Chapter Two: Fundamental Data Types I

C++ for Everyone by Cay Horstmann

Slides by Evan Gallagher & Nikolay Kirov

Lecture Goals

- To be able to define and initialize variables and constants
- To understand the properties and limitations of integer and floating-point numbers
- To write arithmetic expressions and assignment statements in C++
- To appreciate the importance of comments and good code layout

Variables

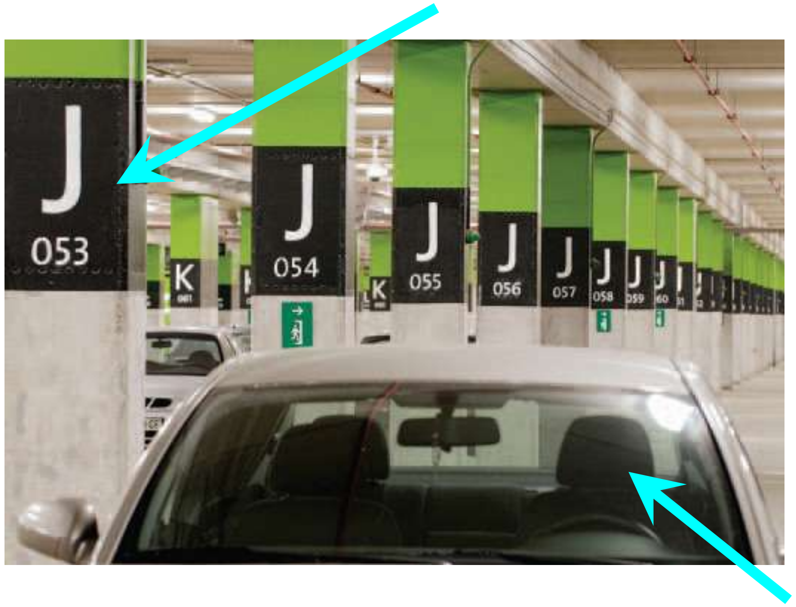
- A variable
  - is used to store information:
    - can contain one piece of information at a time.
  - has an identifier:
    The programmer picks a g*ood* name
    - A good name describes the contents of the variable or what the variable will be used for

Variables

Parking garages store cars.

Variables

Each parking space is identified
– like a variable's identifier



A each parking space in a garage "contains" a car
– like a variable's current contents.

Variables

and

each space can contain only *one* car

and

only cars, not buses or trucks

Variable Definitions

- When creating variables, the programmer specifies the type of information to be stored.
  - (more on types later)
- Unlike a parking space, a variable is often given an initial value.
  - *Initialization* is putting a value into a variable when the variable is created.
  - Initialization is not required.

Variable Definitions

The problem:

- Soft drinks are sold in cans and bottles.
- A store offers a six-pack of 12-ounce cans for the same price as a two-liter bottle.
- Which should you buy?
- (12 fluid ounces equal approximately 0.355 liters.)

Variable Definitions

The following statement defines a variable.

`cans_per_pack` is the variable's name.

```
int cans_per_pack = 6;
```

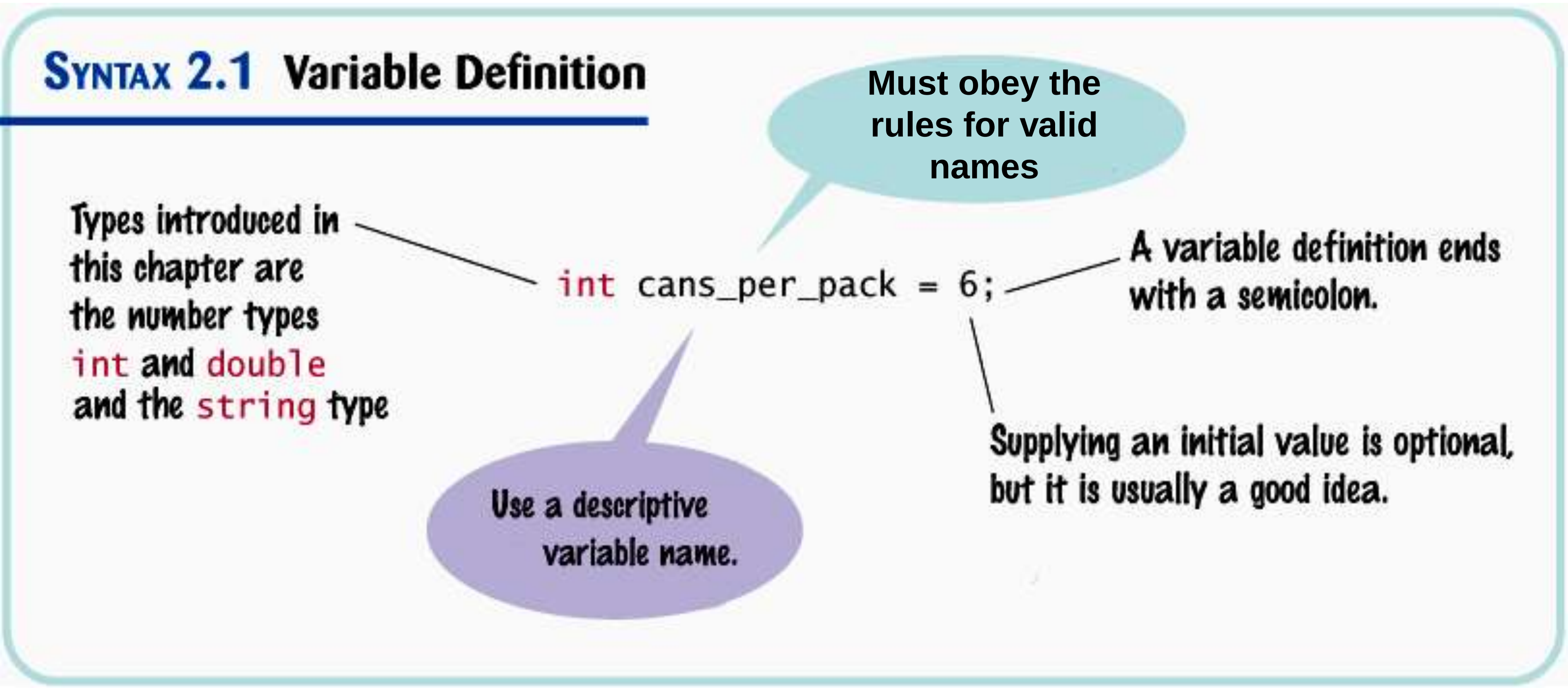
`int`

indicates that the variable **`cans_per_pack`** will be used to hold integers.

`= 6`

indicates that the variable **`cans_per_pack`** will initially contain the value 6.

Variable Definitions


SYNTAX 2.1 Variable Definition
Must obey the rules for valid names
Types introduced in this chapter are the number types int and double and the string type
int cans_per_pack = 6;
A variable definition ends with a semicolon.
Use a descriptive variable name.
Supplying an initial value is optional, but it is usually a good idea.

Variable Definitions

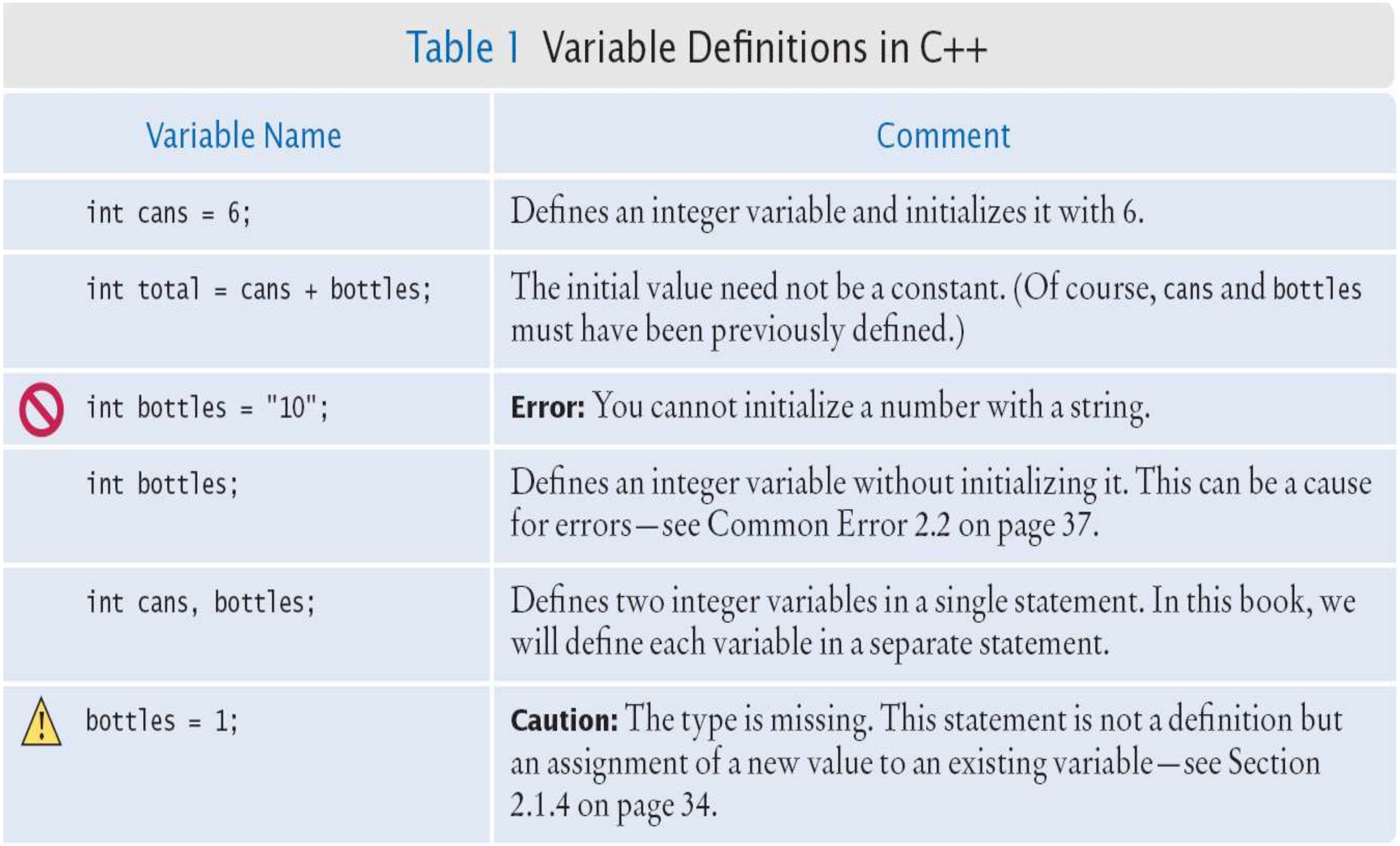


Table 1 Variable Definitions in C++

Variable Name	Comment
`int cans = 6;`	Defines an integer variable and initializes it with 6.
`int total = cans + bottles;`	The initial value need not be a constant. (Of course, `cans` and `bottles` must have been previously defined.)
🚫 `int bottles = "10";`	**Error:** You cannot initialize a number with a string.
`int bottles;`	Defines an integer variable without initializing it. This can be a cause for errors—see Common Error 2.2 on page 37.
`int cans, bottles;`	Defines two integer variables in a single statement. In this book, we will define each variable in a separate statement.
⚠ `bottles = 1;`	**Caution:** The type is missing. This statement is not a definition but an assignment of a new value to an existing variable—see Section 2.1.4 on page 34.

Number Types

A number written by a programmer is called a *number literal*.

There are rules for writing literal values:

Number Types

Number	Type	Comment
6	`int`	An integer has no fractional part.
–6	`int`	Integers can be negative.
0	`int`	Zero is an integer.
0.5	`double`	A number with a fractional part has type `double`.
1.0	`double`	An integer with a fractional part .0 has type `double`.
1E6	`double`	A number in exponential notation: 1×10^6 or 1000000. Numbers in exponential notation always have type `double`.
2.96E-2	`double`	Negative exponent: $2.96 \times 10^{-2} =$ 2.96 / 100 = 0.0296
🚫 100,000		**Error:** Do not use a comma as a decimal separator.
🚫 3 1/2		**Error:** Do not use fractions; use decimal notation: 3.5

Variable Names

- When you define a variable, you should pick a name that explains its purpose.

- For example, it is better to use a descriptive name, such as **`can_volume`**, than a terse name, such as **`cv`**.

Variable Names

In C++, there are a few simple rules for variable names:

Variable Names

1. Variable names must start with a letter or the underscore (_) character, and the remaining characters must be letters numbers, or underscores.
2. You cannot use other symbols such as $ or %. Spaces are not permitted inside names; you can use an underscore instead, as in **`can_volume`**.
3. Variable names are *case-sensitive*, that is, **`can_volume`** and **`can_Volume`** are different names.

   For that reason, it is a good idea to use only lowercase letters in variable names.
4. You cannot use *reserved words* such as **`double`** or **`return`** as names; these words are reserved exclusively for their special C++ meanings.

Variable Names

Table 3 Variable Names in C++

Variable Name	Comment
`can_volume1`	Variable names consist of letters, numbers, and the underscore character.
`x`	In mathematics, you use short variable names such as *x* or *y*. This is legal in C++, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 38).
⚠ `Can_volume`	**Caution:** Variable names are case-sensitive. This variable name is different from `can_volume`.
🚫 `6pack`	**Error:** Variable names cannot start with a number.
🚫 `can volume`	**Error:** Variable names cannot contain spaces.
🚫 `double`	**Error:** You cannot use a reserved word as a variable name.
🚫 `ltr/fl.oz`	**Error:** You cannot use symbols such as / or `.`

The Assignment Statement

- The contents in variables can “vary” over time (hence the name!).
- Variables can be changed by
  - assigning to them
    - The assignment statement
  - using the increment or decrement operator
  - inputting into them
    - The input statement

The Assignment Statement

- An *assignment statement*

  stores a new value in a variable,
  replacing the previously stored value.

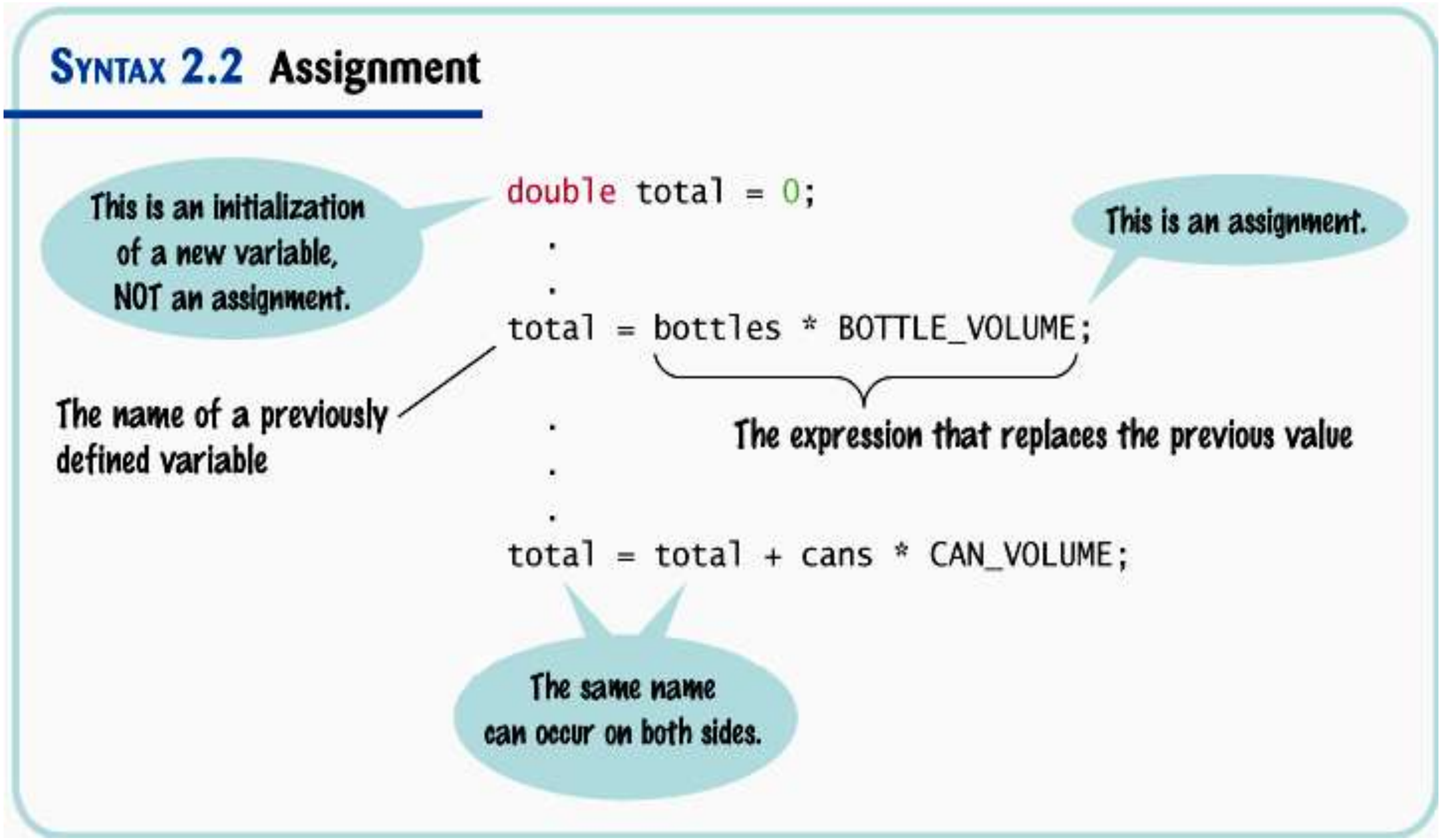
The Assignment Statement

```
cans_per_pack = 8;
```

This assignment statement changes the value stored in **`cans_per_pack`** to be 8.

The previous value is replaced.

The Assignment Statement



Syntax 2.2 Assignment

```
double total = 0;
   .
   .
total = bottles * BOTTLE_VOLUME;
   .
   .
   .
total = total + cans * CAN_VOLUME;
```

This is an initialization of a new variable, NOT an assignment.
This is an assignment.
The name of a previously defined variable
The expression that replaces the previous value
The same name can occur on both sides.

The Assignment Statement

- There is an important difference between a variable definition and an assignment statement:

```
int cans_per_pack = 6; // Variable definition
. . .
cans_per_pack = 8; // Assignment statement
```

- The first statement is the *definition* of **`cans_per_pack`**.
- The second statement is an *assignment statement.* An *existing* variable's contents are replaced.

The Assignment Statement

- The = in an assignment does ***not*** mean the left hand side is equal to the right hand side as it does in math.
- = is an instruction to do something:

  copy the value of the expression on the right
  into the variable on the left.

- Consider what it would mean, mathematically, to state:

```
counter = counter + 2;
```

counter *EQUALS* counter + 1 ?

The Assignment Statement

```
counter = 11; // set counter to 11
counter = counter + 2; // increment
```

The Assignment Statement

```
counter = 11; // set counter to 11
counter = counter + 2; // increment
```

1. Look up what is currently in counter (11)

The Assignment Statement

```
counter = 11; // set counter to 11
counter = counter + 2; // increment
```

1. Look up what is currently in counter (11)
2. Add 2 to that value (13)

The Assignment Statement

```
counter = 11; // set counter to 11
counter = counter + 2; // increment
```

1. Look up what is currently in counter (11)
2. Add 2 to that value (13)
3. *copy* the result of the addition expression *into* the variable on the left, changing counter

```
cout << counter << endl;
```

13 is shown

Constants

- Sometimes the programmer knows certain values just from analyzing the problem, for this kind of information, programmers use the reserved word `const`.
- The reserved word `const` is used to define a constant.
- A `const` is a variable whose contents cannot be changed and must be set when created.
  (Most programmers just call them constants, not variables.)
- Constants are commonly written using capital letters to distinguish them visually from regular variables:

```
const double BOTTLE_VOLUME = 2;
```

Constants

Another good reason for using constants:

```
double volume = bottles * 2;
```

What does that 2 mean?

Constants

If we use a constant there is no question:

```
double volume = bottles * BOTTLE_VOLUME;
```

Any questions?

Constants

And still another good reason for using constants:

```
double bottle_volume = bottles * 2;
double can_volume = cans * 2;
```

What does *that* 2 mean?

— *WHICH 2?*

That 2

is called a "***magic number***"

(so is that one)

because it would require magic to know what 2 means.

It is **not** good programming practice to use magic numbers. Use constants.

Constants

And it can get even worse …

Suppose that the number 2 appears hundreds of times throughout a five-hundred-line program?

Now we need to change the BOTTLE_VOLUME to 2.23 (because we are now using a bottle with a different shape)

How to change ***only*** some of those magic numbers 2’s?

Constants

Constants to the rescue!

```
const double BOTTLE_VOLUME = 2.23;
const double CAN_VOLUME = 2;

...

double bottle_volume = bottles * BOTTLE_VOLUME;
double can_volume = cans * CAN_VOLUME;
```

(Look, no magic numbers!)

Comments

- *Comments*
  are explanations for human readers of your code (other programmers).
- The compiler ignores comments completely.

```
double can_volume = 0.355; // Liters in a 12-ounce can
```

Comment

Comments

Comments can be written in two styles:

- Single line:

```
double can_volume = 0.355; // Liters in a 12-ounce can
```

The compiler ignores everything after `//` to the end of line

- Multiline for longer comments:

```
/*
   This program computes the volume (in liters)
   of a six-pack of soda cans.
*/
```

Notice All the Issues Covered So Far

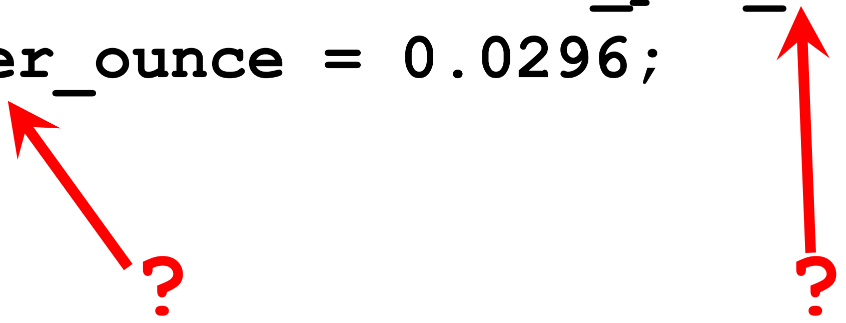
ch02/volume1.cpp

```
/*
This program computes the volume (in liters) of a six-pack of soda
cans and the total volume of a six-pack and a two-liter bottle.
*/
int main()
{
   int cans_per_pack = 6;
   const double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
   double total_volume = cans_per_pack * CAN_VOLUME;

   cout << "A six-pack of 12-ounce cans contains "
      << total_volume << " liters." << endl;

   const double BOTTLE_VOLUME = 2; // Two-liter bottle

   total_volume = total_volume + BOTTLE_VOLUME;

   cout << "A six-pack and a two-liter bottle contain "
      << total_volume << " liters." << endl;
   return 0;
}
```

Common Error – Using Undefined Variables

You must define a variable before you use it for the first time. For example, the following sequence of statements would not be legal:

```
double can_volume = 12 * liter_per_ounce;
double liter_per_ounce = 0.0296;
```



Statements are compiled in top to bottom order.

When the compiler reaches the first statement, it does not know that **`liter_per_ounce`** will be defined in the next line, and it reports an error.

Common Error – Using Uninitialized Variables

Initializing a variable is not required, but there is always a value in every variable, even uninitialized ones.

Some value will be there, the flotsam left over from some previous calculation or simply the random value there when the transistors in RAM were first turned on.

```
int bottles; // Forgot to initialize
int bottle_volume = bottles * 2;// Result is unpredictable
```

What value would be output from the following statement?

```
cout << bottle_volume << endl; // Unpredictable
```

Numeric Types in C++

In addition to the **`int`** and **`double`** types,
C++ has several other numeric types.

C++ has two other floating-point types.

The **`float`** type uses half the storage of the double type that we use in this book, but it can only store 6–7 digits.

Numeric Types in C++

Many years ago, when computers had far less memory than they have today, **`float`** was the standard type for floating-point computations, and programmers would *indulge in the luxury of "double precision"* only when they really needed the additional digits.

Ah, the good old days…

Today, the **`float`** type is rarely used.

The third type is called **`long double`**
and is for quadruple precision.
Most contemporary compilers use this type when a programmer asks for a **`double`** so just choosing **`double`** is what is done most often.

Numeric Types in C++

By the way, these numbers are called “floating-point” because of their internal representation in the computer.

Consider the numbers 29600.0, 2.96, and 0.0296.
They can be represented in a very similar way: namely, as a sequence of the significant digits: 296 and an indication of the position of the decimal point.
When the values are multiplied or divided by 10, only the position of the decimal point changes; it “floats”.

Computers use base 2, not base 10,
but the principle is the same.

Numeric Types in C++

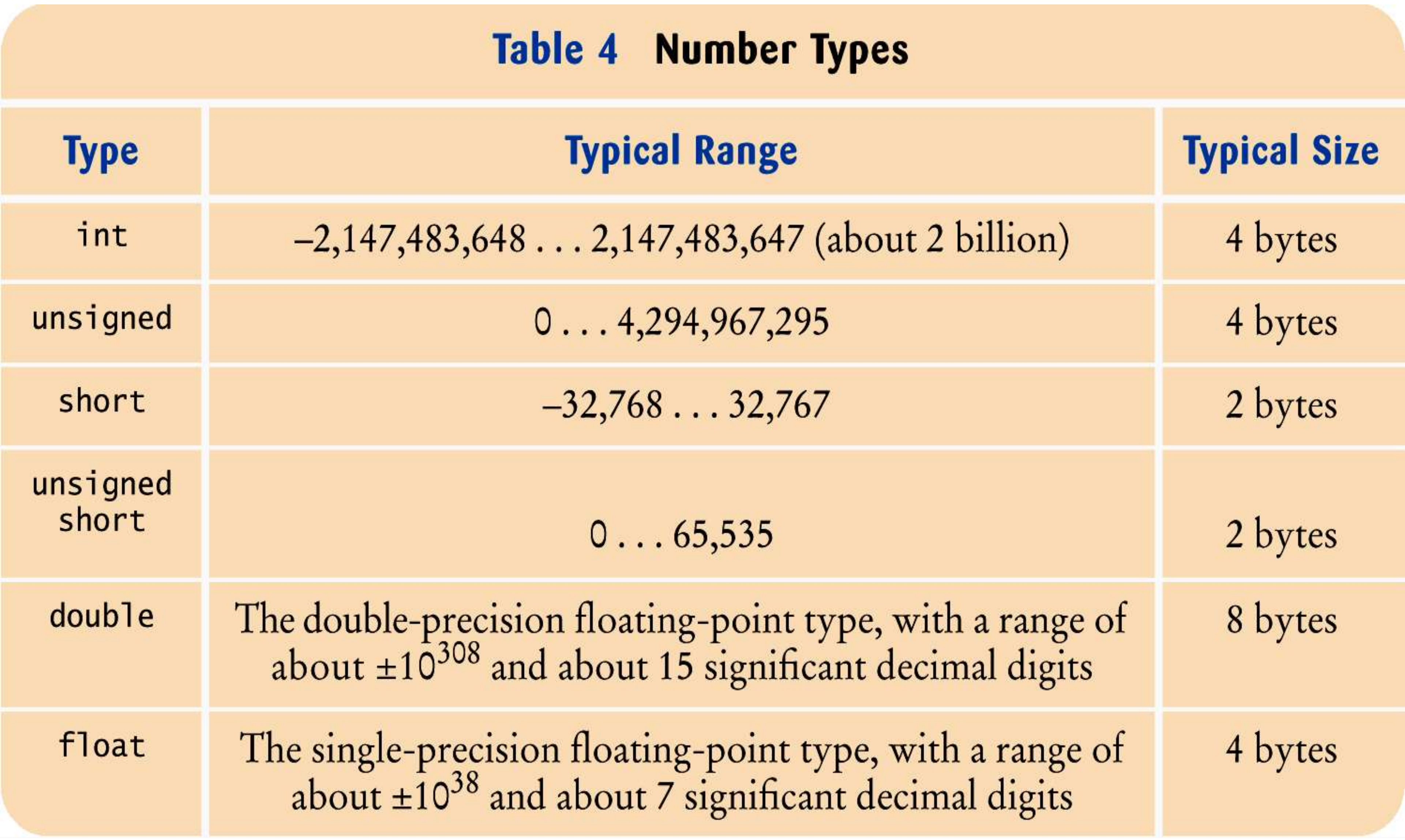


Table 4 Number Types

Type	Typical Range	Typical Size
`int`	–2,147,483,648 . . . 2,147,483,647 (about 2 billion)	4 bytes
`unsigned`	`0` . . . 4,294,967,295	4 bytes
`short`	–32,768 . . . 32,767	2 bytes
`unsigned short`	`0` . . . 65,535	2 bytes
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes

Numeric Types in C++

In addition to the **`int`** type, C++ has these additional integer types: **`short`**, **`long`**.
For each integer type, there is an unsigned equivalent:
`unsigned short`, **`unsigned long`**

For example, the **`short`** type typically has a range from –32768 to 32767, whereas **`unsigned short`** has a range from 0 to 65535. These strange-looking limits are the result of the use of binary numbers in computers.

A **`short`** value uses 16 binary digits, which can encode 2^{16} = 65536 values.

Numeric Types in C++

The C++ Standard does not completely specify the number of bytes or ranges for numeric types.

Table 4 showed typical values.

Numeric Types in C++

Some compiler manufacturers have added other types like:

`long long`

`long long`	–9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807	8 bytes

This type is not in the C++ standard as of this writing.

Numeric Ranges and Precisions

The **`int`** type has a *limited range:*

On most platforms, it can represent numbers up to a little more than two billion.

For many applications, this is not a problem, but you cannot use an **`int`** to represent the world population.

If a computation yields a value that is outside the **`int`** range, the result *overflows.*

No error is displayed.

Instead, the result is *truncated* to fit into an **`int`**, yielding a value that is most likely not what you thought.

Numeric Ranges and Precisions

For example:

```
int one_billion = 1000000000;
cout << 3 * one_billion << endl;
```

displays –1294967296 because the result is larger than an **`int`** can hold.

In situations such as this, you could instead use the **`double`** type.

However, you will need to think about a related issue: *roundoff errors*.

Arithmetic Operators

C++ has the same arithmetic operators as a calculator:

*	for multiplication:	**`a * b`**
	(not **`a · b`** or **`ab`** as in math)	
/	for division:	**a / b**
	(not ÷ or a fraction bar as in math)	
+	for addition:	**a + b**
-	for subtraction:	**a – b**

Arithmetic Operators

Just as in regular algebraic notation,
* and / have higher precedence
than + and –.

In **`a + b / 2`**,
the **`b / 2`** happens first.

Increment and Decrement

- Changing a variable by adding or subtracting 1 is so common that there is a special shorthand for these.

  The increment and decrement operators.

```
counter++; // add 1 to counter
counter--; // subtract 1 from counter
```

Increment and Decrement

C++ was based on C and so it’s one better than C, right?

Guess how C++ got its name!

Integer Division and Remainder

The % operator computes the remainder of an integer division.

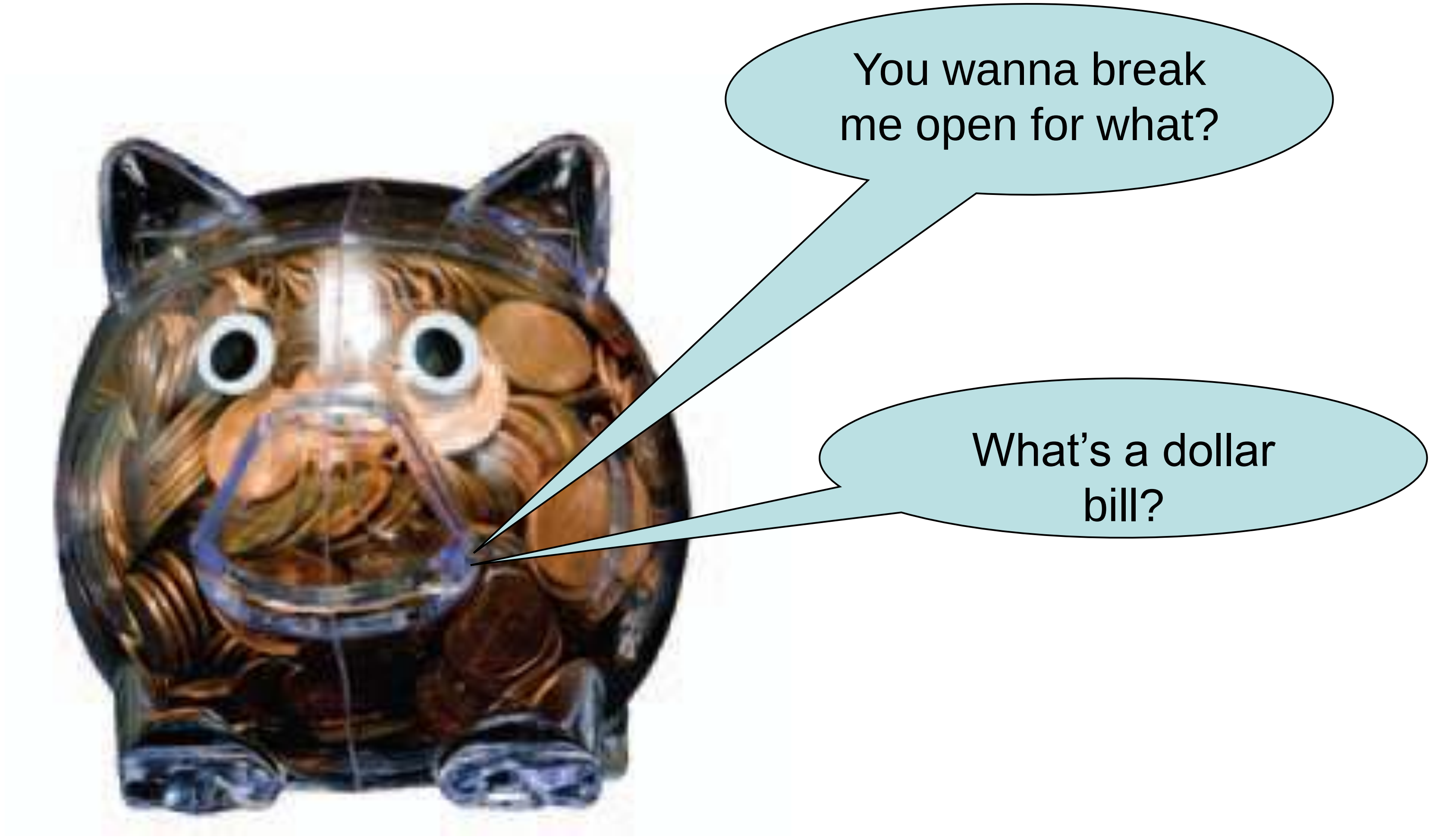
It is called the ***modulus operator*** (also modulo and mod)

It has nothing to do with the % key on a calculator

Integer Division and Remainder

Integer Division and Remainder

Integer Division and Remainder



Integer Division and Remainder

Time to break open the piggy bank.

You want to determine the value in dollars and cents stored in the piggy bank.
You obtain the dollars through an integer division by 100.
The integer division discards the remainder.
To obtain the remainder, use the % operator:

```
int pennies = 1729;
int dollars = pennies / 100; // Sets dollars to 17
int cents = pennies % 100; // Sets cents to 29
```

`(yes, 100 is a magic number)`

Integer Division and Remainder

```
dollars =
```

```
/  100;
```

```
cents =
```

```
% 100;
```

Don’t worry, Penny wasn’t broken or harmed in any way because she’s on the right hand side of the = operator.

Converting Floating-Point Numbers to Integers

- When a floating-point value is assigned to an integer variable, the fractional part is discarded:

```
    double price = 2.55;
    int dollars = price;
          // Sets dollars to 2
```

- You probably want to round to the *nearest* integer.
  To round a positive floating-point value to the nearest integer, add 0.5 and then convert to an integer:

```
int dollars = price + 0.5;
         // Rounds to the nearest integer
```

Powers and Roots

What about this?

$$b + \left(1 + \frac{r}{100}\right)^n$$

Inside the parentheses is easy:

```
1 + (r / 100)
```

But that raised to the *n*?

Powers and Roots

- In C++, there are no symbols for powers and roots. To compute them, you must call *functions*.
- The C++ library defines many mathematical functions such as **`sqrt`** (square root) and **`pow`** (raising to a power).
- To use the functions in this library, called the **`cmath`** library, you must place the line:

  ```
  #include <cmath>
  ```

  at the top of your program file.
- It is also necessary to include

  ```
  using namespace std;
  ```

  at the top of your program file.

Powers and Roots

The power function has the base followed by a comma followed by the power to raise the base to:

```
pow(base, exponent)
```

Using the **`pow`** function:

```
b * pow(1 + r / 100, n)
```

Powers and Roots

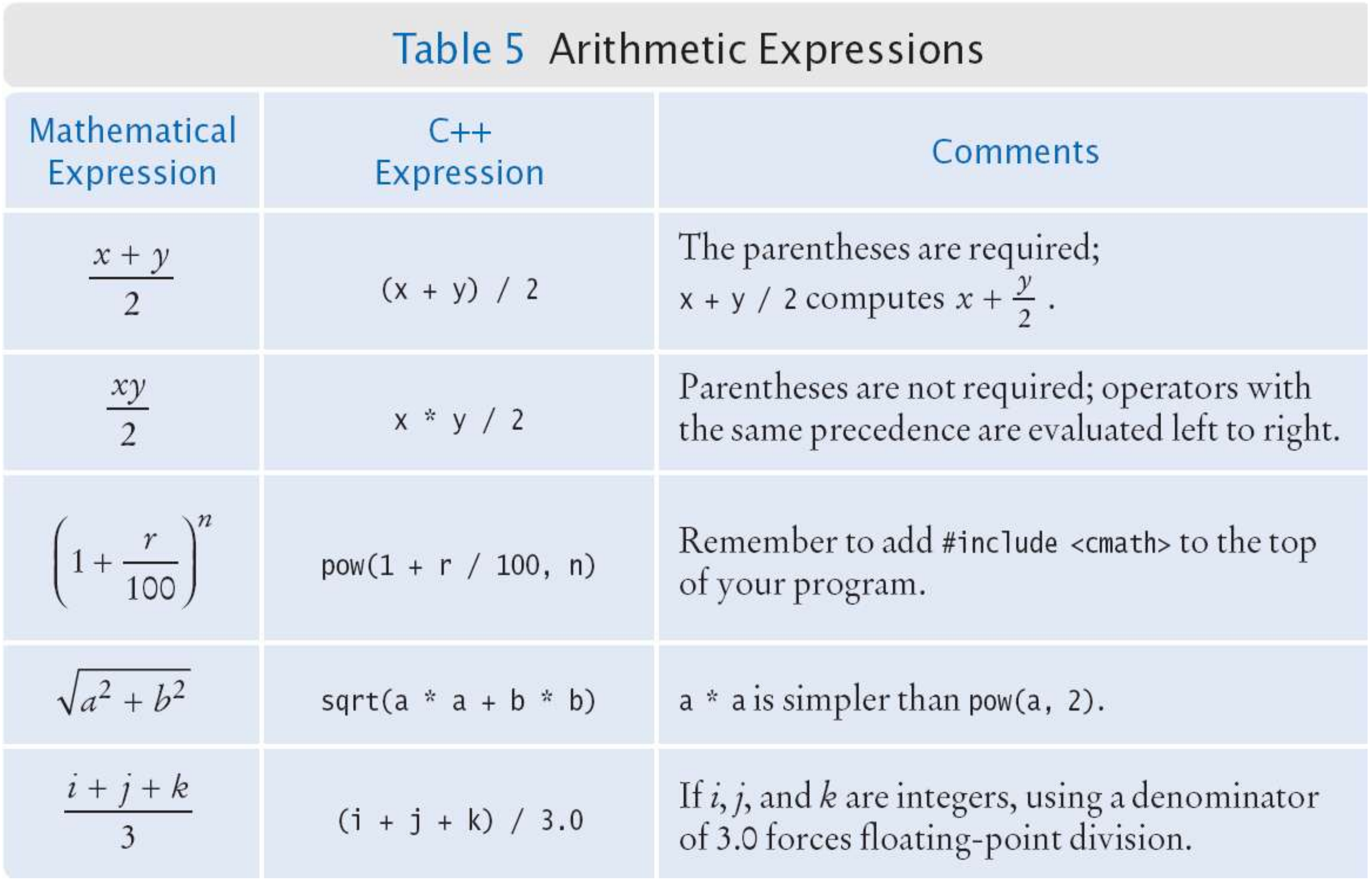


Table 5 Arithmetic Expressions

Mathematical Expression	C++ Expression	Comments
$\frac{x+y}{2}$	`(x + y) / 2`	The parentheses are required; `x + y / 2` computes $x + \frac{y}{2}$.
$\frac{xy}{2}$	`x * y / 2`	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	`pow(1 + r / 100, n)`	Remember to add `#include <cmath>` to the top of your program.
$\sqrt{a^2 + b^2}$	`sqrt(a * a + b * b)`	`a * a` is simpler than `pow(a, 2)`.
$\frac{i+j+k}{3}$	`(i + j + k) / 3.0`	If i, j, and k are integers, using a denominator of 3.0 forces floating-point division.

Other Mathematical Functions

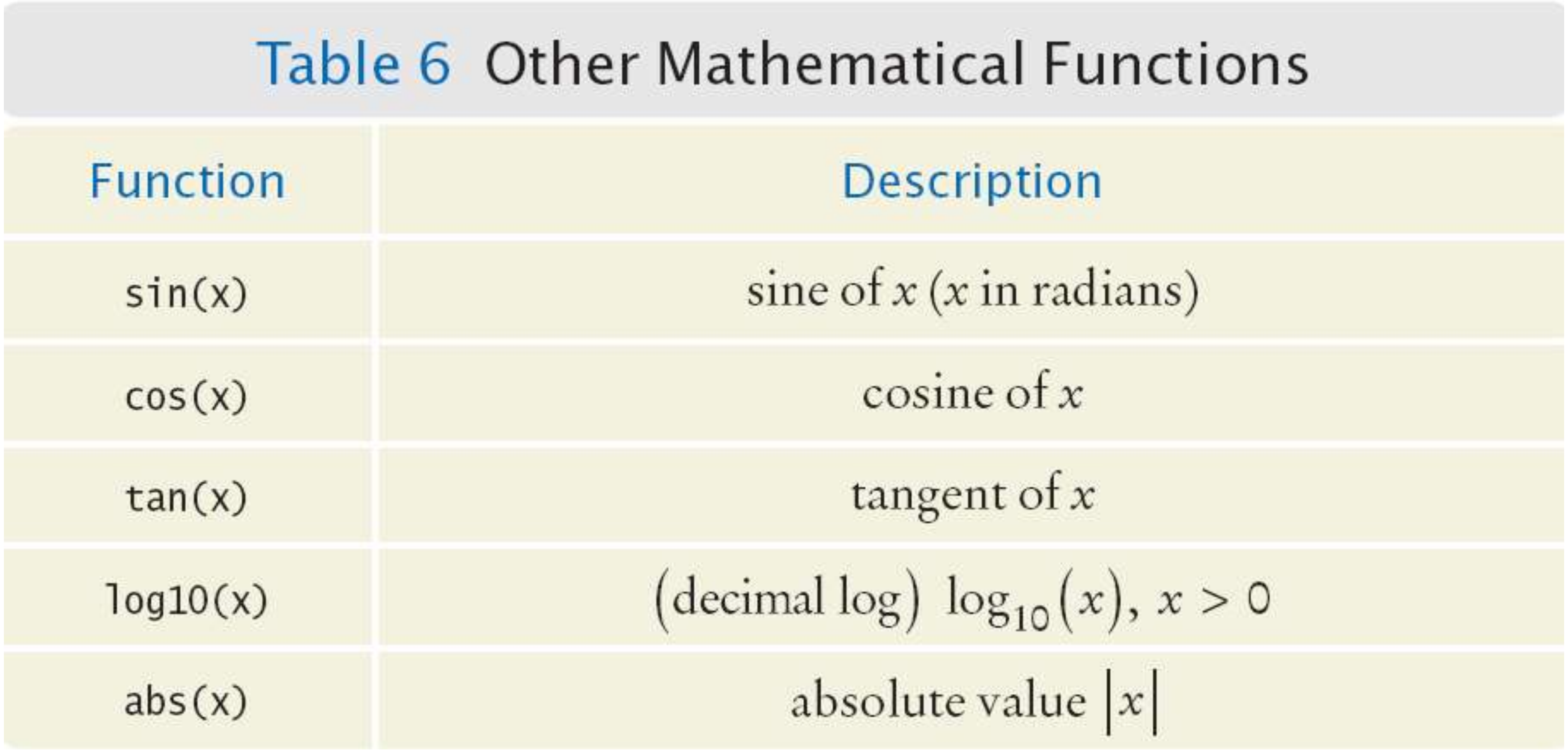


Table 6 Other Mathematical Functions

Function	Description
`sin(x)`	sine of x (x in radians)
`cos(x)`	cosine of x
`tan(x)`	tangent of x
`log10(x)`	(decimal log) $\log_{10}(x)$, $x > 0$
`abs(x)`	absolute value $\lvert x \rvert$

Common Error – Unintended Integer Division

- If both arguments of / are integers, the remainder is discarded:
  `7 / 3` is `2`, not `2.5`
- but
  ```
  7.0 / 4.0
  7 / 4.0
  7.0 / 4
  ```
- all yield `1.75`.

Common Error – Unintended Integer Division

It is unfortunate that C++ uses the same symbol: `/`
for both integer and floating-point division.
These are really quite different operations.

It is a common error to use integer division by accident.
Consider this segment that computes the average of three integers:

```
cout << "Please enter your last three test scores: ";
int s1;
int s2;
int s3;
cin >> s1 >> s2 >> s3;
double average = (s1 + s2 + s3) / 3;  ← ← ← Error
cout << "Your average score is " << average << endl;
```

Common Error – Unintended Integer Division

What could be wrong with that?

Of course, the average of `s1`, `s2`, and `s3` is

```
   (s1+ s2+ s3) / 3
```

Here, however, the `/` does not mean division in the mathematical sense.

It denotes integer division because both `(s1 + s2 + s3)` and `3` are integers.

Common Error – Unintended Integer Division

For example, if the scores add up to 14,
the average is computed to be 4.

WHAT?

Yes, the result of the integer division of 14 by 3 is 4
How many times does 3 evenly divide into 14?
Right!

That integer 4 is then moved into the floating-point variable **`average`**.

So 4.0 is stored.

That's not what I want!

Common Error – Unintended Integer Division

The remedy is to make the numerator or denominator into a floating-point number:

```
double total = s1 + s2 + s3;
double average = total / 3;
```

or

```
double average = (s1 + s2 + s3) / 3.0;
```

Common Error – Unbalanced Parentheses

Consider the expression

```
(-(b * b - 4 * a * c) / (2 * a)
```

?

What is wrong with it?

The parentheses are *unbalanced*.
This is very common with complicated expressions.

Common Error – Unbalanced Parentheses

Now consider this expression

```
-(b * b - (4 * a * c))) / 2 * a)
```

It is still is not correct.

There are too many closing parentheses.

Common Error – Unbalanced Parentheses – A Solution

The Muttering Method

Count (not out loud, of course!)

starting with 1 at the 1st parenthesis
add one for each `(`
subtract one for each `)`

```
- ( b * b - ( 4 * a * c ) )  ) / 2 * a )
  1         2             1 0 -1 OH NO!
                                  (still to yourself – careful!)
```

If your count is not 0 when you finish, or if
you ever drop to -1, STOP, something is wrong.

Common Error – Forgetting Header Files

Every program that carries out input or output needs the **`<iostream>`** header.

If you use mathematical functions such as **`sqrt`**, you need to include **`<cmath>`**.

If you forget to include the appropriate header file, the compiler will not know symbols such as **`cout`** or **`sqrt`**.

If the compiler complains about an undefined function or symbol, check your header files.

Common Error – Forgetting Header Files

Sometimes you may not know which header file to include.

Suppose you want to compute the absolute value of an integer using the `abs` function.

As it happens, this version of `abs` is not defined in the `<cmath>` header but in `<cstdlib>`.

How can you find the correct header file?

Why do you think Tim Berners-Lee invented going online?

A reference site on
the Internet such as: http://www.cplusplus.com
is a great help.

(Note: do not attempt to click the link if this slide is being projected.)

End Fundamental Data Types I