Expressions

An expression gets evaluated and yields some value

2 + 2 *evaluates to* 4

Simple Expressions

Literal values:

- how values of a specific type are represented in C++

Type	Examples		
bool literal	true	false	
int literal	3	0	-10
double literal	30.0	3e1	30.
char literal	‘a’	‘0’	‘\n’
string literal	“this is a string literal”		

Evaluating literal values as expressions:

```
10   // evaluates to 10, represented as an int
10.0 // evaluates to 10, represented as a double
“10” // evaluates to 10, represented as a string
```

Simple Expressions

Variables:

- variables get evaluated as whatever value they store
- the type of the resulting value is always the same as the type of the variable
- to use a variable as an expression, just use its name

Evaluating variables as expressions:

```
int var1 = 10;

var1; // expression evaluates to 10 as an integer

double var2 = 1e2;

var1; // expression evaluates to 100 as an integer

char var3;

var3; // expression evaluates to some char (garbage value)
```

Simple Expressions

Function Calls:

- a function call evaluates as the result, or *return value*, of the function
- functions are called by using the name of a function, followed by a set of parentheses containing a comma-separated list of inputs to be sent to that function
- the type of the resulting value depends on the *return type* of the function

Evaluating function calls as expressions:

```
// calls the acos function with -1 as input
// the return type of acos is double
acos(-1)  // evaluates to 3.14159 as a double
// calls the pow function with 2 and 5 as inputs
// the return type of pow is double
pow(2, 5) // evaluates to 32 as a double
```

Complex Expressions

Complex Expressions:

- one or more simple expressions, joined by at least one operator

Examples:

```
// two literal integers joined by the addition operator

2 + 2

// a single bool literal along with the ‘NOT’ operator (!)

!true

// literals, variables, and function calls joined by

4.0 / 3 * acos(-1) * pow(radius, 3)

// cout (a variable) and literal strings, joined by the output operator

cout << “a string” << “ and another!”
```

Complex Expressions

Complex Expressions:

- one or more simple expressions, joined by at least one operator

Evaluating complex expressions:

- complex expressions get evaluated *one operator at a time*
- the order of evaluation depends on the *precedence* and *associativity* of the operators
- the result of evaluating a complex expression is a *single simple expression*:

Remember:

- *precedence* is the <u>order</u> in which operators are evaluated (order of operations)
- *associativity* determines the <u>direction</u> of evaluation (*left-to-right* or *right-to-left*)

Complex Expressions

Example of evaluating a complex expression:

```
// initial complex expression (assume radius = 3)

volume = 4.0 / 3 * acos(-1) * pow(radius, 3)
```

How to evaluate it:

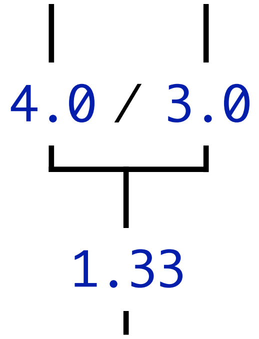
- the assignment operator has lowest precedence, while all the others (division and multiplication) have equal precedence. Evaluate them first.

Complex Expressions

Example of evaluating a complex expression:

```
// initial complex expression (assume radius = 3)

volume = 4.0 / 3 * acos(-1) * pow(radius, 3)
```



4.0 / 3.0

1.33

How to evaluate it:

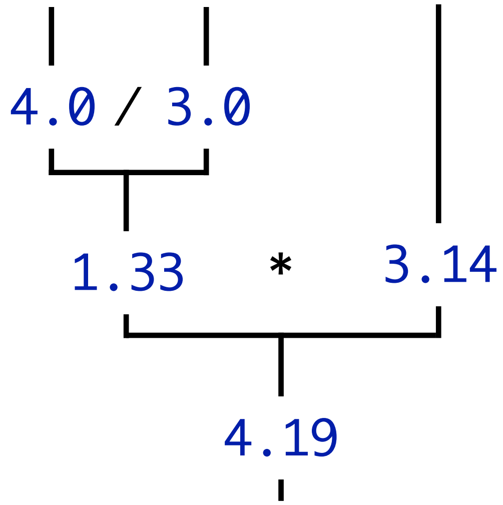
- binary arithmetic operators are left-to-right associative, so start with `4.0 / 3`

Complex Expressions

Example of evaluating a complex expression:

```
// initial complex expression (assume radius = 3)

volume = 4.0 / 3 * acos(-1) * pow(radius, 3)
```



4.0 / 3.0

1.33 * 3.14

4.19

How to evaluate it:

- binary arithmetic operators are left-to-right associative, so next comes the result of the previous step `(1.33) * acos(-1)`

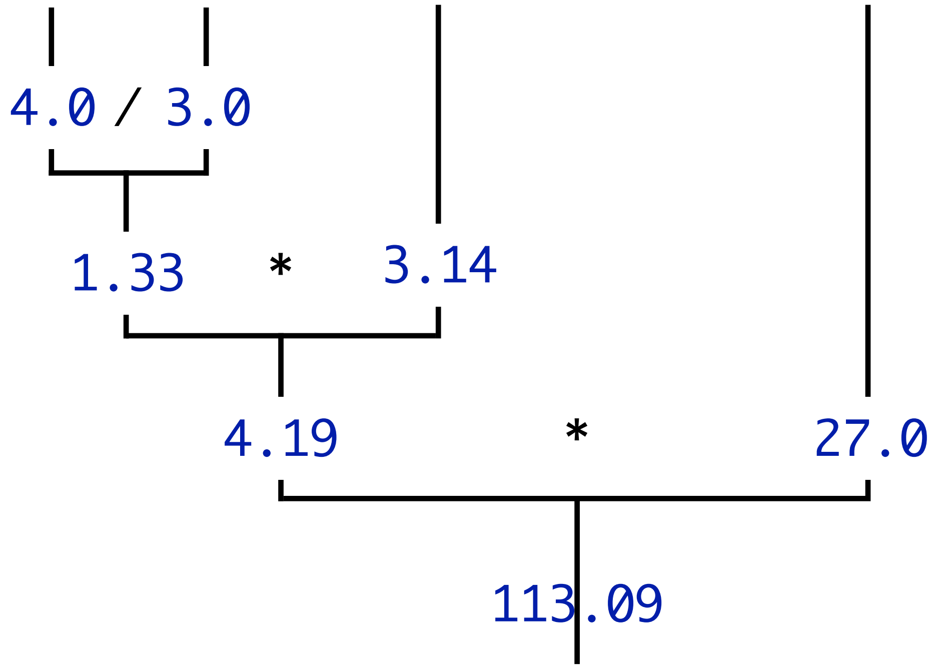
- `acos(-1)` is a function call that evaluates to `3.14159` (pi)

Complex Expressions

Example of evaluating a complex expression:

```
// initial complex expression (assume radius = 3)

volume = 4.0 / 3 * acos(-1) * pow(radius, 3)
```



How to evaluate it:

- binary arithmetic operators are left-to-right associative, so next comes the result of the previous step (4.19) * pow(radius, 3)

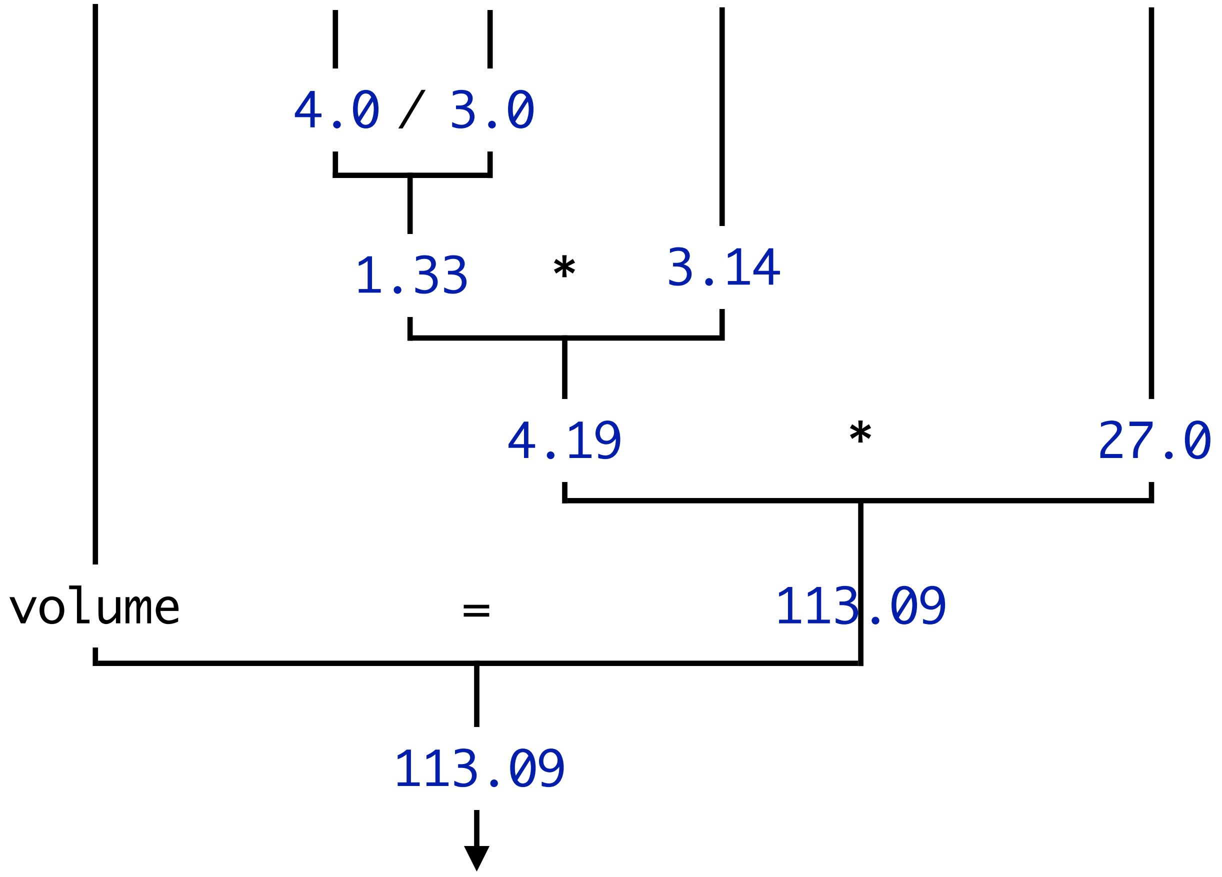
- pow(radius, 3) is a function call that evaluates to 27.0

Complex Expressions

Example of evaluating a complex expression:

```
// initial complex expression (assume radius = 3)

volume = 4.0 / 3 * acos(-1) * pow(radius, 3)
```



4.0 / 3.0

1.33 * 3.14

4.19 * 27.0

volume = 113.09

113.09

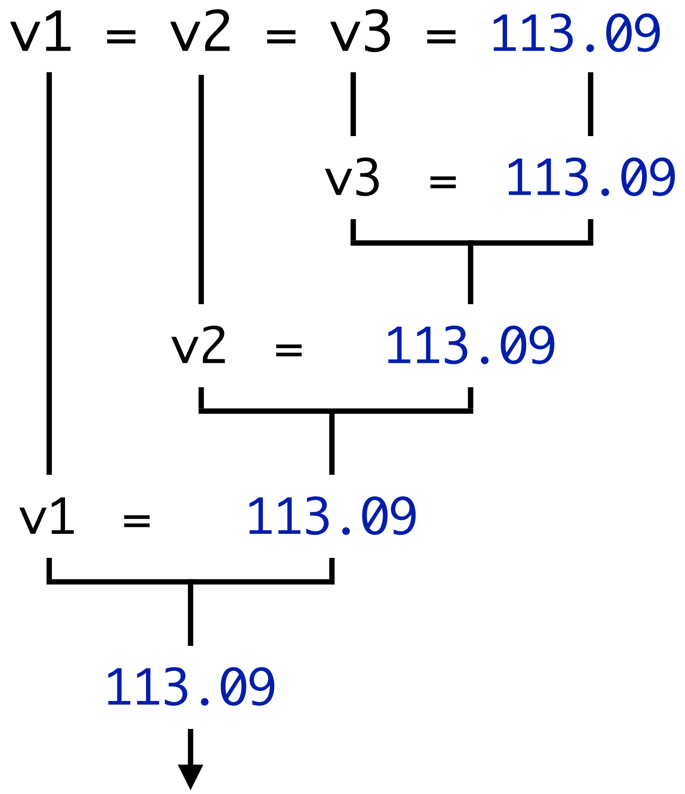
How to evaluate it:

- the assignment operator (the lone remaining operator) now has the highest precedence
- change the value of the `volume` variable... then the assignment expression evaluates to `113.09`. Assignment operators evaluate to the value that was assigned!

Complex Expressions

Example of evaluating a complex expression:

```
// initial complex expression (assignment chaining)
```



```
v1 = v2 = v3 = 113.09

         v3  =  113.09

   v2  =   113.09

v1  =   113.09

     113.09
```

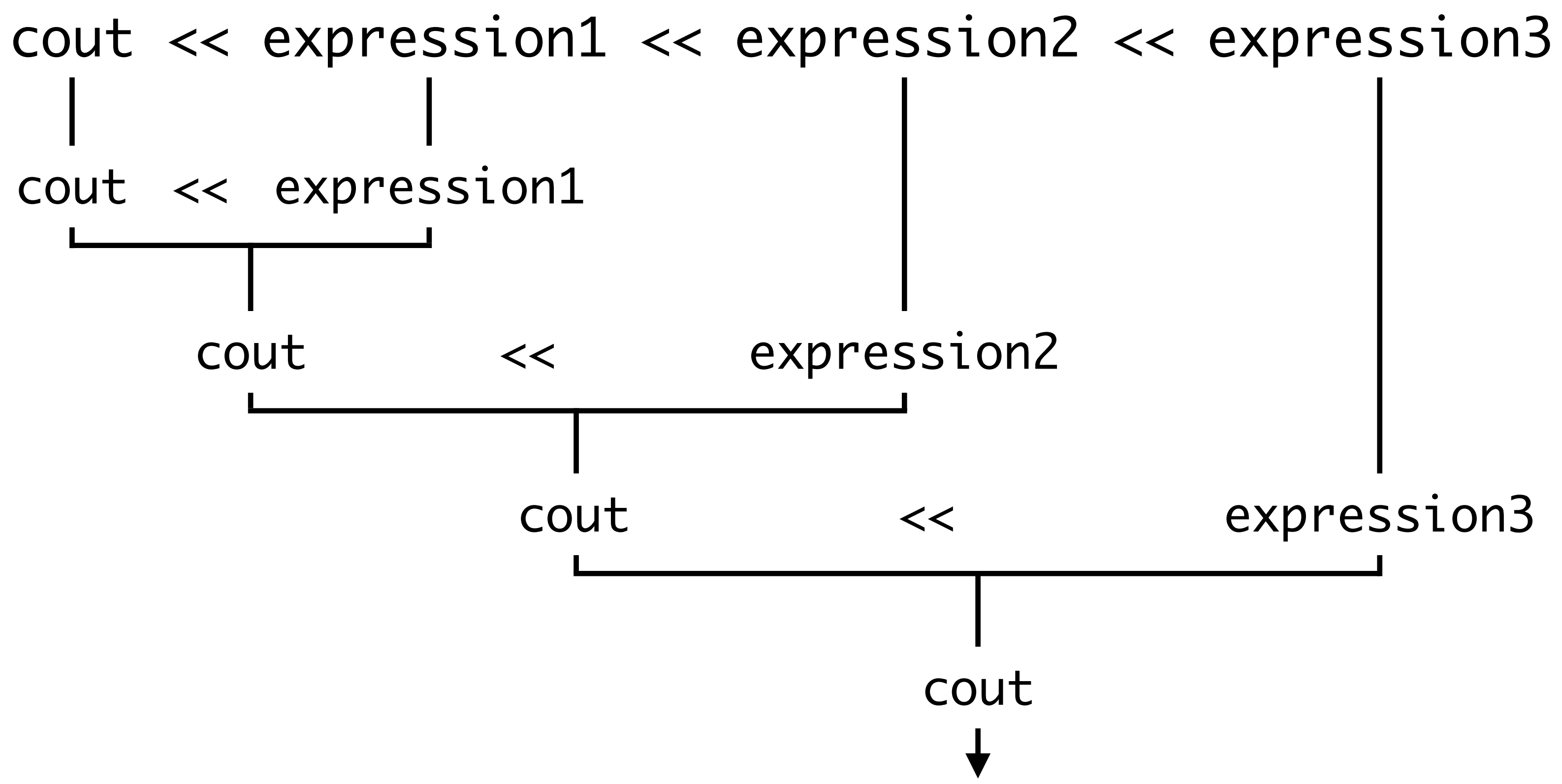
Assignment operators evaluate to the value being assigned

- this is why we can chain assignment operations!
- assignment is a *side-effect* of this operator (something changes as a result)

Complex Expressions

Example of evaluating a complex expression:

```
// initial complex expression (output chaining)
```



```
cout << expression1 << expression2 << expression3

cout  <<  expression1

     cout        <<        expression2

                cout          <<          expression3

                               cout
```

I/O operators evaluate to the stream object on the left

- this is why we can chain input and output statements!
- I/O is a *side-effect* of this operator (something gets output or input as a result)

No-Side-Effect Statements

What do I mean by “side effects”?

- did a variable change?
- was something output to the console?
- did something *happen*???

Example of no-side-effect statements (nothing happens):

```
3 + 2;                // C++ does the calculation, then discards the results
“This does nothing”;  // C++ creates the string, then discards it
sqrt(4);              // C++ calculates the result as 2, then discards it
;;;;;;;;              // 8 “empty” statements (just semicolons!)
```

Valid C++!

- though you may see this from the compiler: `warning: statement has no effect`

C++ becomes easier to understand if you understand the concept of expressions.