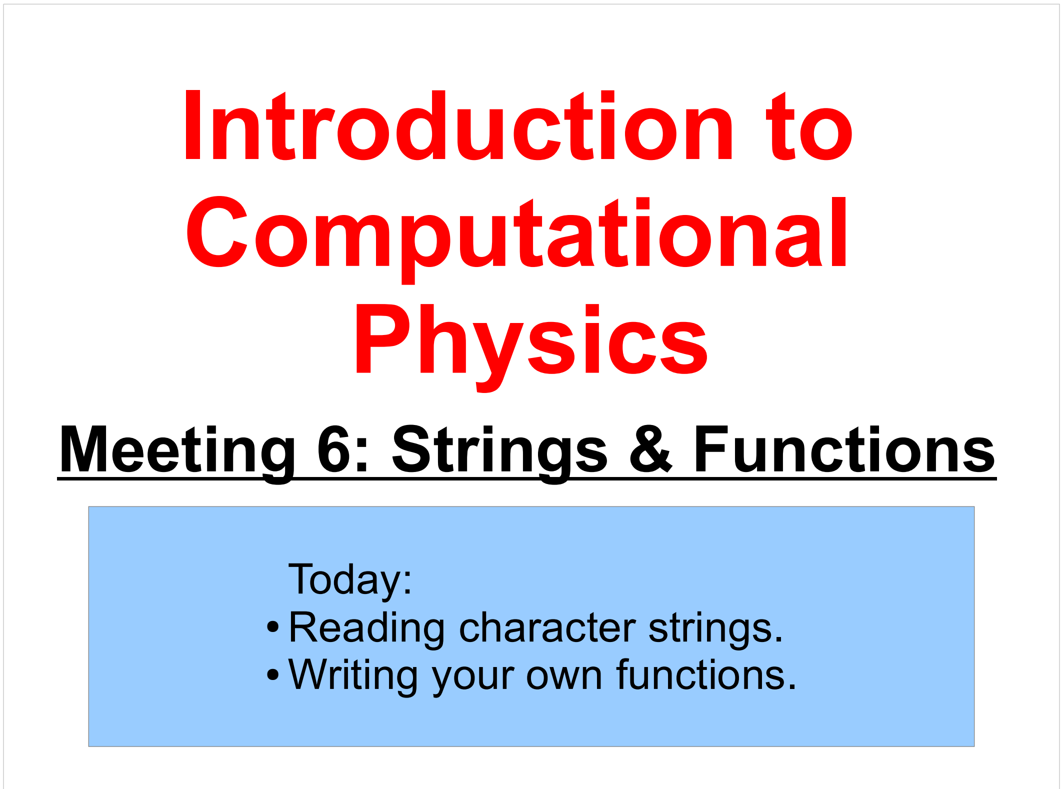


Welcome back!

Remember that you can try out these programs using your account on Galileo. For instructions on how to use Galileo, see:

http://galileo.phys.virginia.edu/compfac/courses/comp_intro/connecting.html

Today we'll be looking at arrays, which are lists of related variables. We'll be using them to make vectors and histograms.

Let's get started!

Last Week's Practice Problem:

What if we didn't zero out out counters in the histogram program? Try writing, compiling and running this program to see what happens:

```
#include <stdio.h>
int main () {
  int data[10];
  int i;

  for ( i=0; i<10; i++ ) {
    printf ("%d %d\n", i, data[i]);
  }

}
```

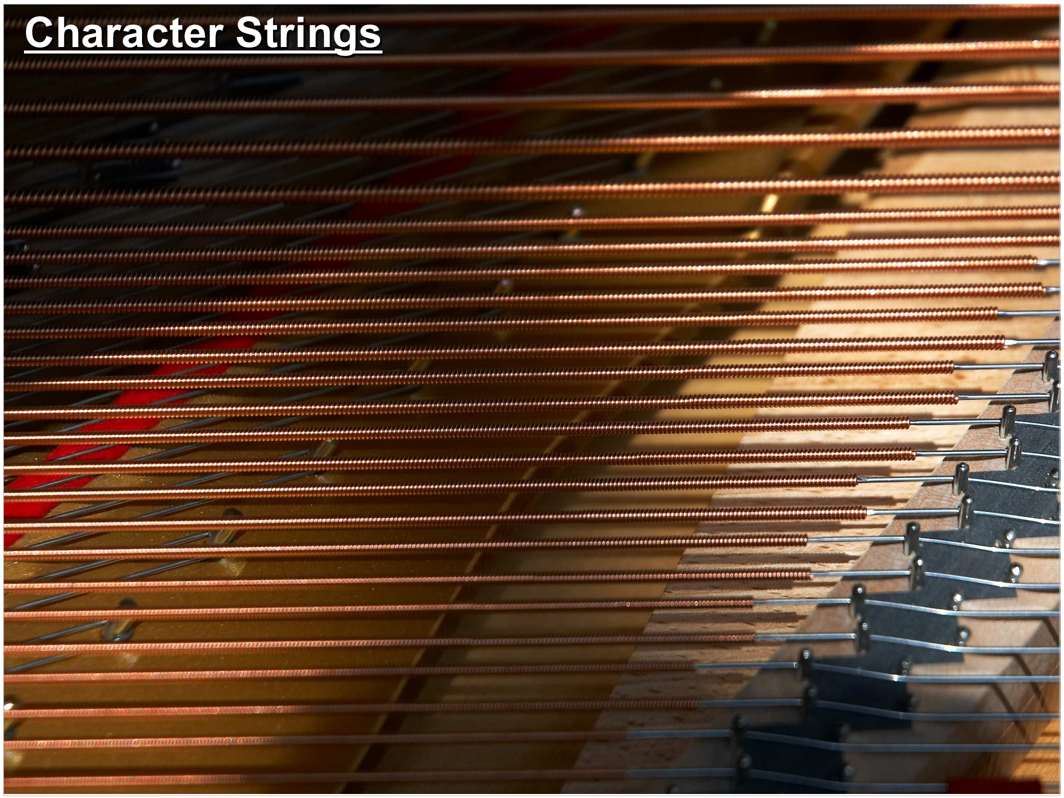
```
0 -1074441112
1 134513457
2 6643456
3 0
4 -1074441064
5 134513897
6 2759781
7 7304660
8 -1074441080
9 3989492
```

If we don't set the array elements to zero, they just contain whatever random junk was left over in that memory location from the last program that was run!

Do you see why it's important to set counters to zero before using them?

Why does this happen? It's because C has reserved a space in memory to hold the 10 elements of “data”, but C doesn't automatically clear the contents of that memory for you.

Never assume that a variable has a value of zero when you start using it. If it needs to be zero, you should set it to that explicitly.



As we've noted before, character strings are just arrays of characters. Let's look at them a little more closely.

What are Character Strings?

As we've seen, there are several different types of variables in C. We've used “int” for integers and “double” for floating-point numbers. Another type is “char”. A “char” variable can hold one character (letter, number, punctuation, etc.)

We can use an array of “char” variables to hold a chunk of text. We call such an array a “character string”.

```
char name[10] = “NICOLE”;
```

As we'll see, we need to remember that a character string is an array instead of a single value. Fortunately C provides some standard functions to make it easier to deal with character arrays.

In the following we're going to write several tiny programs that illustrate some problems you might run into when you use character strings in your programs. In each case, we'll show you the “right” way to do it.

Defining Character Strings:

As we saw last time, C variables to contain text. In programming, we usually refer to a chunk of text as a “character string”. Character strings in C are just arrays of characters. They can be initialized in several ways:

```
#include <stdio.h>

int main () {
  char string0[10];
  char string1[20] = "this is a test.";
  char string2[20] = {'t','h','i','s',' ',
                      'i','s',' ','a',' ',
                      't','e','s','t','.'};
  char string3[] = “this is a test.”;

  printf ("%s\n",string1);
  printf ("%s\n",string2);
  printf ("%s\n",string3);
}
```

Empty string, for filling in later.

Works, but why do it?

We can let the compiler figure out the length.

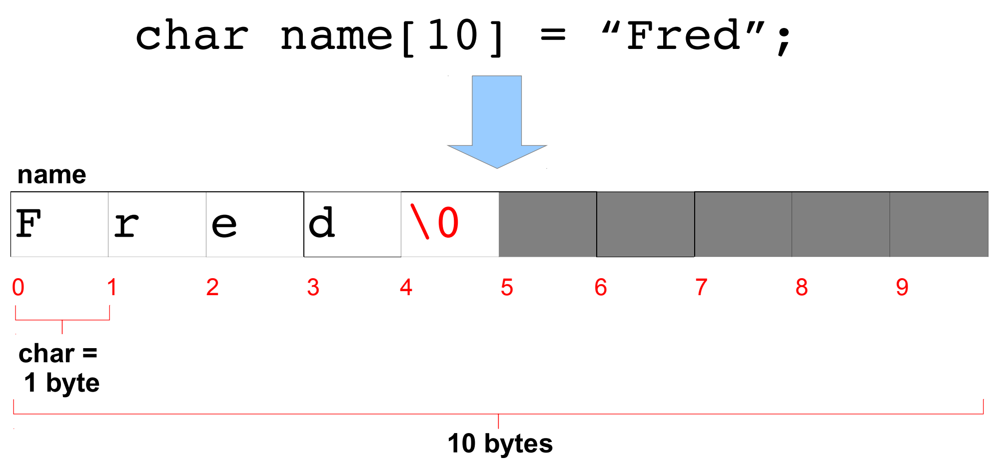
The %s format is for strings.

As you can see in “string2”, character strings really are arrays of single characters, and you can look at (and set) the individual characters if you want to. But usually there's no reason to do it that way.

As we've seen before, there's a special format specifer (%s) for strings. You can use this to write them out with printf or fprintf.

Take care when using scanf with strings, though. We'll see why soon.

How Strings are Stored in Memory:



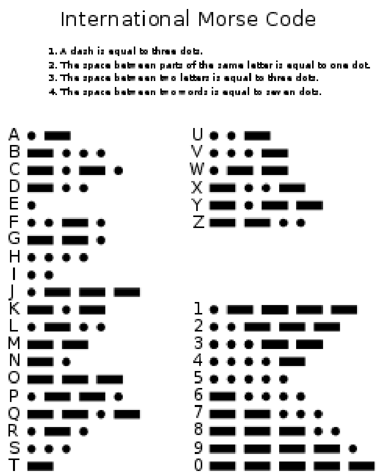
"\0" is a special non-printable character called NUL. It marks the end of the string. When you define a character array, you always need to leave room for the NUL at the end, so the length of the array should be at least 1 more than the biggest string you want to store.

The NUL tells printf where to stop, for example, when it's printing out the string. Without this indicator, printf would just keep printing whatever random garbage is in the other elements of the character array.

In fact, since C forgets how long an array is as soon as you define it, printf would just keep on printing bytes until it happened to find a NUL somewhere in memory or caused the program to crash, since it wouldn't know where the character array ended.

Character Encoding:

1840s:



1963:

American Standard Code for Information Interchange (ASCII)

01000001	A	01010101	U
01000010	B	01010110	V
01000011	C	01010111	W
01000100	D	01011000	X
01000101	E	01011001	Y
01000110	F	01011010	Z
01000111	G		
01001000	H	00110001	1
01001001	I	00110010	2
01001010	J	00110011	3
01001011	K	00110100	4
01001100	L	00110101	5
01001101	M	00110110	6
01001110	N	00110111	7
01001111	O	00111000	8
01010000	P	00111001	9
01010001	Q		
01010010	R	...etc.	
01010011	S		
01010100	T		

00000000 = “NUL”

Prior to the 1960s, the most widespread way of communicating data electronically was morse code. When a telegram was sent, its text was encoded in morse code and transmitted through air or a wire to its destination, where it was decoded back into text.

Morse code was fine for human telegraphers, but it was clumsy for computers. In the 1960s the “American Standards Association” published a new, more computer-friendly way of transmitting text. This was called the American Standard Code for Information Interchange (ASCII).

In ASCII, each character is represented by 8 bits of information (1 byte). When you store text in a file on disk, the text is stored as ASCII characters. ASCII characters are also the way communications between a terminal (or pseudo-terminal) and a computer are encoded.

(Actually, other encodings like UTF-8 may be used these days, but the principle is the same. For simplicity, let's just assume everything is ASCII.)

Characters in Memory:

```
char day[] = "Tuesday";
```

Each character takes up one byte (8 bits) in memory. A character string is just an array of characters.

Here's what the string above would look like in the computer's memory:

	day
T	01010100
u	01110101
e	01100101
s	01110011
d	01100100
a	01100001
y	01111001
\0	00000000

Each character is represented by an 8-bit (1 byte) ASCII code.

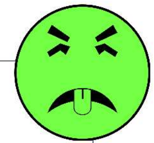
Notice again that the end of the string is indicated by the special NUL character, which has the ASCII code “00000000”.

The array named “day” has eight elements in this example. The eighth one holds the NUL.

The special character “NUL” is a non-printable character that's represented by a string of eight zeros in memory. We sometimes write it as “\0”.

The Wrong Way to Compare Strings:

The following won't work. Why?



```
#include <stdio.h>
int main () {

  char s[] = ”junk”;
  char t[] = ”junk”;

  if ( s == t ) {
   printf (“They match.\n”);
  } else {
   printf (“They don't match.\n”);
  }
}
```

Try it!

Why doesn't this work? Because “s” and “t” are arrays. Think about it: if we had two “double”s, x and y, we could compare their values with “if (x==y)”. Similarly, if we had two arrays of doubles , a[10] and b[10], we could compare two of their elements with “if (a[1] == b[1])”. But what would we mean if we typed “if(a==b)”?

It turns out that, in C, if you type just the name of an array, you get the memory address of the beginning of the array. Since “s” and “t” in the example above are two different arrays, each of which has its own allocated section of memory, each of them will have a different address. So, “if(s==t)” will never be true.

Apparently, that's not the right way to compare two strings.

The Right Way to Compare Strings:

The strcmp function (defined in string.h) compares two strings:

```
int strcmp(char* S1, char *S2);
```

Returns:

0	if S1 = S2
>0	if S1 > S2
<0	if S1 < S2

```
#include <stdio.h>
#include <string.h>
int main () {
  char s[] = "junk";
  char t[] = "junk";
  if ( !strcmp( s, t ) ) {
   printf ("They match.\n");
  } else {
   printf ("They don't match.\n");
  }
}
```

Note the “reversed” logic!

Try it!

This is the right way to compare strings.

strcmp compares strings “lexicographically” (i.e., in dictionary order). One string is “greater than” another if it would come later in the dictionary. So, strcmp would say that “aardvark” is less than “zebra” because “aardvark” comes earlier in the dictionary.

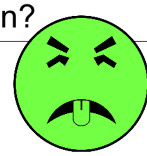
When we say “if (something)”, the “something” is true if it has a non-zero value, and false otherwise. Because strcmp returns 0 if the strings are equal, we need to use a ! (read “not”) to logically invert this into a true value.

Reading Strings The Wrong Way:

As we saw when comparing strings, the name of an array is actually just the memory address of the beginning of the array. This means we can leave off the “&” when we read a character array with scanf:

```
scanf (“%d”,&number);
scanf (“%d”, string );
```

So what's wrong with the following then?



```
#include <stdio.h>
int main () {
  char string[10];
  printf (“Enter some text:”);
  scanf("%s",string);
  printf("%s",string);
}
```

Try it!

First try it with “hello”.

Then try it with: “abcdefghijklmno pqrstuvwxyz”.

(For now, don't type anything that has spaces in it.)

When you try this you'll find that short strings like “hello” work fine, but long strings are likely to cause errors (segmentation faults). This is because scanf will just keep reading letters as long as we type them, and stuffing those letters into more and more locations in memory, even past the end of the “string” array. Sooner or later, scanf will try to stuff something into a memory location that doesn't belong to this program, and the program will crash.

So, how can we keep this from happening?

Reading Strings Safely:

We can fix our program by just adding one letter: change “%s” to “%9s”. This tells scanf to read no more than 9 characters.

```
#include <stdio.h>
int main () {
  char string[10];
  printf (“Enter some text:”);
  scanf("%9s",string);
  printf("%s",string);
}
```

Try your program now with “abcdefghijklmnopqrstuvwxyz”. What happens?

But what happens if you enter “the end”?

It turns out that C lets us insert a number in the middle of “%s” to specify how many characters (at most) we should read. In the example above, we say “%9s”, since our array has a length of 10, but we need to allow one element to hold the final NUL character at the end of the string.

Now, if we type:
  abcdefghijklmnoprstuvwxyz
the program will print:
  abcdefghi
(just the first 9 characters).

But, if you try entering “the end”, you'll find that the program thinks you just entered “the”. Why is that?

Reading Strings Containing Spaces:

scanf will stop reading a string when it sees a "white space" character (a space or a tab). This may not be what you want your program to do.

If you need to read strings containing spaces, a better choice is "fgets". fgets reads a specified number of characters from a file.

C automatically opens a special file called "stdin". This "file" is really just our keyboard.


```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text:");
  fgets( string, 9, stdin );
  printf("%s",string);
}
```

So why does "%s" stop at white spaces? It's so we can do things like this:

```
char name[10];
int year;
printf ("Enter your last name and birth year:");
scanf("%9s %d", name, &year);
```

or like this:

```
char firstname[10], lastname[10];
scanf("%9s %9s", firstname, lastname);
```

If scanf didn't stop at white spaces, the first example would try to stick things like "Wright 1961" into "name". It would never know when you were done typing the string, and had started typing something else.

Assigning Strings the Wrong Way:

Since strings are arrays, we also need to take care when assigning values to them in our programs:

```
#include <stdio.h>
int main () {
  char s[10] = "Testing";
  char t[10];

  t = s;
  printf( "%s", t);
}
```

This won't work!

Try it!

"In function 'int main()':
error: ISO C++ forbids assignment of arrays"

Again, remember that "t" and "s" are arrays, not single values.. The C compiler is telling you that it can't figure out what you want to do here.

What we're trying to do is make each element of the "t" array be the same as the corresponding element of the "s" array. If these were arrays of numbers, we could write a "for" loop to go through all of the array elements and do that, but there's an easier way to do it with character arrays.

Assigning Strings the Right Way:

We can use the “sprintf” function to “print” the value of one string into another string:

```
#include <stdio.h>
int main () {
  char s[10] = "Testing";
  char t[10];

  sprintf( t, “%9s”, s);
  printf( "%s", t);
}
```

This says: “Print the value of s into t.”

Try it!

“sprintf” is another variation on “printf”. The first argument of sprintf is just the name of a character string.

We could also to things like this:

sprintf (t, “Hello world!\n”);

which would put the text “Hello world!\n” into t.

Internally, sprintf just does the same thing as looping through all of the characters in the arrays, one by one, and setting their values.

String Recap:

- Comparing Strings:

```
if ( !strcmp( s, t ) ) {...}
```

- Reading Strings:

OK if no spaces in string:

```
scanf ( “%9s”, string );
```

If you need to read spaces or tabs:

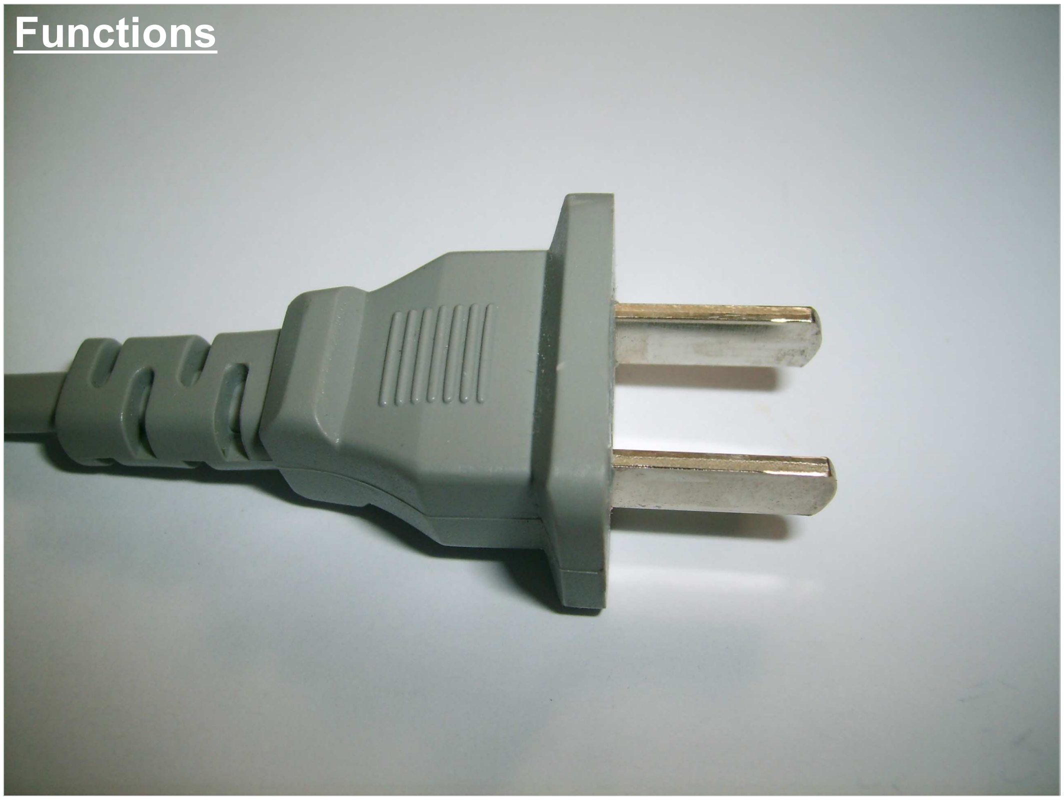
```
fgets ( string, 9, stdin );
```

- Assigning Values to Strings:

```
sprintf( t, “%9s”, s);
```

Writing past the end of a string array is a very common program bug. It often leads to crashes, and is responsible for many security flaws. Sticking to the methods above will help you avoid these problems in your programs.

Functions



Despite what you may think after these lectures, C is really a very simple language with a small vocabulary. It's extended through functions. These are found in standard libraries that are usually installed along with the compiler, but you can also create functions of your own to extend C's functionality.

“Intrinsic” Functions:

The functions we've used so far are sometimes called “intrinsic” functions. They're standard functions that available as soon as you install a C compiler on a computer.

printf	sqrt
fprintf	cos
scanf	sin
fscanf	rand
fgets	pow

There are lots of other intrinsic functions. But C also lets us define our own functions, to do things peculiar to our own programs.

Wouldn't it be nice...

In the galaxy example I sent out last week, we did a lot of the following:

```
r = sqrt( x*x + y*y + z*z );
```

Wouldn't it have been nice if we had a function that would just give us the value of r? Something like this, maybe:

```
r = radius(x,y,z);
```

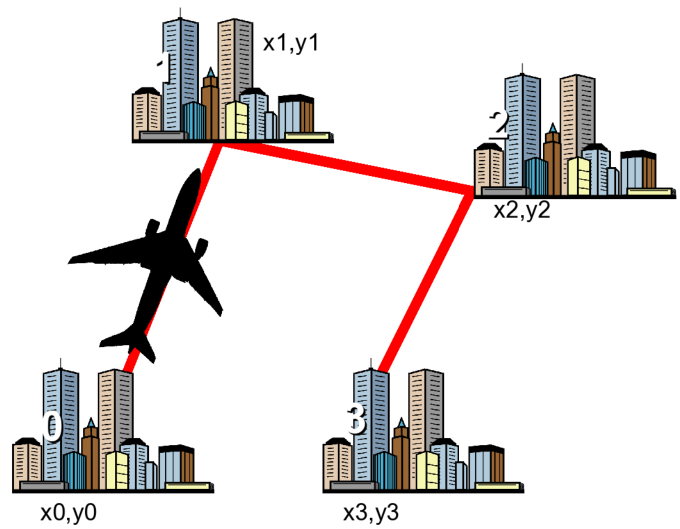
Maybe you can already see a couple of the advantages of a function like this:

•It can make our code more readable, so it's easier for us (and others) to understand in the future.

•It can reduce the likelihood of typos. Instead of typing out some long procedure every time we need it, and possible mis-typing something, we type the name of a function that will always do the same thing, every time we use it.

An Example:

Consider a program that calculates the total flying distance for a plane travelling a route connecting four cities. We're given the map coordinates of each city.

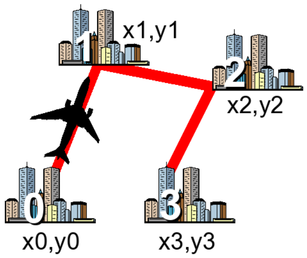


These cities are on a 2-d map, so we only have two coordinates for each city's position.

Let's look at a program to find the total distance.

The Long Way to Do It:

Here's one way to do it. Whew! that looks like a lot of typing!



We're doing the same thing over and over again.

Is there some way to avoid typing almost the same thing again and again?

```
double p0[2] = {0,0};
double p1[2] = {1,2};
double p2[2] = {4,1};
double p3[2] = {3,0};
double leg, d = 0;
int i;

leg = 0;
for ( i=0; i<2; i++ ) {
  leg += pow( p0[i]-p1[i], 2 );
}
d += sqrt(leg);

leg = 0;
for ( i=0; i<2; i++ ) {
  leg += pow( p1[i]-p2[i], 2 );
}
d += sqrt(leg);

leg = 0;
for ( i=0; i<2; i++ ) {
  leg += pow( p2[i]-p3[i], 2 );
}
d += sqrt(leg);

printf ("distance is %lf\n",d);
```

Length of 1st leg.

Length of 2nd leg.

Length of 3rd leg.

There's a lot of repetition in this program, and a lot of opportunities for typos! Also, it's really hard to read. Anyone else would have a hard time figuring out what this program is supposed to do.

Wouldn't it be nicer if we just had a “distance” function that would tell us the lengths? Unfortunately, C doesn't provide that, but maybe we can write our own.

Defining a New Function:

Here's one way we might write a function to make our program easier:

Type of value the function will produce.

Function arguments (two arrays of doubles, in this example).

```
double distance ( double a[], double b[] ) {
  // Calculate distance from point A to point B.
  double d=0;
  int i;
  for ( i=0; i<2; i++ ) {
    d += pow( b[i]-a[i], 2 );
  }
  d = sqrt(d);

  return(d);
}
```

Returns the calculated value to our program.

The general form of a function definition is:

```
type name( type1 arg1, type2 arg2, ....) {
...
}
```

We'll be able to use the function just the same way we use functions like "sqrt":

x = sqrt(y);

We give the function some arguments, and it gives us back a value.

Adding the Function to Our Program:

```
#include <stdio.h>
#include <math.h>

double distance ( double a[], double b[] ) {
  // Calculate distance from point A to point B.
  double d=0;
  int i;
  for ( i=0; i<2; i++ ) {
    d += pow( b[i]-a[i], 2 );
  }
  d = sqrt(d);

  return(d);
}

int main () {
....
```

Let's add the function just above our “main()” statement.

Notice the similarity?

You can put the function definition in other places, too, but for now let's put it here.

Notice that “int main() {...}” looks like a function definition too? That's because “main” really is just another function. When we run our C program, it invokes the “main” function.

In a later talk we'll see that we can even give arguments to “main”, and we can make use of any value it might return.

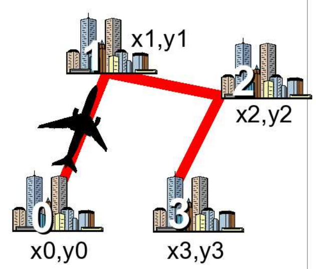
The Full Program:

"distance" function

"main" function

Try it!

```
#include <stdio.h>
#include <math.h>

double distance ( double a[], double b[] ) {
  double d=0;
  int i;
  for ( i=0; i<2; i++ ) {
    d += pow( b[i]-a[i], 2 );
  }
  d = sqrt(d);

  return(d);
}

int main () {
  double p0[2] = {0,0};
  double p1[2] = {1,2};
  double p2[2] = {4,1};
  double p3[2] = {3,0};
  double d;

  d = distance(p0,p1) +
      distance(p1,p2) +
      distance(p2,p3);
  printf ("distance is %lf\n",d);

}
```



So here's our complete program, using the "distance" function. I think it's much more readable now, don't you?

Functions have lots of other advantages. I hope we have time to talk about some of them later.

Practice Problem:

Can you modify the “distance” function so that it can calculate distances in higher dimensions? Hint: add another argument that specifies the number of dimensions.

Give this a try! Can your function calculate the distance between two points in a ten-dimensional Euclidean space?

The End

Thanks!