Pointers and `scanf()`

Steven R. Bagley

Recap

- Programs are a series of statements
- Defined in functions
- Can call functions to alter program flow
- `if` statement can determine whether code gets run
- Loops can execute code multiple times

Function calls

- Function calls can take parameters
- The parameter passes the *value* not the variable
- This means that if the function alters the value of the parameter the original variable won't be affected
- Referred to as *pass-by-value*

```
void func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
}

int main(int argc, char *argv[])
{
   int a = 42;

   printf("In main, a = %d\n", a);
   func(a);
   printf("In main again, a = %d\n", a);
}
```

Run this program -- should print out 42, 43, 42

```
int a = 42;
func(a);
```

```
int a = 42;
func(a);
```

```
int a = 42;
func(a);
```

```
int a = 42;
func(a);



void func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
}
```

a 42

```
int a = 42;
func(a);
```



```
void func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
}
```

```
int a = 42;
func(a);
```

a 42

```
void func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
}
```

x 42

```
int a = 42;
func(a);
```

a 42

```
void func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
}
```

x 43

```
int a = 42;
func(a);



void func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
}
```

a 42

```
int a = 42;
func(a);
```

Parameters

- When a function is called, a *copy* of the value is passed
- Copy placed in the parameter variable
- Functions can alter the value of this parameter
- But because it only contained a *copy* it doesn't affect the original

Return Values

- Functions can return a single value
- Use `return` keyword
- The return value is *copied* and passed back to the original function
- Where it does something with it (assign to variable, pass as another parameter, etc…)

```
int a = 42;
a = func(a);
```

```
int a = 42;
a = func(a);
```

```
int a = 42;
a = func(a);
```

```
int a = 42;
a = func(a);



int func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
  return x;
}
```

a 42

```
int a = 42;
a = func(a);
```

a 42

```
int func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
  return x;
}
```

x

```
int a = 42;
a = func(a);
```

a 42

```
int func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
  return x;
}
```

x 42

```
int a = 42;
a = func(a);
```

a 42

```
int func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
  return x;
}
```

x 43

```
int a = 42;
a = func(a);



int func(int x)
{
  x++;
  printf("Printing from func, x = %d\n", x);
  return x;
}
```

a 42

```
int a = 42;
a = func(a);
```

```
int a = 42;
a = 43;
```

a 42

the value of 43 returned by func is assigned into a -- equivalent to a = 43;

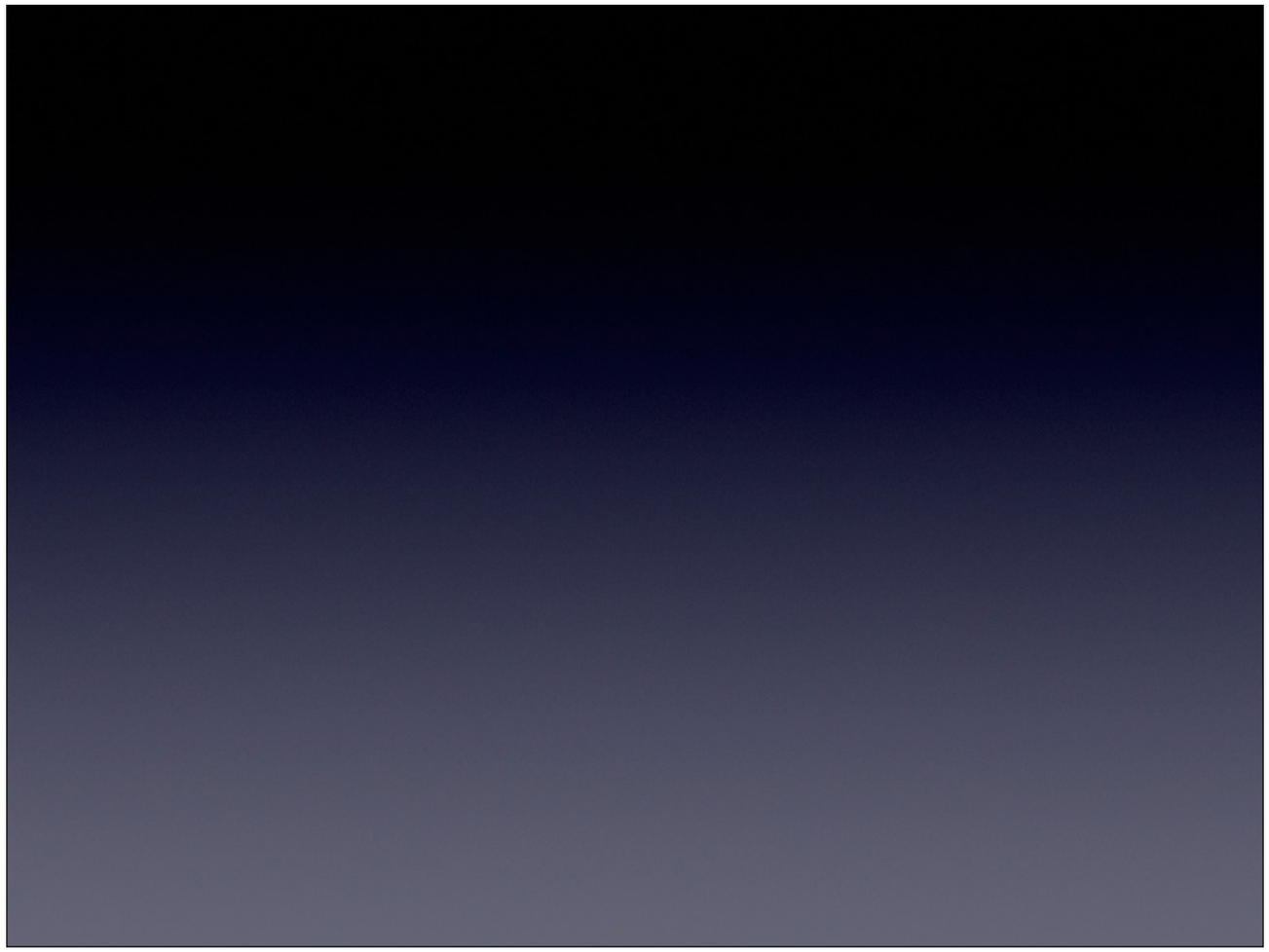
```
int a = 42;
a = 43;
```

a 43

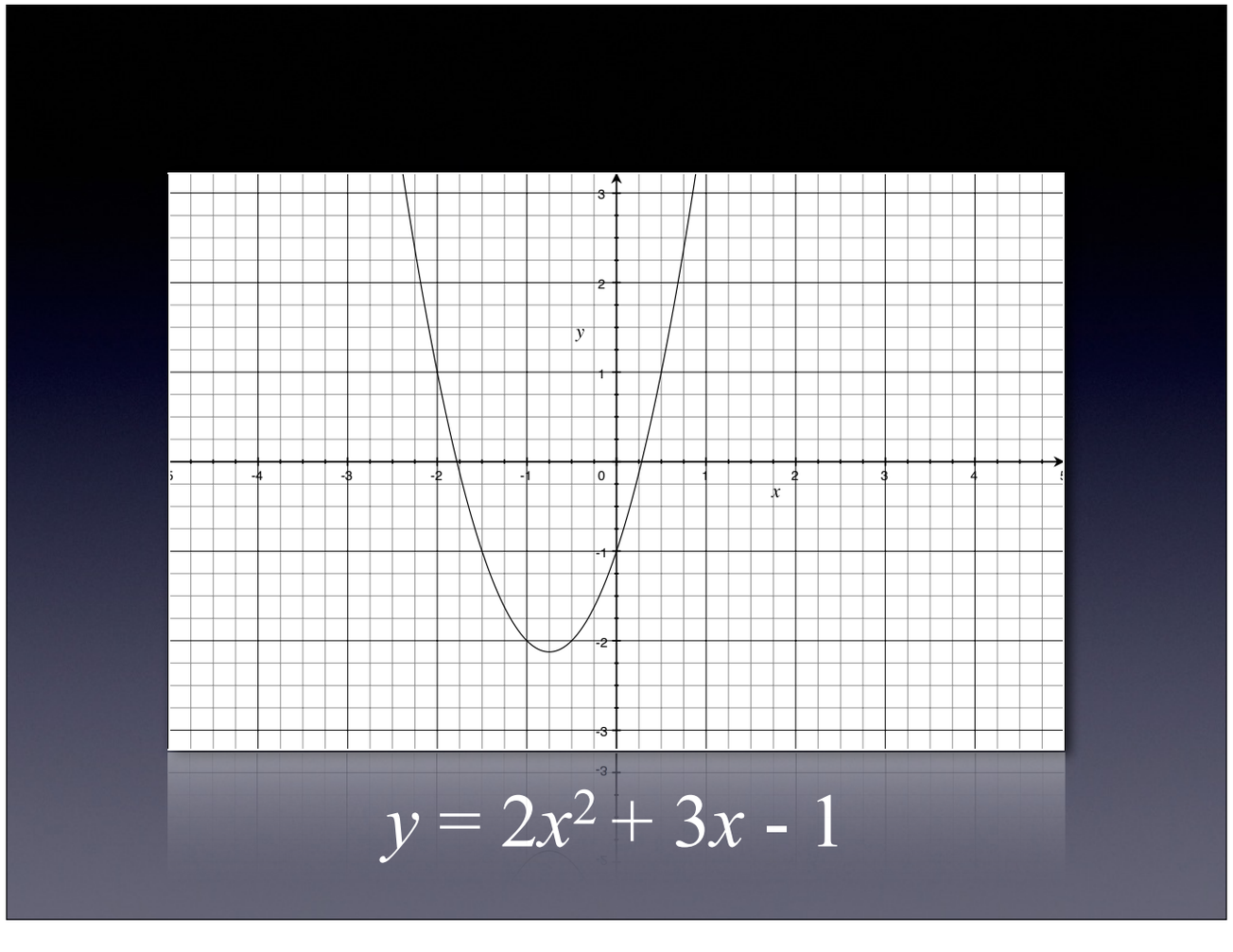
the value of 43 returned by func is assigned into a -- equivalent to a = 43;

Return Values

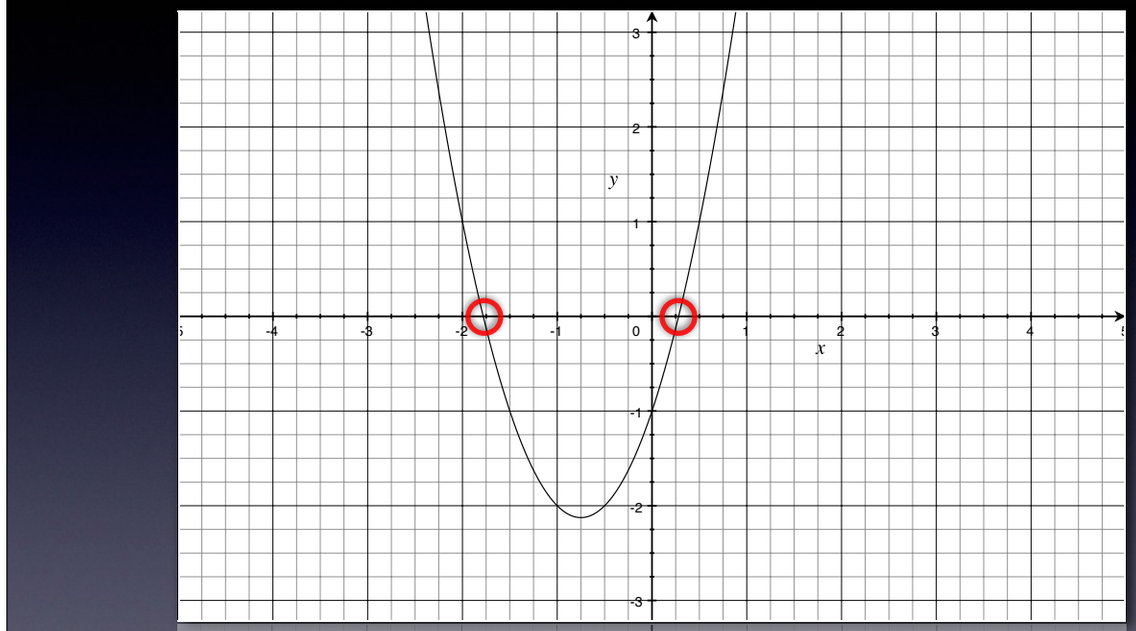
- What if you need to return more than two values?
- For example, solving a quadratic equation
- Each quadratic equation has two possible answers
- How would we write a function to calculate these?



Solving the equation means finding the two values of x which mean y is 0



Solving the equation means finding the two values of x which mean y is 0



$$y = 2x^2 + 3x - 1$$

Solving the equation means finding the two values of x which mean y is 0

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
double solveQuadratic(double a, double b, double c)
{
    double x;

    x = (-b + sqrt(b*b - 4*a*c))/(2*a);
    return x;
}
```

sqrt() calculates the square root, and requires to include math.h

This calculates the positive root, but what about the negative root

Multiple returns

- Can sometimes get around it by writing two functions
- But we end up calculating things twice…
- In other cases, you can't write it as multiple functions e.g. `scanf ( )`
- Fortunately, C provides with a solution…

```
double solveQuadraticNeg(double a, double b, double c)
{
    double x;

    x = (-b - sqrt(b*b - 4*a*c))/(2*a);
    return x;
}
```

sqrt() calculates the square root, and requires to include math.h

This calculates the negative root

```
double solveQuadratic(double a, double b, double c)
{
    double x;

    x = (-b + sqrt(b*b - 4*a*c))/(2*a);
    return x;
}
```

sqrt() calculates the square root, and requires to include math.h

This calculates the positive root, but what about the negative root

Multiple returns

- Can sometimes get around it by writing two functions
- But we end up calculating things twice…
- In other cases, you can't write it as multiple functions e.g. `scanf ( )`
- Fortunately, C provides with a solution…

Value, or where to find it?

- So far, we've been passing values into parameters
- But we could also pass something else
- The *address* of where to find the value
- All variables are stored in the computer's memory at a specific address

Variables and Memory

- Internally, C knows the location of where each variable is stored in memory
- When accessed it looks up the value from the memory location
- But we can also do this manually if we know the location (or address)

Point at the value

- If we know the address of the variable, then we can modify its contents
- Will look at this in more detail in CSA
- So if we pass the function the address of the variable, rather than its value it could alter the original variable

Pointers

- Tend to call variables that contain the address of another variable, a *pointer*
- The variable points at some value (somewhere over there)
- Defined by using a `*` when declaring the variable

```
int *p;
```

p is a variable that points to an int (somewhere in memory)

http://www.cdecl.org

Using Pointers

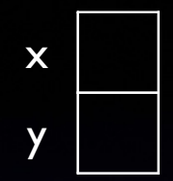
- Can assign a value to `p` just as before
- But this sets where `p` points
- To access what `p` points at, you need to place a `*` before it, e.g.

```
printf("%d", *p);
*p = 42;
```

Accessing the value pointed to is sometimes called dereferencing
Print the integer that is pointed at by p
Set the value pointed to by p to be 42

Address of

- Need to assign the pointer to point to something (like another variable)
- Do this by using the *address-of* operator, `&`
- So `&x` gets the address of the variable, `x`
- Note the types should match, an `int` pointer should point to an `int` and so on…

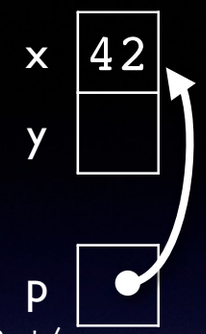



```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf(“%d”, *p); /* print value pointed to by p, 42 */

x = 31;
printf(“%d”, *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf(“%d”, x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```

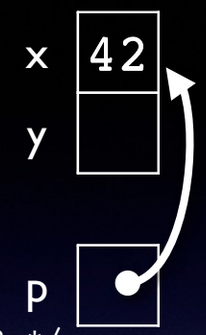
```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```

```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```

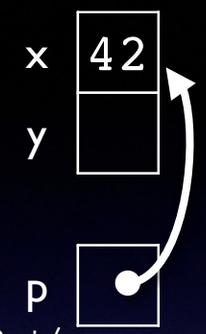
```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



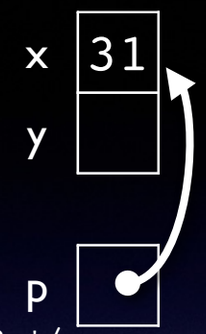
```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



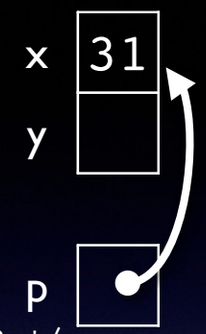
```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



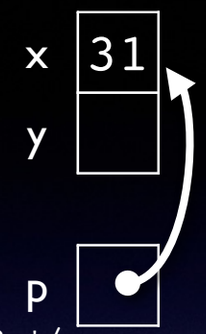
```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



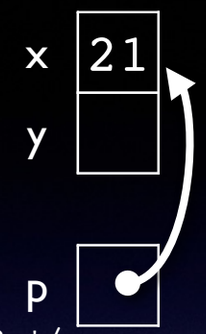
```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



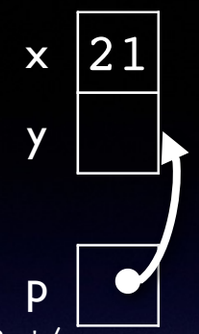
```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



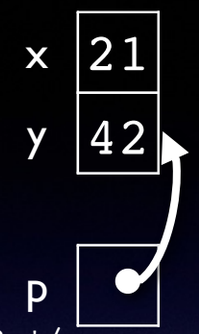
```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



```
int x,y; /* Declare two integers, x and y */
int *p; /* Declare a pointer to an integer */

x = 42;

p = &x; /* Set p to point at the address of x */
printf("%d", *p); /* print value pointed to by p, 42 */

x = 31;
printf("%d", *p); /* print value pointed to by p, 31 */

*p = 21; /* Set the int that p points at to 21 */
printf("%d", x); /* prints 21 */

p = &y; /* p now points at y */
*p = 42; /* value of y changed, but x left the same*/
```



Returning Multiple Values

- Can use this to return multiple values in our quadratic function
- Takes five parameters now
- The three co-efficients, `a`, `b`, and `c`
- Two pointers, `*pos`, and `*neg` which point to where to store the result
- What do we `return`?

Return nothing in this case, could also return a status value, or one of the answers

```
double solveQuadratic(double a, double b, double c,
                      double *pos, *neg)
{
    double x;
    double s, twoa;

    twoa = 2 * a;
    s = sqrt(b*b - 4 * a * c);

    *pos = (-b + s) / twoa;
    *neg = (-b - s) / twoa;
}
```

sqrt() calculates the square root, and requires to include math.h

Calculates both roots

Pointers

- Pointers are incredibly powerful
- We shall see them used all over the place
- Both visibly like here, and behind the scenes (e.g. in Java)
- This is only scratching the surface
- We shall return to them