WALL by MIGNON G. EBERHART
A TOWER MAGAZINE

This is the final part of the "mystery of scanf": scanf needs to use indirection because it has to give back more than one value. Typically, we give scanf a list of variable addresses, and it fills the variables up, using data read from the keyboard.

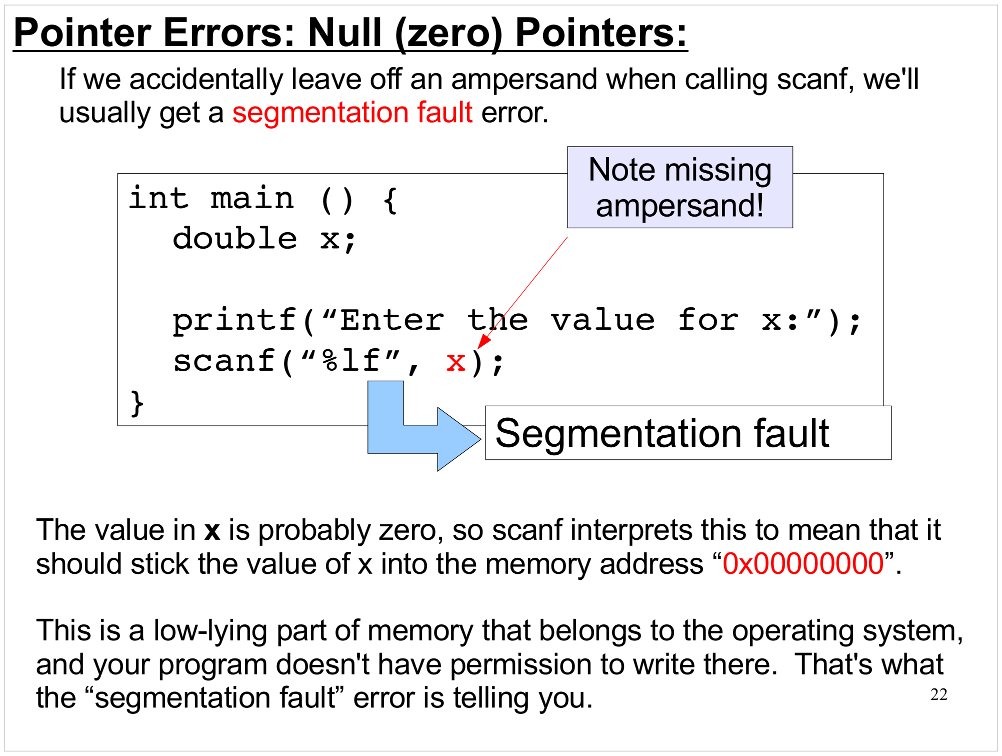
With indirection, the function can give back any number of values. Consider the following function that returns area and circumference simultaneously:

```
void c_and_a(float r,
             float *a,
             float *c) {
  *a = PI*r*r;
  *c = 2*PI*r;
}
```

There are actually a couple of other scanf mysteries that we'll look at later:

- Why don't we use an ampersand when we give scanf a character string?

- How can scanf (and printf) have a variable number of arguments?



Pointer Errors: Null (zero) Pointers:

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

```
int main () {
  double x;

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

Note missing ampersand!

Segmentation fault

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.

If we give scanf an arbitrary value as a memory location, it's a good bet that this isn't a section of memory that your program is allowed to write into. This will generate a segmentation fault.

Pointer Errors: Mis-casting:

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

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

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

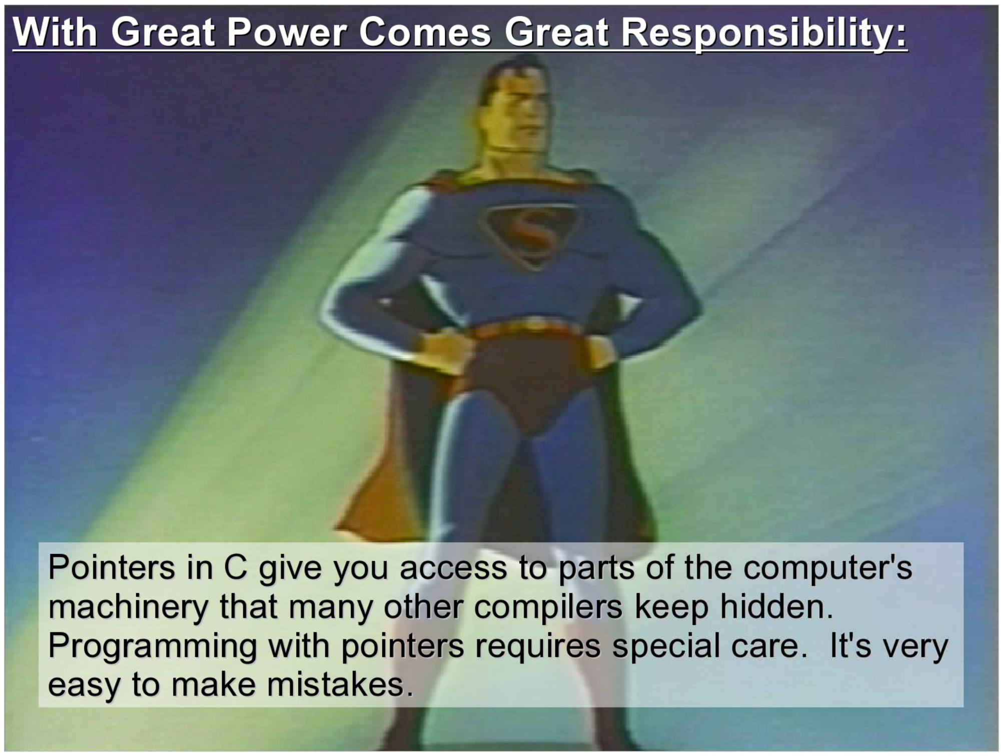
0.00000000

Remember that different variable types have very different representations in memory. Since each pointer variable has a type, the compiler tries to keep you from pointing to the wrong type of data.

Override this at your peril.

int i = 4;	00000100 00000000 00000000 00000000
float f = 4;	00000000 00000000 10000000 01000000
char c = '4';	00110100

With Great Power Comes Great Responsibility:

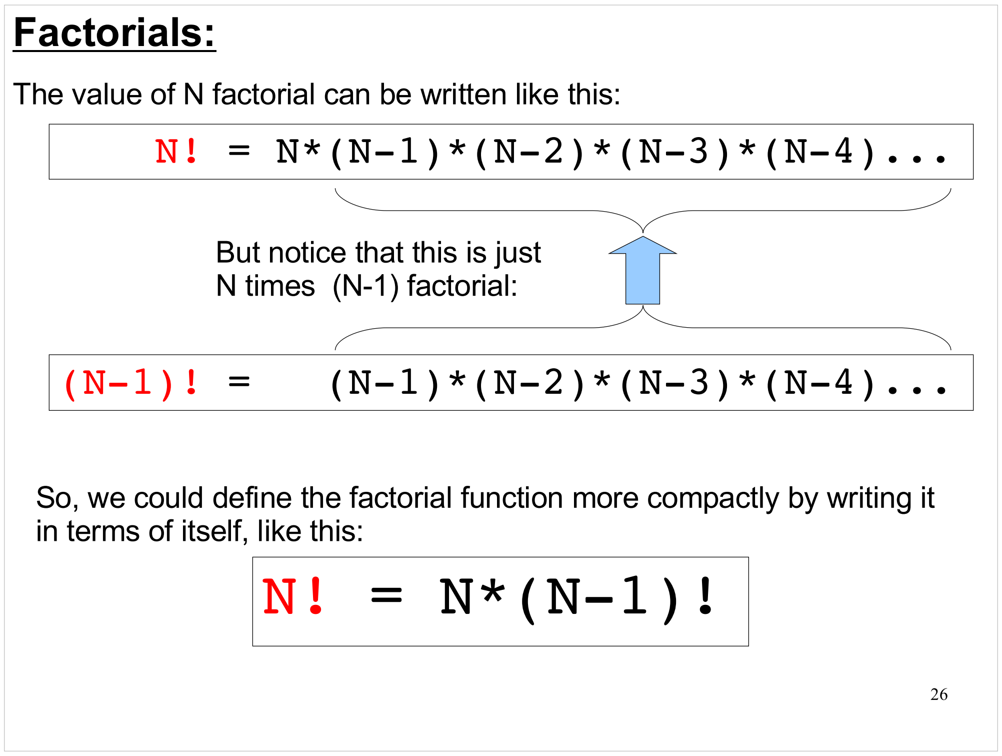


Pointers in C give you access to parts of the computer's machinery that many other compilers keep hidden. Programming with pointers requires special care. It's very easy to make mistakes.

Part 4: Recursion

Imagine you're sitting in a barber shop with one mirror in front of you and another behind you. In the mirror in front of you, you see the mirror behind, and contained within it you see a smaller image of the mirror in front. In other words, the image you see in front of you depends upon...the image you see in front of you!

This is called recursion, and in C, functions can do it.



Factorials:

The value of N factorial can be written like this:

```
    N!  = N*(N-1)*(N-2)*(N-3)*(N-4)...
```

But notice that this is just
N times (N-1) factorial:

```
(N-1)!  =   (N-1)*(N-2)*(N-3)*(N-4)...
```

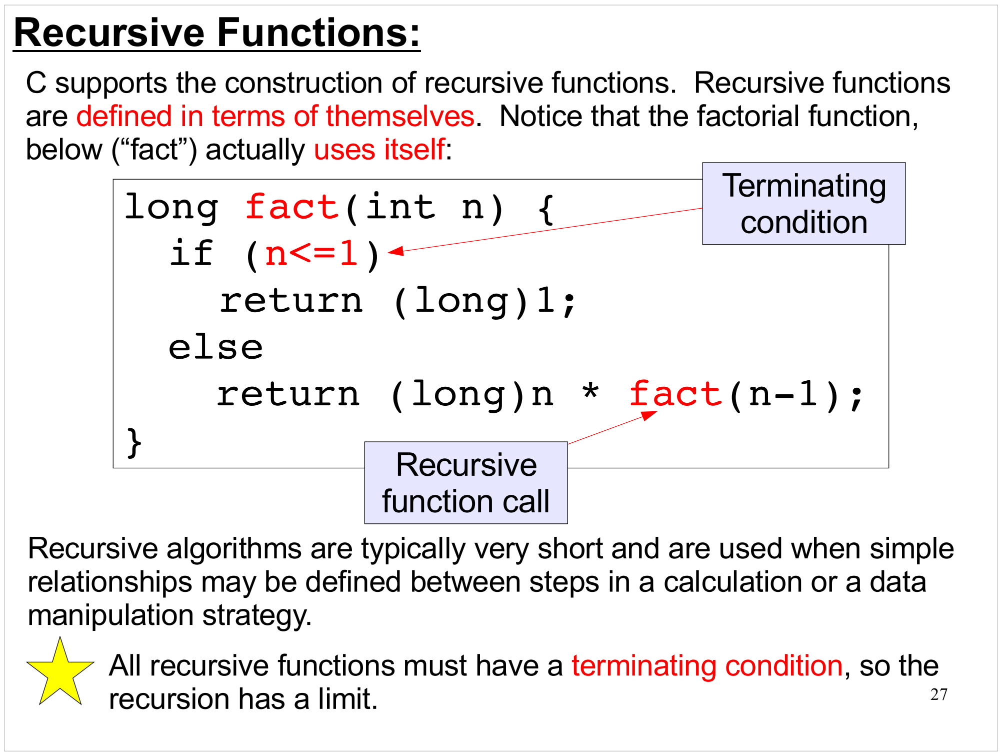
So, we could define the factorial function more compactly by writing it in terms of itself, like this:

```
N!  = N*(N-1)!
```

26

In the bottom line, we've defined the factorial operation recursively (i.e., in terms of itself). This is a nice compact form, but what use is it?

As it turns out, C allows functions to refer to themselves. Because of that, we can use the definition above to write a very compact "factorial" function in C, using recursion.



Recursive Functions:

C supports the construction of recursive functions. Recursive functions are defined in terms of themselves. Notice that the factorial function, below ("fact") actually uses itself:

```
long fact(int n) {
  if (n<=1)
    return (long)1;
  else
    return (long)n * fact(n-1);
}
```

Terminating condition
Recursive function call

Recursive algorithms are typically very short and are used when simple relationships may be defined between steps in a calculation or a data manipulation strategy.

All recursive functions must have a terminating condition, so the recursion has a limit.

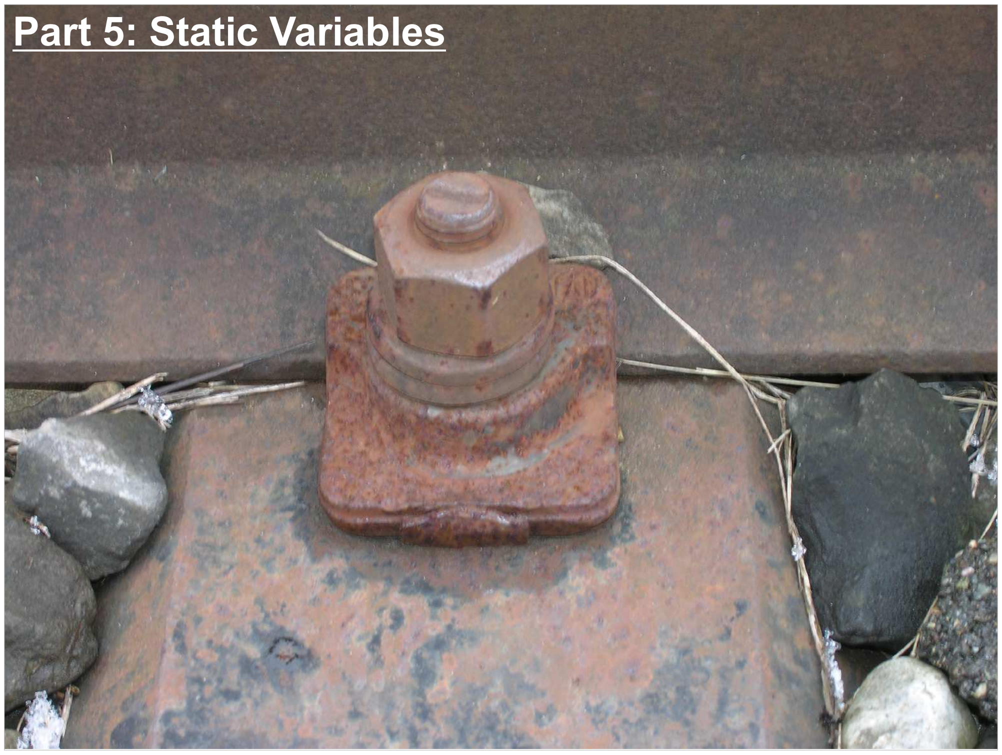
Why "<="? Because 0! is defined to be 1. Someday we may want to use this function to give us the factorial of zero.

We use "long" integers here because factorials can get very large.

Without a terminating condition, the recursion would continue to go deeper and deeper, infinitely, until all available resources were exhausted and the program crashed.

Try working through this function by hand, starting with fact(3).

Part 5: Static Variables



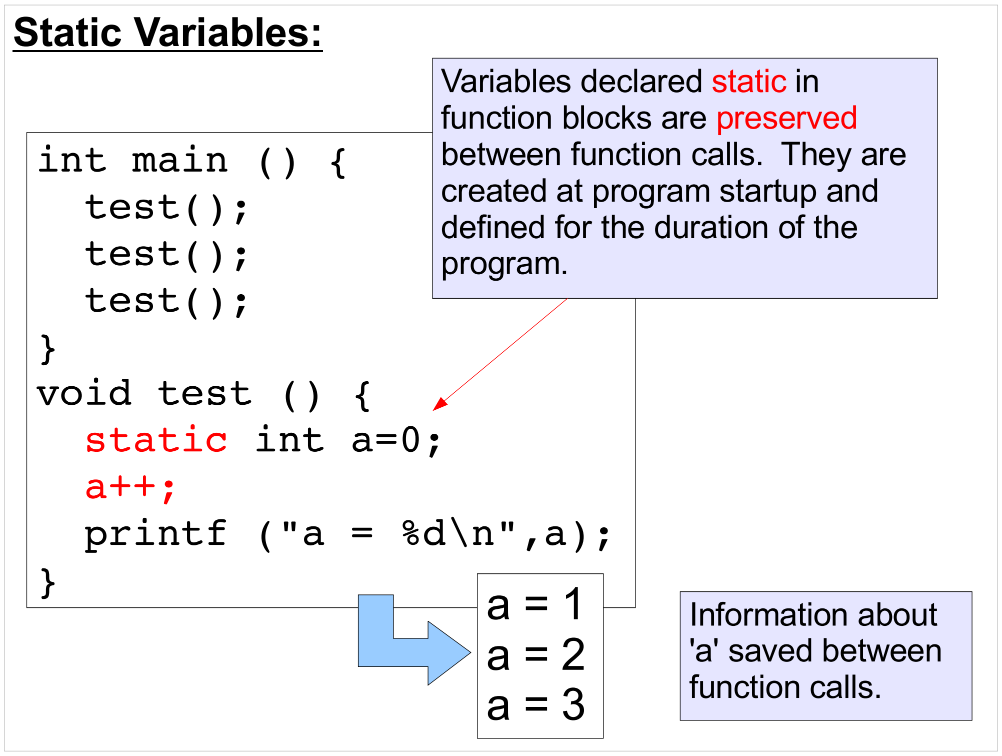
Ephemeral Local Variables:

```
int main () {
  test();
  test();
  test();
}
void test () {
  int a=0;
  a++;
  printf ("a = %d\n",a);
}
```

By default, variables declared in function blocks are ephemeral. They are created when the function is entered, and destroyed when the function is exited.

```
a = 1
a = 1
a = 1
```

No information about 'a' is saved between function calls. 'a' is recreated, starts out fresh, each time test() is called.



Static Variables:

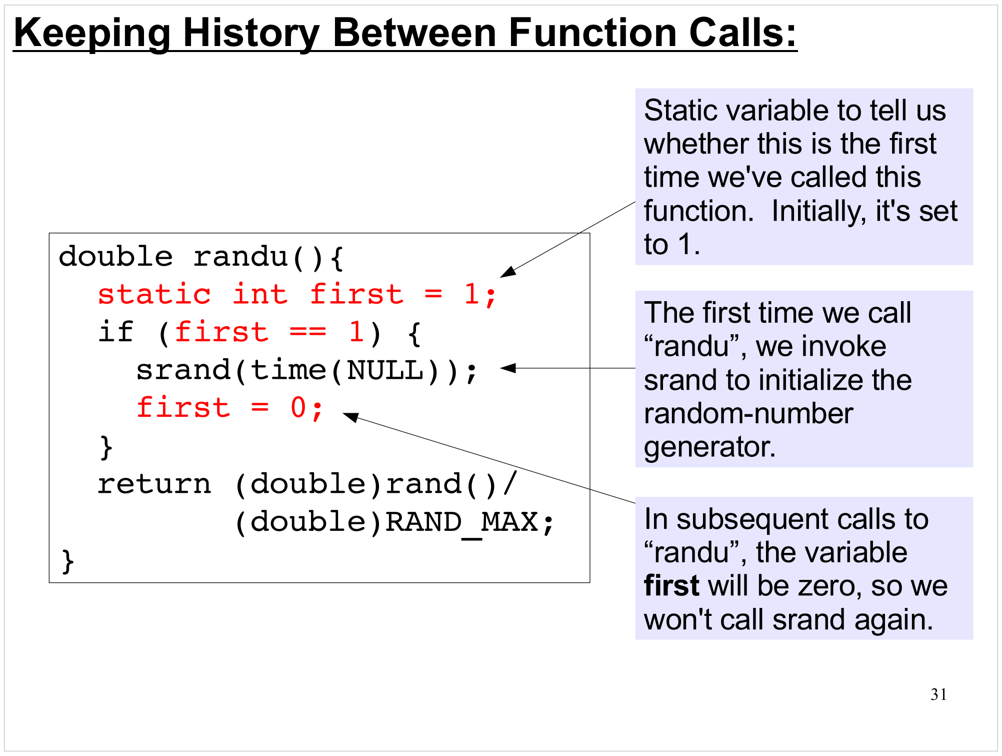
```
int main () {
  test();
  test();
  test();
}
void test () {
  static int a=0;
  a++;
  printf ("a = %d\n",a);
}
```

Variables declared static in function blocks are preserved between function calls. They are created at program startup and defined for the duration of the program.

```
a = 1
a = 2
a = 3
```

Information about 'a' saved between function calls.

The only change from the previous slide is that now we've added the word “static”.



Keeping History Between Function Calls:

```
double randu(){
  static int first = 1;
  if (first == 1) {
    srand(time(NULL));
    first = 0;
  }
  return (double)rand()/
         (double)RAND_MAX;
}
```

Static variable to tell us whether this is the first time we've called this function. Initially, it's set to 1.

The first time we call “randu”, we invoke srand to initialize the random-number generator.

In subsequent calls to “randu”, the variable **first** will be zero, so we won't call srand again.

Here's a common use for static variables. We check to see if we've already called “srand”. If not, we call it.

By doing it this way, we avoid the possibility of forgetting to call srand in our main function.

Other Uses for Static Variables:

- keep track of how many times a function is called,
- keep track of previous parameters sent to a function,
- keep track of a previous result of function.

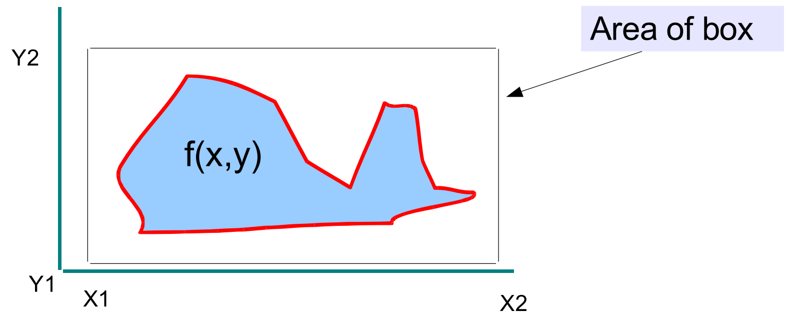
Part 6: Random Numbers

Early on, students of probability and statistics were treated like second-class citizens by other mathematicians. There was a widespread belief that these subjects were only relevant to games, and that games weren't serious mathematics.

Now we know that random numbers, and the study of them through probability and statistics, can be highly valuable tools. Let's look at one example.

Integrating Over an Arbitrary Area:

Say we have some horrible function in two dimensions, like the one shown below, and we want to calculate the area enclosed within the curve.



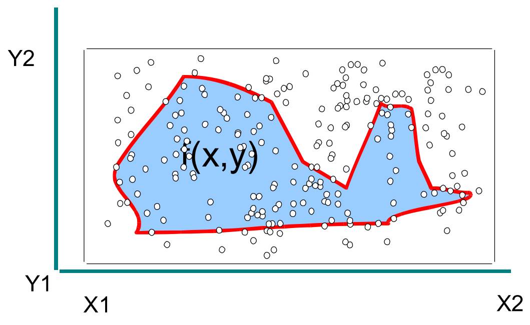
We could arrive at a very rough estimate of the area like this:

Define a box as shown, where
Y2 >= Max Y value of f(x,y) in the range X1 <= x <= X2
Y1 <= Min Y value of f(x,y) in the range Y1 <= y <= Y2

Area of the box = (X2-X1)*(Y2-Y1) > Area enclosed by the function

Obviously, the area of the box is a poor estimate of the area of the whale-shaped blue region. How can we improve our estimate?

Using Random Numbers to Estimate the Area:



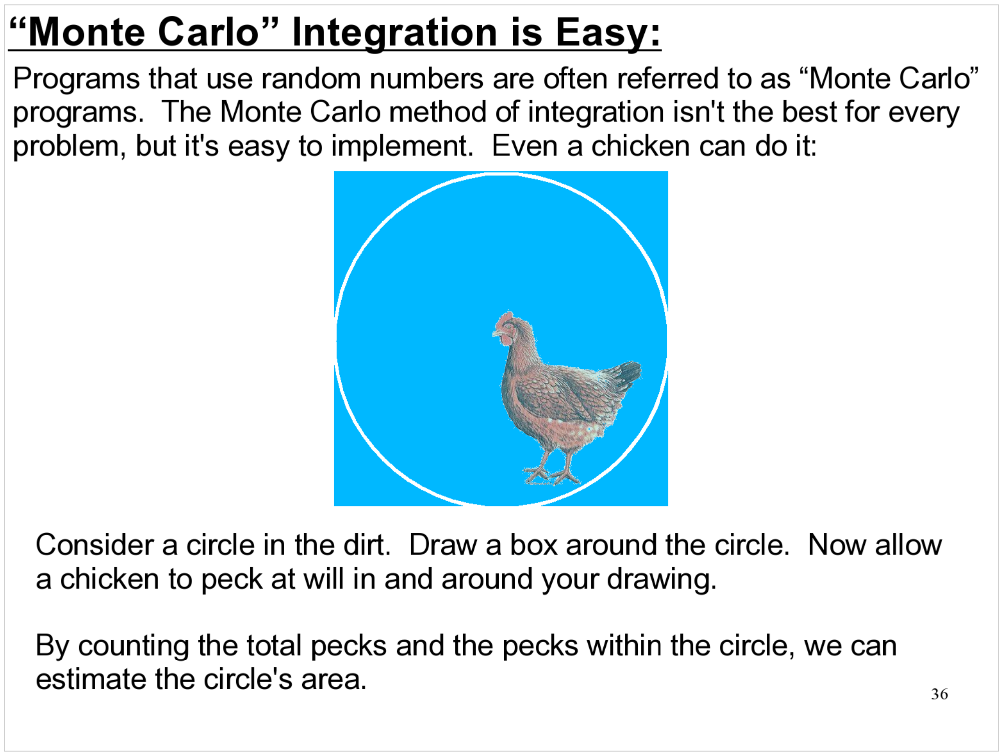
If we generate random points within the box, we can use them to get a better estimate of the area:

$$r = \frac{\text{Points inside shape}}{\text{Total points}}$$

$$A_{\text{shape}} \approx A_{\text{box}} * r$$

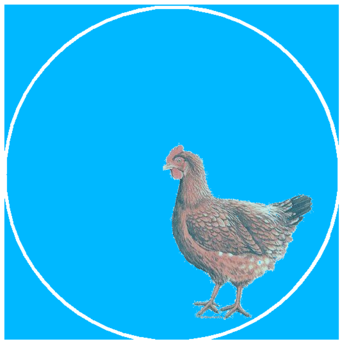
As we generate more and more points we asymptotically approach the exact answer.

```
n_in = 0;
for (i=0 ; i< num_trials <i++) {
   x = (double)rand()/RAND_MAX * (X2-X1) + X1;
   y = (double)rand()/RAND_MAX * (Y2-Y1) + Y1;
   if (in_fcn(x, y)) n_in ++;
}
area = (X2-X1)*(Y2-Y1) * n_in / num_trials;
```



“Monte Carlo” Integration is Easy:

Programs that use random numbers are often referred to as “Monte Carlo” programs. The Monte Carlo method of integration isn't the best for every problem, but it's easy to implement. Even a chicken can do it:

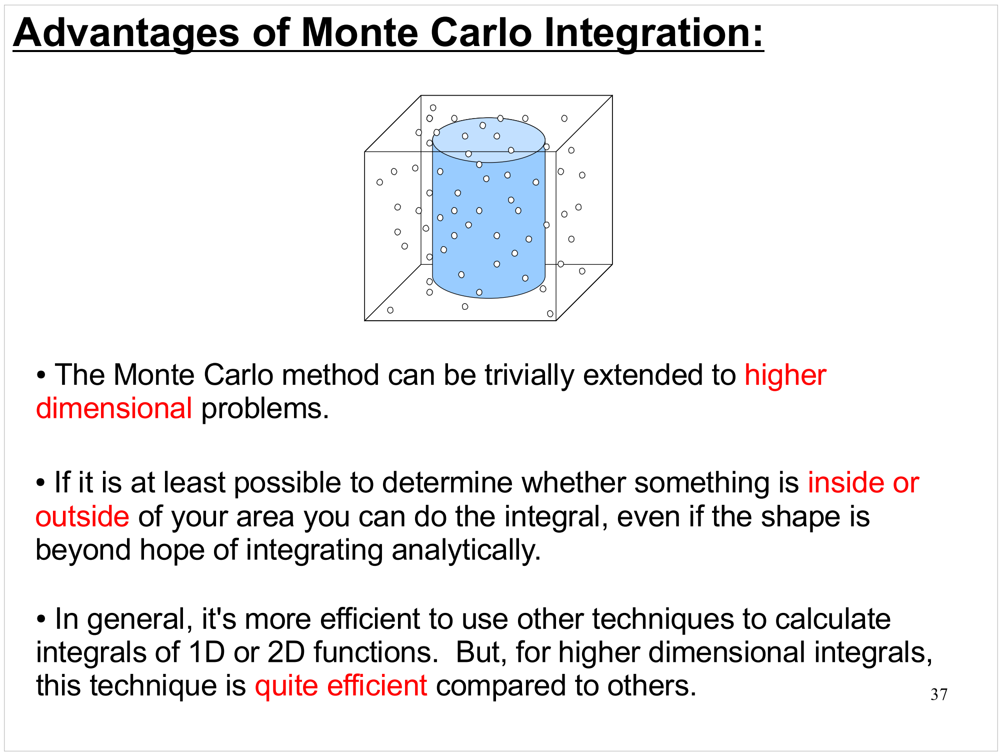


Consider a circle in the dirt. Draw a box around the circle. Now allow a chicken to peck at will in and around your drawing.

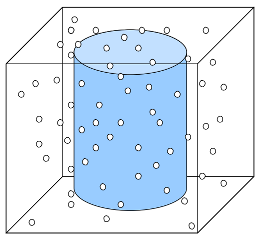
By counting the total pecks and the pecks within the circle, we can estimate the circle's area.

One way to estimate the uncertainty of your Monte-Carlo-estimated area is to run your calculation multiple times, then use the average of your trials as your answer and use the standard deviation of the mean as an estimate of your uncertainty.

If efficiency is not an issue, the MC technique is attractive, because of its simplicity.



Advantages of Monte Carlo Integration:



• The Monte Carlo method can be trivially extended to higher dimensional problems.

• If it is at least possible to determine whether something is inside or outside of your area you can do the integral, even if the shape is beyond hope of integrating analytically.

• In general, it's more efficient to use other techniques to calculate integrals of 1D or 2D functions. But, for higher dimensional integrals, this technique is quite efficient compared to others.

Consider the case of an 11-dimensional integral in String Theory. This might be utterly impossible to solve analytically, but by generating thousands of sets of 11 coordinate values, we could estimate its value by Monte Carlo methods.

Part 7: What We Know About C Now

C

Now for a quick review of what we've learned so far about the C language.

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.

Input/Output:

- 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.

Files:

- 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.

Conditionals:

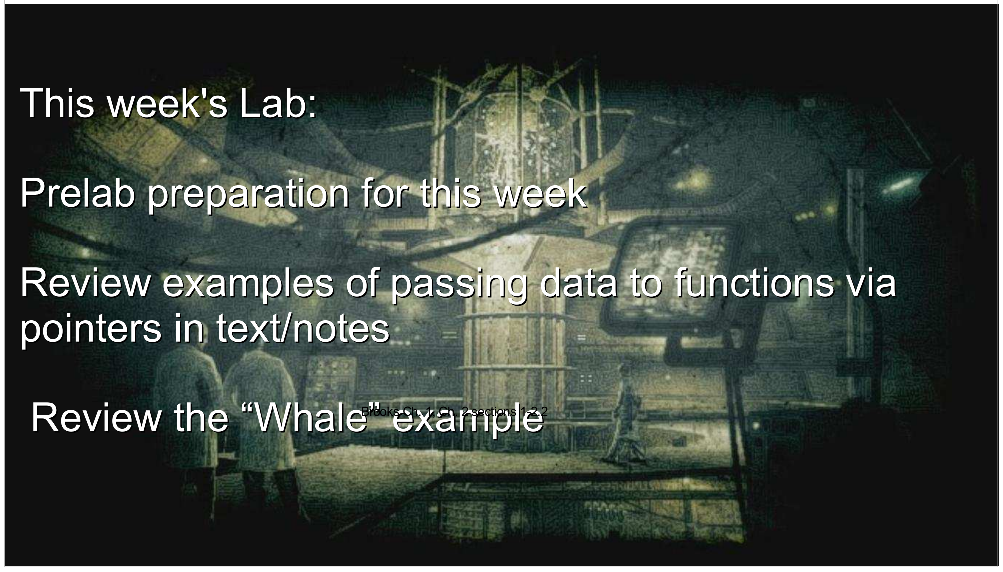
- if statements:
  - Only do something if a given statement is true.
- if/else statements:
  - Pick between two options, depending on whether a given statement is true or false.
- if/else if/else statements:
  - Choose based on the first true expression you find.
- switch statements:
  - Choose between 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.

Next Time:

- Arrays
- Making your own libraries
- Pointers to functions, etc.



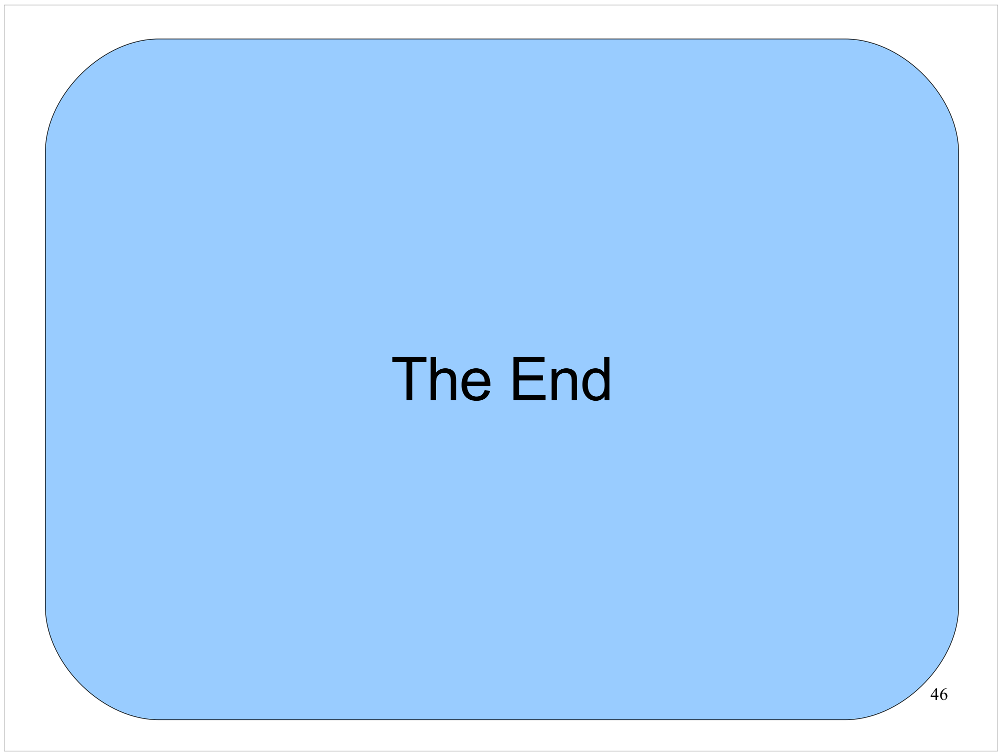
This week's Lab:

Prelab preparation for this week

Review examples of passing data to functions via pointers in text/notes

Review the “Whale” example

Brooks Ch. 1, Ch. 2 sections 1-2.2



The End

Thanks!