Lecture 5

C++ Operators

Shibo Li

shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

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:

```
int x, y = 5, z;
// 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 +
```

Operators

- **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 Operators

- 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). Avariable or a reference to a location. (name comes from "left" side of assignment operator

```
variable name = expression
```

Assignment Operators

- 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 Operators

- 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
```

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

- Modulus % is not legal for floating point types. / gives floating point results

```
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

Operator Precedence

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

Operator Precedence

- Arithmetic has usual precedence
- When in doubt, can always use parentheses

```
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

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
```

Increment and Decrement Operators

- Pre-increment: incrementing is done **first** and the **updated** value of x is used in the rest of the expression
- Post-increment: incrementing is done **first** but **a copy of** the old value of x is used in the rest of the expression
- Note - this only matters if the variable is actually used in another expression. The two statements (x++ and ++x)by themselves have the same discernible effect, even if the post increment operation returns a copy of the old value.

Increment and Decrement Operators

Example 1:

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

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

Increment and Decrement Operators

Example 1:

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

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

Increment and Decrement Operators

Example 2:

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

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

Increment and Decrement Operators

Example 2:

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

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

Increment and Decrement Operators

Example 3:

```
int x = 5, count = 7;
result = x * (++count);

int x = 5, count = 7;
result = x * (count++);
```

Increment and Decrement Operators

Example 3:

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

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

Increment and Decrement Operators

Example 4:

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

int x = 5, count = 7++;
result = x * (count++);
```

Increment and Decrement Operators

Example 4:

```
int x = 5, count = ++7; // illegae statement, we
    cannot change the literal 7
result = x * count;

int x = 5, count = 7++; // illegae statement, we
    cannot change the literal 7
result = x * count;
```

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-intypes.
- Rules can be created for doing automatic type conversions between user-defined types, too

Automatic Type Conversions

- For atomic data types, can go from "smalle" to "larger" types when loading a value into a storage location.
- General rule of thumb: Allowed if no chance for partial data

```
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`