Arithmetic, input and Output

Due this week

- **Homework 1**
  - Submit pdf file on Canvas. PDF
- Start going through the textbook readings and watch the videos
  - Take **Quiz 2**.
- Participation: 3-2-1 (published on Friday)
- Check the due date! **No late submissions!!**

Today

- Arithmetic
- Console input
- Formatted output

Arithmetic

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

- 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**

Just like in regular math, * and / have higher precedence than + and –

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

- 10/4 has a remainder of 2, so **10 % 4 = 2**

Increment and Decrement

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

- Increment (add 1): `count++; // add 1 to count`
- Decrement (subtract 1): `count--; // subtract 1 from count`

Example: What is the value of `count` after the code below?

```
int count = 3;
count--;
count = count + 2;
count++;
```

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
```

- **Note:** rounding 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
```

Casts

- Occasionally, you need to store a value into a variable of a different type, or print it in a different way
- A **cast** is a conversion from one type (e.g., int) to another type (e.g., double)

Example: How can we print or capture the exact quotient from two int variables?

```
   int x= 25;
   int y = 10;
   cout << "The quotient is " << x / y;
//gives int quotient of 2; not what we want
```

Casts

The ***cast*** conversion syntax:

```
   static_cast<newtype>( data_to_convert)
```

Example, to get an exact quotient, we cast one of the int variables to a double before dividing:

```
   int x= 25;
   int y = 10;
   cout << x / static_cast<double>(y);
     //gives double quotient of 2.5
```

An older version of the cast conversion syntax also works, but its use is discouraged:

```
      (newtype)data_to_convert

   cout << x / (double)y;
     //gives double quotient of 2.5
```

Combining Assignment and Arithmetic

- In C++, you can combine arithmetic and assignments.
- For example, the statement

```
    total += cans * CAN_VOLUME;
```

  is a shortcut for

```
    total = total + cans * CAN_VOLUME;
```

- Similarly,

```
    total *= 2;
```

  is another way of writing

```
    total = total * 2;
```

- Many programmers *prefer* using this form of coding.

Powers and Roots

- In C++, there are no symbols for powers and roots.

- To compute them, you must call *functions*. Don’t forget to include the *cmath* library

```
#include <cmath>
using namespace std;
```

Example of `pow()` function call

The pow() function has two arguments:

- Base
- exponent

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

Using the **pow** function:

```
double balance = b * pow(2, n);
```

Other Mathematical Functions (from `<cmath>`)

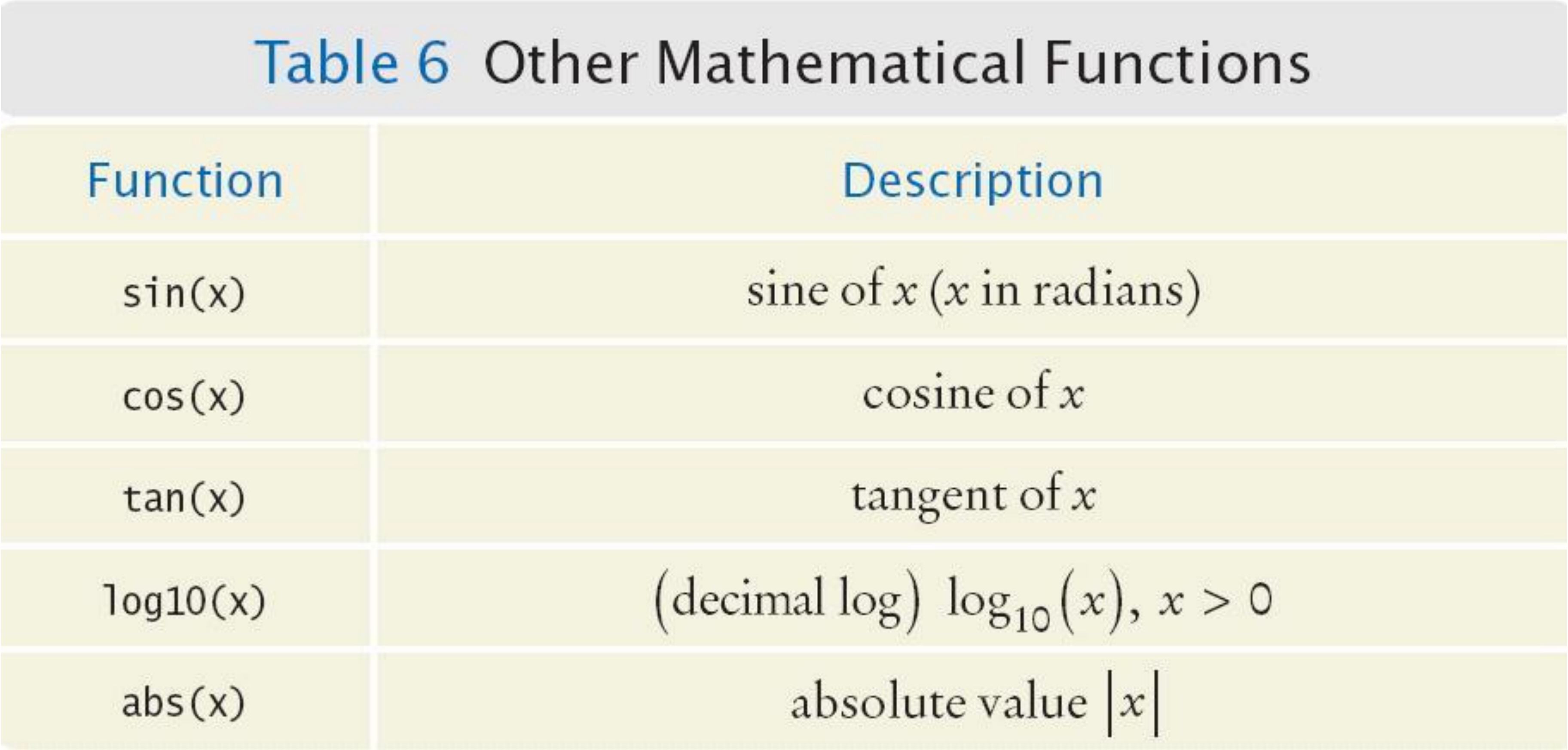

Table 6 Other Mathematical Functions

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 $\lvert x \rvert$

Example:

```
    double population = 73693997551.0;
    double decimal_log = log10(population);
```

Input and Output

Input

- Sometimes the programmer does not know what value should be stored in a variable – but the user does.
- The programmer must get the input value from the user
    - Users need to be prompted -- *how else would they know they need to type something?*
    - Prompts are done in output statements
- The keyboard needs to be read from
    - This is done with an input statement

Input with `cin >>`

The **input** statement

- To read values from the keyboard, you input them from an object called **`cin`.**
- The "double greater than" operator **>>** denotes the “send to” command.

```
cin >> bottles;
```

is an *input statement*.

Of course, the variable **`bottles`** must be defined earlier.

Input with `cin >> to multiple variables`

You can read more than one value in a single input statement:

```
cout << "Enter the number of bottles and cans: ";
cin >> bottles >> cans;
```

The user can supply both inputs on the same line:

```
Enter the number of bottles and cans: 2 6
```

Alternatively, the user can press the *Enter* key or *tab* key after each input, as `cin` treats all blank spaces the same

Formatted Output

Formatted Output

- When you print an amount in dollars and cents, you want it to be *rounded* to two significant digits.
- You learned earlier how to round off and store a value but, for output, we want to round off *only* for display.
- A ***manipulator*** is something that is sent to **`cout`** to specify how values should be formatted.
- To use manipulators, you must include the **`iomanip`** header in your program:

  ```
  #include <iomanip>
  ```

  and of course

  ```
  using namespace std;
  ```

  is also needed

Formatted Output for Dollars and Cents: `setprecision()`

Which do you think the user prefers to see on her gas bill?

```
Price per liter: $1.22
```

or

```
Price per liter: $1.21997
```

Table 4: Formatted Output Examples

Output Statement	Output	Comment
cout << 12.345678;	12.3457	By default, a number is printed with 6 significant digits.
cout << fixed << setprecision(2) << 12.3;	12.30	The fixed and setprecision manipulators control the number of digits after the decimal point.
cout << ":" << setw(6) << 12;	: 12	Four spaces are printed before the number, for a total width of 6 characters.
cout << ":" << setw(2) << 123;	:123	If the width not sufficient, it is ignored.
cout << setw(6) << ":" << 12;	:12	The width only refers to the next item. Here, the : is preceded by five spaces.

Formatted Output, Dollars and Cents

- You can combine manipulators and values to be displayed into a single statement:

```
price_per_liter = 1.21997;
cout << fixed << setprecision(2)
   << "Price per liter: $"
   << price_per_liter << endl;
```

- This code produces this output:

```
Price per liter: $1.22
```

Formatted Output with `setw()` to Align Columns

- Use the **`setw`** manipulator to set the *width* of the next output field.
- The width is the total number of characters, including digits, the decimal point, and spaces.
- If you want aligned columns of certain widths, use the **`setw()`** manipulator.
- For example, if you want a number to be printed, right justified, in a column that is eight characters wide, you use

```
      << setw(8)
```

before EVERY COLUMN's DATA.

Exercise: Formatting Examples

- Given **`int quantity = 10;  double price = 19.95;`**

What do the following statements print?

```
cout << "Quantity:" << setw(4) << quantity;

cout << "Price:" << fixed << setw(8) << setprecision(2) << price;

cout << "Price:" << fixed << setprecision(2) << price;

cout << fixed << setprecision(3) << price;

cout << fixed << setprecision(1) << price;
```

Formatted Output, Another Example

This code:

```
price_per_ounce_1 = 10.2372;
price_per_ounce_2 = 117.2;
price_per_ounce_3 = 6.9923435;
cout << setprecision(2);
cout << setw(8) << price_per_ounce_1;
cout << setw(8) << price_per_ounce_2;
cout << setw(8) << price_per_ounce_3;
cout << "---------" << endl;
```

produces this output:

```
   10.24
  117.20
    6.99
---------
```

`setprecision` versus `setw`: Persistence

- There is a notable difference between the **`setprecision`** and **`setw`** manipulators.

- Once you set the precision, that precision is used for all floating-point numbers until the next time you set the precision.

- But **`setw`** affects only the *next* value.

- Subsequent values are formatted without added spaces.

Additional Slides for Curious Minds

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`, and `7.0 / 4.0` all yield `1.75`

Remember: if at least one of the operands is a double, then the result will be a double.

Common Error – Unintended Integer Division

- It is unfortunate that C++ uses the same symbol **/** for both integer and floating-point division.
- It is a common error to use integer division by accident. Consider this segment that computes the average of three integers:

```
int score1 = 2
int score2 = 3
int score3 = 5
double average = (score1 + score2 + score3) / 3;
cout << “Your average score is ” << average << endl;
```

Common Error – Unintended Integer Division

- Here, however, the **/** denotes **integer division** because both `(score1 + score2 + score3)` and `3` are integers.
- **FIX:** make the numerator or denominator into a floating-point number:

```
double total = score1 + score2 + score3;
double average = total / 3;
```

or

```
double average = (score1 + score2 + score3) / 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*
- very common with complicated expressions

- Check out **The Muttering Method** - textbook

Spaces in Expressions

It is easier to read

```
x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);
```

than

```
x1=(-b+sqrt(b*b-4*a*c))/(2*a);
```

Itreallyiseasiertoreadwithspaces!

So always **use spaces** around all operators: **+ - * / % =**

Spaces in Expressions

- **Unary minus:** A minus sign - used to negate a single quantity like: -b
- **Binary minus:** A minus sign taking the difference between two quantities: a - b
- We do not put a space after a unary minus.
  - Helps distinguish it from a binary one.
- It is customary not to put a space between a function name and the parentheses.

  Write: **sqrt(x)**

  not **sqrt (x)**