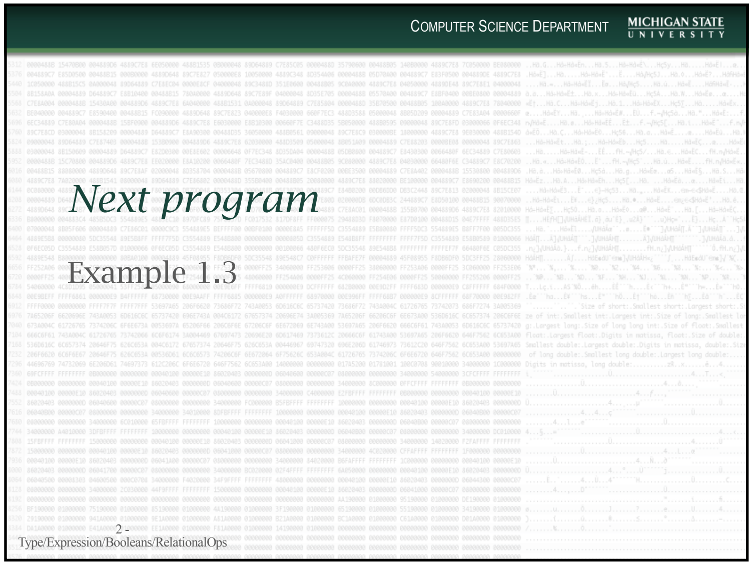
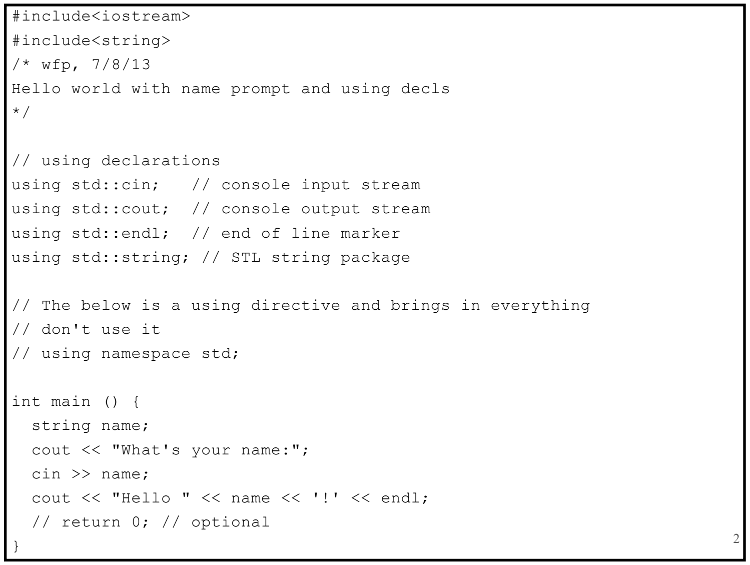
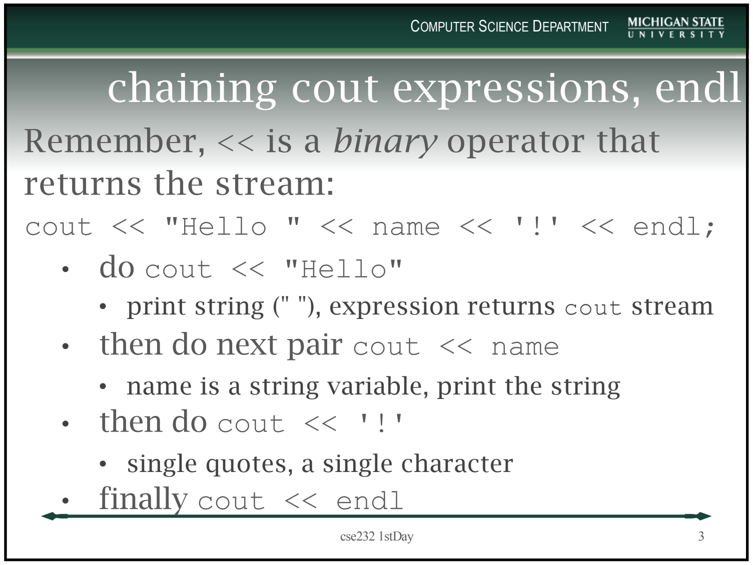


## *Next program*

Example 1.3



```
#include<iostream>
#include<string>
/* wfp, 7/8/13
Hello world with name prompt and using decls
*/

// using declarations
using std::cin;   // console input stream
using std::cout;  // console output stream
using std::endl;  // end of line marker
using std::string; // STL string package

// The below is a using directive and brings in everything
// don't use it
// using namespace std;

int main () {
  string name;
  cout << "What's your name:";
  cin >> name;
  cout << "Hello " << name << '!' << endl;
  // return 0; // optional
}
```

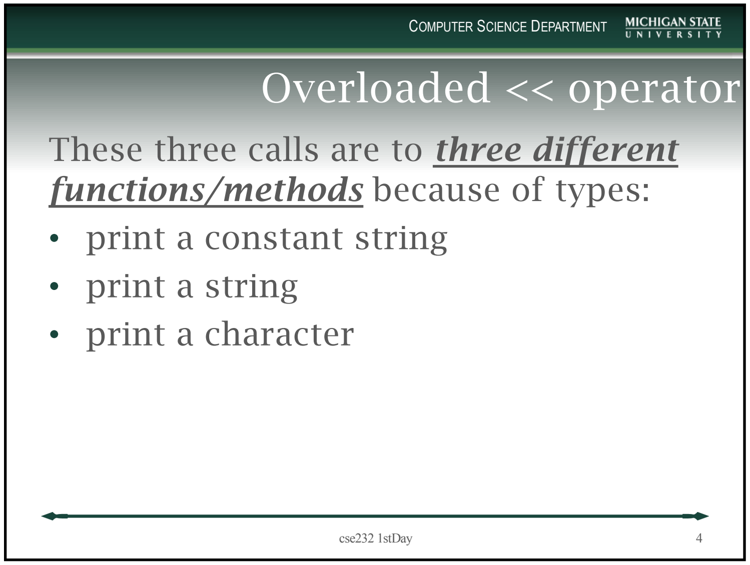


## chaining cout expressions, endl

Remember, << is a *binary* operator that returns the stream:

`cout << "Hello " << name << '!' << endl;`

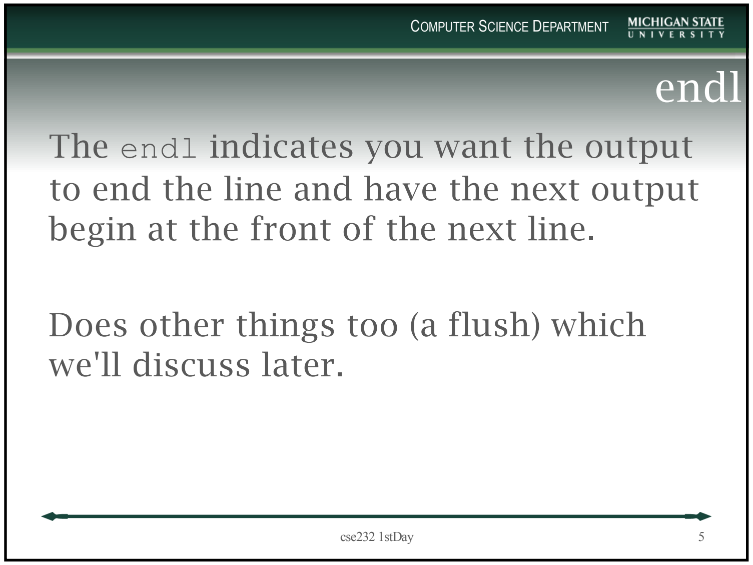
- do `cout << "Hello"`
  - print string (" "), expression returns `cout` stream
- then do next pair `cout << name`
  - name is a string variable, print the string
- then do `cout << '!'`
  - single quotes, a single character
- finally `cout << endl`



## Overloaded << operator

These three calls are to ***three different functions/methods*** because of types:

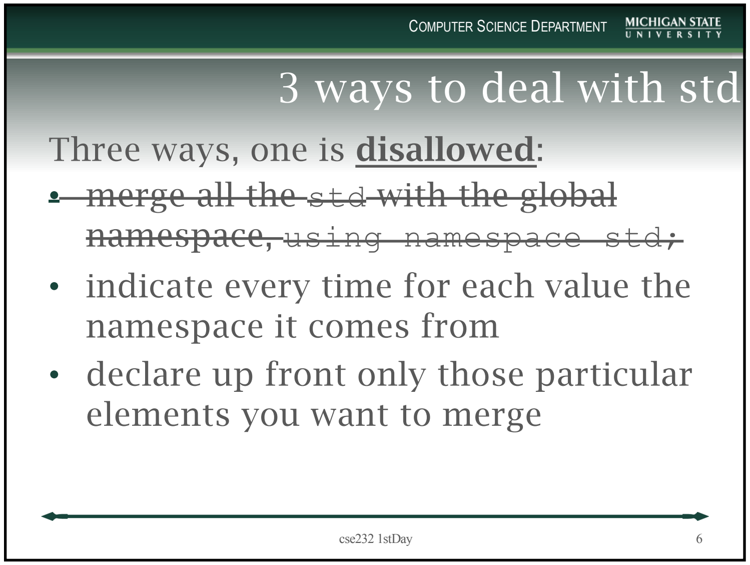
- print a constant string
- print a string
- print a character



## endl

The `endl` indicates you want the output to end the line and have the next output begin at the front of the next line.

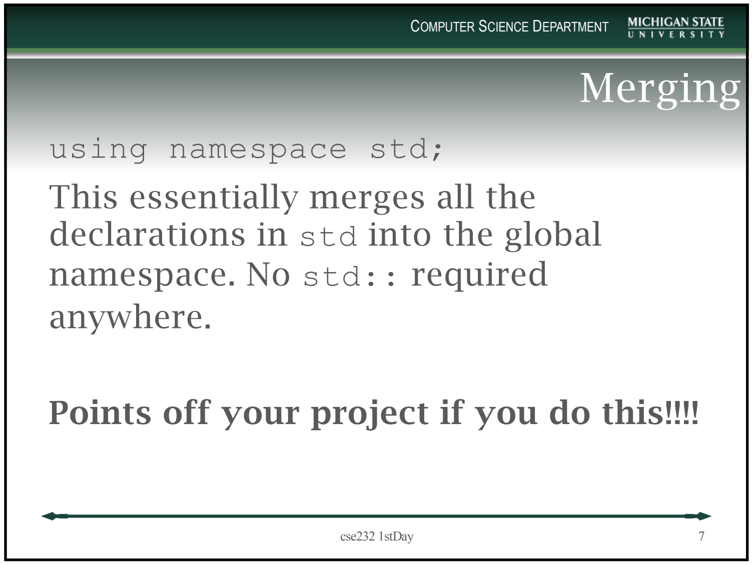
Does other things too (a flush) which we'll discuss later.



## 3 ways to deal with std

Three ways, one is **disallowed**:

- ~~merge all the `std` with the global namespace, `using namespace std;`~~
- indicate every time for each value the namespace it comes from
- declare up front only those particular elements you want to merge

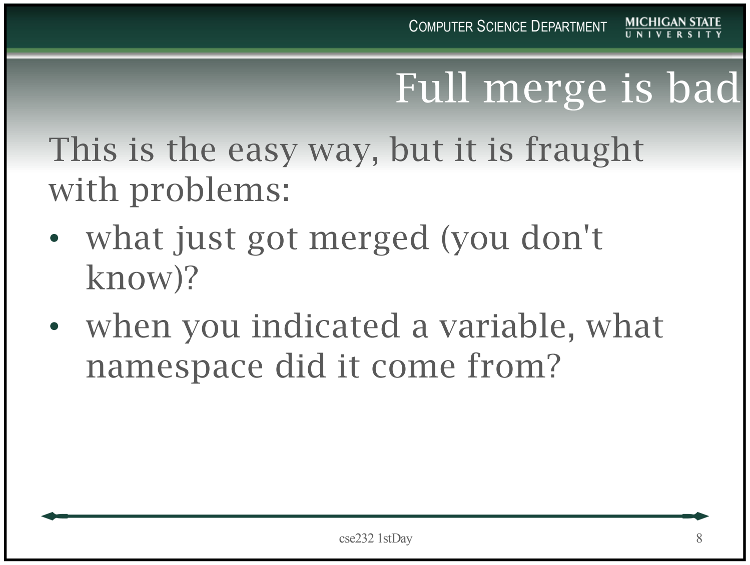


## Merging

```
using namespace std;
```

This essentially merges all the declarations in `std` into the global namespace. No `std::` required anywhere.

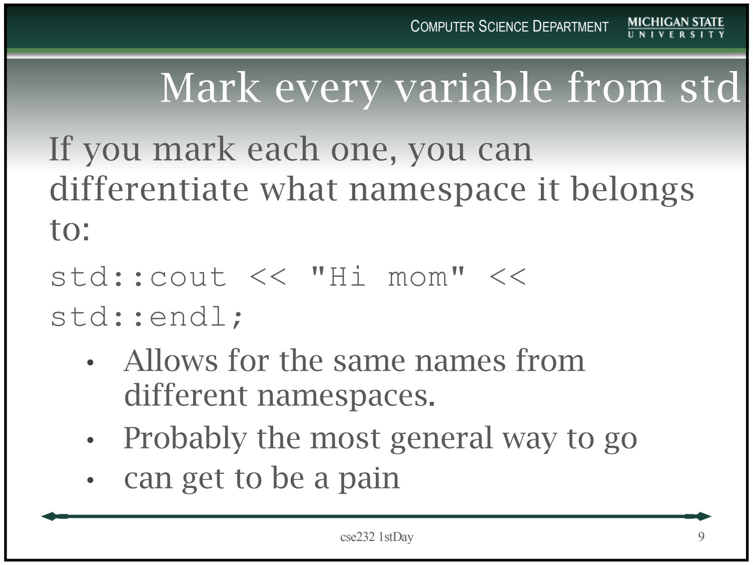
**Points off your project if you do this!!!!**



## Full merge is bad

This is the easy way, but it is fraught with problems:

- what just got merged (you don't know)?
- when you indicated a variable, what namespace did it come from?



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

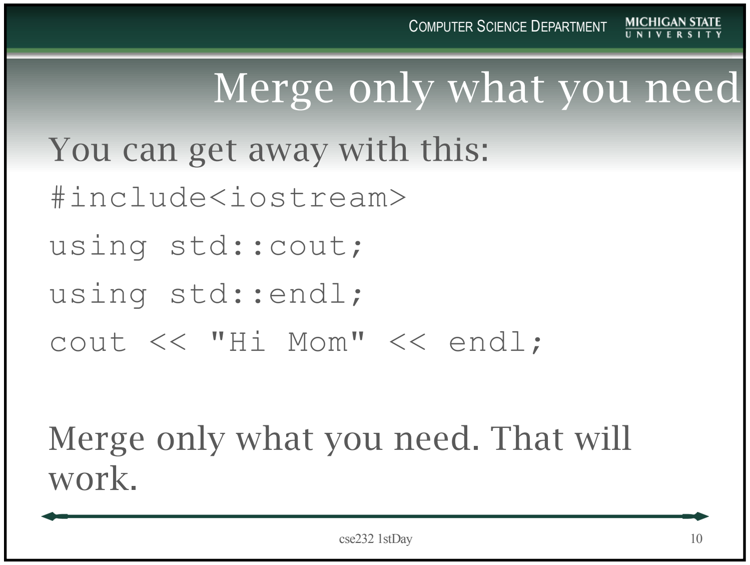
## Mark every variable from std

If you mark each one, you can differentiate what namespace it belongs to:

```
std::cout << "Hi mom" <<
std::endl;
```

- Allows for the same names from different namespaces.
- Probably the most general way to go
- can get to be a pain

cse232 1stDay 9



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

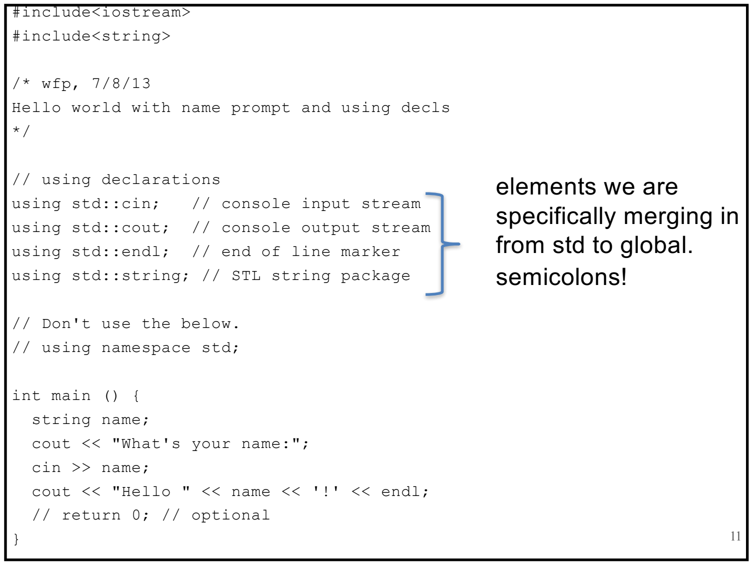
## Merge only what you need

You can get away with this:

```
#include<iostream>
using std::cout;
using std::endl;
cout << "Hi Mom" << endl;
```

Merge only what you need. That will work.

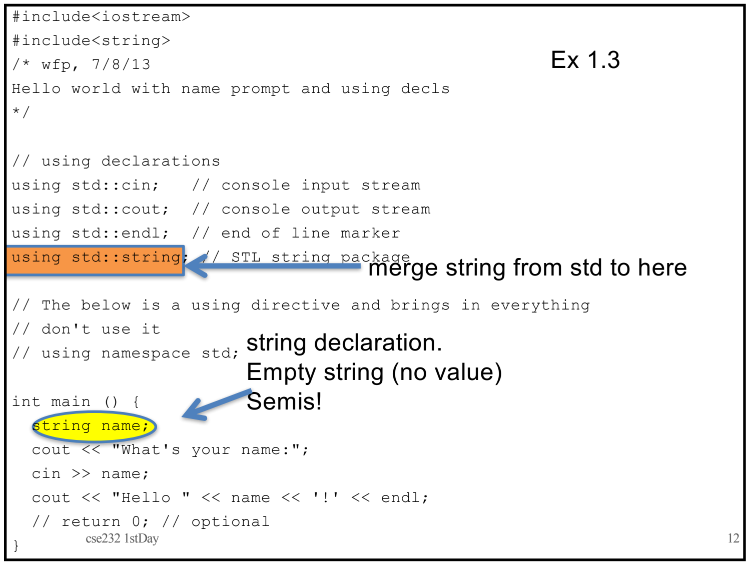
cse232 1stDay 10



```
#include<iostream>
#include<string>

/* wfp, 7/8/13
Hello world with name prompt and using decls
*/

// using declarations
using std::cin;   // console input stream
using std::cout;  // console output stream
using std::endl;  // end of line marker
using std::string; // STL string package

// Don't use the below.
// using namespace std;

int main () {
  string name;
  cout << "What's your name:";
  cin >> name;
  cout << "Hello " << name << '!' << endl;
  // return 0; // optional
}
```

elements we are specifically merging in from std to global. semicolons!

11



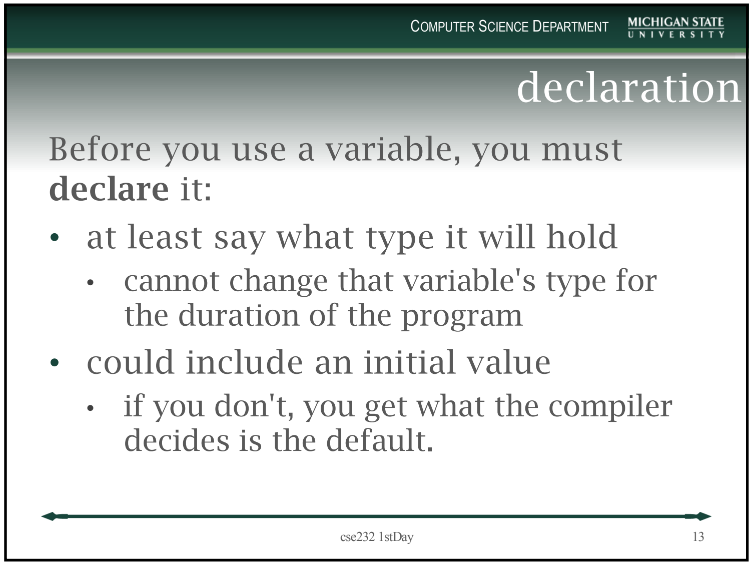
Ex 1.3

```
#include<iostream>
#include<string>
/* wfp, 7/8/13
Hello world with name prompt and using decls
*/

// using declarations
using std::cin;   // console input stream
using std::cout;  // console output stream
using std::endl;  // end of line marker
using std::string; // STL string package

// The below is a using directive and brings in everything
// don't use it
// using namespace std;

int main () {
  string name;
  cout << "What's your name:";
  cin >> name;
  cout << "Hello " << name << '!' << endl;
  // return 0; // optional
}
```

merge string from std to here

string declaration.
Empty string (no value)
Semis!

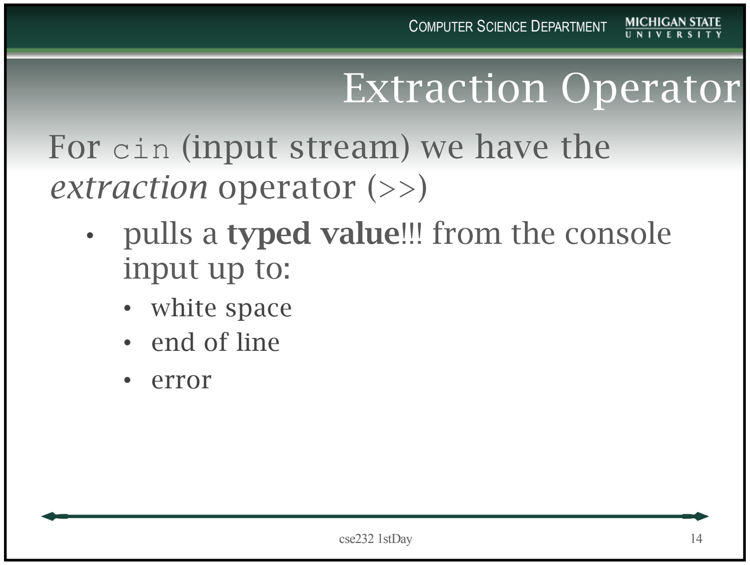
cse232 1stDay 12



## declaration

Before you use a variable, you must **declare** it:

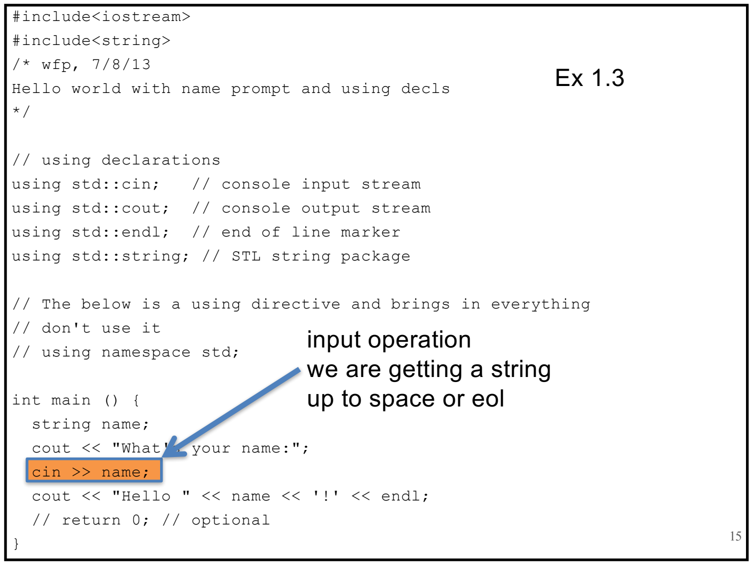
- at least say what type it will hold
  - cannot change that variable's type for the duration of the program
- could include an initial value
  - if you don't, you get what the compiler decides is the default.



## Extraction Operator

For `cin` (input stream) we have the *extraction* operator (>>)

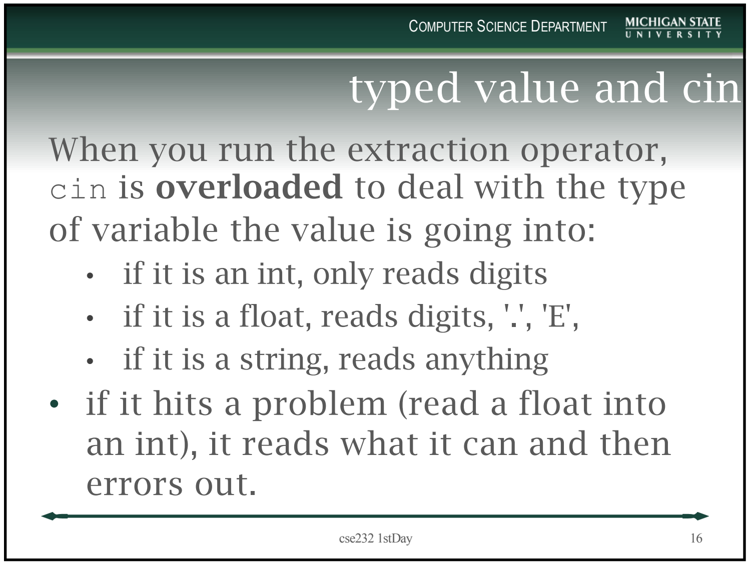
- pulls a **typed value**!!! from the console input up to:
  - white space
  - end of line
  - error



Ex 1.3

```
#include<iostream>
#include<string>
/* wfp, 7/8/13
Hello world with name prompt and using decls
*/

// using declarations
using std::cin;   // console input stream
using std::cout;  // console output stream
using std::endl;  // end of line marker
using std::string; // STL string package

// The below is a using directive and brings in everything
// don't use it
// using namespace std;

int main () {
  string name;
  cout << "What's your name:";
  cin >> name;
  cout << "Hello " << name << '!' << endl;
  // return 0; // optional
}
```

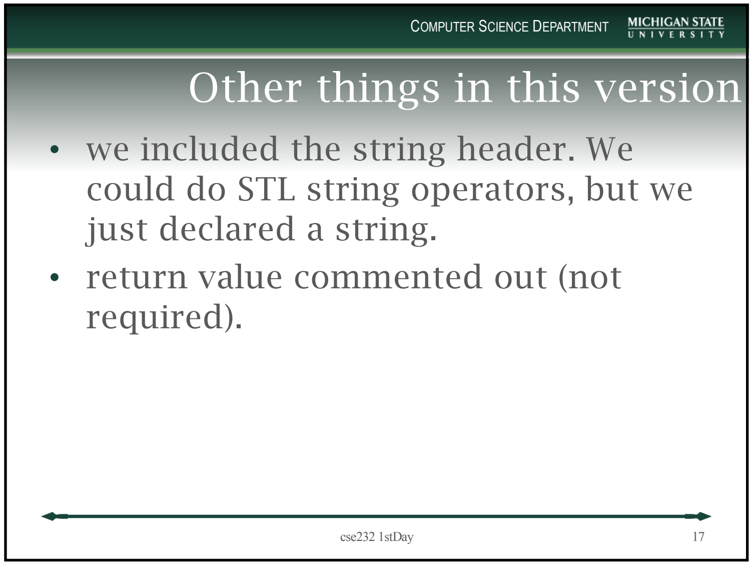
input operation
we are getting a string
up to space or eol



## typed value and cin

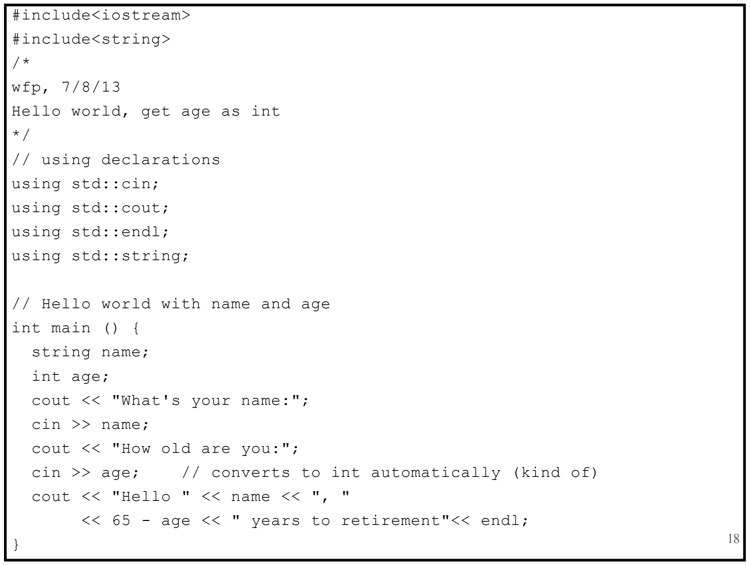
When you run the extraction operator, `cin` is **overloaded** to deal with the type of variable the value is going into:

- if it is an int, only reads digits
- if it is a float, reads digits, '.', 'E',
- if it is a string, reads anything
- if it hits a problem (read a float into an int), it reads what it can and then errors out.

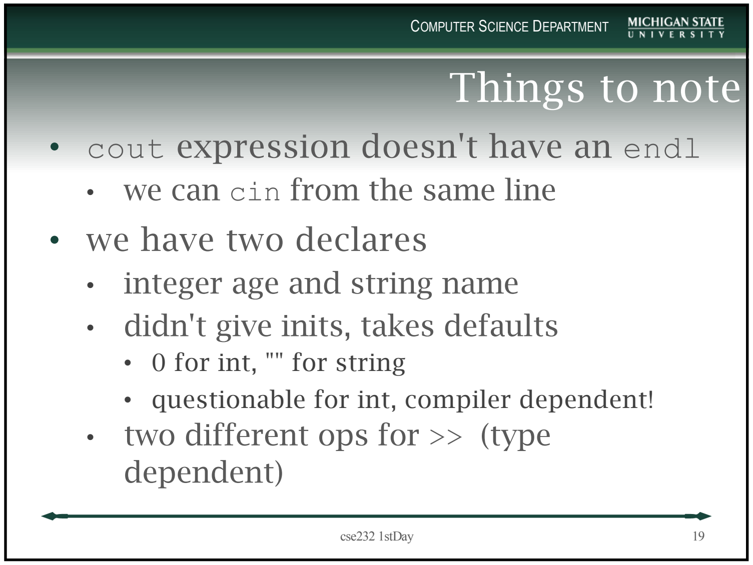


## Other things in this version

- we included the string header. We could do STL string operators, but we just declared a string.
- return value commented out (not required).

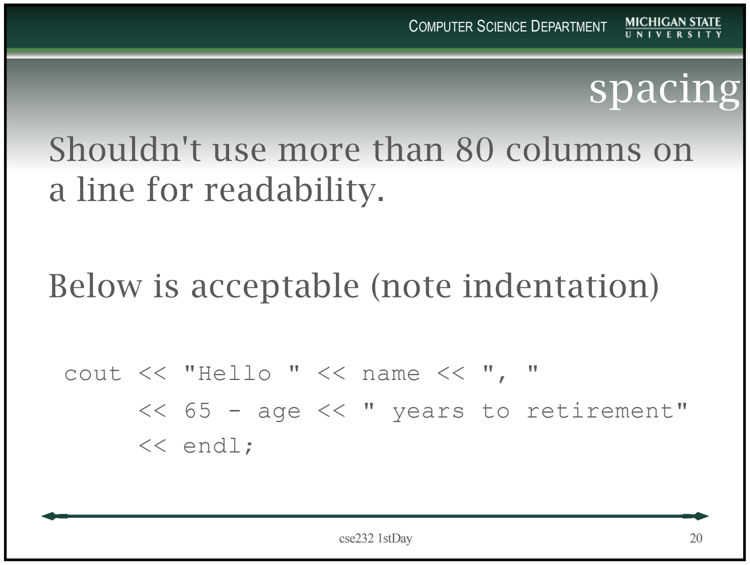


```
#include<iostream>
#include<string>
/*
wfp, 7/8/13
Hello world, get age as int
*/
// using declarations
using std::cin;
using std::cout;
using std::endl;
using std::string;

// Hello world with name and age
int main () {
  string name;
  int age;
  cout << "What's your name:";
  cin >> name;
  cout << "How old are you:";
  cin >> age;    // converts to int automatically (kind of)
  cout << "Hello " << name << ", "
       << 65 - age << " years to retirement"<< endl;
}
```



## Things to note

- `cout` expression doesn't have an `endl`
  - we can `cin` from the same line
- we have two declares
  - integer age and string name
  - didn't give inits, takes defaults
    - 0 for int, "" for string
    - questionable for int, compiler dependent!
  - two different ops for >> (type dependent)

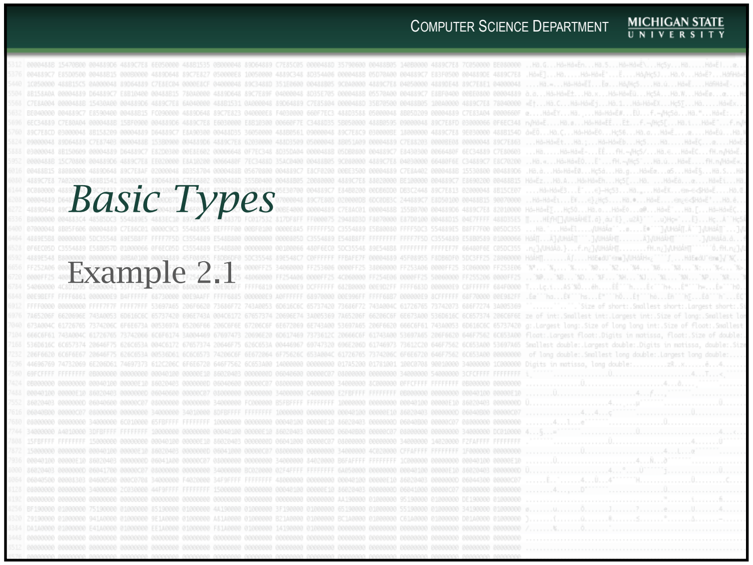


## spacing

Shouldn't use more than 80 columns on a line for readability.

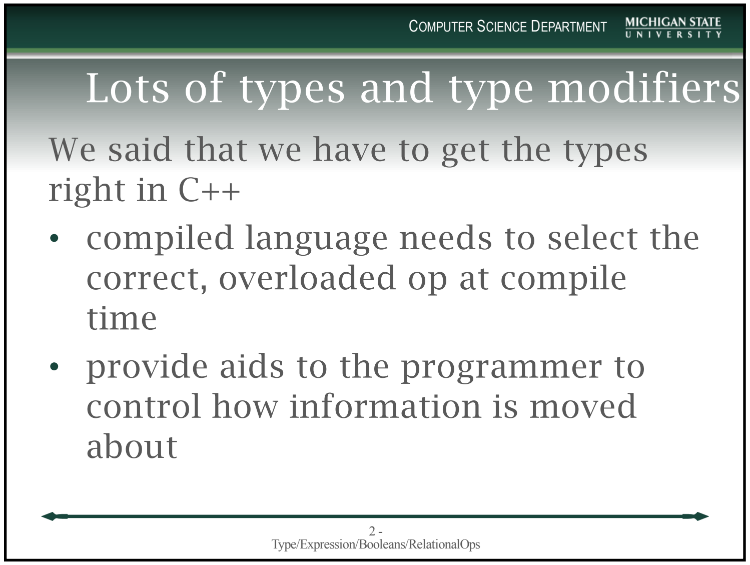
Below is acceptable (note indentation)

```
cout << "Hello " << name << ", "
     << 65 - age << " years to retirement"
     << endl;
```



## *Basic Types*

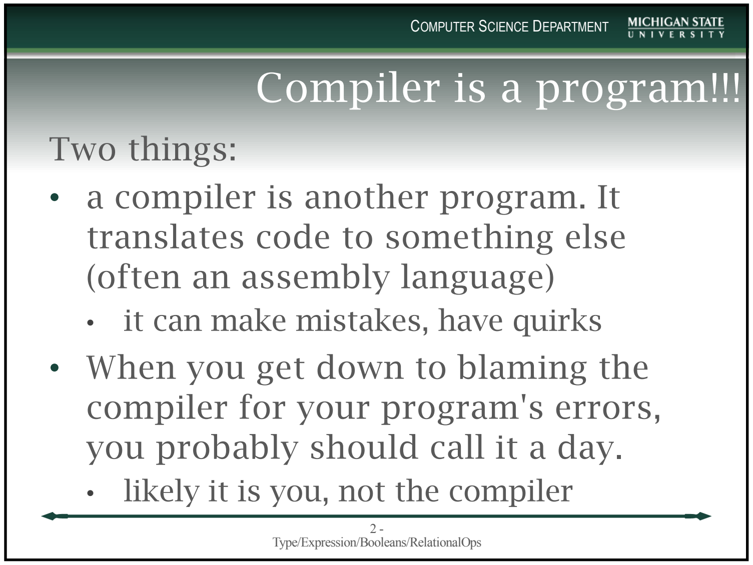
Example 2.1



## Lots of types and type modifiers

We said that we have to get the types right in C++

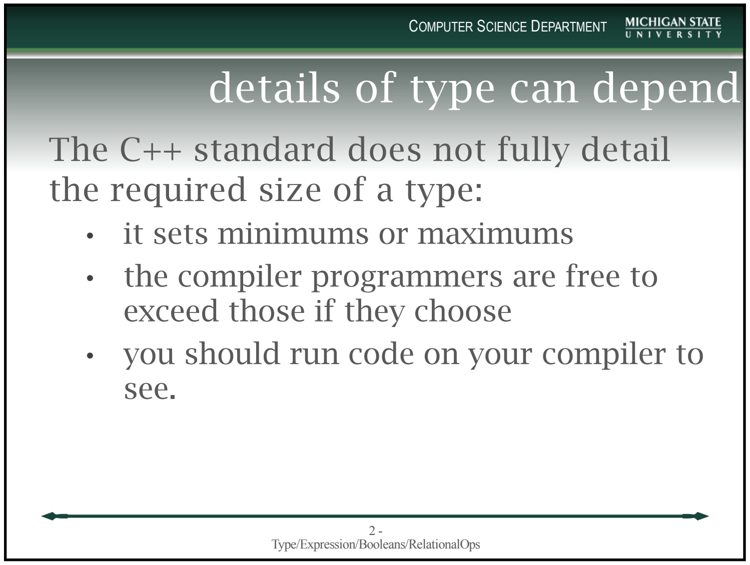
- compiled language needs to select the correct, overloaded op at compile time
- provide aids to the programmer to control how information is moved about



## Compiler is a program!!!

Two things:

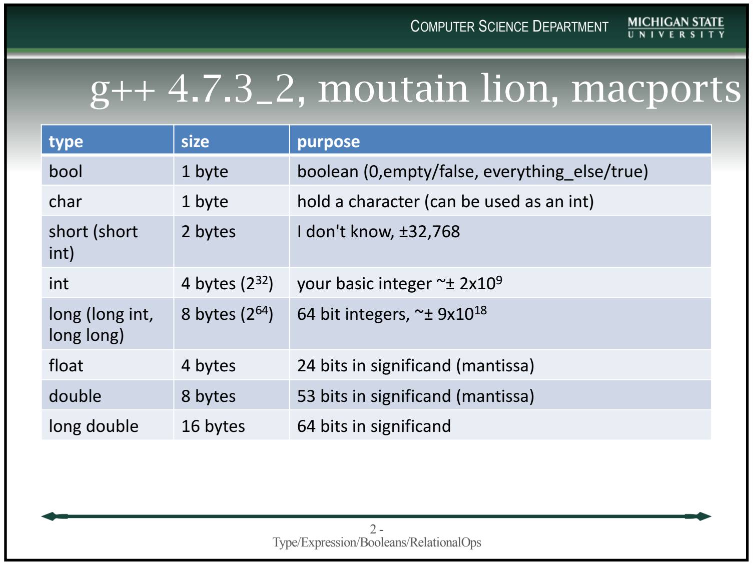
- a compiler is another program. It translates code to something else (often an assembly language)
  - it can make mistakes, have quirks
- When you get down to blaming the compiler for your program's errors, you probably should call it a day.
  - likely it is you, not the compiler



## details of type can depend

The C++ standard does not fully detail the required size of a type:

- it sets minimums or maximums
- the compiler programmers are free to exceed those if they choose
- you should run code on your compiler to see.

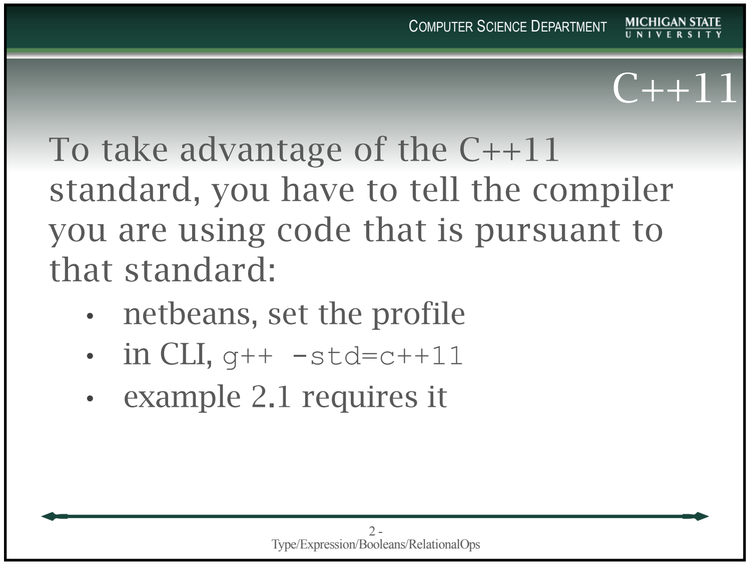


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## g++ 4.7.3_2, moutain lion, macports

| type | size | purpose |
|---|---|---|
| bool | 1 byte | boolean (0,empty/false, everything_else/true) |
| char | 1 byte | hold a character (can be used as an int) |
| short (short int) | 2 bytes | I don't know, ±32,768 |
| int | 4 bytes ($2^{32}$) | your basic integer ~± $2x10^9$ |
| long (long int, long long) | 8 bytes ($2^{64}$) | 64 bit integers, ~± $9x10^{18}$ |
| float | 4 bytes | 24 bits in significand (mantissa) |
| double | 8 bytes | 53 bits in significand (mantissa) |
| long double | 16 bytes | 64 bits in significand |

2 -
Type/Expression/Booleans/RelationalOps



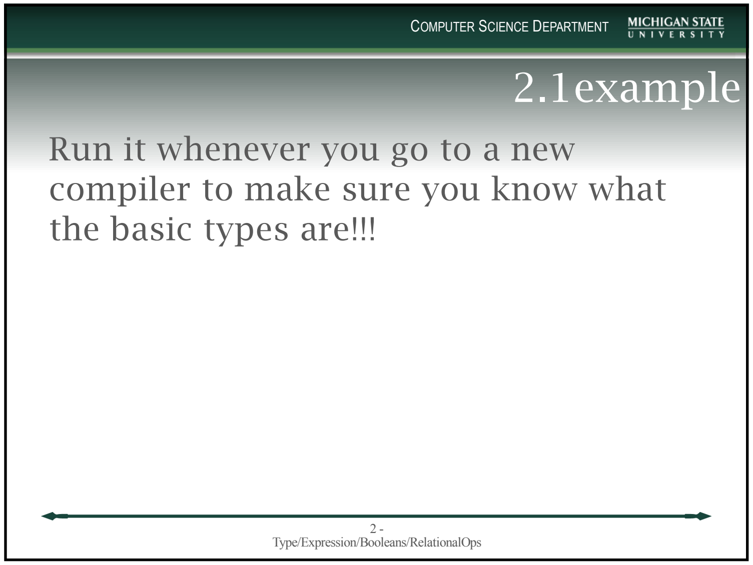
COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## C++11

To take advantage of the C++11 standard, you have to tell the compiler you are using code that is pursuant to that standard:

- netbeans, set the profile
- in CLI, `g++ -std=c++11`
- example 2.1 requires it

2 -
Type/Expression/Booleans/RelationalOps

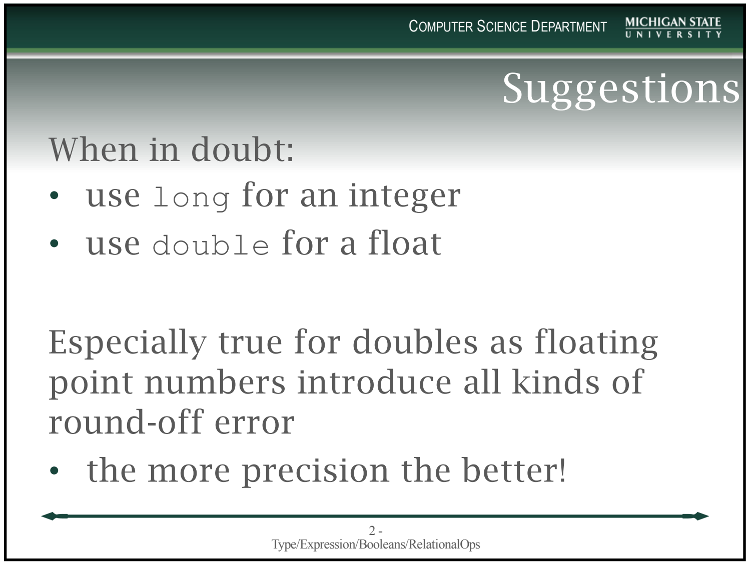


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## 2.1example

Run it whenever you go to a new compiler to make sure you know what the basic types are!!!

2 -
Type/Expression/Booleans/RelationalOps



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## Suggestions

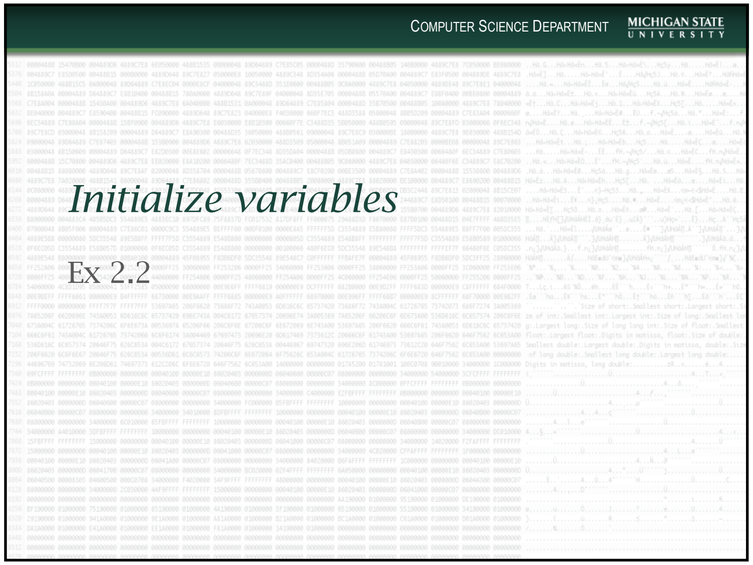
When in doubt:

- use `long` for an integer
- use `double` for a float

Especially true for doubles as floating point numbers introduce all kinds of round-off error

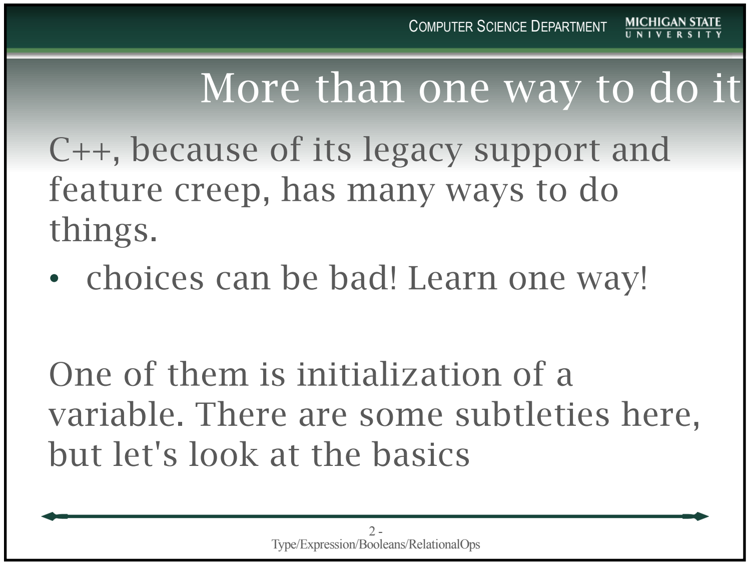
- the more precision the better!

2 -
Type/Expression/Booleans/RelationalOps



## *Initialize variables*

Ex 2.2

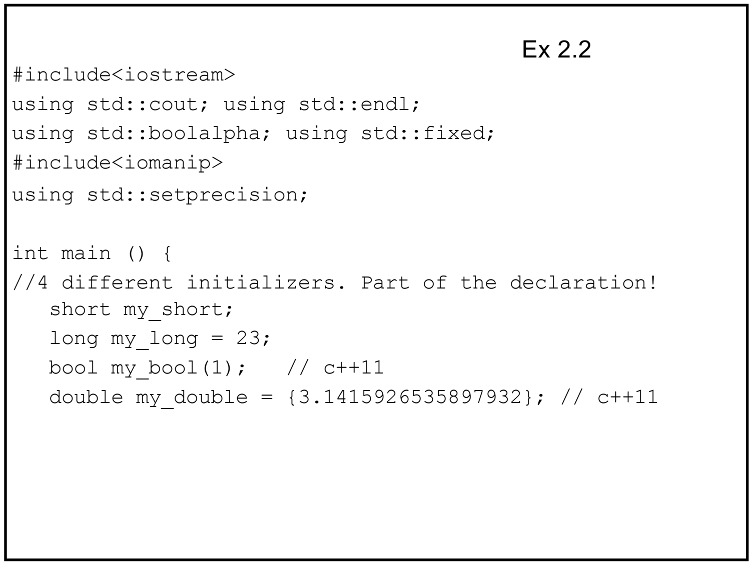


## More than one way to do it

C++, because of its legacy support and feature creep, has many ways to do things.

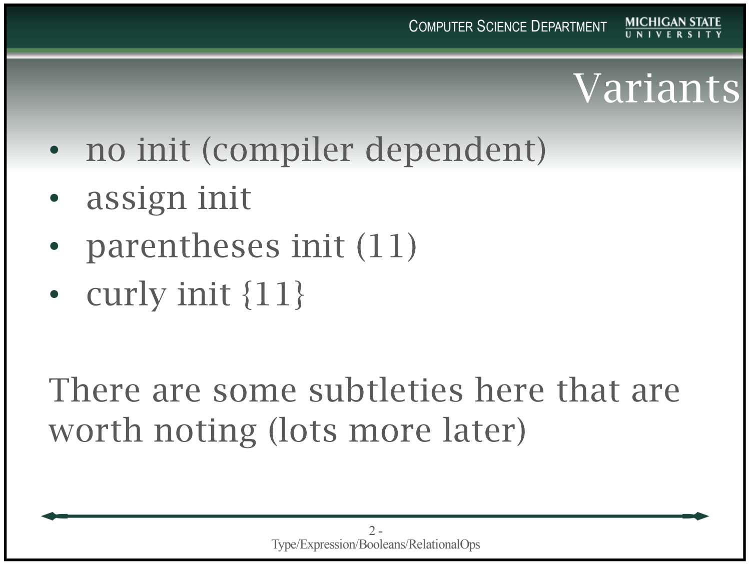
- choices can be bad! Learn one way!

One of them is initialization of a variable. There are some subtleties here, but let's look at the basics



Ex 2.2

```
#include<iostream>
using std::cout; using std::endl;
using std::boolalpha; using std::fixed;
#include<iomanip>
using std::setprecision;

int main () {
//4 different initializers. Part of the declaration!
   short my_short;
   long my_long = 23;
   bool my_bool(1);    // c++11
   double my_double = {3.1415926535897932}; // c++11
```



## Variants

- no init (compiler dependent)
- assign init
- parentheses init (11)
- curly init {11}

There are some subtleties here that are worth noting (lots more later)

## Three types

These are initializations because they are ***declaring*** a variable

Direct Initialization (both equivalent)

```
long my_long(my_int);
int my_int = 123;
```

Initializer List (depends on type)

```
long another_long{1};
```

## C++ and efficiency

Unlike in Python, in C++ we worry about efficiency.

- one of the main reasons to use C++
- can cause us complications (but that is kind of the point)

For efficiency's sake, we want to avoid copies (because they can be expensive)

## What does = mean?

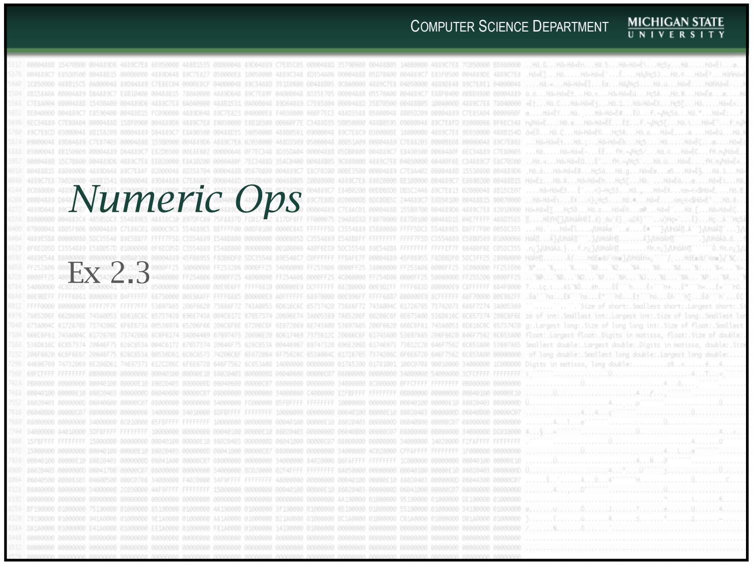
Remember the context problem for C++?

The = (equal) sign means different things in different ***contexts***:

```
int my_int = 23;  // initialization
my_int = 123;     // different op, assign
```

## *Expressions*

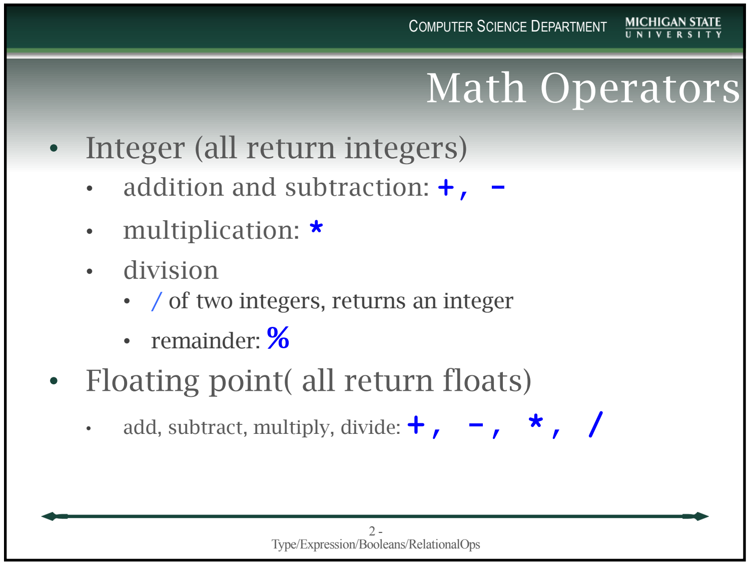
"Never express yourself more clearly than you are able to think." - Niels Bohr



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

# *Numeric Ops*

Ex 2.3

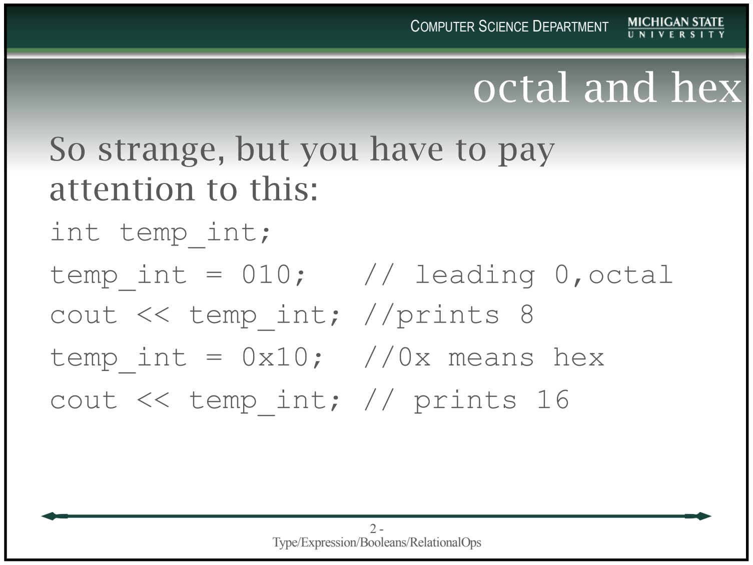


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## Math Operators

- Integer (all return integers)
  - addition and subtraction: `+, -`
  - multiplication: `*`
  - division
    - `/` of two integers, returns an integer
    - remainder: `%`
- Floating point( all return floats)
  - add, subtract, multiply, divide: `+, -, *, /`

2 -
Type/Expression/Booleans/RelationalOps



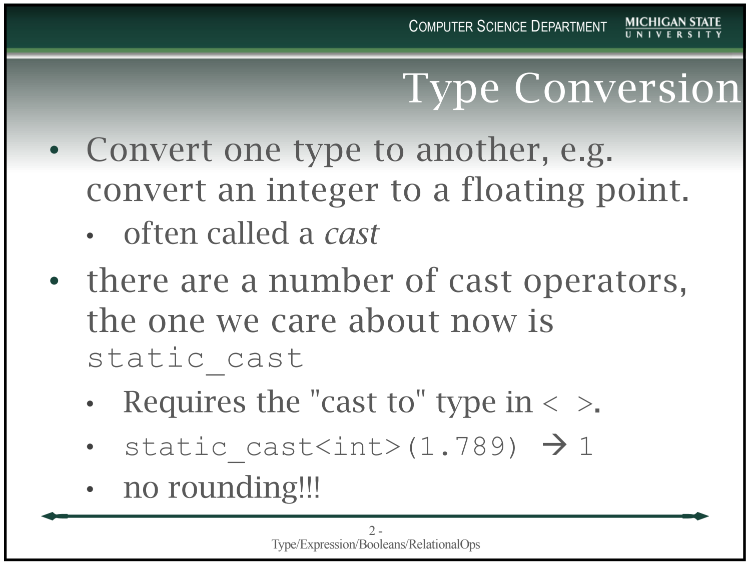
COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## octal and hex

So strange, but you have to pay attention to this:

```
int temp_int;
temp_int = 010;   // leading 0,octal
cout << temp_int; //prints 8
temp_int = 0x10;  //0x means hex
cout << temp_int; // prints 16
```

2 -
Type/Expression/Booleans/RelationalOps

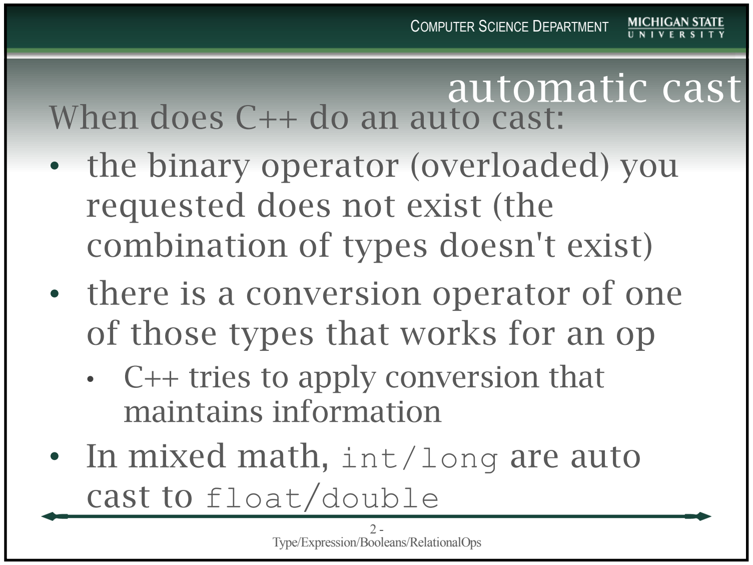


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## Type Conversion

- Convert one type to another, e.g. convert an integer to a floating point.
  - often called a *cast*
- there are a number of cast operators, the one we care about now is `static_cast`
  - Requires the "cast to" type in < >.
  - `static_cast<int>(1.789)` → 1
  - no rounding!!!

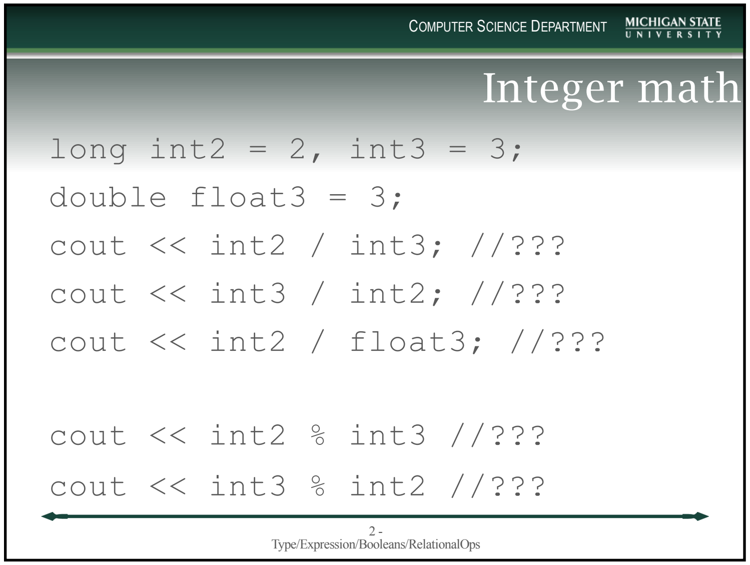
2 -
Type/Expression/Booleans/RelationalOps



## automatic cast

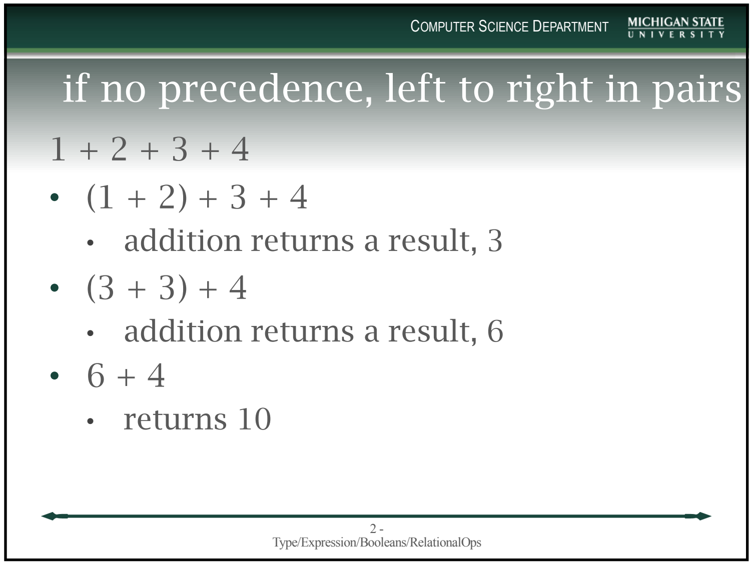
When does C++ do an auto cast:

- the binary operator (overloaded) you requested does not exist (the combination of types doesn't exist)
- there is a conversion operator of one of those types that works for an op
  - C++ tries to apply conversion that maintains information
- In mixed math, `int/long` are auto cast to `float/double`



## Integer math

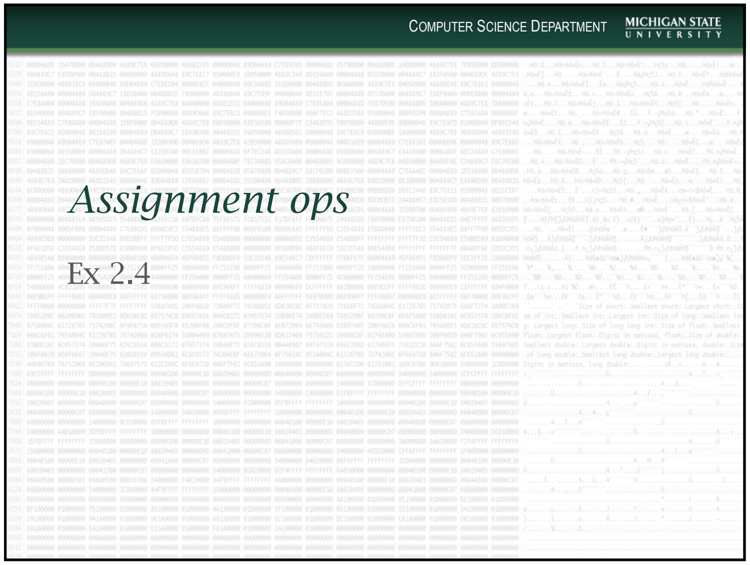
```
long int2 = 2, int3 = 3;
double float3 = 3;
cout << int2 / int3; //???
cout << int3 / int2; //???
cout << int2 / float3; //???

cout << int2 % int3 //???
cout << int3 % int2 //???
```



## if no precedence, left to right in pairs

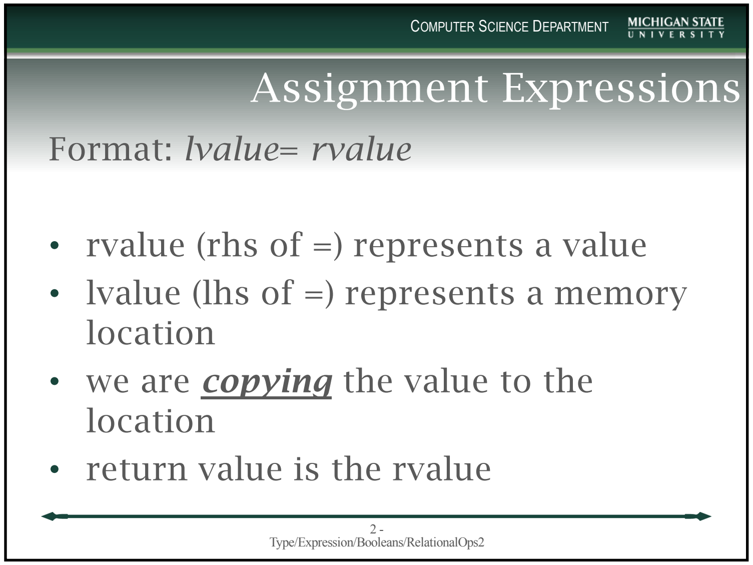
1 + 2 + 3 + 4

- (1 + 2) + 3 + 4
  - addition returns a result, 3
- (3 + 3) + 4
  - addition returns a result, 6
- 6 + 4
  - returns 10



## *Assignment ops*

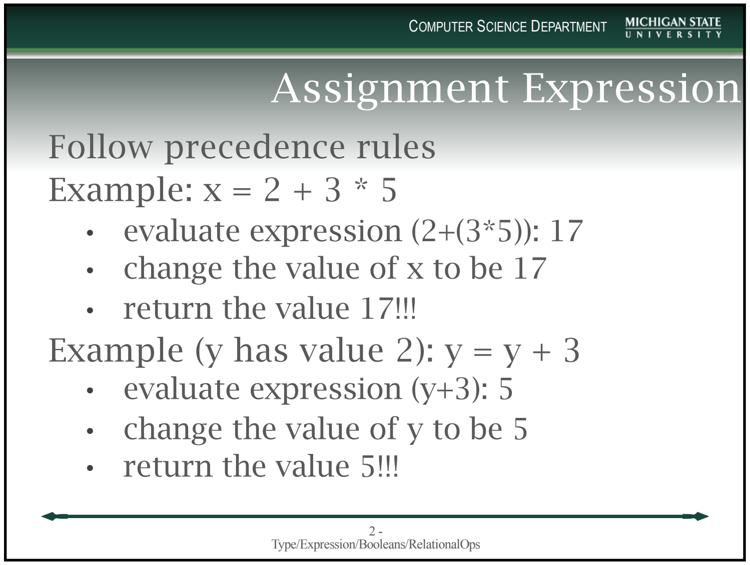
Ex 2.4



## Assignment Expressions

Format: *lvalue*= *rvalue*

- rvalue (rhs of =) represents a value
- lvalue (lhs of =) represents a memory location
- we are ***copying*** the value to the location
- return value is the rvalue



## Assignment Expression

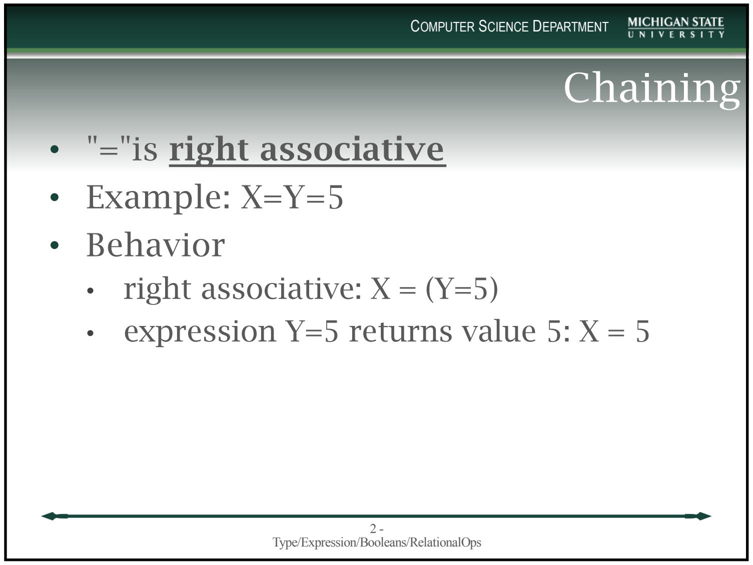
Follow precedence rules

Example: x = 2 + 3 * 5

- evaluate expression (2+(3*5)): 17
- change the value of x to be 17
- return the value 17!!!

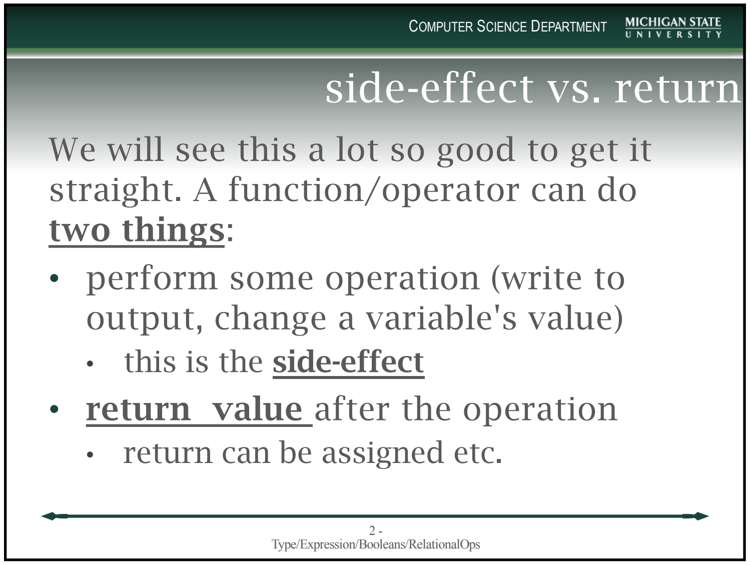
Example (y has value 2): y = y + 3

- evaluate expression (y+3): 5
- change the value of y to be 5
- return the value 5!!!



## Chaining

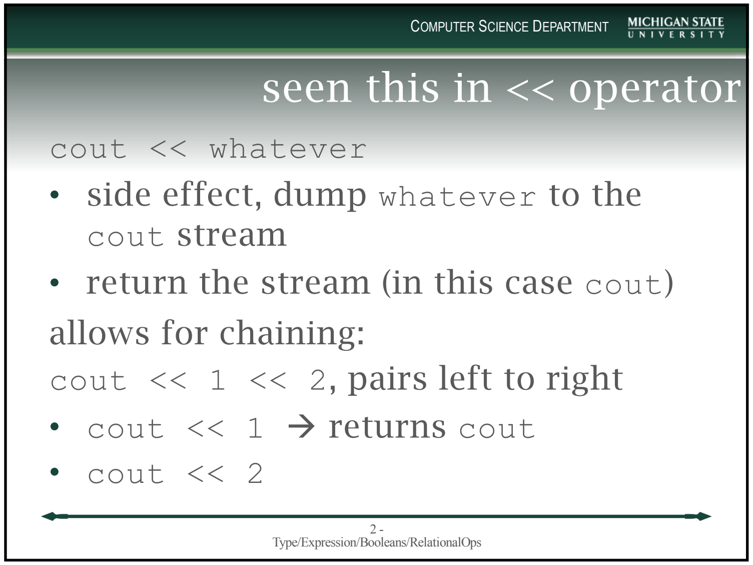
- "="is **right associative**
- Example: X=Y=5
- Behavior
  - right associative: X = (Y=5)
  - expression Y=5 returns value 5: X = 5



## side-effect vs. return

We will see this a lot so good to get it straight. A function/operator can do **two things**:

- perform some operation (write to output, change a variable's value)
  - this is the **side-effect**
- **return value** after the operation
  - return can be assigned etc.



## seen this in << operator

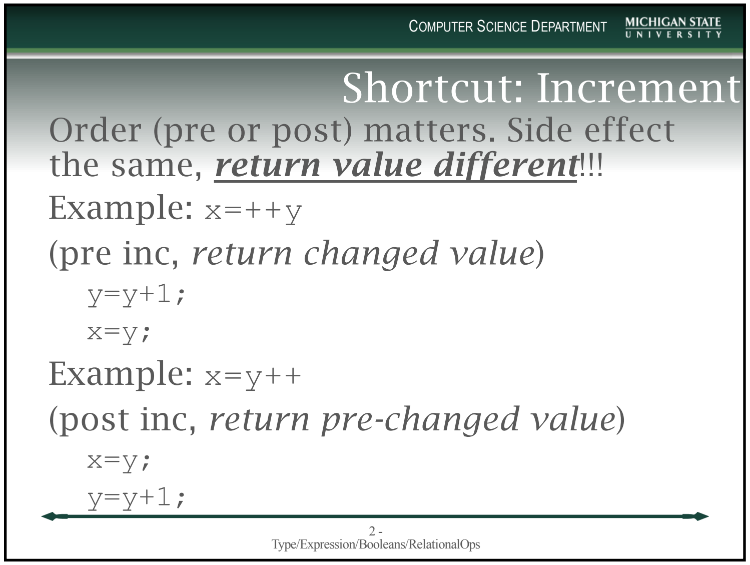
`cout << whatever`

- side effect, dump `whatever` to the `cout` stream
- return the stream (in this case `cout`)

allows for chaining:

`cout << 1 << 2`, pairs left to right

- `cout << 1` → returns `cout`
- `cout << 2`



## Shortcut: Increment

Order (pre or post) matters. Side effect the same, ***return value different***!!!

Example: `x=++y`

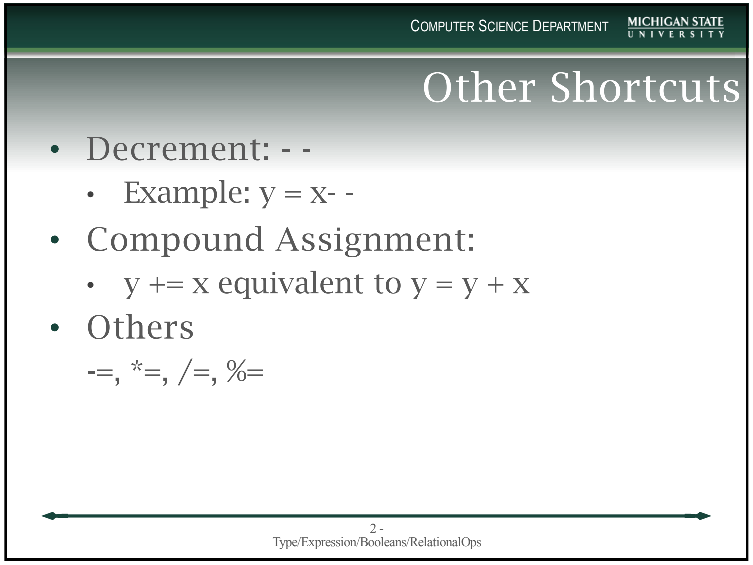
(pre inc, *return changed value*)

```
y=y+1;
x=y;
```

Example: `x=y++`

(post inc, *return pre-changed value*)

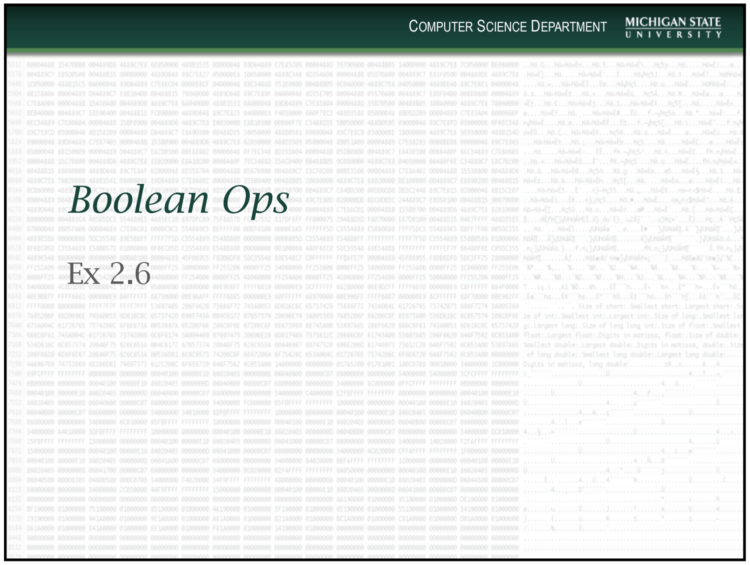
```
x=y;
y=y+1;
```



## Other Shortcuts

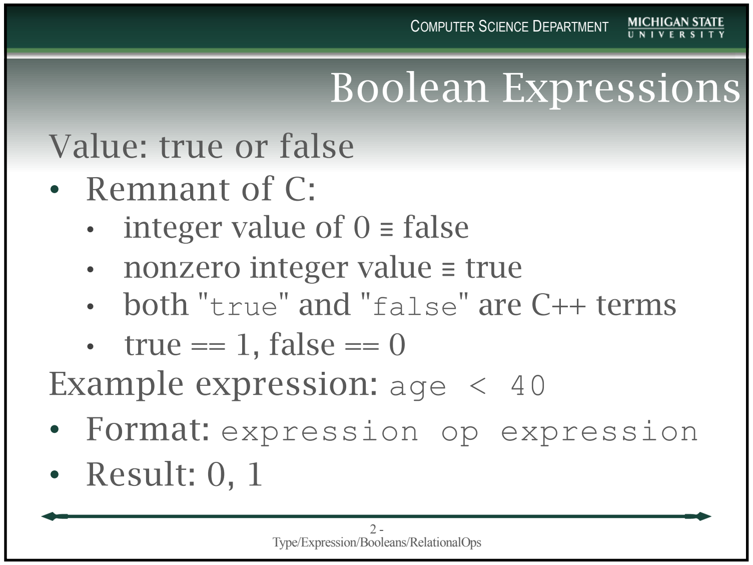
- Decrement: - -
  - Example: y = x- -
- Compound Assignment:
  - y += x equivalent to y = y + x
- Others

  -=, *=, /=, %=



## *Boolean Ops*

Ex 2.6



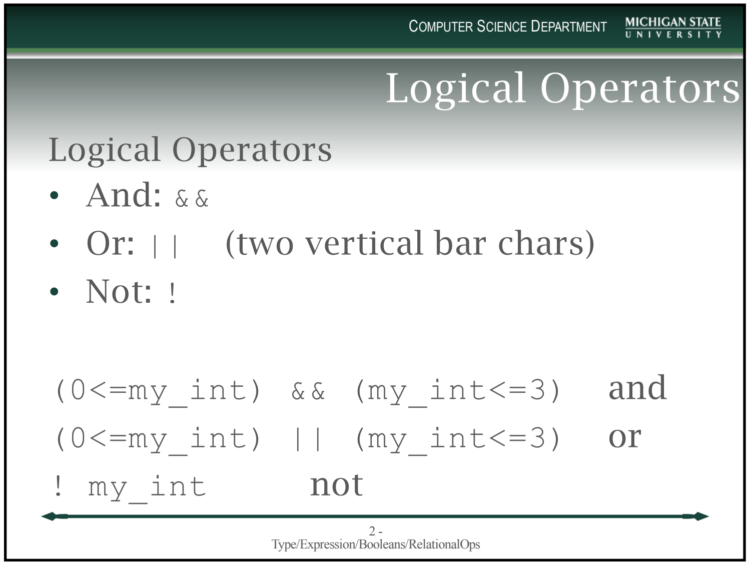
## Boolean Expressions

Value: true or false

- Remnant of C:
  - integer value of 0 ≡ false
  - nonzero integer value ≡ true
  - both "`true`" and "`false`" are C++ terms
  - true == 1, false == 0

Example expression: `age < 40`

- Format: `expression op expression`
- Result: 0, 1

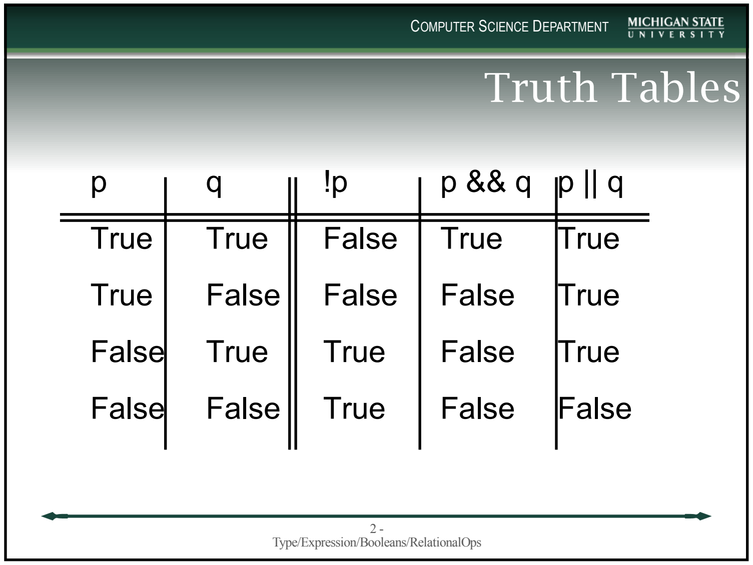


## Logical Operators

Logical Operators

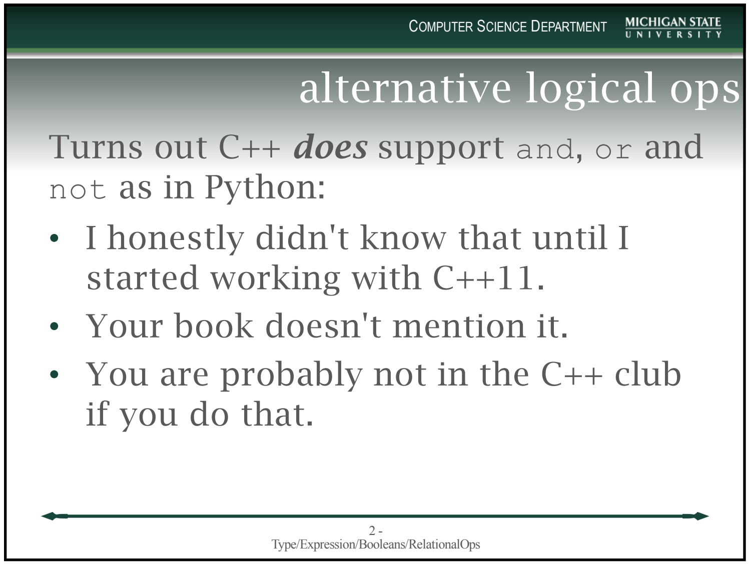
- And: `&&`
- Or: `||` (two vertical bar chars)
- Not: `!`

```
(0<=my_int) && (my_int<=3)   and
(0<=my_int) || (my_int<=3)   or
! my_int       not
```



## Truth Tables

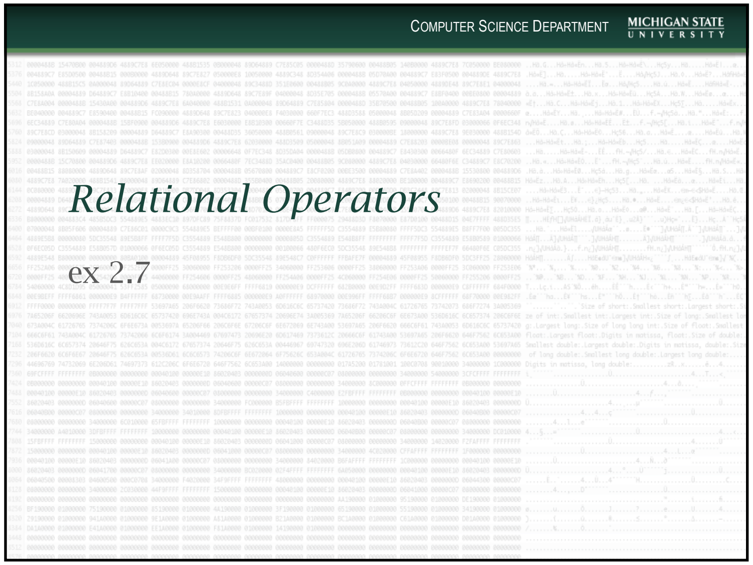
| p | q | !p | p && q | p \|\| q |
|---|---|---|---|---|
| True | True | False | True | True |
| True | False | False | False | True |
| False | True | True | False | True |
| False | False | True | False | False |



## alternative logical ops

Turns out C++ ***does*** support `and`, `or` and `not` as in Python:

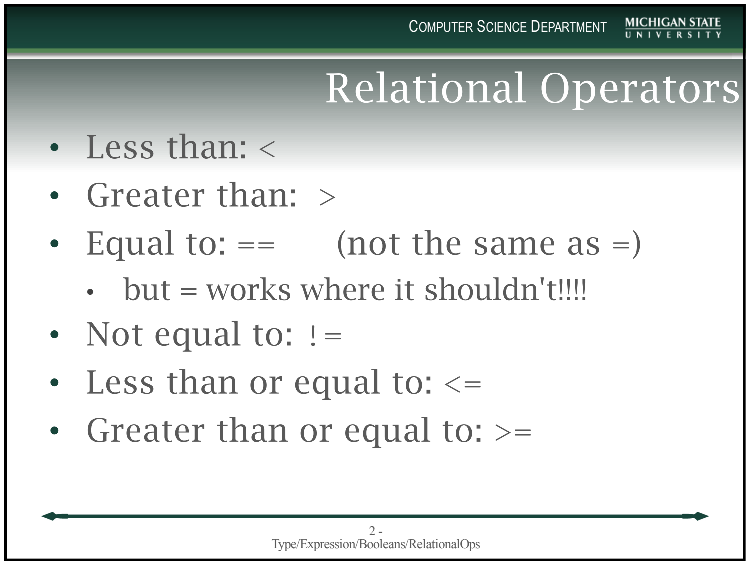
- I honestly didn't know that until I started working with C++11.
- Your book doesn't mention it.
- You are probably not in the C++ club if you do that.



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

# *Relational Operators*

ex 2.7

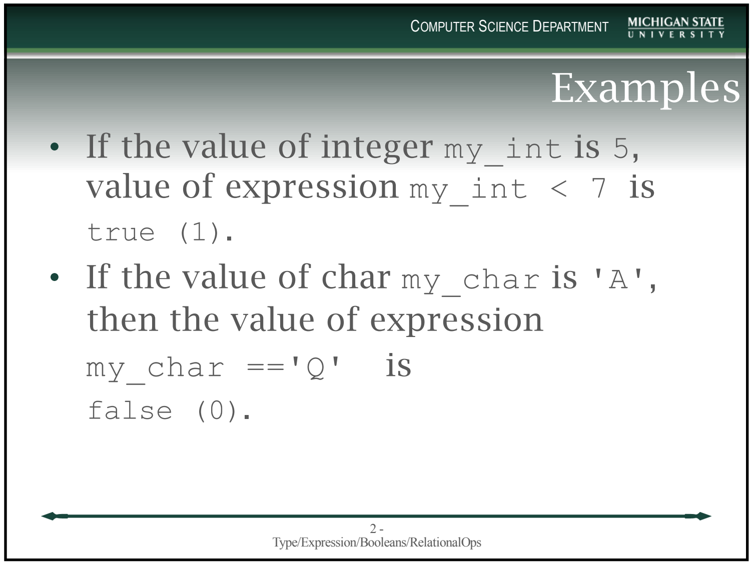


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## Relational Operators

- Less than: <
- Greater than: >
- Equal to: == (not the same as =)
  - but = works where it shouldn't!!!!
- Not equal to: !=
- Less than or equal to: <=
- Greater than or equal to: >=

2 -
Type/Expression/Booleans/RelationalOps

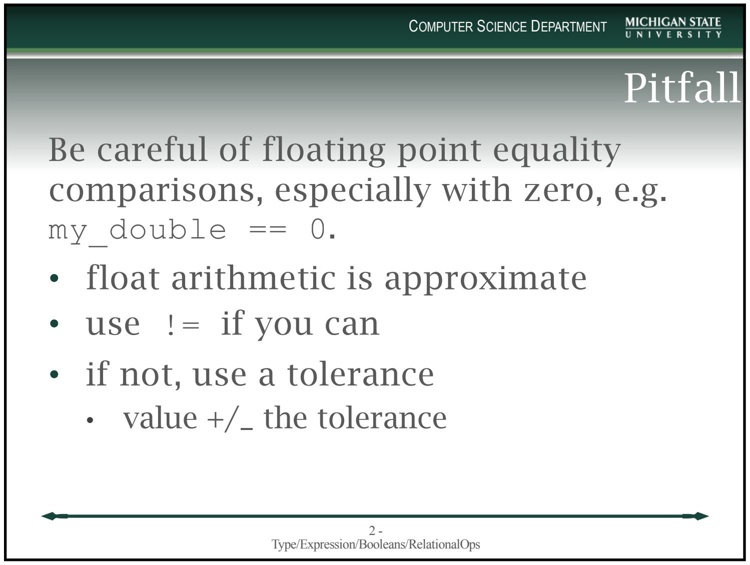


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## Examples

- If the value of integer `my_int` is `5`, value of expression `my_int < 7` is `true (1).`
- If the value of char `my_char` is `'A'`, then the value of expression `my_char =='Q'` is `false (0).`

2 -
Type/Expression/Booleans/RelationalOps



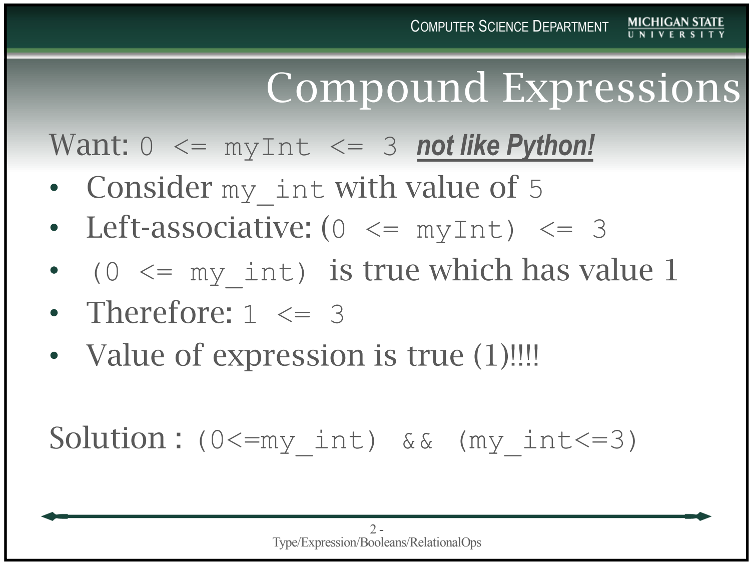
COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## Pitfall

Be careful of floating point equality comparisons, especially with zero, e.g. `my_double == 0.`

- float arithmetic is approximate
- use `!=` if you can
- if not, use a tolerance
  - value +/_ the tolerance

2 -
Type/Expression/Booleans/RelationalOps



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

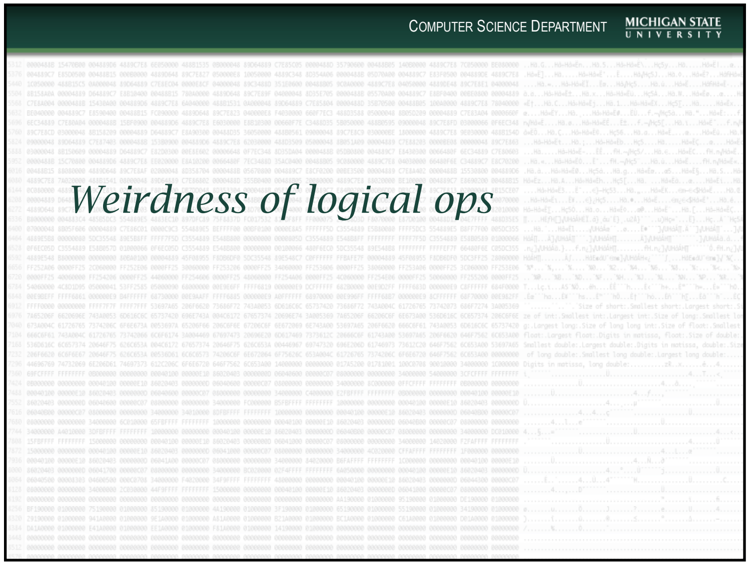
## Compound Expressions

Want: `0 <= myInt <= 3` ***not like Python!***

- Consider `my_int` with value of `5`
- Left-associative: `(0 <= myInt) <= 3`
- `(0 <= my_int)` is true which has value 1
- Therefore: `1 <= 3`
- Value of expression is true (1)!!!!

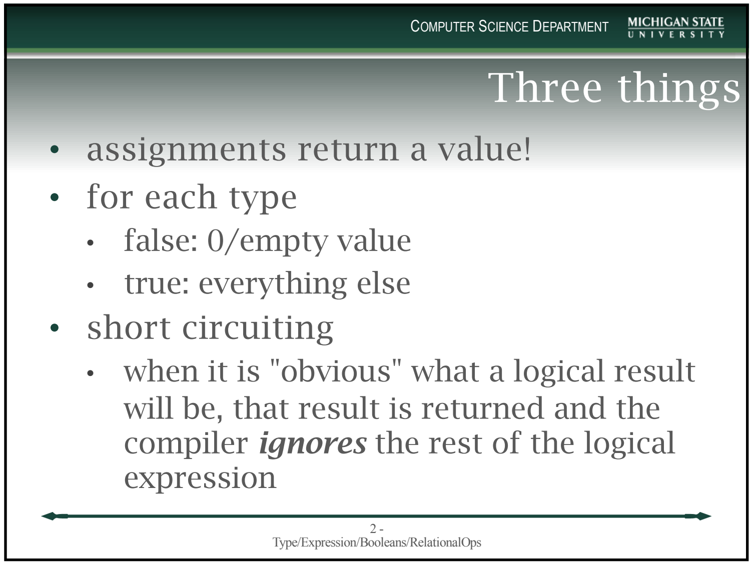
Solution : `(0<=my_int) && (my_int<=3)`

2 -
Type/Expression/Booleans/RelationalOps



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## *Weirdness of logical ops*

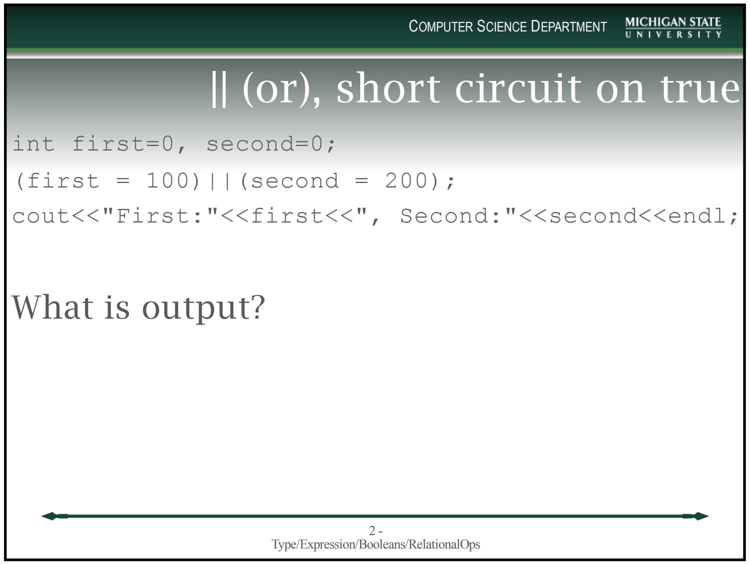


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## Three things

- assignments return a value!
- for each type
  - false: 0/empty value
  - true: everything else
- short circuiting
  - when it is "obvious" what a logical result will be, that result is returned and the compiler ***ignores*** the rest of the logical expression

2 -
Type/Expression/Booleans/RelationalOps



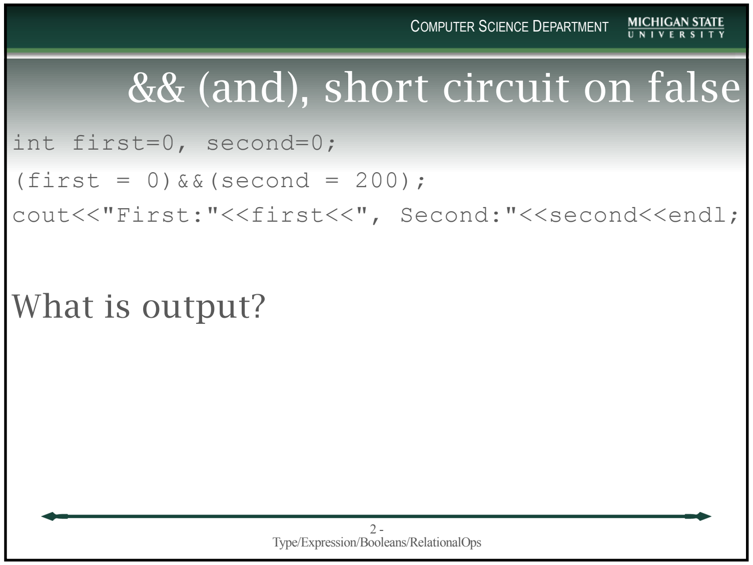
COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## || (or), short circuit on true

```
int first=0, second=0;
(first = 100)||(second = 200);
cout<<"First:"<<first<<", Second:"<<second<<endl;
```

What is output?

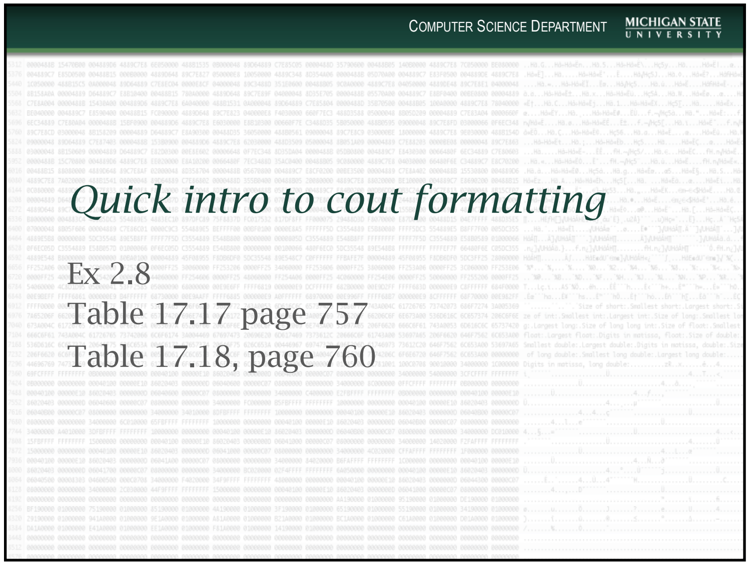
2 -
Type/Expression/Booleans/RelationalOps



## && (and), short circuit on false

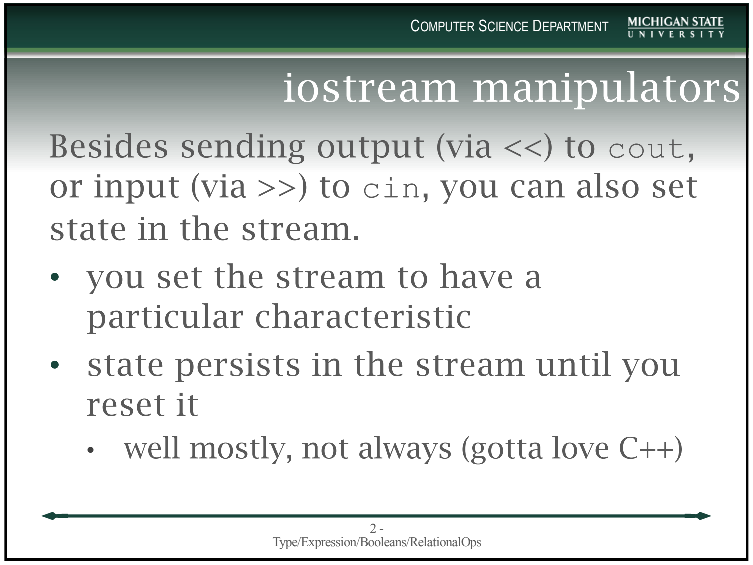
```
int first=0, second=0;
(first = 0)&&(second = 200);
cout<<"First:"<<first<<", Second:"<<second<<endl;
```

What is output?



## *Quick intro to cout formatting*

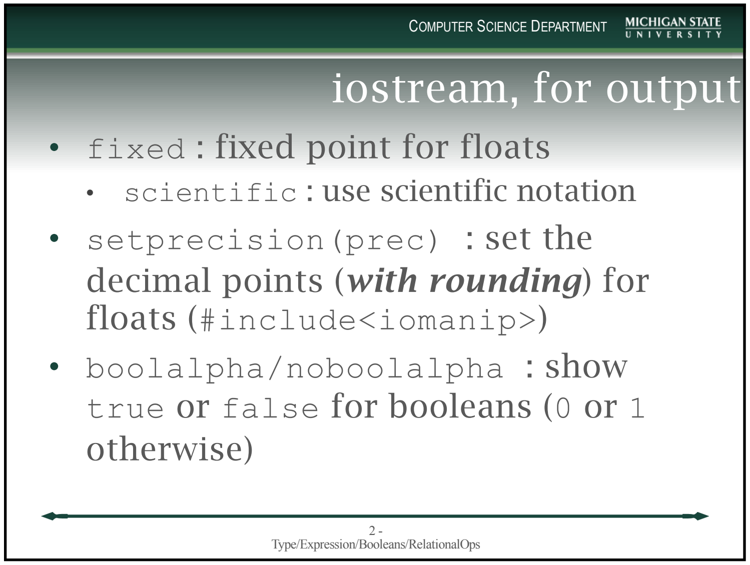
Ex 2.8
Table 17.17 page 757
Table 17.18, page 760



## iostream manipulators

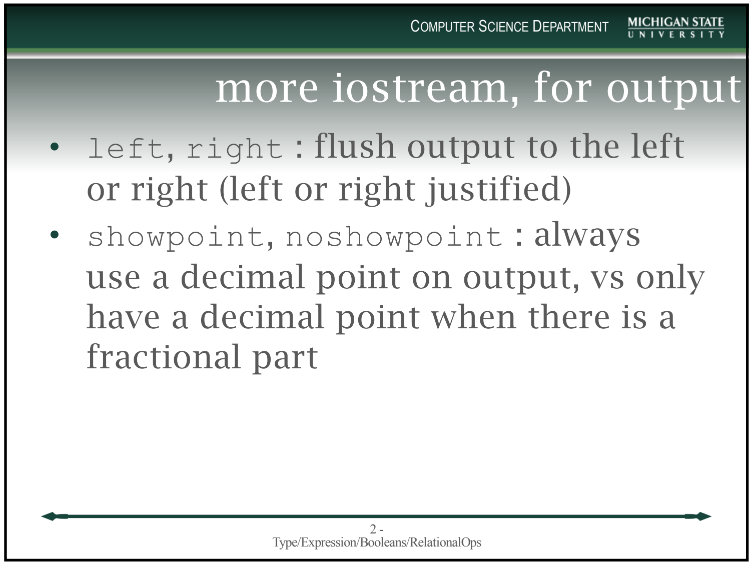
Besides sending output (via <<) to `cout`, or input (via >>) to `cin`, you can also set state in the stream.

- you set the stream to have a particular characteristic
- state persists in the stream until you reset it
  - well mostly, not always (gotta love C++)



## iostream, for output

- `fixed` : fixed point for floats
  - `scientific` : use scientific notation
- `setprecision(prec)` : set the decimal points (***with rounding***) for floats (`#include<iomanip>`)
- `boolalpha/noboolalpha` : show `true` or `false` for booleans (`0` or `1` otherwise)

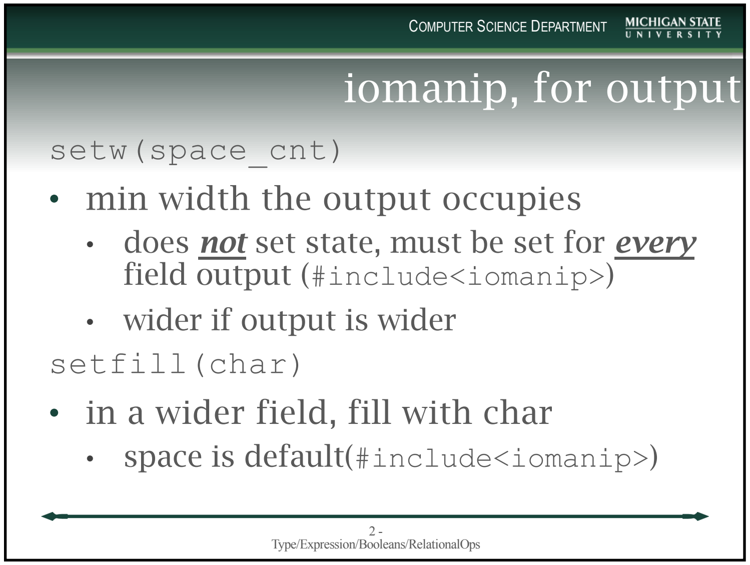


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## more iostream, for output

- `left`, `right` : flush output to the left or right (left or right justified)
- `showpoint`, `noshowpoint` : always use a decimal point on output, vs only have a decimal point when there is a fractional part

2 -
Type/Expression/Booleans/RelationalOps



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## iomanip, for output

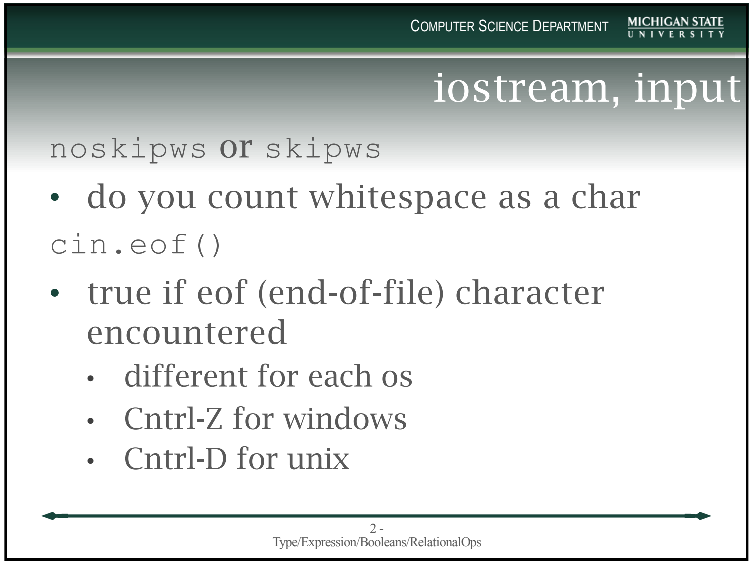
`setw(space_cnt)`

- min width the output occupies
  - does ***<u>not</u>*** set state, must be set for ***<u>every</u>*** field output (`#include<iomanip>`)
  - wider if output is wider

`setfill(char)`

- in a wider field, fill with char
  - space is default(`#include<iomanip>`)

2 -
Type/Expression/Booleans/RelationalOps



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

## iostream, input

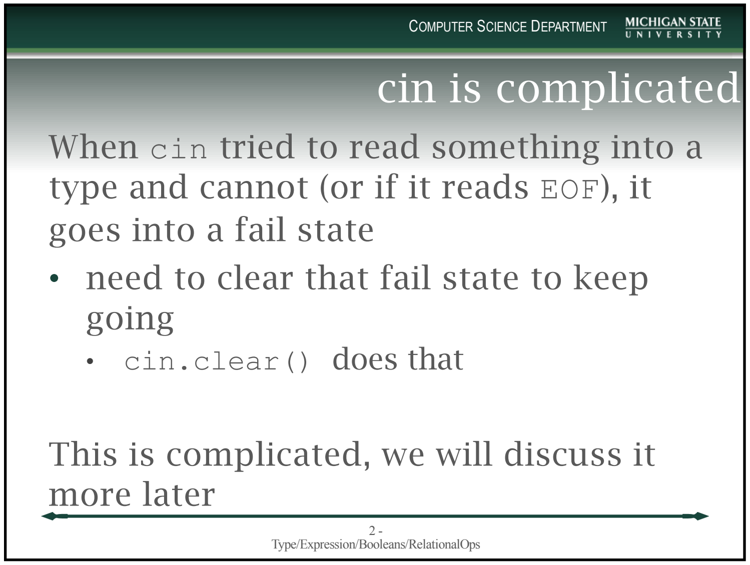
`noskipws` or `skipws`

- do you count whitespace as a char

`cin.eof()`

- true if eof (end-of-file) character encountered
  - different for each os
  - Cntrl-Z for windows
  - Cntrl-D for unix

2 -
Type/Expression/Booleans/RelationalOps



COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

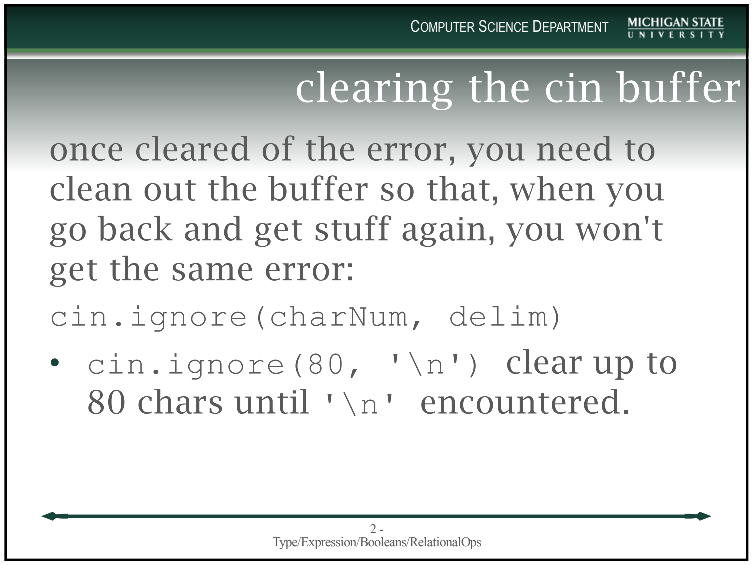
## cin is complicated

When `cin` tried to read something into a type and cannot (or if it reads `EOF`), it goes into a fail state

- need to clear that fail state to keep going
  - `cin.clear()` does that

This is complicated, we will discuss it more later

2 -
Type/Expression/Booleans/RelationalOps

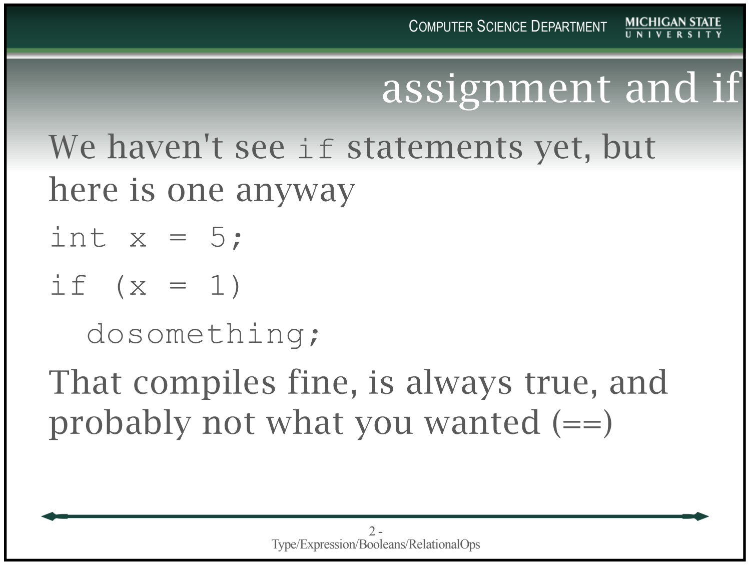


## clearing the cin buffer

once cleared of the error, you need to clean out the buffer so that, when you go back and get stuff again, you won't get the same error:

```
cin.ignore(charNum, delim)
```

- `cin.ignore(80, '\n')` clear up to 80 chars until `'\n'` encountered.



## assignment and if

We haven't see `if` statements yet, but here is one anyway

```
int x = 5;
if (x = 1)
  dosomething;
```

That compiles fine, is always true, and probably not what you wanted (==)