Variables (2)

More operations, constants, and mixed expressions
By the end of this lesson you will be able to:

- Understand how expressions involving integral and floating-point values are evaluated
- Use casting to change one data type to another
- Understand mathematical operations using characters and Boolean values
- Understand why variable initialization is important
- Create constants
- Use combined assignment operators
- Increment and decrement variables
Mixed expressions

- A **mixed expression** is one that involves integral and floating-point data types
  - Integral data types: `bool`, `char`, `int`
  - Floating-point data types: `double`

- Binary mathematical operation:
  - Two integral operands – result is integral
  - Two floating-point operands – result is floating-point
  - One integral, one floating-point – integral is converted to floating-point before evaluation, result is floating-point
Mixed expressions examples

- $5.0 / 2.0 = 2.5$
- $5.0 / 2 = 2.5$
  - 2 is converted to a floating-point first
- $5 / 2.0 = 2.5$
  - 5 is converted to a floating-point first
- $5 / 2 = 2$
  - The result is truncated, not rounded (e.g. the remainder is discarded)
A more complicated example

- Remember that expressions are evaluated left-to-right and considering order of operations
- \( \frac{5}{3} + \frac{7}{2} + \frac{11.0}{10} \) – \( \frac{5}{3} \) is evaluated to 1
  - 5 / 3 is evaluated to 1
- \( \frac{7}{2} \) is evaluated to 3
  - 7 / 2 is evaluated to 3
- \( \frac{11.0}{1} \) is evaluated to 1.1
  - 11.0 / 1 is evaluated to 1.1
- The result is 5.1
Typecasting

- **Typecasting, type casting, type coercion, and type conversion** all refer to the process of converting a value of one data type to another data type.

- The evaluation of mixed expressions are an *implicit* form of typecasting that happens automatically.

- If you want to force the typecast *explicitly* (e.g. you want \( \frac{5}{2} \) to evaluate to 2.5), you can do so in a variety of ways.
Typecasting in C++

- There are several ways of basic typecasting in C++
  - Functional cast: `datatype(expression)`
  - C-style cast: `(datatype)expression`
  - Static cast: `static_cast<datatype>(expression)`

- The standard way of doing this in C++ is with a **static cast**

- Note that type-casting a floating-point value to an integral value will truncate (not round) the value
char details

- Recall how characters are represented by ASCII values
- If you print out a `char` (e.g. 'A') it will display the character
- If you print out a `char` plus an integer value, it will display an integer
- Use `static_cast` to display the character
bool details

- Remember that a bool has values of true (1) or false (0)
- If you print out a bool it will display the number
- Doing math with a bool will always convert it to the decimal value of 0 or 1
Bad implicit casts

- It is bad style to do implicit casts just by using assignment.
- Some programmers will create useless variables to avoid casting.
- Good style:
  ```cpp
  char c = 'Q';
  cout << static_cast<int>(c) << endl;
  ```
- Bad style:
  ```cpp
  char c = 'Q';
  int ascii_value = c;
  cout << ascii_value << endl;
  ```
Variable initialization

- Recall that a variable can be initialized when declared
  \[
  \text{int } x = 5; \\
  \]

- If you declare a variable and try to use it without explicitly placing a value in it you may:
  - Get a compiler warning or error
  - Get an unexpected or erroneous result

- Before using a variable you should do one of the following:
  - Initialize it at declaration
  - Assign a value to it after declaration
  - Have the user input the value (e.g. using \texttt{cin})
Constants

- **Constants** are useful for values that you need in your program but are never going to change.

- For example, if you are going to use an interest rate of 0.05 throughout your program, there are many ways of doing this:
  - **Hard coding** the value of 0.05 when you need it
  - Creating a constant using `const` that has a name (e.g. `RATE`)
  - Using a `#define` preprocessor directive

- Advantages of using a constant:
  - If the value needs to change (e.g. interest rate is not 0.05 but 0.08) you need only to change it in one location
  - With a name, other people understand what that value is used for
Constants

- The standard way of using a constant is to use the `const` keyword before the datatype
  
  ```
  const double PI = 3.14159265358979;
  ```

- It is not required, but conventional, to use all uppercase identifiers for constants (e.g. `E`, `PI`, `GOLDEN_RATIO`)

- If you try to re-assign a value to a constant, your program will not compile
Combined assignment operators

- It is very common to need to do something like add a value to a variable and store the result in that same variable

- A statement like `x = x + 3;` will work fine, but there are also combined assignment operators:

  - `+=`  
    
    `a += expr;`  →  
    
    `a = a + (expr);`

  - `-=`  
    
    `a -= expr;`  →  
    
    `a = a - (expr);`

  - `*=`  
    
    `a *= expr;`  →  
    
    `a = a * (expr);`

  - `/=`  
    
    `a /= expr;`  →  
    
    `a = a / (expr);`

  - `%=`  
    
    `a %= expr;`  →  
    
    `a = a % (expr);`
Increment and decrement

- It is very common to want to add/subtract 1 from a variable
- Use the increment operator, `++`, or the decrement operator, `--`
- All of the following statements work similarly:
  
  \[
  \begin{align*}
  x &= x + 1; \\
  z &= z - 1; \\
  x &= x + 1; \\
  z &= z - 1; \\
  x &= ++x; \\
  z &= --z;
  \end{align*}
  \]

- Whitespace is optional, but `x++;` is more conventional than `x  ++;`
Pre- and post-increment/decrement

- You can place the increment or decrement operators before or after the variable.
- In a statement by itself, there’s no difference: `x++;` and `++x;` are the same.
- As part of an expression, **pre-increment** (`++x`) will increment first and then evaluate, while **post-increment** (`x++`) will evaluate first and then increment.

```
int x = 1;

cout << ++x << endl; // increment x then output 2
cout << x++ << endl; // output 2 then increment x
cout << x << endl; // output 3
```
- **Pre-decrement** (`--x`) and **post-decrement** (`x--`) work similarly.
Exploration

- Write programs that do the following:
  - Ask the user for a character input, then output the decimal ASCII
  - Ask the user for a character input, then output the next 3 characters in the ASCII table
  - Ask the user for integer input for a Fahrenheit temperature, then output the temperature in Celsius
  - Calculate the number of molecules in a user-input number of moles (use the value of 6.022e23 for Avogadro’s number)