Combined Assignment

- the most common statement in programming is assignment:
  `foo = bar;`
- the second most common statement pattern is an arithmetic operation on a variable followed by assignment to that variable:
  `foo = foo + 3;`
- this pattern is very common in C++
- it involves the name of the variable typed twice
- C++ has a shortcut form that combines arithmetic and assignment in one symbol

```
foo = foo + 3;
foo += 3;
```

Combined Assignment

`foo += 3;`	retrieve the current value of foo, add three to it, and store the result in foo
`foo *= 3;`	retrieve the current value of foo, multiply it by three, and store the result in foo
`foo /= bar - 5;`	retrieve the current value of bar, subtract 5 from that value, use the result as the divisor of the current value of foo, and store the final result in foo; bar is not changed

Operator Precedence and Associativity

- when multiple operators exist in a single statement
- the order in which they are evaluated depends on a combination of precedence and associativity
- for the operators we have seen, the precedence and associativity are as follows

Operator	Precedence	Associativity
− (unary)	1	left-to-right
* / %	2	left-to-right
+ − (binary)	3	left-to-right
= *= /= += −= %=	4	right-to-left

Multiple Operators

- when multiple operators of different precedence exist in a single statement
- precedence determines order

```
foo = bar * bim - bam;
```

- in order of precedence, the operators are executed:
  1. * (highest precedence)
  2. − (medium precedence)
  3. = (lowest precedence)

Multiple Operators

- when multiple operators of the same precedence exist in a single statement
- associativity determines order

```
foo / bar * bim % bam
```

- an expression, not a complete statement
- these operators have equal precedence and left-to-right associativity
- they are executed:
  1. / (leftmost)
  2. * (middle)
  3. % (rightmost)

Multiple Operators

- when multiple operators of the same precedence exist in a single statement
- associativity determines order

```
foo = bar = bim = 5;
```

- these operators have equal precedence and right-to-left associativity
- they are executed:
  1. bim = 5 (rightmost)
  2. bar = result of 1 (middle)
  3. foo = result of 2 (leftmost)
- foo, bar, and bim all get the value 5

Algebra vs. Programming

- human algebra uses some syntax and shortcuts that are not available in C++

Algebraic Expression	**C++ Equivalent**
$6b$	`6 * b`
$(3)(12)$	`3 * 12`
$x = \frac{a+b}{c}$	`x = (a + b) / c;`
$y = 3\frac{x}{2}$	`y = 3 * (x / 2);`

Exponentiation

- C++ does not have an exponentiation operator
- the pow function is provided by the cmath library
- regardless of arguments, pow returns a double value

```
#include <cmath>
        ...
 foo = pow(bar, 2.5);
```

- for simple exponentiation like squaring, it's faster and easier to simply multiply the variable by itself
  `foo = bar * bar;`

Typecasting and Formatting

Skipped Content

- we will not explicitly cover:
  - the material from page 122 to the middle of page 126: cin.get, cin.ignore, string functions
  - the material from the bottom of page 126 to the bottom of 128: additional math library functions (but look at the table on page 127 to see that the functions exist)
  - sections 3.10 and 3.11 — please read over them, but we won't cover them in a lecture and they will not specifically be on the test

Online dating advices:
Hang on tight, it can be a tough ride!

Numerical Types

- remember there are three fundamentally different families of numerical data types
- they have very different purposes

Family	**Purpose**
unsigned integers	counted quantities
signed integers	whole numbers that might need to be negative
floating point	measured or calculated quantities that might have fractional parts

Keep Data Types Separate

- the arithmetic operators are defined for identical data types
  - `unsigned = unsigned + unsigned;`
  - `double = double + double;`
- to the greatest extent possible, you should avoid mixing data types in expressions

- however, sometimes you must mix data types in a single expression
- the compiler has a set of rules to try to convert one into the other
- the purpose of the rules is to avoid information loss

Type Ranking

- C++ ranks types by the largest value each can hold
  1. long double
  2. double
  3. float
  4. unsigned long
  5. long
  6. unsigned
  7. int
  8. unsigned short
  9. short

Terminology

coercion: convert a value of one type to a different type (floating $\leftrightarrow$ integral or signed $\leftrightarrow$ unsigned)

promotion: convert a value to a higher-ranked type

demotion: convert a value to a lower-ranked type

Mixing Sizes

- remember some of the integer sizes

Name	# Bytes (ice)	Range of Values
short	2 bytes	$-32,768 \ldots 32,767$
unsigned short	2 bytes	$0 \ldots 65,535$
int	4 bytes	$-2,147,483,648 \ldots 2,147,483,647$
unsigned int	4 bytes	$0 \ldots 4,294,967,296$

- a signed short's value can always fit into an int location
- an unsigned short's value can always fit into an int location
- a signed short's value might not fit into an unsigned int location
- an int value might not fit into an unsigned int location
- an unsigned int value might not fit into an int location

Mixing Sizes

- the compiler will not allow an attempt to convert to a type that might not be able to hold the value

```
int foo = 10;
unsigned int bar = foo;
warning: implicit conversion changes
   signedness: 'int' to 'unsigned int'

int foo = 10;
short bar = foo;
warning: implicit conversion loses
   integer precision: 'int' to 'short'

float foo = 10.0;
int bar = foo;
warning: implicit conversion turns
   floating-point number into integer: 'float' to 'int'
```

Mixed Types

- there are several automatic conversions that it is ok to use
- the compiler does the conversions for you
- this differs somewhat from what your textbook says
- the clang-llvm compiler is much more strict than older, classic compilers

- the following pairs of mixed types are "safe"
- but you still need a good reason to mix them

Types	Result Type
two signed integer types	the larger type
two unsigned integer types	the larger type
an integer type and a floating type	the floating type

Concise

- there is a fine line between being concise and being sloppy
- being concise involves
  - keep it short
  - don't use more words if fewer words will suffice
  - don't use a longer expression if a shorter one gets the same results

- however, sometimes being short is not concise, it's sloppy: `double weight_of_material = 0;`
- weight_of_material is declared as a double because it will involve a measured quantity
- a double has a whole part and a fractional part
- the correct initialization is: `double weight_of_material = 0.0;`
- this is a signal that you the programmer are consciously choosing the correct data type

Type Casting

- sometimes you need to mix types that are "unsafe"
- sometimes you need to explicitly convert types
  1. you need to convert an integer into a floating point to perform floating point division
  2. the compiler would not normally allow an automatic conversion, but you the programmer know it is safe

Typecasting 1

- calculate a floating point average value, given two integer types
  `double average = tantrum_sum / NUMBER_OF_VALUES;`
- no errors or warnings
- integer division (truncates)
- result has no fractional part, so it's the "wrong" answer
- solution: typecast

```
double average =
 static_cast<double>(tantrum_sum) / NUMBER_OF_VALUES;
```