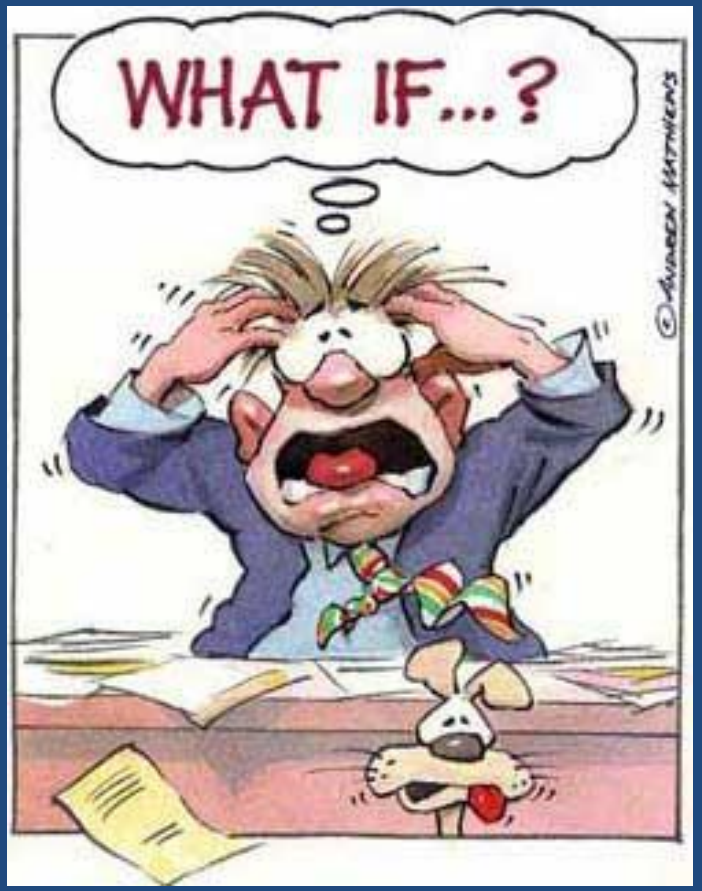
Advanced Software Techniques

Input Validation

A question heard often in the

C course was:

"Do I have to worry about bad input?"



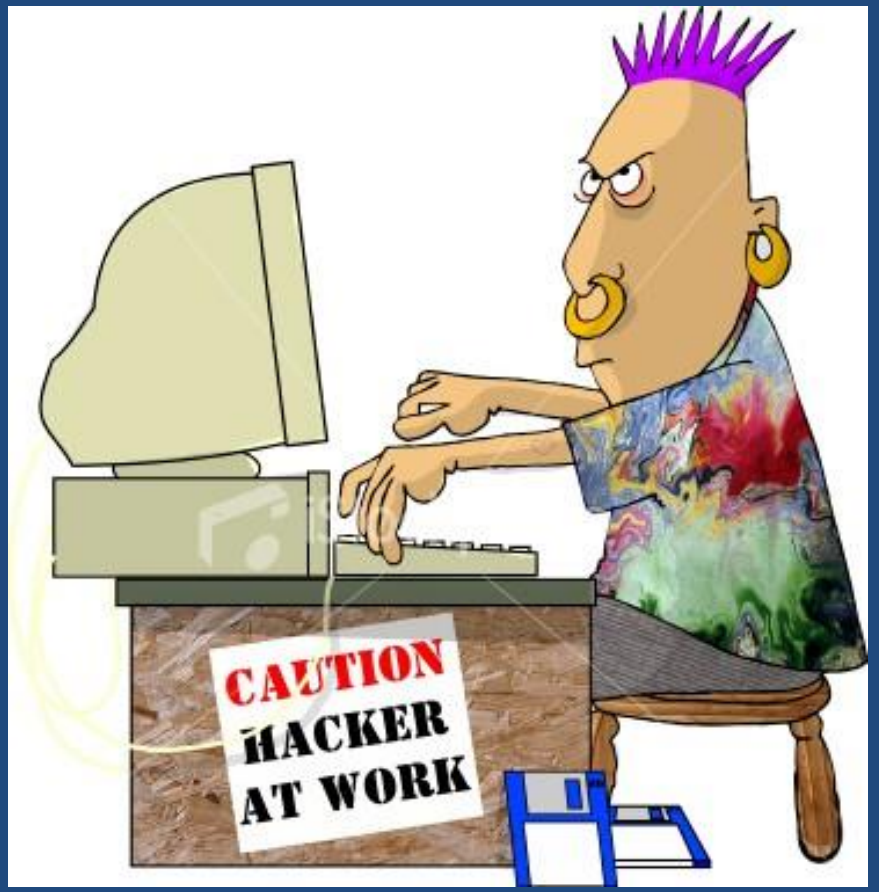
In the C course,
often the answer was
"no"
or
"just this type of bad input ..."

In the real world ...


YES!

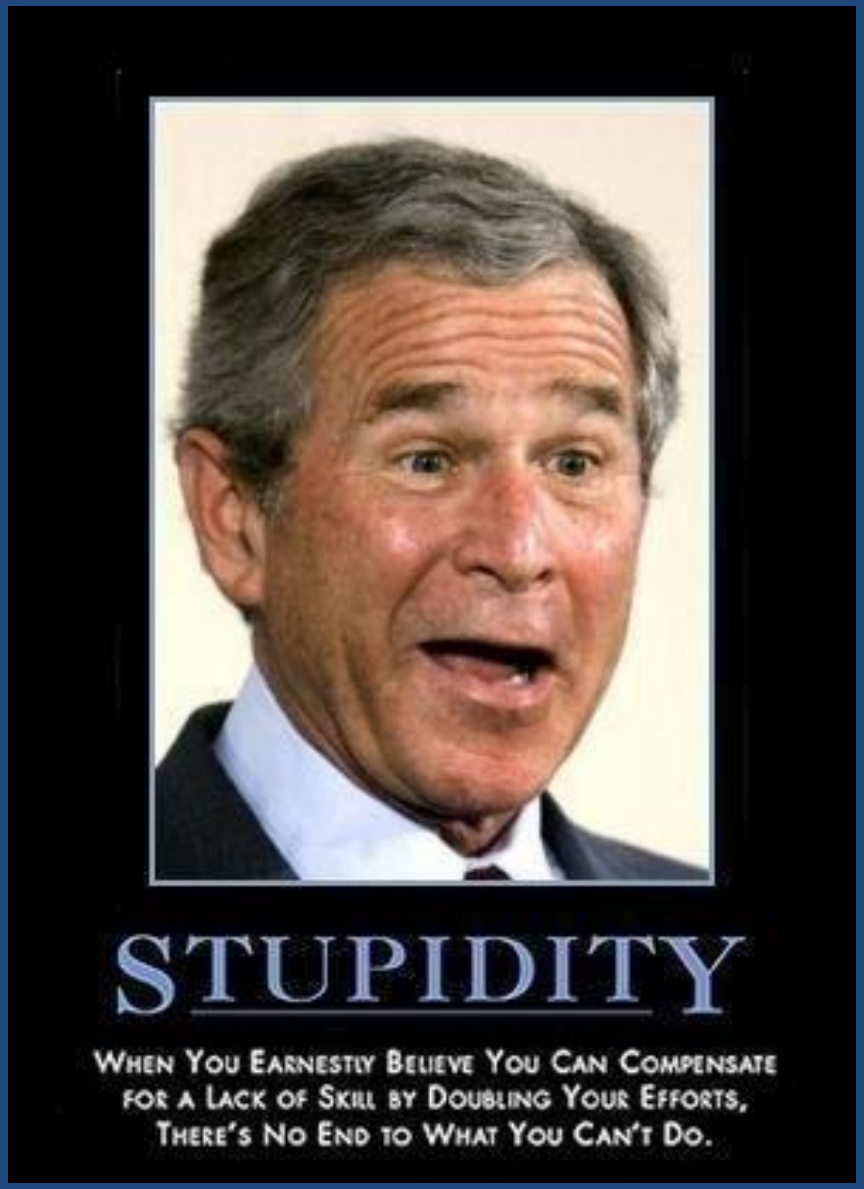
Users will be stupid

Users will be malicious

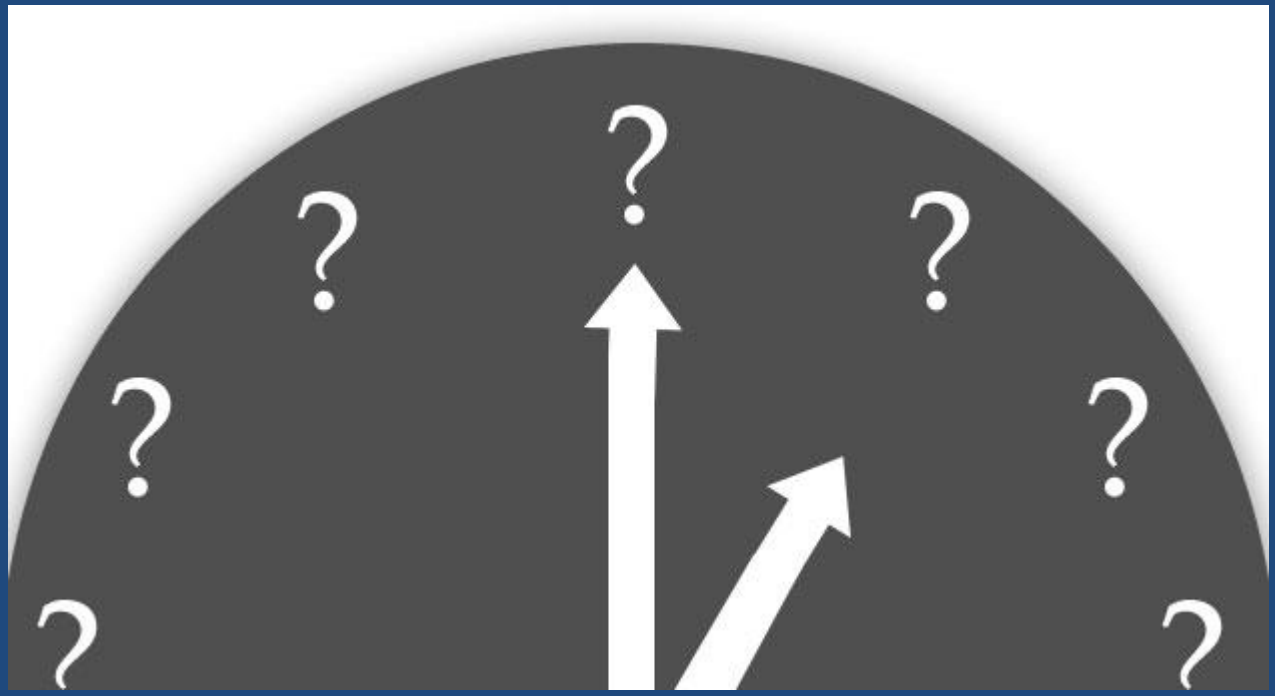


Users will be cats

Google said I should use this image for "stupid computer user"



OK, so when should we check data?



When the user enters it?

Later, just before using it?

It's best to be safe

A Bad Strategy

How about checking for illegal input?

Not a good idea on its own

Attackers can be creative

A Better Strategy

Identify legal inputs

and

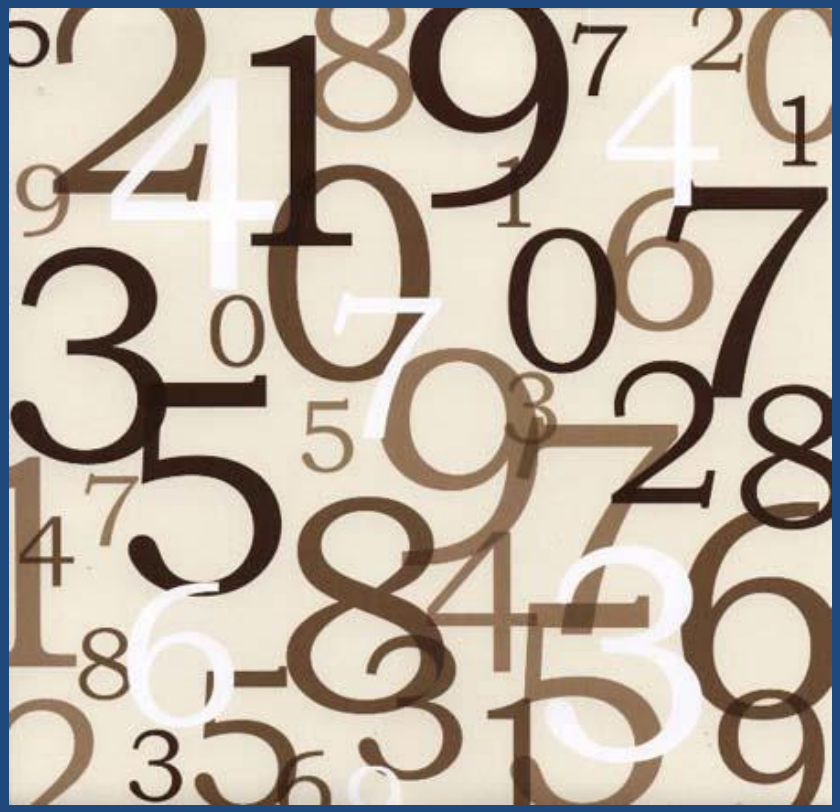
only allow them

And, if you happen to know about dangerous input, explicitly disallow it too

Let's look at the example of numbers

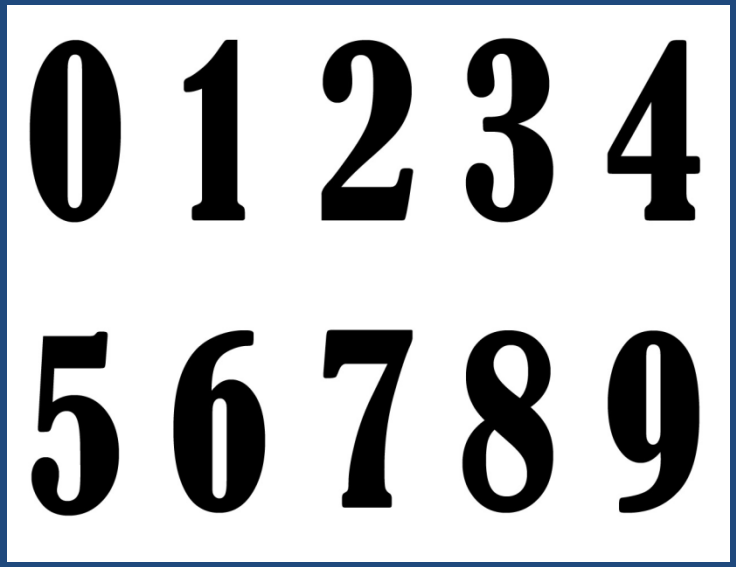
Numbers are easier

to define legal inputs for



Allow the digits

0 through 9



Maybe allow -

(if negative numbers are allowed)

Maybe allow $
(if dealing with currency)

Maybe allow .
(if dealing with
real numbers)

Maybe allow SPACE

(if dealing with credit card numbers or SINs)

(OK, maybe this is the wrong kind of space)

And then deal with
minimum
and
maximum values

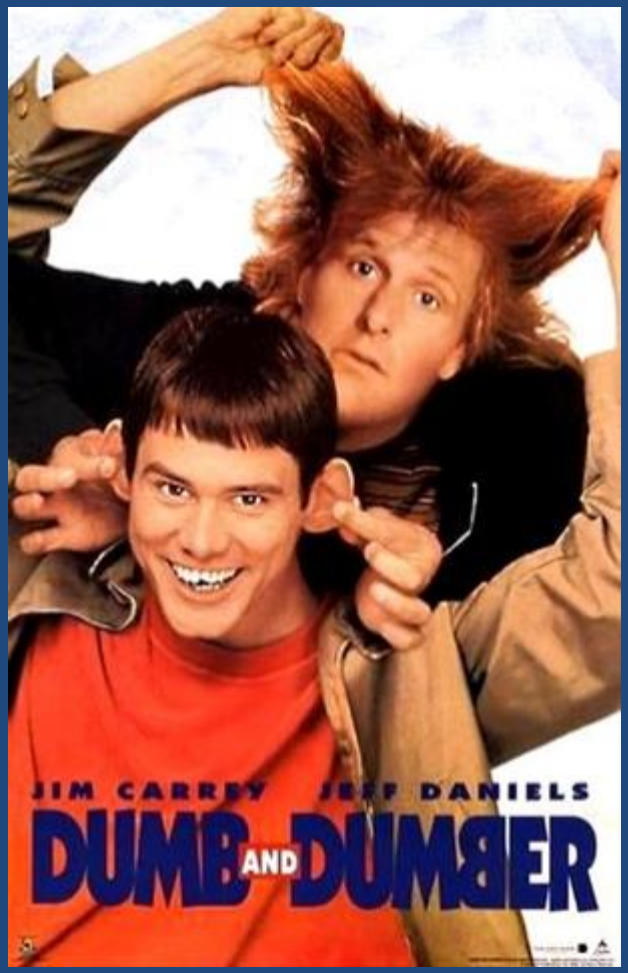

if...


if


if...

Hopefully,
it's obvious
that context is vitally important

Even with something as (apparently) simple as validating a number



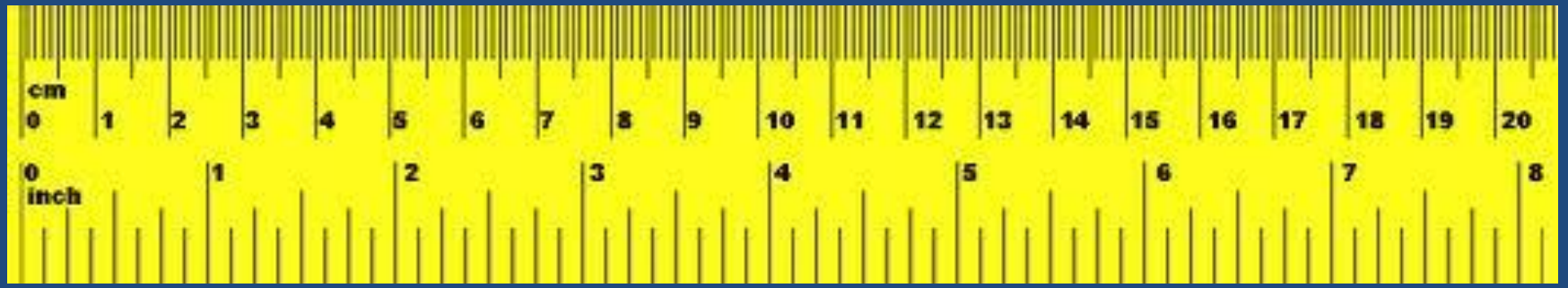
Definition

What is input validation?

"Make sure that the data is:
strongly typed,

correct syntax,

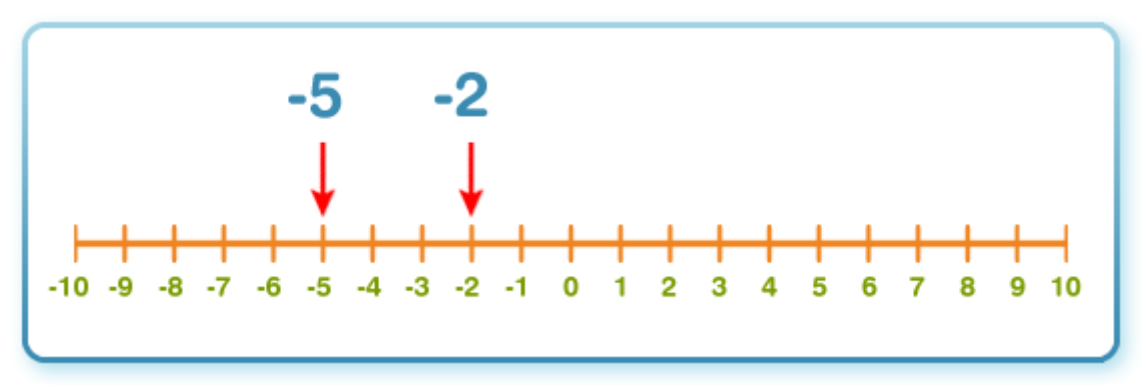
within length boundaries,



contains only permitted characters,

or that numbers are correctly signed

and



within range boundaries"

(from http://www.owasp.org/index.php/Data_Validation)

Beyond that:

"ensure that data is not only validated but business rule correct.

For example, interest rates fall within permitted boundaries." (also from the OWASP webpage)

Business Rule?

"a statement that defines or constrains some aspect of the business"

(from http://en.wikipedia.org/wiki/Business_rule)

Wow

Yes, there's a lot to this stuff

Revisiting the Definition

Strongly Typed

This goes beyond

"what data type is your variable?"

Also, can you

safely

convert to the

data type you

need

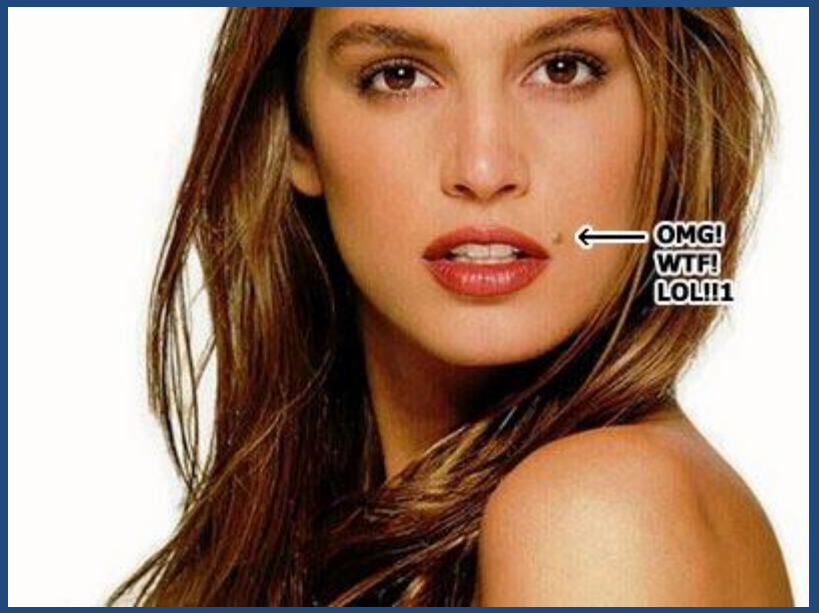
You'll often take in strings and need numbers

How can you safely do that conversion?

atoi()?

sscanf()?

Both have flaws



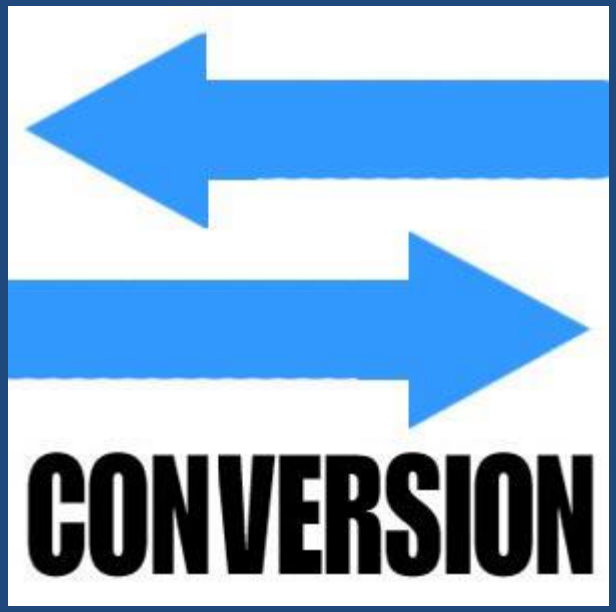
How about strtol()?

```
long strtol(
const char * nptr,
char ** endptr,
int base)
```

And don't forget

```
#include <stdlib.h>
```

The first parameter is the string to convert



We'll get to the second parameter in a minute

The third parameter is the numeric base (usually 10)

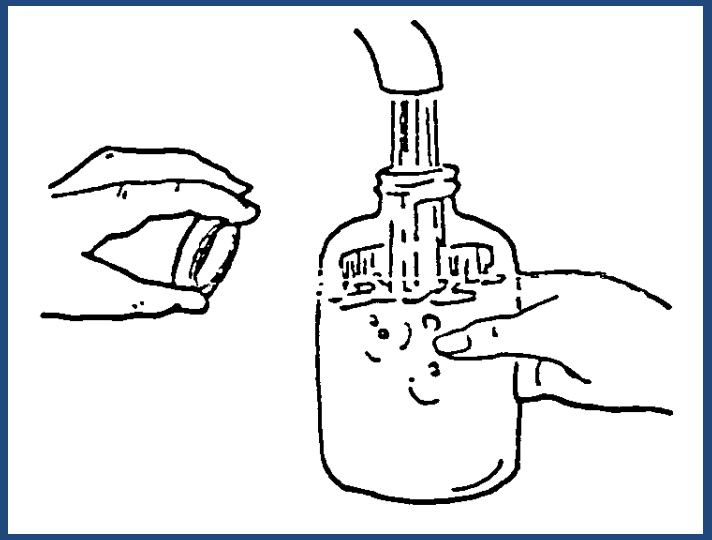
The function converts strings to long variables, similar to atol()

It'll return the converted value

or

0, if it can't convert

The second parameter
is filled in
by the function



endptr is filled in

as a pointer

to the first invalid character

or

NULL

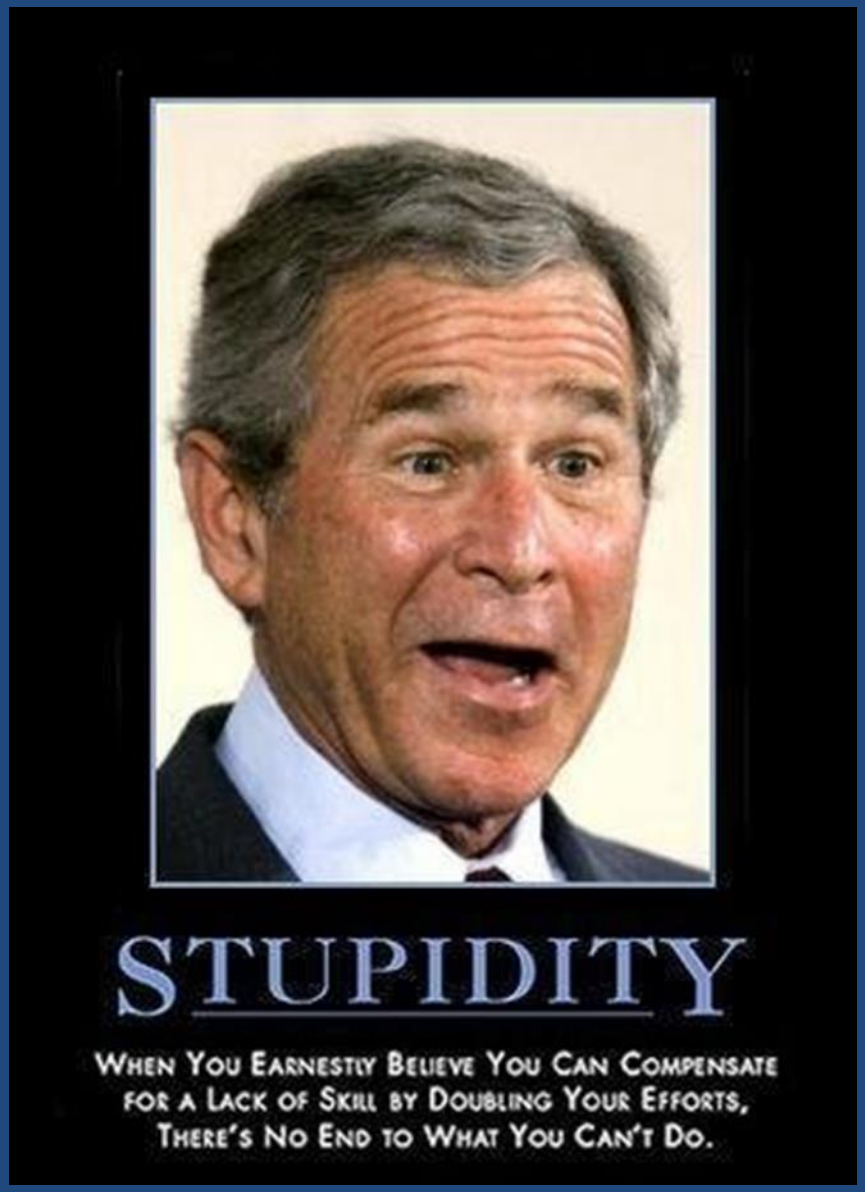
If endptr is
NULL
or equal to the
start of the string,
it didn't succeed in
conversion

Correct Syntax

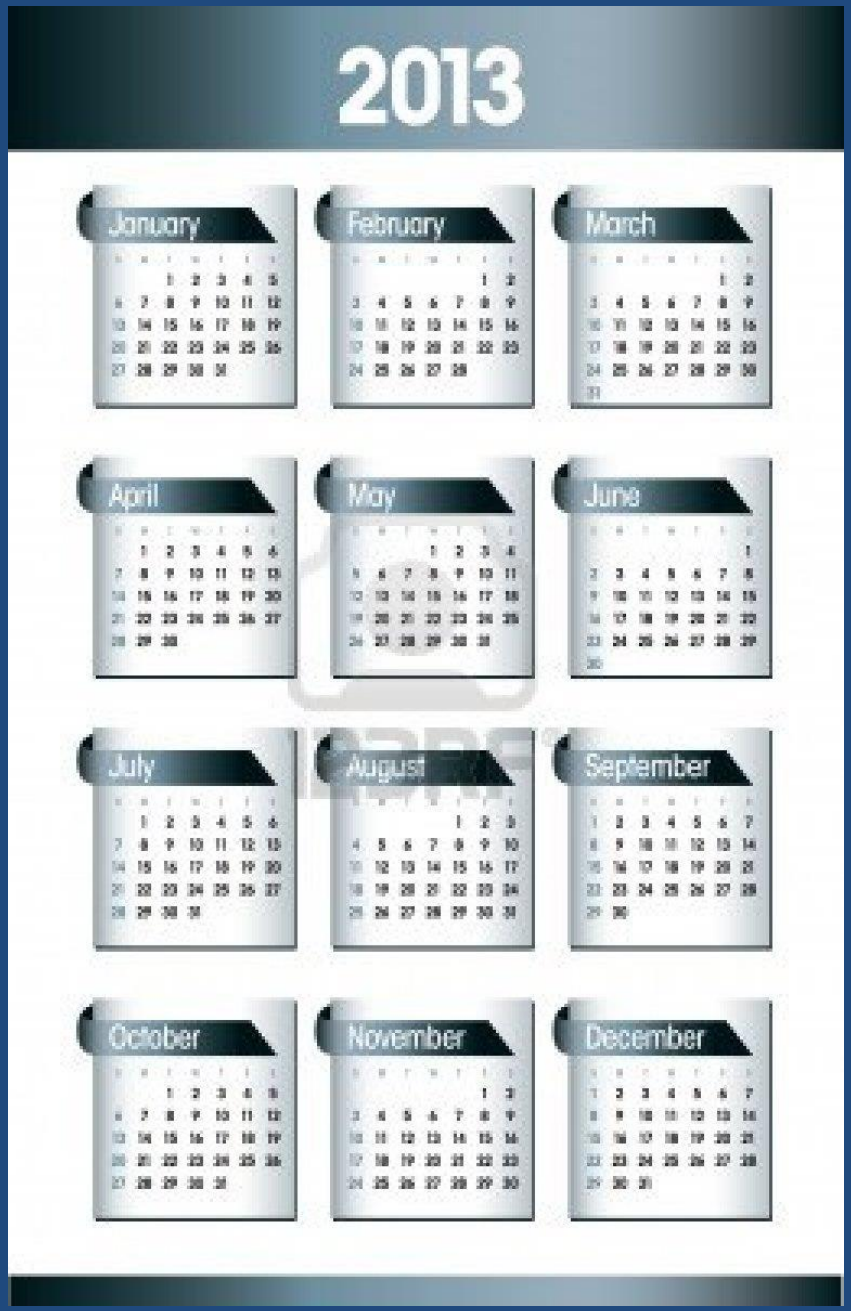
First, it's YOUR responsibility to communicate what the correct syntax is

If the user has to guess, they'll guess wrong

Make it easy for the stupid user



For example, dates can have many different formats



Times have fewer different formats but are still problematic

Create
a validation
function
that takes the
input
and validates the
syntax

In C, a valuable tool is

strtok()

Prototype:

```
char * strtok ( char * str, const char * delimiters );
```

It looks for one of the many specified delimiters

(skipping over leading delimiters)

If it finds one, it NULL-terminates the string where it finds it

The string ending
at that
NULL-termination
is called the "token"

Yes, it does change
the string it's looking at

It returns either:

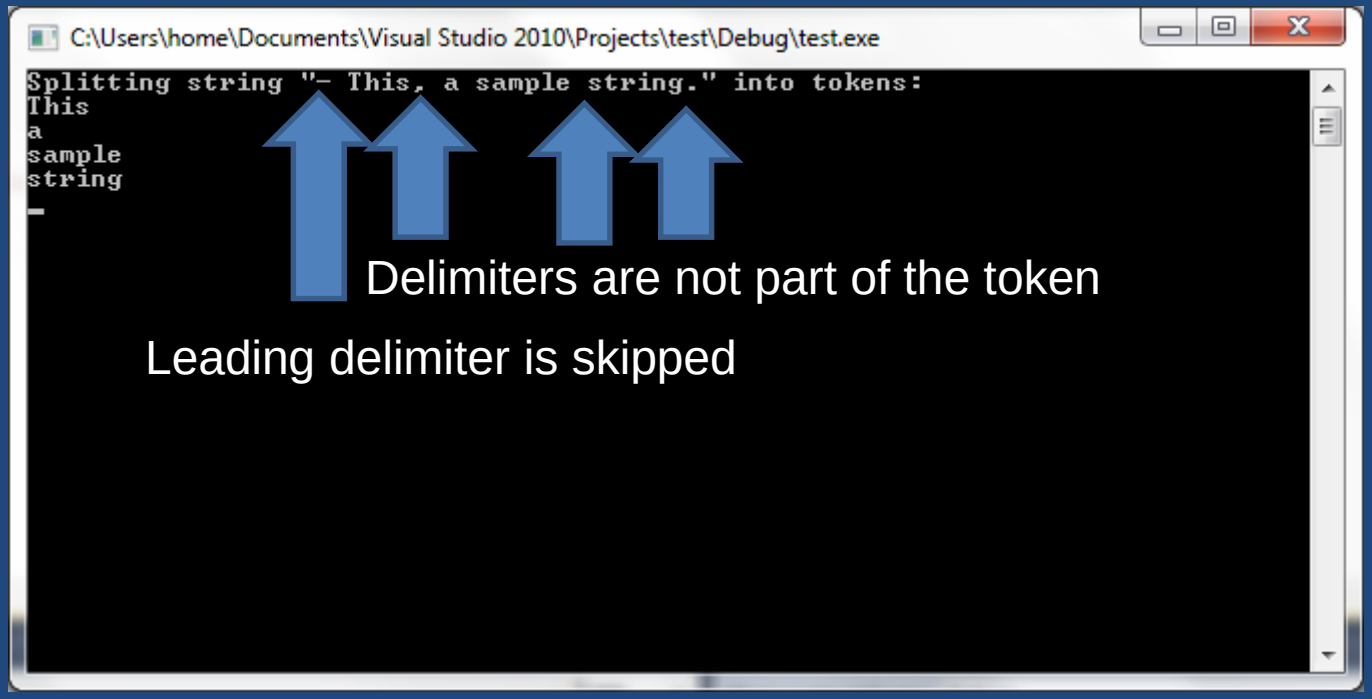
- a pointer to the token it found
- NULL, indicating no tokens

To find the next token
in that same original string,
call strtok() again,
this time with NULL
as the first parameter

Yes, it keeps track of the string it's working on

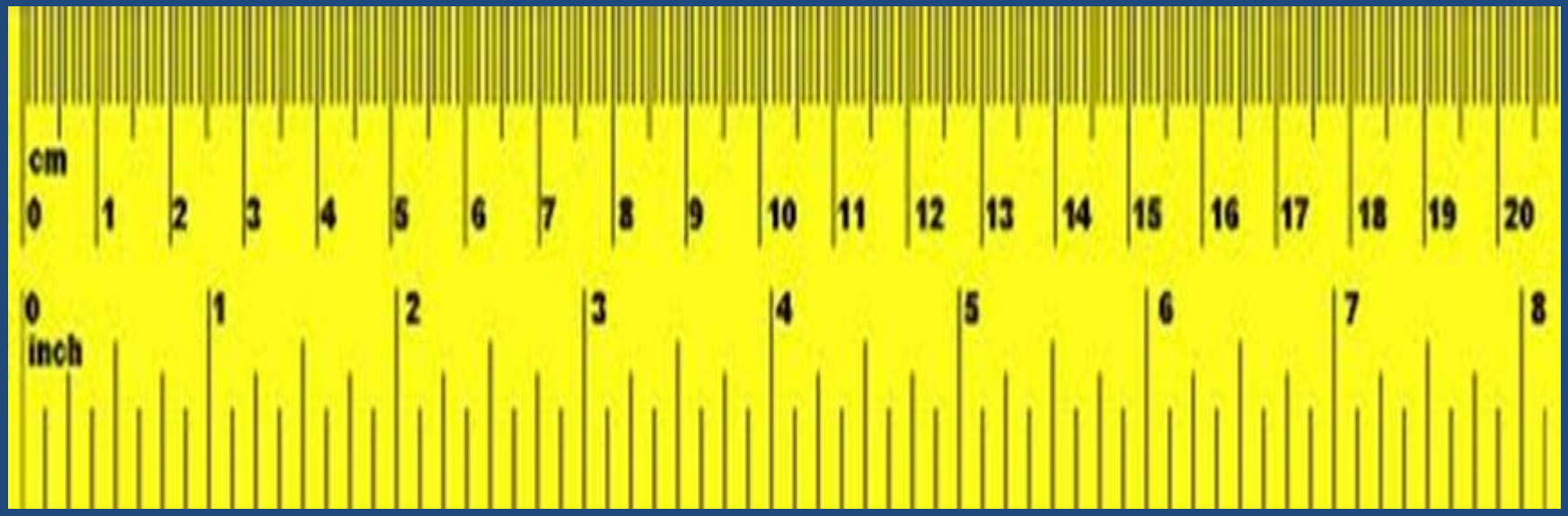
```
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] ="- This, a sample string.";
  char * pch = NULL;
  printf ("Splitting string \"%s\" into tokens:\n",str);
  pch = strtok (str," ,.-");
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok (NULL, " ,.-");
  }
  return 0;
}
/* from http://cplusplus.com/reference/clibrary/cstring/strtok */
```

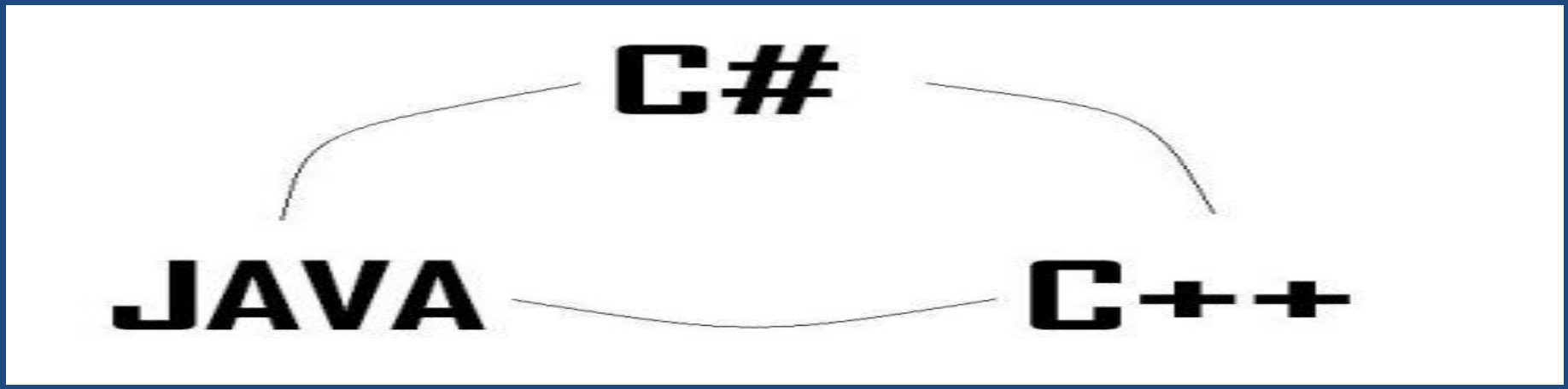

C:\Users\home\Documents\Visual Studio 2010\Projects\test\Debug\test.exe
Splitting string "- This, a sample string." into tokens:
This
a
sample
string
Delimiters are not part of the token
Leading delimiter is skipped

Within Length Boundaries

This is a LARGE problem in C



More modern languages
have learned from
C's mistakes
(with String classes)

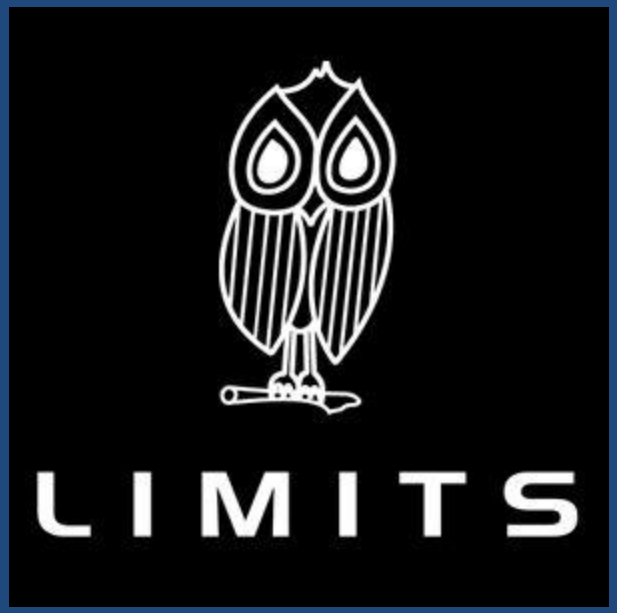


In C++
(and many other languages),
the capacity field of
the string class
contains
the maximum length

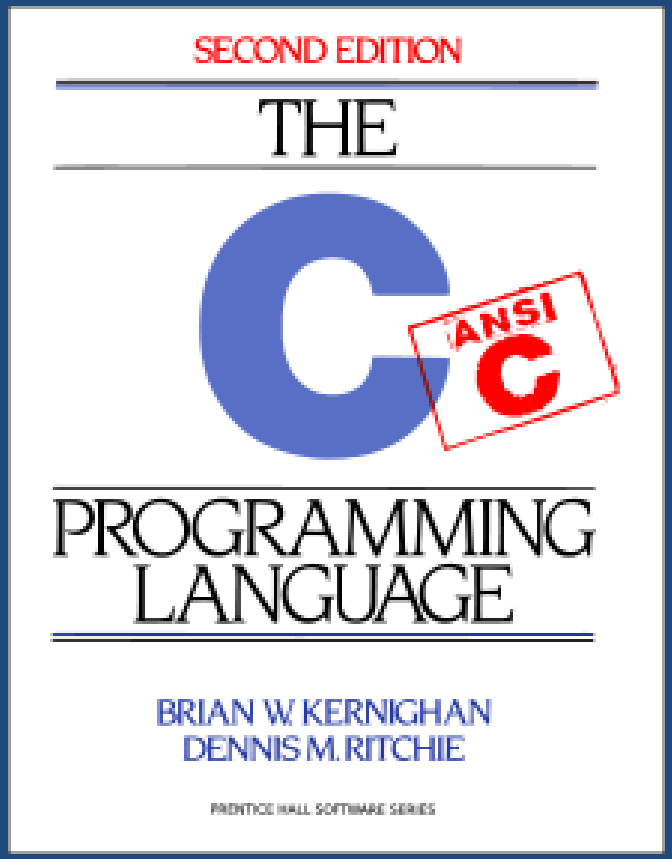
The length limit for the Visual C++ (and C#)
string class
is 2^30 characters
(just over 1 billion)

But you might still have

limits internal to your application

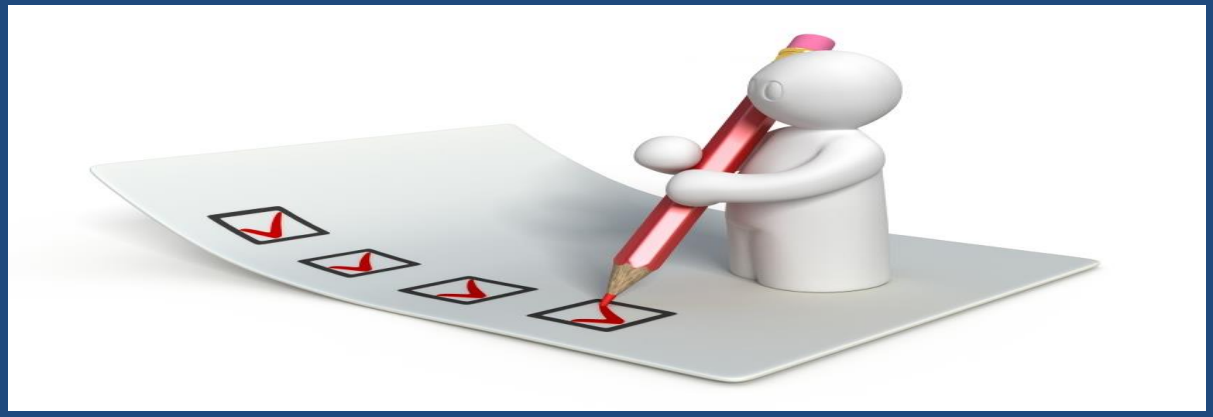


Those limits are inherent to using C-style NULL-terminated strings



If you're using C-style strings, you MUST use a safe method of input

If you're copying
the data into a
C-style string or
other limited-space data field,
make sure there's enough room BEFORE copying



In Visual C++,
they provide functions
that help with this
(we'll take about this later in
Computer Security)

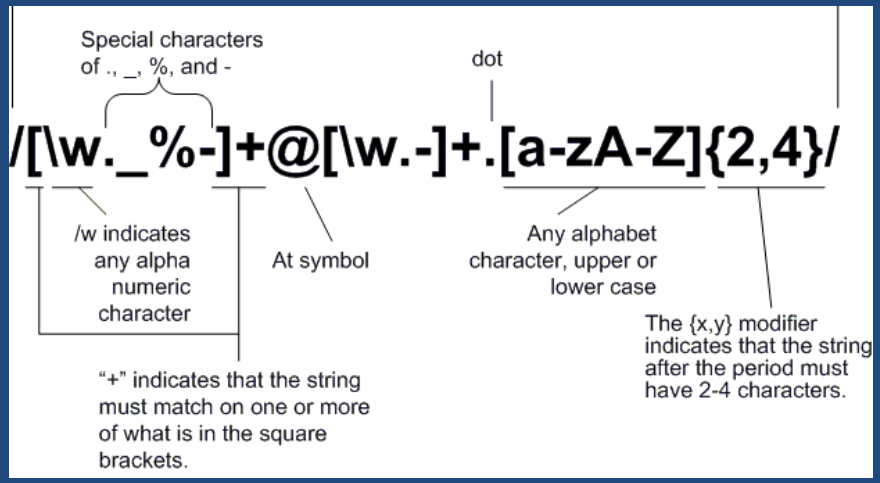
Contains only Permitted Characters

It is best if you know exactly which characters are legal (for this particular input)

Each input might be different

So it's up to you to determine it

We'll be looking at regular expressions in another class

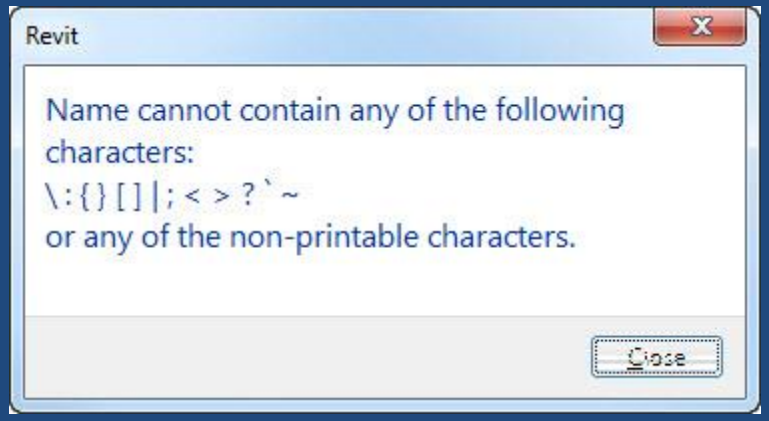


Even without
regular expressions,
you can use other functions

Like sscanf's scansets

e.g. sscanf(str, "%[a-z]", letters);

Of course,
you can always
look for illegal characters



Using sscanf's inverted scansets

e.g. sscanf(str, "%[^@&%]", letters);

or

```
const char * strchr ( const char
                      * str,
                int character );
```

or

```
const char * strpbrk( const char
* str1,
const char * str2 );
```

Here's some examples
of common illegal
characters used
by malicious users:

'\0'

'\r'

'\n'

0x0f

"

'

<

>

;

&

!

0xff

,

/

\

0x9 (TAB)

any character below ASCII value 0x20

any character above ASCII value 0x7f

And let's not forget
the most irritating
mistake
in determining
valid characters ...

UPPERCASE

vs.

lowercase

Remember to accept both uppercase and lowercase unless accepting both would make it wrong

Example:

postal codes should be accepted in both cases

Example:
Searching for a name should be accepted in both cases

Example:

Getting confirmation

(y/n)

should be accepted in both cases

But don't forget
that storing the data
might require a conversion

Example:

If the user enters a postal code of "n2t 4s5",

you should probably store it as "N2T 4S5"

Numbers are Correctly Signed

Look for minus signs in the input

Numbers are Within Range Boundaries

This is

ABSOLUTELY

vital if the input

will be used as an index

Check the value where it's most appropriate

How about
WAY
out of range?

Remember strtol()?

It returns
LONG_MAX or LONG_MIN and
sets the errno variable
to ERANGE
if it's out of range

Business Rule Correct

Context-dependent

Wow!

Yes, there WAS really a lot to this stuff

Now let's test it out

in

the major assignment

for the course!