Operators

Lecture 3
COP 3014 Fall 2021

August 31, 2021

Operators

- Special built-in symbols that have functionality, and work on operands
- **operand** – an input to an operator
- **Arity** - how many operands an operator takes
  - *unary operator* – has one operand
  - *binary operator* – has two operands
  - *ternary operator* – has three operands
- Examples:

```
int x, y = 5, z;
z = 10;        // assignment operator (binary)
x = y + z;    // addition (binary operator)
x = -y;       //-y is a unary operation (negation)
x++;          // unary (increment)
```

Operators

- **cascading** - linking of multiple operators, especially of related categories, together in a single statement:

```
cascading arithmetic operators
x = a + b + c - d + e;
// cascading assignment operators
x = y = z = 3;
```

- **Precedence** - rules specifying which operators come first in a statement containing multiple operators

```
x = a + b * c; // b * c happens first, since *
               //has higher precedence than +
```

- **Associativity** - rules specifying which operators are evaluated first when they have the same level of precedence.
  - Most (but not all) operators associate from left to right.

Assignment Operator

- Value on the right side (R-value) is assigned to (i.e. stored in) the location (variable) on the left side (L-value)
  - **R-value** – any expression that evaluates to a single value (name comes from "right" side of assignment operator)
  - **L-value** – A storage location! (not any old expression). A variable or a reference to a location. (name comes from "left" side of assignment operator
  - Typical Usage
    `variable_name = expression`
- The assignment operator returns a reference to the L-value
- Example:

```
x = 5;
y = 10.3;
z = x + y; // right side can be an expression
a + 3 = b; // ILLEGAL! Left side must be a
           // storage location
```

Assignment Operator

- Associates right-to-left

```
x = y = z = 5; // z = 5 evaluated first, returns
                  z, which is stored in y and so on
```

- Use appropriate types when assigning values to variables:

```
int x, y;
x = 5843;
y = -1234; // assign integers to int variables

double a, b;
a = 12.98;
b = -345.8; //assign decimal numbers to floats

char letter, symb;
letter = 'Z';
symb = '$'; // character literals to char types
```

- Be careful to not confuse assignment $=$ with comparison $==$

Arithmetic Operators

Name	Symbol	Arity	Usage
Add	+	binary	x + y
Subtract	-	binary	x - y
Multiply	*	binary	x * y
Divide	/	binary	x / y
Modulus	%	binary	x % y
Minus	-	unary	-x

- Division is a special case
- Modulus % not legal for floating point types. / gives floating point result

```
double x = 19.0, y = 5.0, z;
z = x / y;          // z is now 3.8
```

Arithmetic Operators

- For integer types, / gives the quotient, and % gives the remainder (as in long division)

```
int x = 19, y = 5, q, r;
q = x / y;              // q is 3
r = x % y;              // r is 4
```

- An operation on two operands of the same type returns the same type
- An operation on mixed types (if compatible) returns the "larger" type

```
int x = 5;
float y = 3.6;
z = x + y;          // what does z need to be?
                    // x + y returns a float.
```

Operator Precedence

- Arithmetic has usual precedence
  1. parentheses
  2. Unary minus
  3. *, /, and %
  4. + and -
  5. operators on same level associate left to right
- Many different levels of operator precedence (about 18)
- When in doubt, can always use parentheses
- Example

```
z = a - b * -c + d / (e - f);
          // 7 operators in this statement
```

What order are they evaluated in?

Some short-cut assignment operators (with arithmetic)

v += e;	means	v = v + e;
v -= e;	means	v = v - e;
v *= e;	means	v = v * e;
v /= e;	means	v = v / e;
v %= e;	means	v = v % e;

Please look at the Note on Operator Precedence on the course website.

Increment and Decrement Operators

- These are shortcut operators for adding or subtracting 1 from a variable.
- Shortcut for x=x+1

```
++x; // pre-increment (returns reference to new x)
x++; // post-increment (returns value of old x)
```

- Shortcut for x=x-1

```
--x; // pre-decrement
x--; // post-decrement
```

- Pre-increment: incrementing is done **before** the value of x is used in the rest of the expression
- Post-increment: incrementing is done **after** the value of x is used in the rest of the expression

Increment and Decrement Operators

- Note - this only matters if the variable is actually used in another expression.
- The two statements (x++ and ++x)by themselves have the same effect.
- Examples

```
int x = 5, count = 7;
result = x * ++count; // result = 40, count = 8

int x = 5, count = 7;
result = x * count++; // result = 35, count = 8
```

Automatic Type Conversions

- Typically, matching types are expected in expressions
- If types don't match, ambiguity must be resolved
- There are some legal automatic conversions bewteen built-in types.
- Rules can be created for doing automatic type conversions between user-defined types, too
- For atomic data types, can go from "smaller" to "larger" types when loading a value into a storage location.
- General rule of thumb: Allowed if no chance for partial data loss.

  `char -> short -> int -> long -> float -> double -> long double`
- Should avoid mixing unsigned and signed types, if possible

Automatic Type Conversions: Examples

```
int i1, i2;
double d1, d2;
char c1;
unsigned int u1;

d1 = i1; // legal.
c1 = i1; // illegal.  trying to stuff int into char
i1 = d1; // illegal.  Might lose decimal point data.
i1 = c1; // legal
u1 = i1; // dangerous (possibly no warning)
d2 = d1 + i2; // result of double + int is a double
d2 = d1 / i2; // floating point division (at least
              // one operand a float type)
```

Explicit type conversions (casting)

- Older C-style cast operations look like:
  `c1 = (char)i2; // cast a copy of the value of i2 as a char, and assign to c1`
  `i1 = (int)d2; // cast a copy of the value of d2 as an int, and assign to i1`
- Better to use newer C++ cast operators. For casting between regular variables, use static_cast
  `c1 = static_cast<char>(i2);`
  `i1 = static_cast<int>(d2);`
- Just for completeness, the newer C++ cast operators are:
  - static_cast
  - dynamic_cast
  - const_cast
  - reinterpret_cast