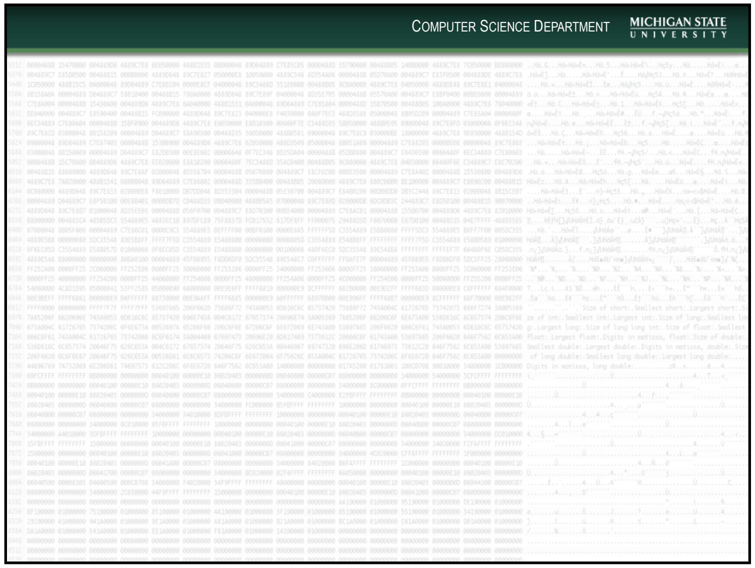
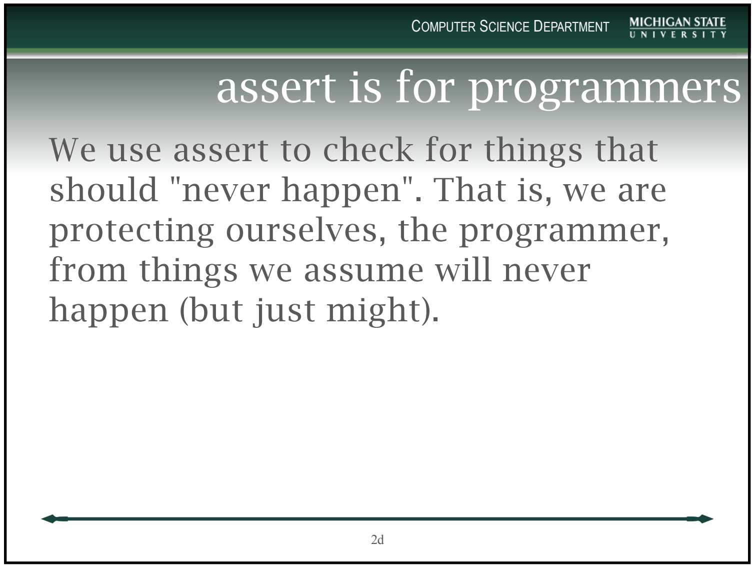
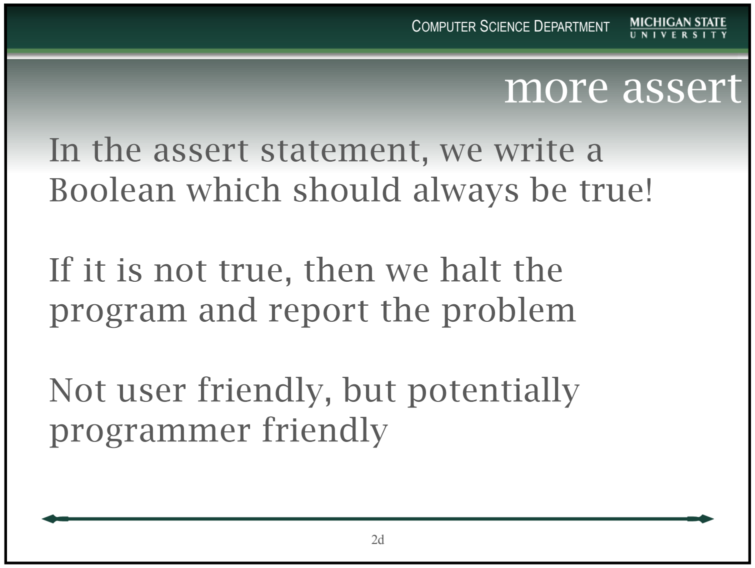

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

assert is for programmers

We use assert to check for things that should "never happen". That is, we are protecting ourselves, the programmer, from things we assume will never happen (but just might).

2d


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

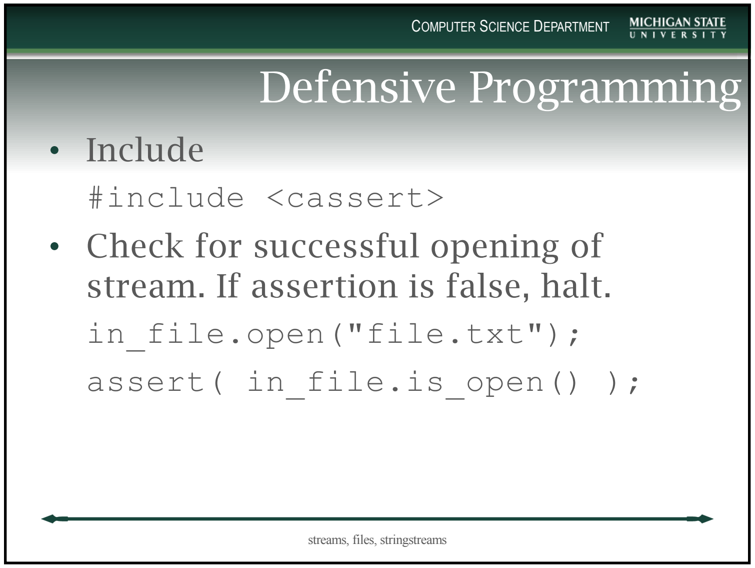
more assert

In the assert statement, we write a Boolean which should always be true!

If it is not true, then we halt the program and report the problem

Not user friendly, but potentially programmer friendly

2d


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

Defensive Programming

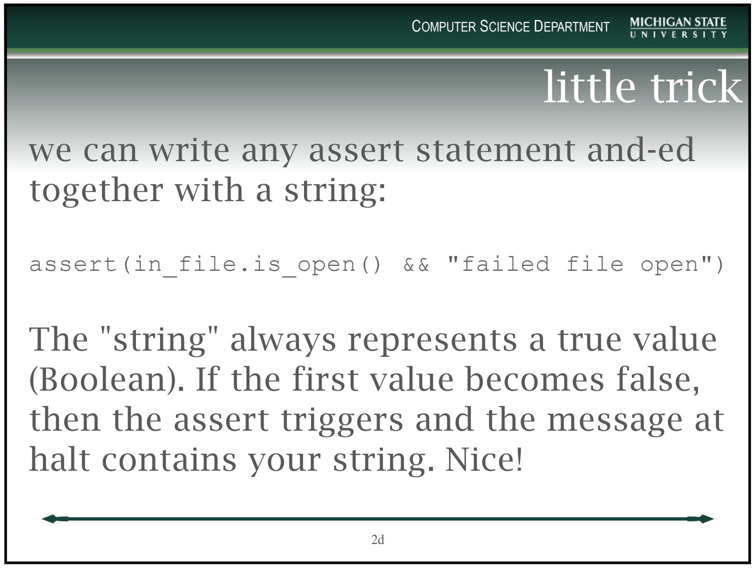
- Include

```
#include <cassert>
```

- Check for successful opening of stream. If assertion is false, halt.

```
in_file.open("file.txt");
assert( in_file.is_open() );
```

streams, files, stringstreams



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

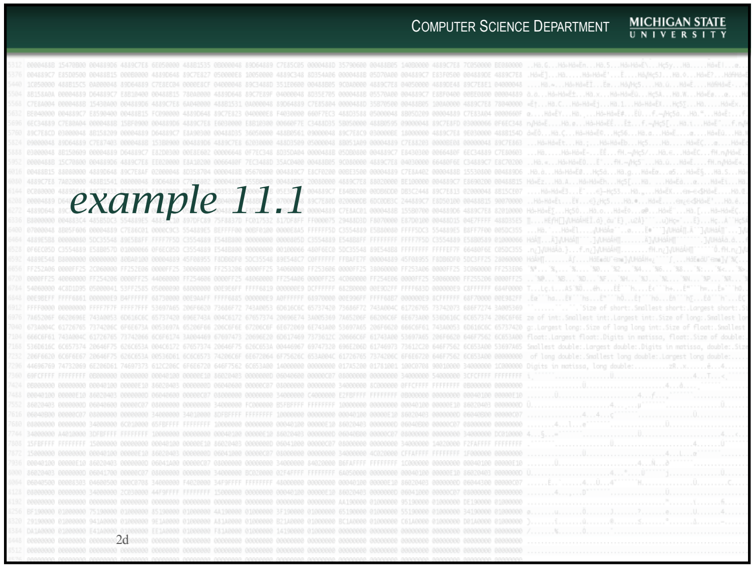
little trick

we can write any assert statement and-ed together with a string:

```
assert(in_file.is_open() && "failed file open")
```

The "string" always represents a true value (Boolean). If the first value becomes false, then the assert triggers and the message at halt contains your string. Nice!

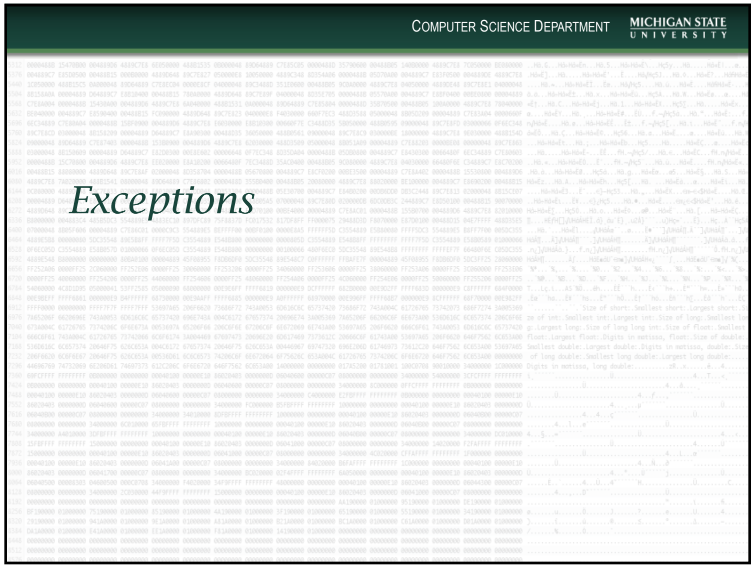
2d



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

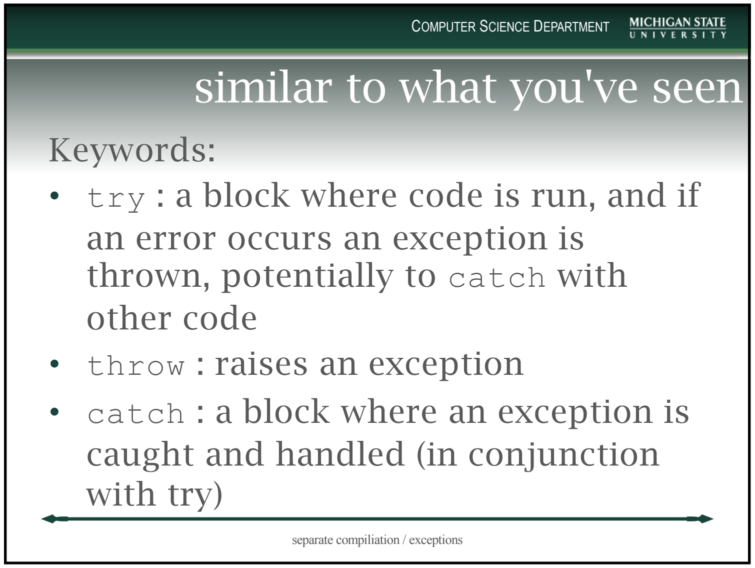
example 11.1

2d



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

Exceptions



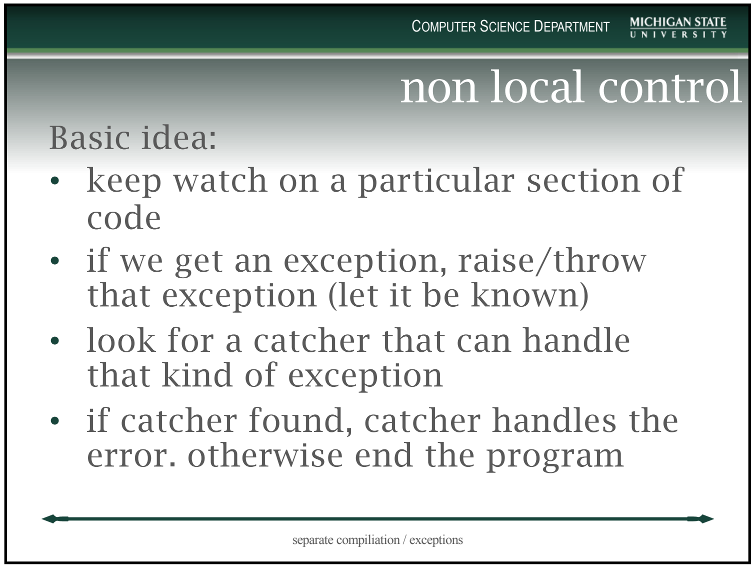
COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

similar to what you've seen

Keywords:

- `try` : a block where code is run, and if an error occurs an exception is thrown, potentially to `catch` with other code
- `throw` : raises an exception
- `catch` : a block where an exception is caught and handled (in conjunction with try)

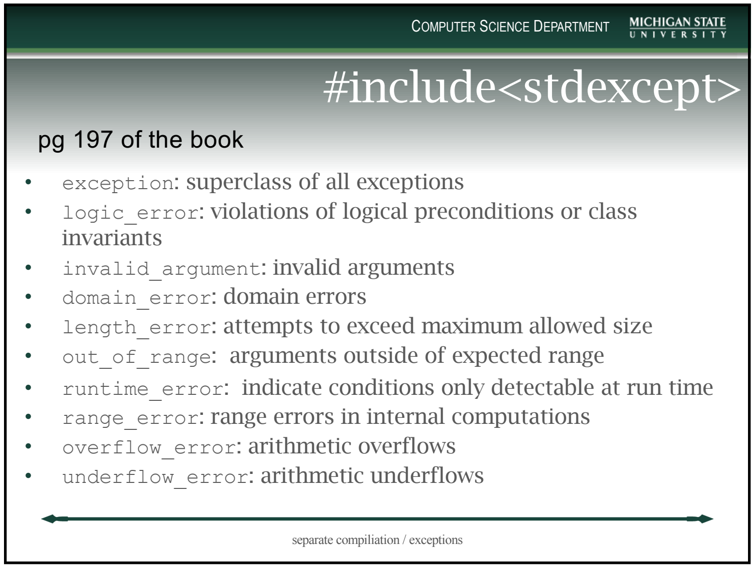
separate compiliation / exceptions



non local control

Basic idea:

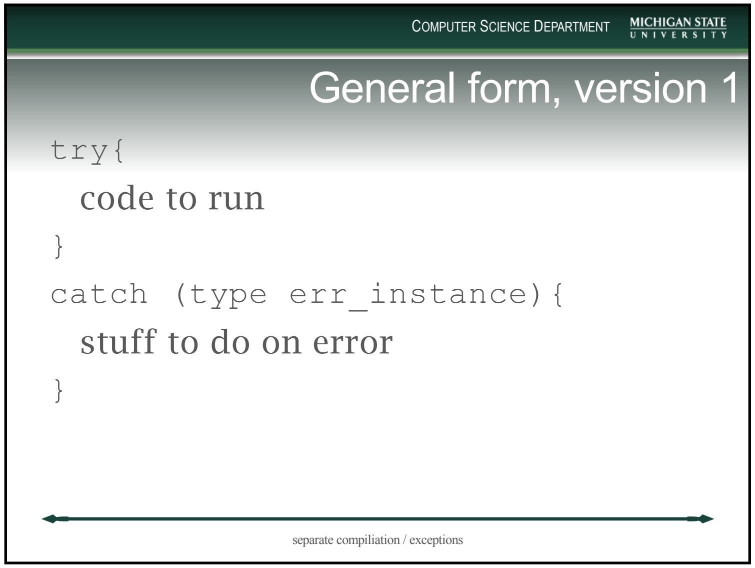
- keep watch on a particular section of code
- if we get an exception, raise/throw that exception (let it be known)
- look for a catcher that can handle that kind of exception
- if catcher found, catcher handles the error. otherwise end the program



#include<stdexcept>

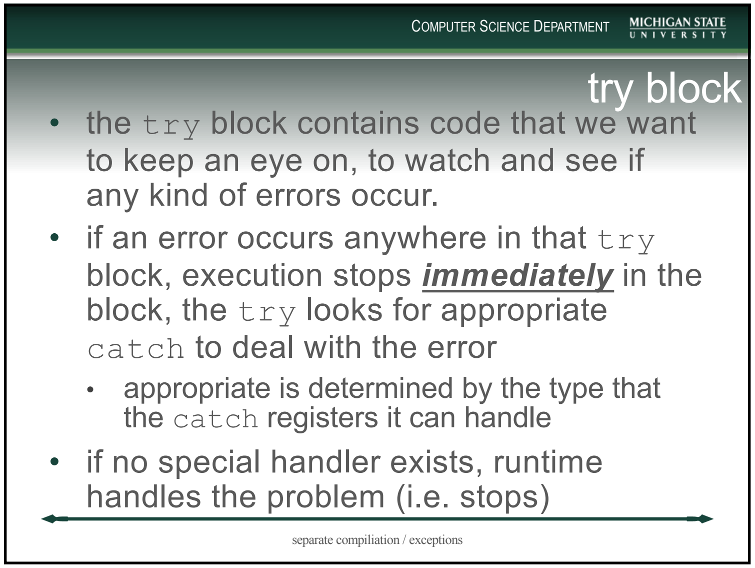
pg 197 of the book

- `exception`: superclass of all exceptions
- `logic_error`: violations of logical preconditions or class invariants
- `invalid_argument`: invalid arguments
- `domain_error`: domain errors
- `length_error`: attempts to exceed maximum allowed size
- `out_of_range`: arguments outside of expected range
- `runtime_error`: indicate conditions only detectable at run time
- `range_error`: range errors in internal computations
- `overflow_error`: arithmetic overflows
- `underflow_error`: arithmetic underflows



General form, version 1

```
try{
  code to run
}
catch (type err_instance){
  stuff to do on error
}
```



try block

- the `try` block contains code that we want to keep an eye on, to watch and see if any kind of errors occur.
- if an error occurs anywhere in that `try` block, execution stops ***immediately*** in the block, the `try` looks for appropriate `catch` to deal with the error
  - appropriate is determined by the type that the `catch` registers it can handle
- if no special handler exists, runtime handles the problem (i.e. stops)

exception block

- an `catch` block (perhaps multiple `catch` blocks) is associated with a `try` block.
- the `catch` block names the type of exception it is capable of handling
  - the type can be a subtype of a more general exception type
- if the error that occurs in the `try` block matches the `catch` type, then that `catch` block is activated.

try exception combination

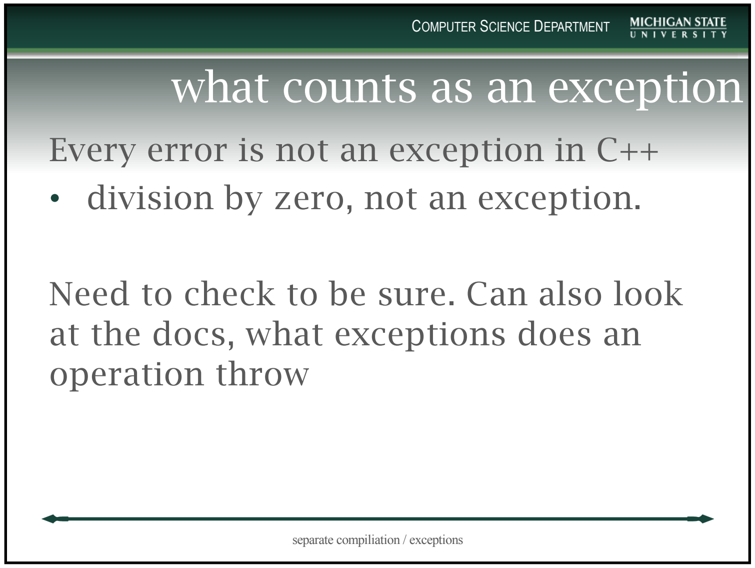
- if no exception in the `try` block, skip past all the `catch` blocks to the following code
- if an error occurs in a `try` block, look for the right `catch` by type
  - including a super-type of the exception
- if `catch` is matched, run that `catch` block and then skip past the `try/catch` blocks to the next line of code
- if no exception handling found, give the error to the runtime

throw

When you do a `throw`, you create an instance of an exception and you can provide, in the constructor, a string to describe the problem:

- except for the superclass exception

example 11.2

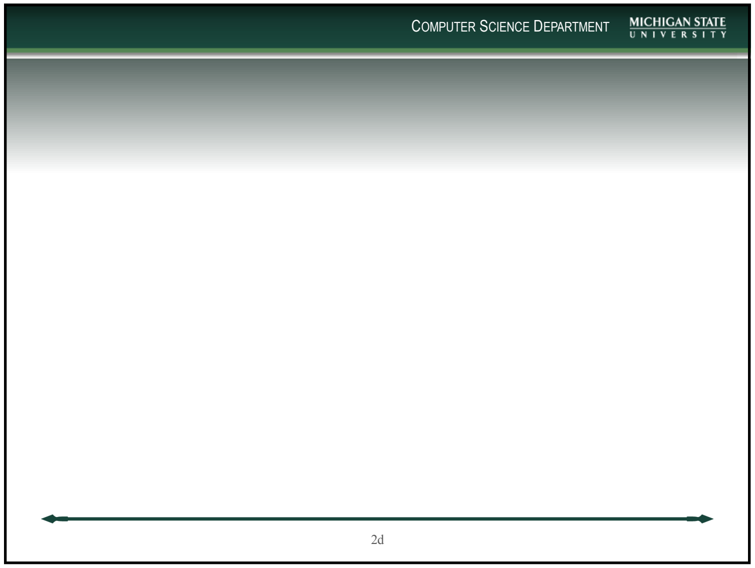


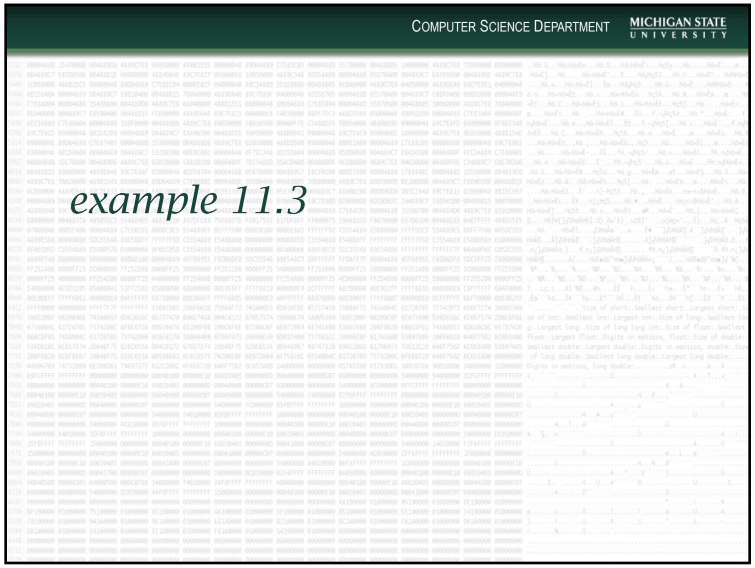
what counts as an exception

Every error is not an exception in C++

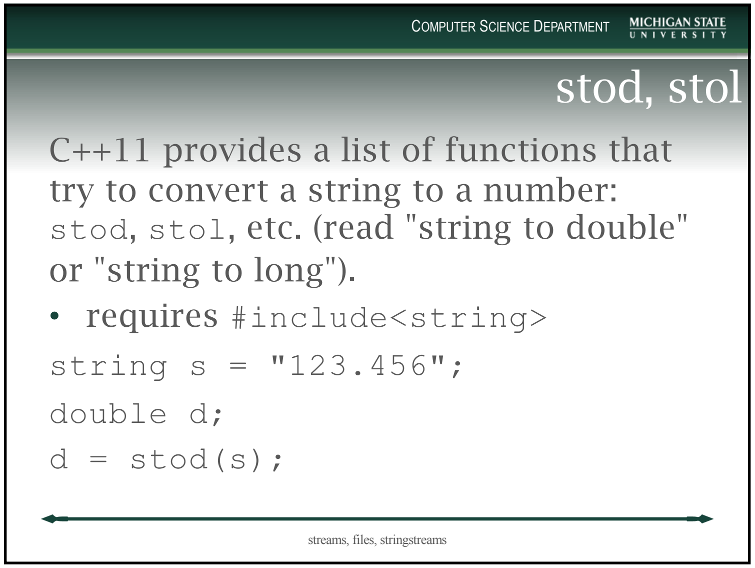
- division by zero, not an exception.

Need to check to be sure. Can also look at the docs, what exceptions does an operation throw





example 11.3

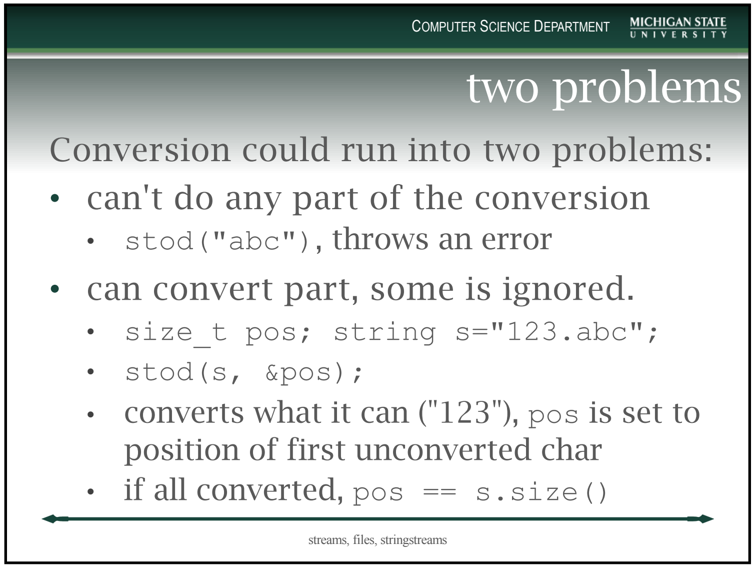


stod, stol

C++11 provides a list of functions that try to convert a string to a number: `stod`, `stol`, etc. (read "string to double" or "string to long").

- requires `#include<string>`

```
string s = "123.456";
double d;
d = stod(s);
```



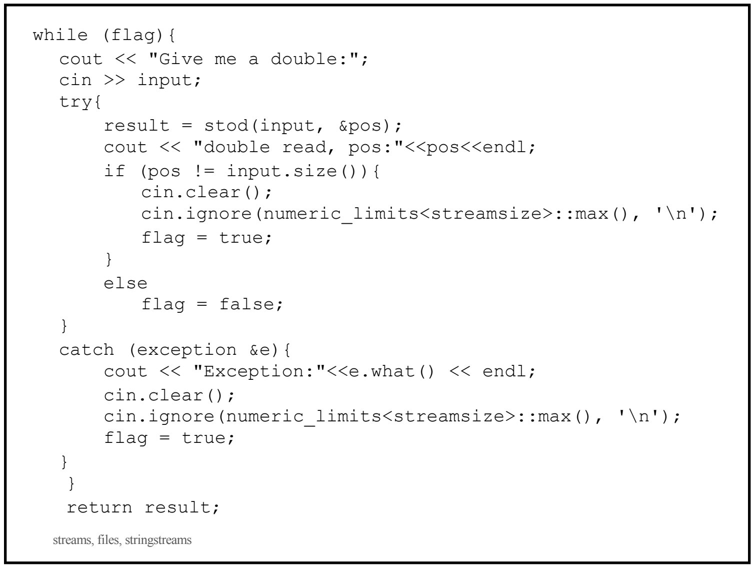
COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

two problems

Conversion could run into two problems:

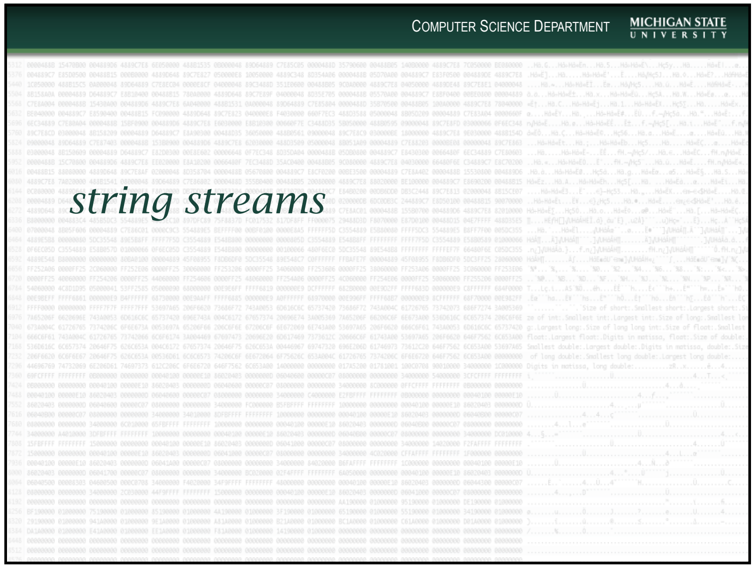
- can't do any part of the conversion
  - `stod("abc")`, throws an error
- can convert part, some is ignored.
  - `size_t pos; string s="123.abc";`
  - `stod(s, &pos);`
  - converts what it can ("123"), `pos` is set to position of first unconverted char
  - if all converted, `pos == s.size()`

streams, files, stringstreams



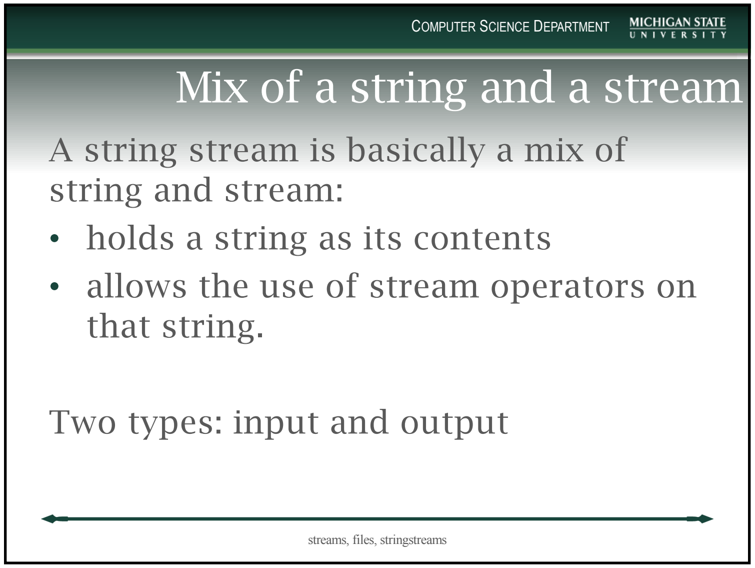
```
while (flag){
   cout << "Give me a double:";
   cin >> input;
   try{
        result = stod(input, &pos);
        cout << "double read, pos:"<<pos<<endl;
        if (pos != input.size()){
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            flag = true;
        }
        else
            flag = false;
   }
   catch (exception &e){
        cout << "Exception:"<<e.what() << endl;
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        flag = true;
   }
    }
    return result;
```

streams, files, stringstreams



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

string streams



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

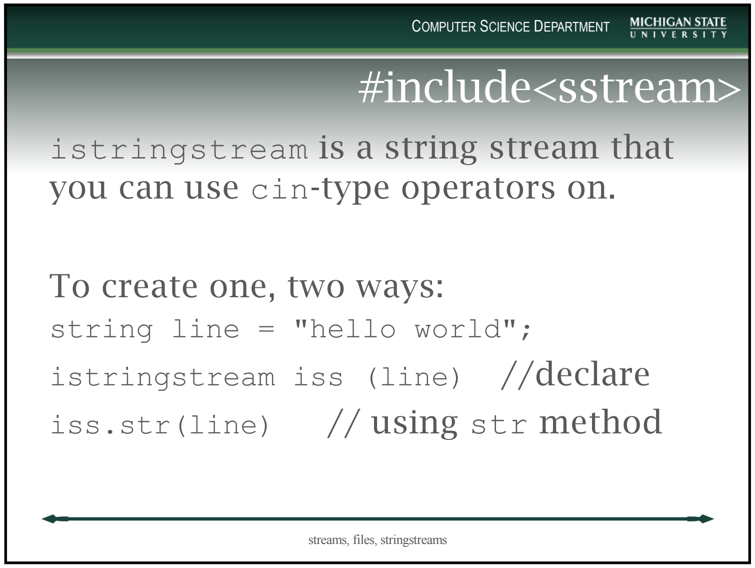
Mix of a string and a stream

A string stream is basically a mix of string and stream:

- holds a string as its contents
- allows the use of stream operators on that string.

Two types: input and output

streams, files, stringstreams



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

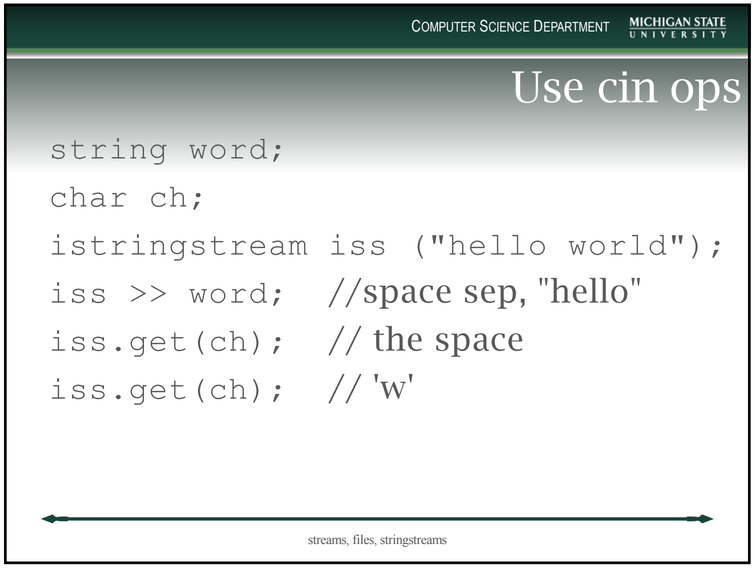
#include<sstream>

`istringstream` is a string stream that you can use `cin`-type operators on.

To create one, two ways:

```
string line = "hello world";
istringstream iss (line)   //declare
iss.str(line)    // using str method
```

streams, files, stringstreams

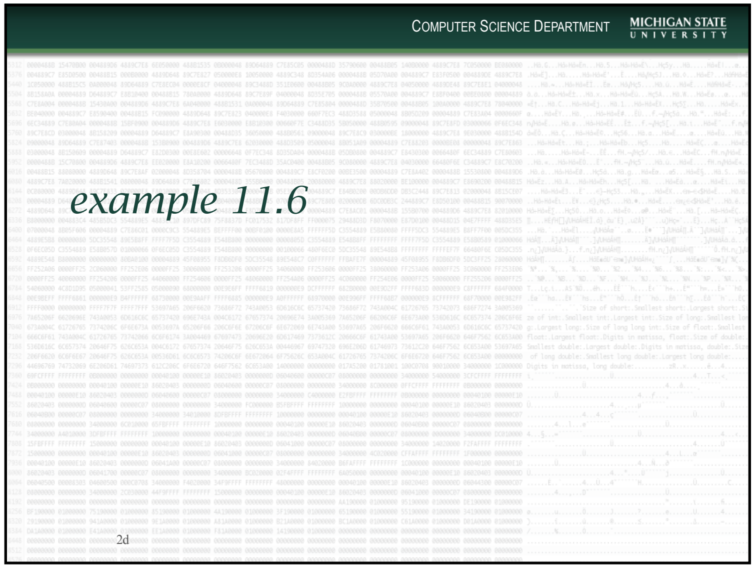


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

Use cin ops

```
string word;
char ch;
istringstream iss ("hello world");
iss >> word;   //space sep, "hello"
iss.get(ch);   // the space
iss.get(ch);   // 'w'
```

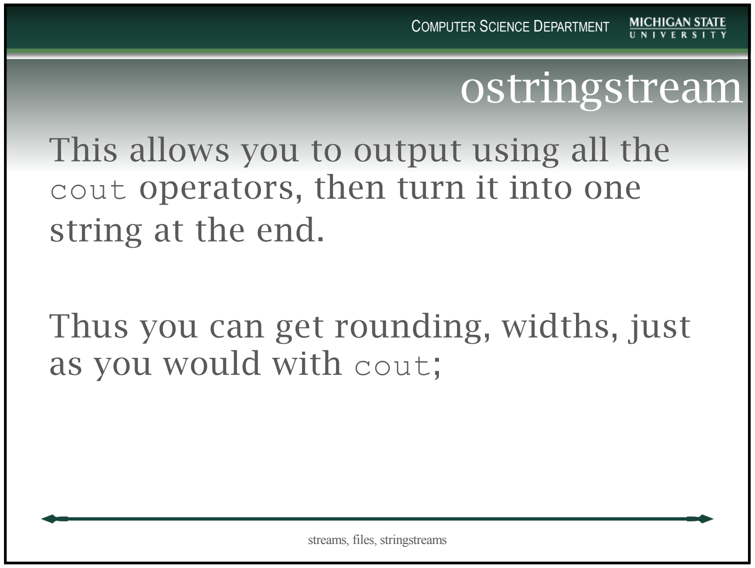
streams, files, stringstreams



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

example 11.6

2d



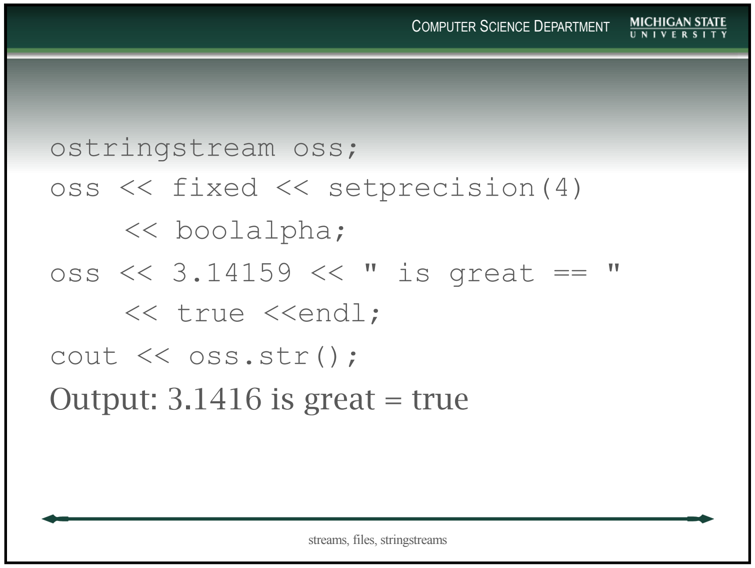
COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

ostringstream

This allows you to output using all the `cout` operators, then turn it into one string at the end.

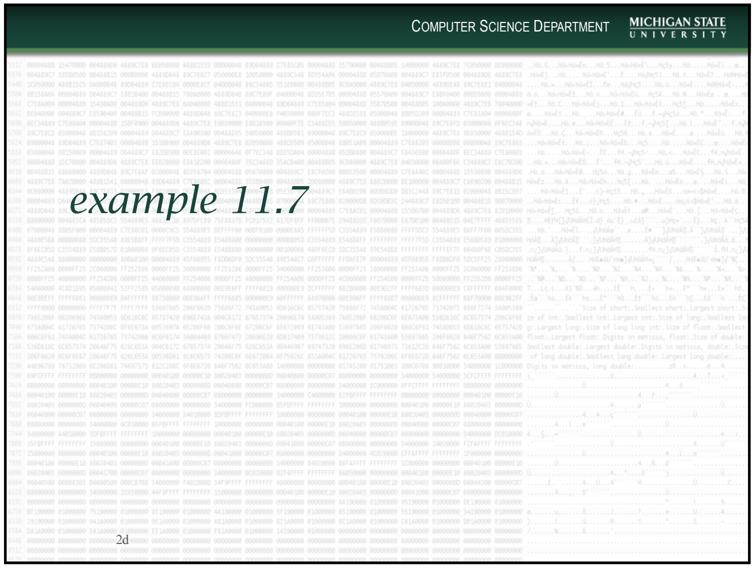
Thus you can get rounding, widths, just as you would with `cout`;

streams, files, stringstreams

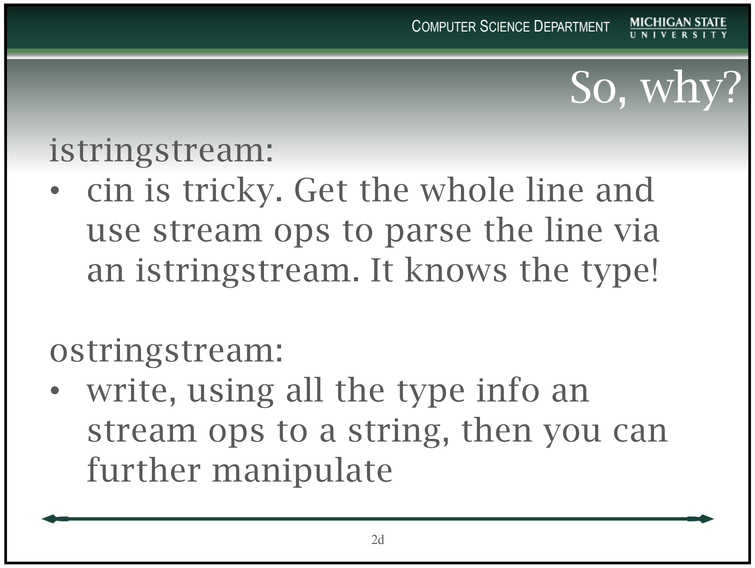


```
ostringstream oss;
oss << fixed << setprecision(4)
     << boolalpha;
oss << 3.14159 << " is great == "
     << true <<endl;
cout << oss.str();
```

Output: 3.1416 is great = true



example 11.7



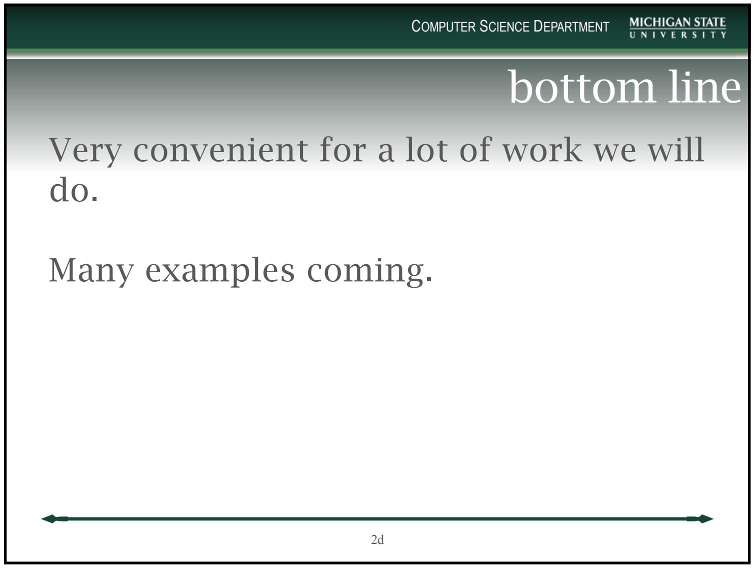
So, why?

istringstream:

- cin is tricky. Get the whole line and use stream ops to parse the line via an istringstream. It knows the type!

ostringstream:

- write, using all the type info an stream ops to a string, then you can further manipulate



bottom line

Very convenient for a lot of work we will do.

Many examples coming.