CS16 Week 2 Part 2

Kyle Dewey

Overview

- Type coercion and casting
- More on assignment
- Pre/post increment/decrement
- `scanf`
- Constants
- Math library
- Errors

Type Coercion / Casting

Last time...

- Data is internally represented as a series of bits
- The data type (i.e. `char`, `int`, etc.) determines how many bits are used

Recall

```
unsigned char x = 255;
x = x + 1;
```

```
  11111111
+        1
----------
 100000000
```

Size of Data Type

What about...

- Assume integers (`int`) are 4 bytes (32 bits)

```
unsigned char x = 255;
unsigned int y;
y = x + 1;
// what is the value of y?
```

Data Sizes

- It doesn't matter where it has been
- It only matters where it's going

Binary Operators

- When dealing with variables/expressions of different types, there are different rules for different operators
- Always go for the "bigger" type
  - `double` > `int` > `char`
  - The "bigger" type will be the type of the expression
  - Going for a bigger type is called "type coercion"

Division

```
int x = 5 / 2;
int y = 6 / 2;
int z = 7 / 0;
```

```
double x = 5 / 2;
double y = 6 / 2;
double z = 7 / 0;
```

Division

```
double x = 5 / 2;
double y = 6 / 2;
double z = 7 / 0;
```

```
double x = 5 / 2.0;
double y = 6 / 2.0;
double z = 7 / 0.0;
```

Question

```
double x = 5.5;
x = x + 6 / 4;
// what is the value of x?
```

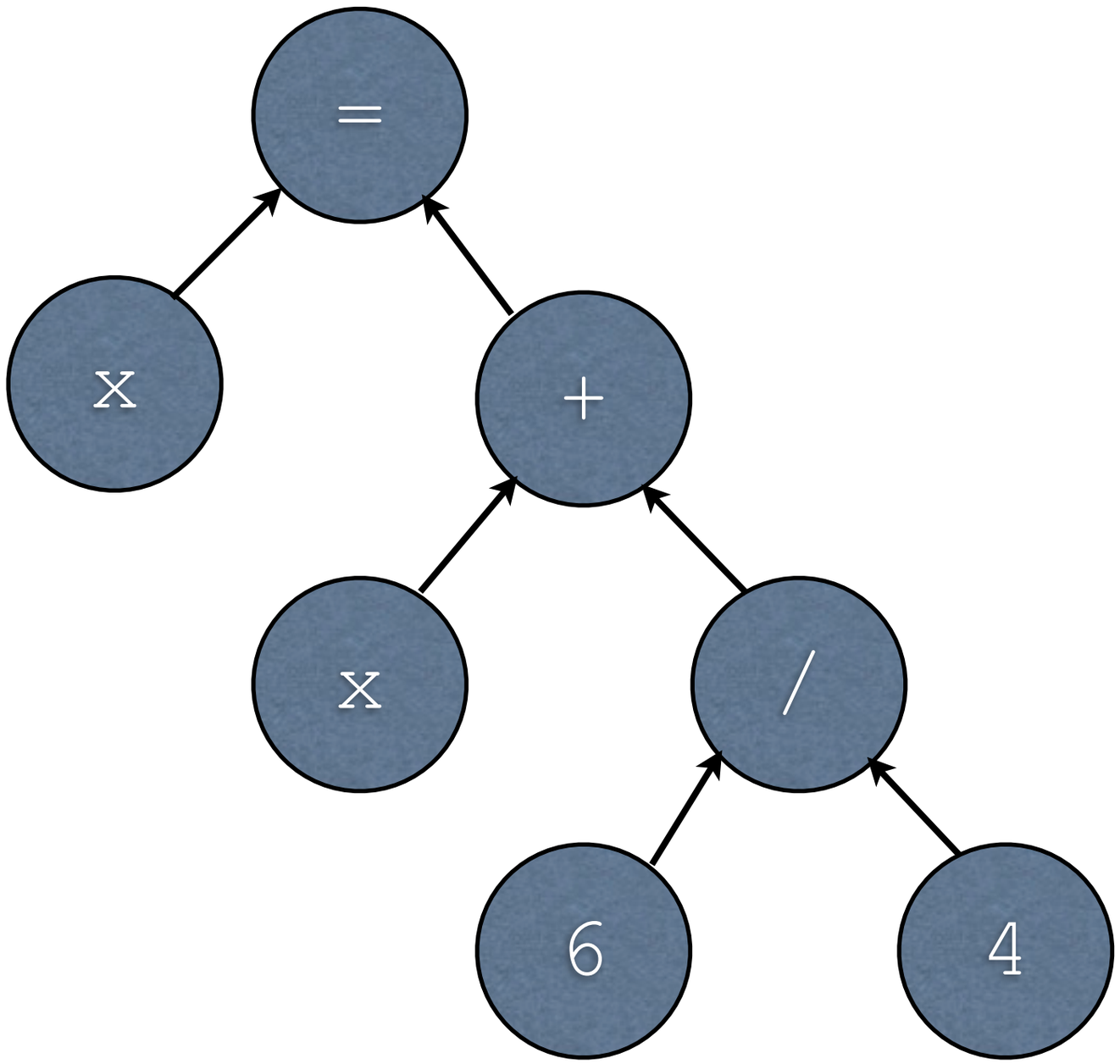
Answer

```
double x = 5.5;


x = x + 6 / 4;
x = x + (6 / 4);
x = (x + ( 6 / 4 );
(x = (x + ( 6 / 4 ));
```

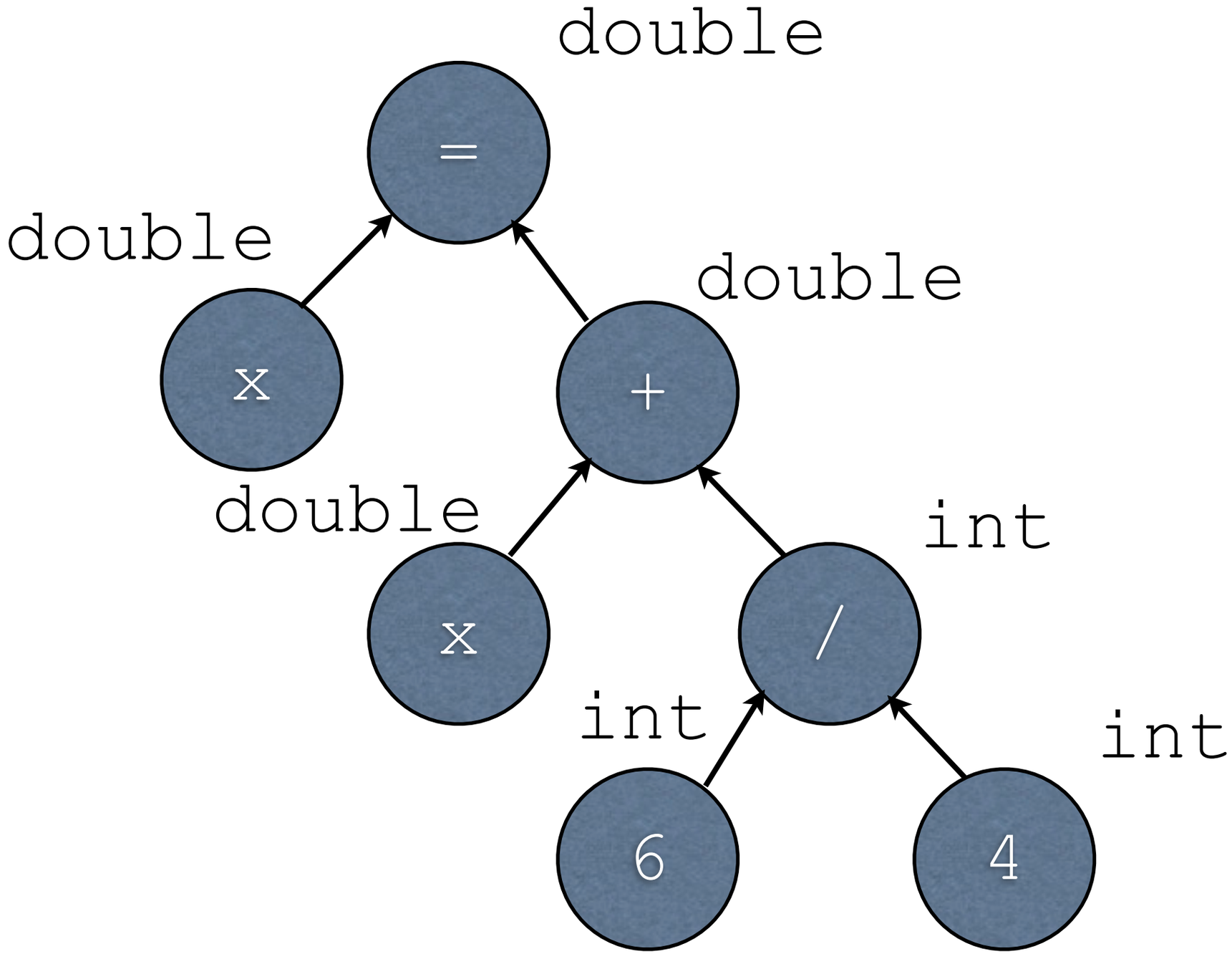
Answer

```
(x = (x + ( 6 / 4 ));
```



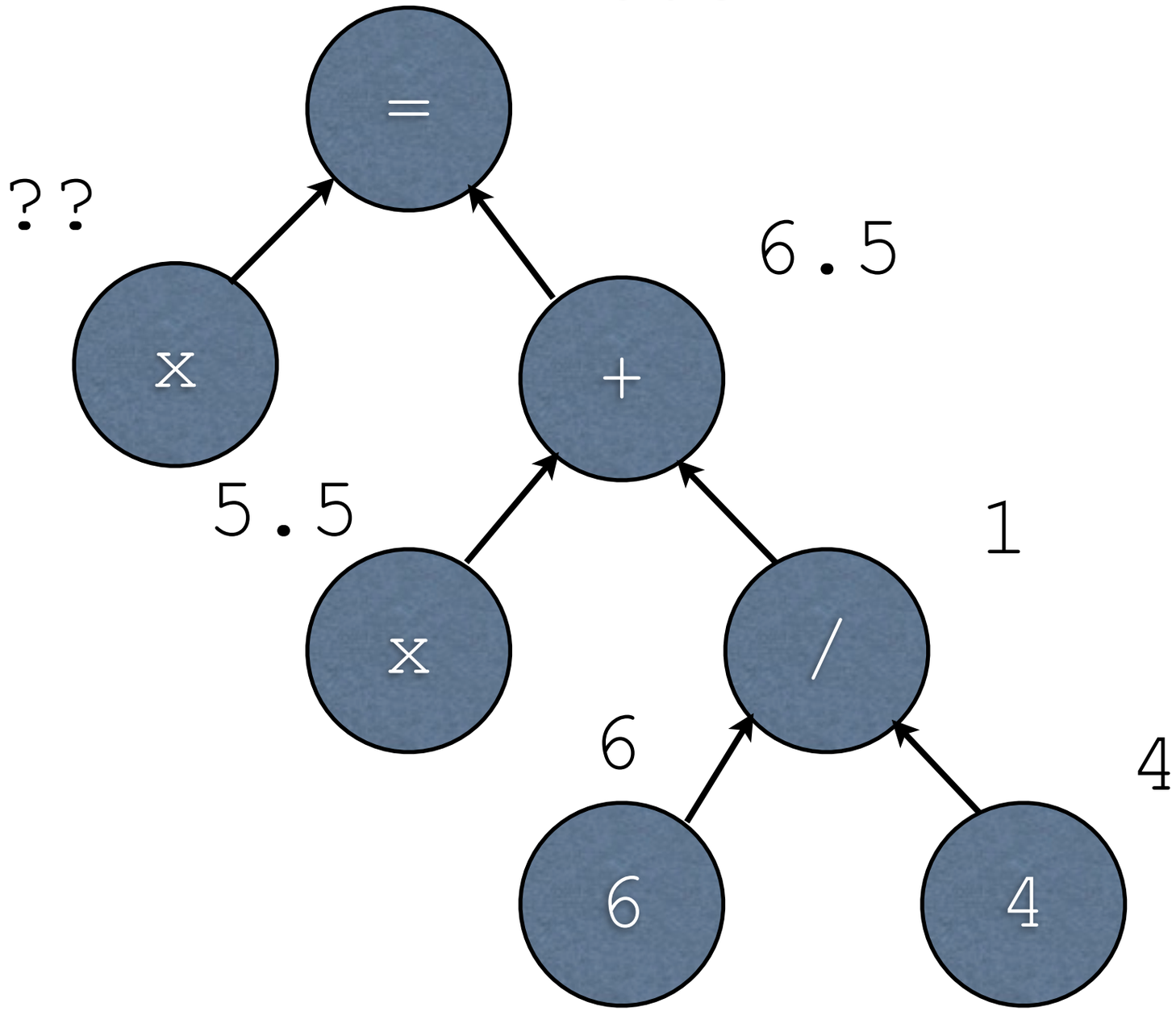
Answer

```
(x = (x + ( 6 / 4 ));
```



Answer

```
(x = (x + ( 6 / 4 ));
```

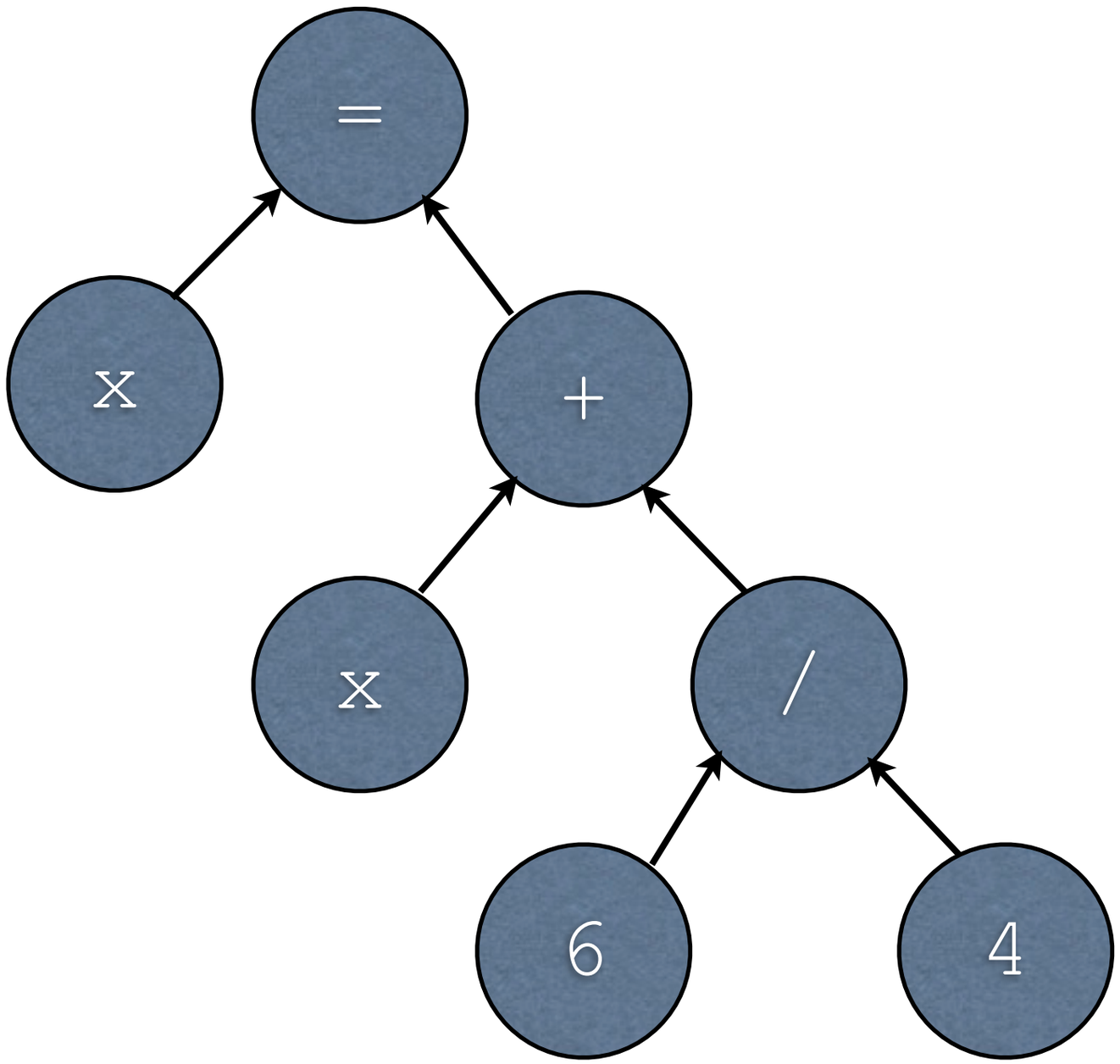


Question

```
double x = 5.5;
x = x + 6 / 4.0;
// what is the value of x?
```

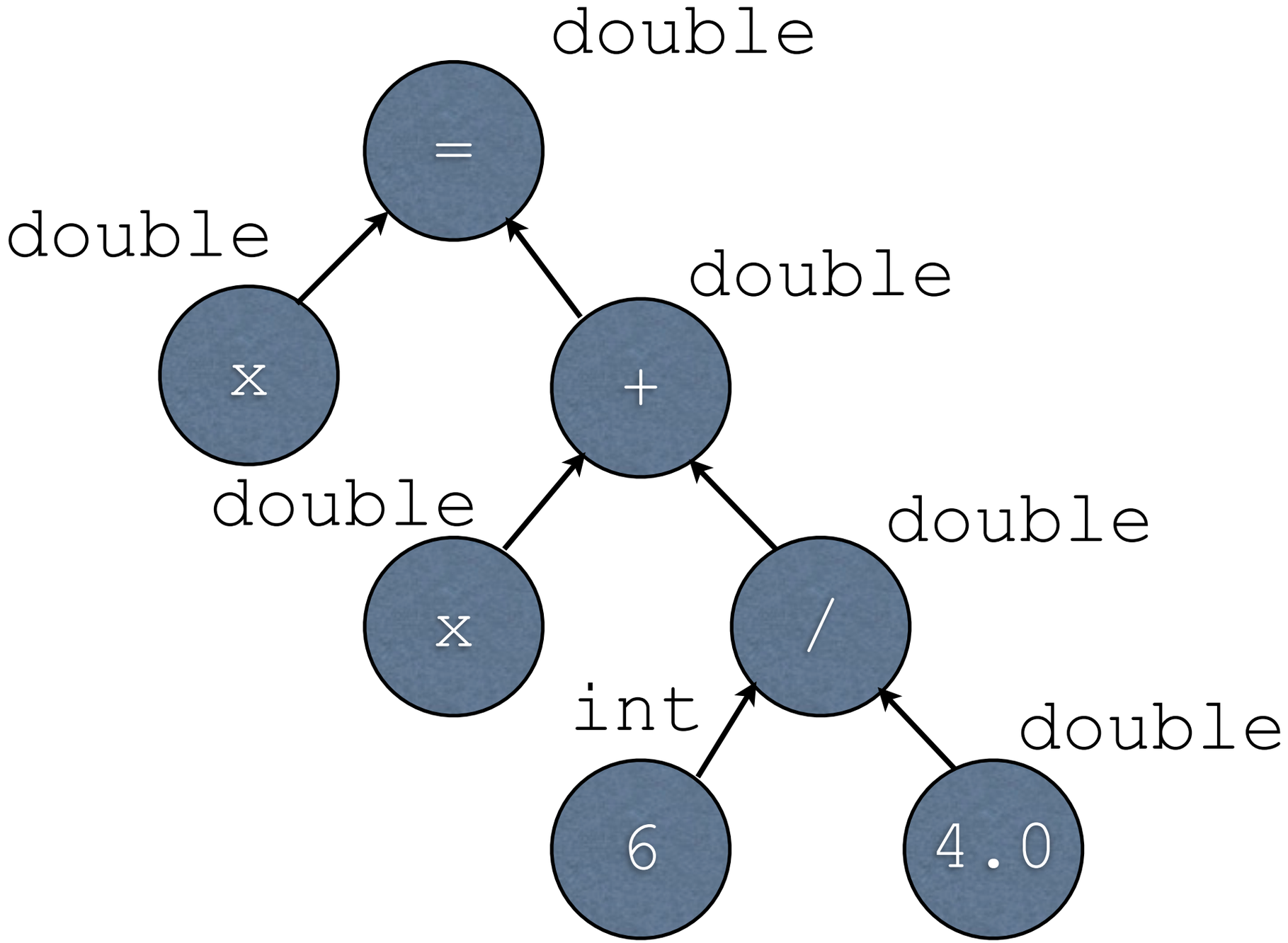
Answer

```
(x = (x + ( 6 / 4.0 ));
```



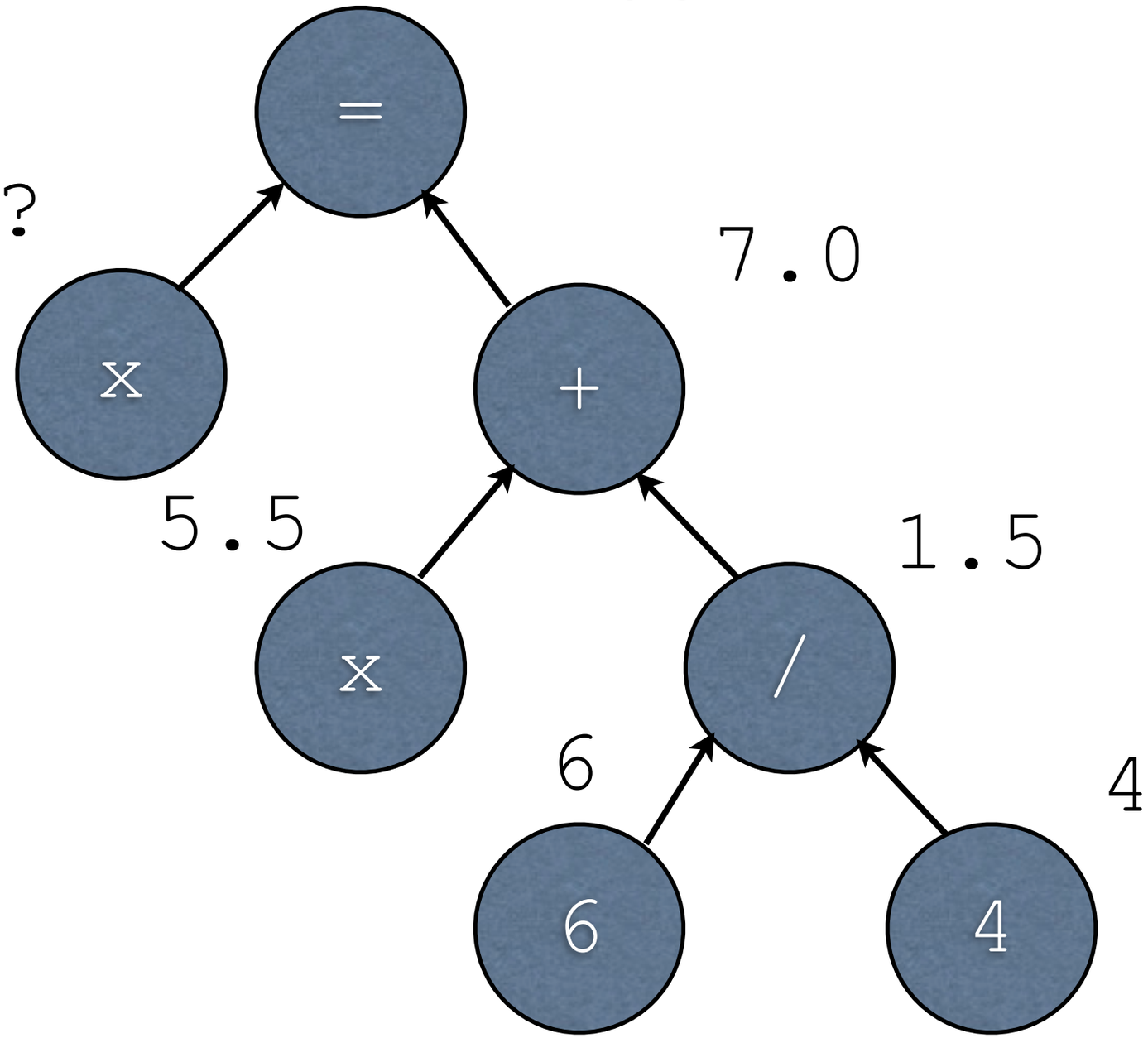
Answer

```
(x = (x + ( 6 / 4.0 ));
```



Answer

```
(x = (x + ( 6 / 4.0 ));
```



Casting

- Sometimes we want to specify the type explicitly
  - Especially when we want to go down the chain (i.e. `double` to `int`)
- This can be done by putting the type itself in parenthesis before the operation

Casting Examples

```
(double)5 // 5.0
(int)5.5 // 5? 6?
(unsigned int)((unsigned char)255)
(unsigned int)((unsigned char)256)
```

More on Expressions

Last time...

- Expressions return values
- Arithmetic consists of nested expressions

```
3 * 2 + 7 - 8 * 2
```

Assignment

```
int x;
x = 6 * 7;
```

Assignment

- Perfectly legal C:

```
int x, y;
x = y = 5 + 1;
// what are the values
// of x and y?
```

Assignment Expression

```
x = y = 5 + 1;
x = y = (5 + 1);
x = (y = (5 + 1));
(x = (y = (5 + 1)));
```

Question

- Is this legal C?

```
int x, y;
x = y + 1 = 3 * 2;
```

Answer

- Is this legal C?

```
int x, y;
x = y + 1 = 3 * 2;
```

- No; the portion `y + 1 = 3 * 2` has no meaning

Question

```
int x, y;
x = 5 + (y = 2 + 1) * 3;
// what are the values
// of x and y?
```

Question

```
int x, y;
x = 5 + y = 2 + 1 * 3;
// what are the values
// of x and y?
```

Answer

```
int x, y;
x = 5 + y = 2 + 1 * 3;
// what are the values
// of x and y?
```

- Trick question!
- This is an ambiguity that needs parenthesis

Precedences

1. ()

2. *, /, %

3. +, -

4. =

Pre/post increment/ decrement

Pre/post inc/dec

- Specifically for variables
- Adding or subtracting one is so common there is are special shorthand operators for it
- Add one: ++
- Subtract one: --

Pre-increment

- ++x
- Semantics: add one to x and return the resulting value

Pre-decrement

- `--x`
- Semantics: subtract one from x and return the resulting value

Post-increment

- x++
- Semantics: return the current value of x and then add one to it

Post-decrement

- `x--`
- Semantics: return the current value of x and then subtract one from it

Example #1

```
int x = 5;
int y = x++ + 1;
// what is x and y?
```

Example #2

```
int x = 5;
int y = ++x + 1;
// what is x and y?
```

Personally...

- Putting these directly in expressions can be **very** confusing
- Shorthand for `x = x + 1`, etc.
- Aside: people can introduce subtle bugs because of undefined behavior

Subtle Bug

```
int x = 0;
int y = x++ + x--;
```

- ch: -1
- gcc: 0

Precedences

1.()

2.++,--

3.*, /, %

4.+, -

5.=

scanf

`scanf`

- The dual to `printf`
- Instead of printing something to the terminal, it reads something from the terminal
- Understands placeholders
- Returns the number of items read in

Reading Something In

- Need a place to put it
- Adds a bit of complexity

Data vs. Data Location

```
unsigned char foo = 0;
foo; // what's the value of foo?
&foo; // where is foo?
```

Key Difference

```
int input = 0;
scanf( “%i”, input );
```

```
...
scanf( “%i”, &input );
```

Simple Examples

```
int input1, input2;
char character;
...
scanf( “%i%i”, &input1, &input2 );
scanf( “%i%c”, &input1, &character );
scanf( “%i %i”, &input1, &input2 );
```

On Whitespace

- `scanf` will ignore whitespace

Format String

- Can have non-placeholders in the format string
- Format string is a pattern of **everything** that must be read in (whitespace treated equally)

```
int input1, input2;
scanf( “%ifoo%i”, &input1, &input2 );
```

scanf.c

Constants

Constants

- Values which never change
- Specific values are constants
  - `55`
  - `27.2`
  - `'a'`
  - `"foobar"`

Constants

- Specifically in the program text
  - Constant in that `52` always holds the same value
    - We cannot redefine `52` to be `53`

Symbolic Constants

- Usually when programmers say "constant", they mean "symbolic constant"
- Values that never change, but referred to using some symbol
  - i.e. π (pi - `3.14...`)
- Mapping between symbol and value is explicitly defined somewhere

In C

- Use `#define`
- By convention, constants should be entirely in caps

```
#define PI 3.14
...
int x = PI * 5;
```

Mutability

- Constants are, well, constant!
- Cannot be changed while code runs

```
#define PI 3.14
...
PI = 4; // not valid C!
```

What `#define` Does

- Defines a **text substitution** for the provided symbol
- This text is replaced during compilation by the **C preprocessor** (cpp)

Example #1

Code

```
#define PI 3.14
...
int x = PI * 5;
```

After Preprocessor

```
int x = 3.14 * 5;
```

Example #2

Code

```
#define PI 3.14
...
PI = 4;
```

After Preprocessor

```
3.14 = 4;
```

Best Practices

- Generally, all constants should be made symbolic
  - Easier to change if needed later on
  - Gives more semantic meaning (i.e. `PI` is more informative than `3.14...`)
  - Possibly less typing

Errors

Errors

- Generally, expected result does not match actual result
- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

Errors

- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

Syntax Error

- A "sentence" was formed that does not exist in the language
- For example, "Be an awesome program" isn't valid C

Syntax Error

- Easiest to correct
- Compiler will not allow it
- *Usually* it will say where it is exactly

On Syntax Errors

- ...sometimes the compiler is really bad at figuring out where the error is

```
#include <stdio.h>

int main() {
  printf( "moo" )
  printf( "cow" );
  return 0;
}
```

Reality

```
#include <stdio.h>

int main() {
   printf( "moo" )
   printf( "cow" );
   return 0;
}
```

- Missing semicolon at line 4

GCC

```
#include <stdio.h>

int main() {
  printf( "moo" )
  printf( "cow" );
  return 0;
}
```

```
syntax.c: In function ‘main’:
syntax.c:5: error: expected ‘;’ before
‘printf’
```

Ch

```
#include <stdio.h>

int main() {
   printf( "moo" )
   printf( "cow" );
     return 0;
  }
```

```
ERROR: multiple operands together
ERROR: syntax error before or at line 5
in file syntax.c
  ==>:   printf( "cow" );
  BUG:   printf( "cow" )<== ???
```

The Point

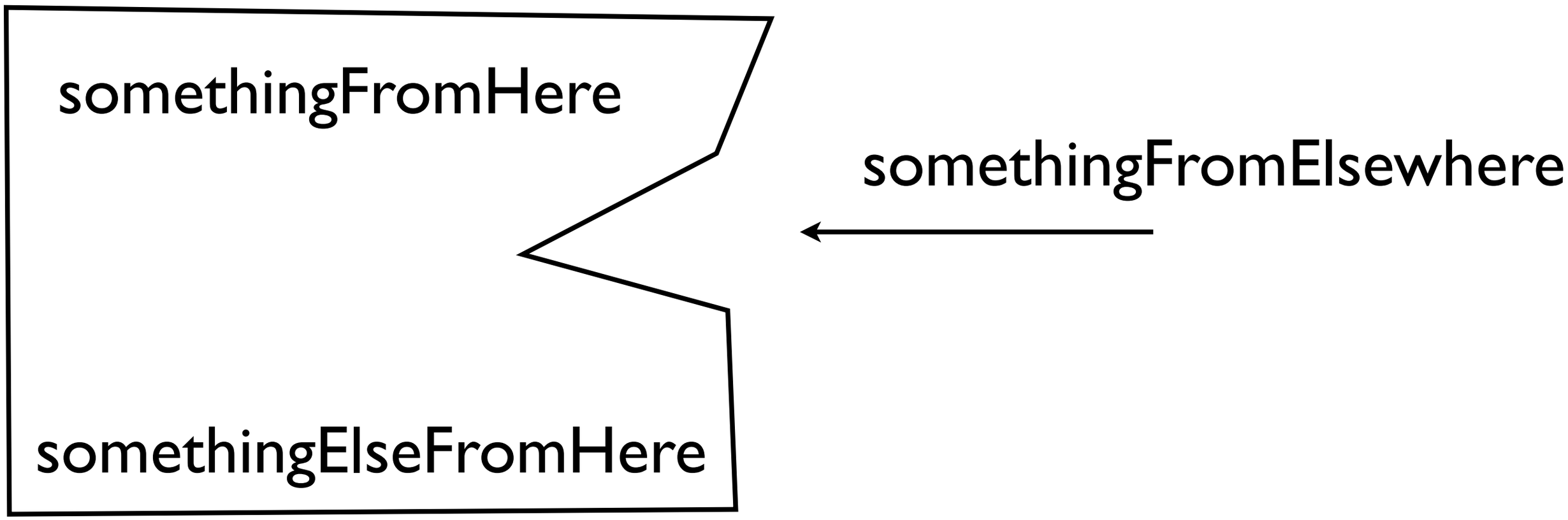
- Compilers are just other programs
- Programs can be wrong
- Programs are not as smart as people

Errors

- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

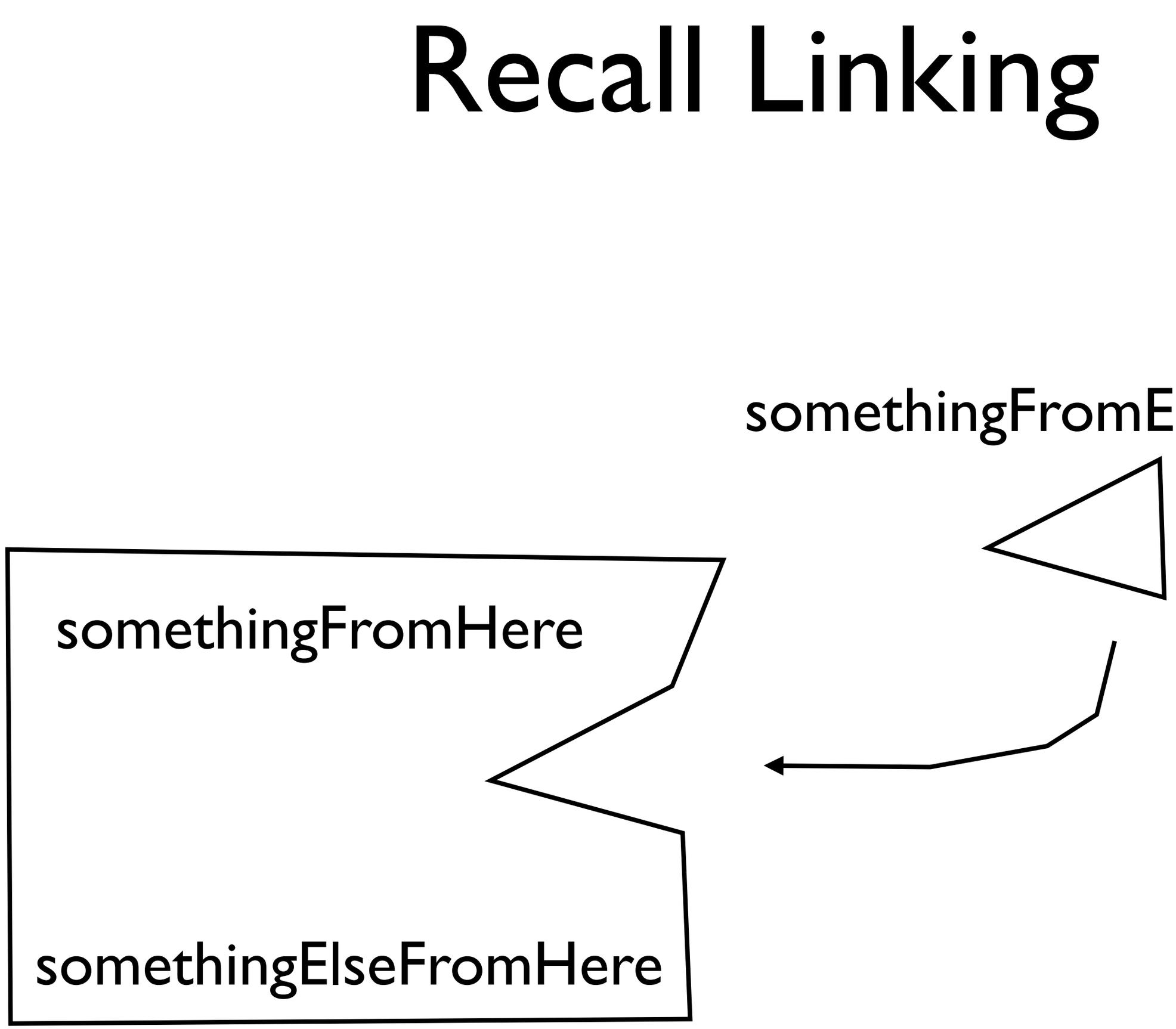
Recall Linking

```
1: somethingFromHere();
2: somethingFromElsewhere();
3: somethingElseFromHere();
```



Recall Linking

somethingFromElsewhere



Linker Errors

- What if `somethingFromElsewhere` is nowhere to be found?
- Missing a piece
- Cannot make the executable

Example

```
int something();

int main() {
    something();
    return 0;
}
```

- `int something();` **tells the compiler that** `something` **exists somewhere, but it does not actually give** `something`

Example

```
int something();

int main() {
   something();
   return 0;
}
```

```
Undefined symbols for architecture
x86_64:
  "_something", referenced from:
      _main in ccM6c8aW.o
ld: symbol(s) not found for
architecture x86_64
```

Errors

- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

Runtime Errors

- Error that occurs while the code is running
- Compilation and linking must have succeeded to get to this point

Examples

- Overflow

```
unsigned char x = 255;
x = x + 1;
```

- Underflow

```
unsigned char x =0;
x = x - 1;
```

Examples

- Divide by zero (especially for integers!)

```
unsigned int x = 5 / 0;
```

- **Wrong** `printf` **placeholder**

```
printf( “%s”, 57 );
```

Errors

- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

Logic Errors

- It works!
- ...but it doesn't do what you wanted
- Like getting the wrong order at a restaurant

Examples

- Transcribed an equation incorrectly
- Using the wrong variable
- Lack of understanding of problem
- etc. etc. etc...

Logic Errors

- By far, the most difficult to debug
- It might be done **almost** correctly
- This is why testing is so important!