Operators (right shift is >>)

Operator, Category, Duty	Operator, Category, Duty	Operator, Category, Duty
=, Assignment, Equals	!=, Comparison, Is not equal to	>, Bitwise, Shift bits right
+, Mathematical, Addition	&&, Logical, AND	~, Bitwise, One's complement
–, Mathematical, Subtraction	||, Logical, OR	+, Unary, Positive
*, Mathematical, Multiplication	!, Logical, NOT	–, Unary, Negative
/, Mathematical, Division	++, Mathematical, Increment by 1	*, Unary, Pointer
%, Mathematical, Modulo	--, Mathematical, Decrement by 1	&, Unary, Address
>, Comparison, Greater than	&, Bitwise, AND	sizeof, Unary, Returns the size of an object
>=, Comparison, Greater than or equal to	|, Bitwise, Inclusive OR	., Structure, Element access
<, Comparison, Less than	^, Bitwise, Exclusive OR (XOR or EOR)	->, Structure, Pointer element access
<=, Comparison, Less than or equal to	<<, Bitwise, Shift bits left	?:, Conditional , Funky if operator expression
==, Comparison, Is equal to		

Operators – continued

- C operators can be classified according to the number of operands which they take.
- C has unary operators, binary operators, and one ternary operator (the conditional operator **?** :)
- The operands of C operators are expressions, which can be constants, variables, or expressions which contain one or more operators.
- Some operators share the same symbol: - and & both have more than one meaning
- Expressions will always be *evaluated* by the code which the compiler generates; that is, an expression has **a value** (which has **a type**, of course).
- There is a table of operators posted on Piazza which shows which operators are unary, binary or ternary, and the precedence and associativity of each operator (precedence/associativity covered below).

Precedence and Associativity

- Precedence refers to the relationship between two operators in terms of the order in which the operations are performed.
- Precedence is a binary relation, that is, it is defined with respect to pairs of (adjacent) operators. Binary operators are adjacent if they have one operand in common; unary operators are adjacent if they have the same operand.
- We can always enforce a precedence different from the precedence specified by the language for 2 operators by using parentheses, because operations inside parentheses are done first (Have the highest precedence).
- If two adjacent operators have *the same precedence*, then *associativity* is relevant.
- L-R associativity means that the operation specified by the leftmost operator is done first, and then the one specified by the rightmost operator. R-L associativity, of course, means the opposite order.

Associativity

```
#include <stdio.h>
/* program to show associativity of the “/” operator */
int main() {
    float num1;
    float num2;
    float num3;
    num1 = 2.0 / 1.0 / 2.0;
    num2 = (2.0 / 1.0) / 2.0;
    num3 = 2.0 / (1.0 / 2.0);
    printf("num1 is: %f \n", num1);
    printf("num2 is: %f \n", num2);
    printf("num3 is: %f \n", num3);
}
```

- What is printed?

Associativity

```
#include <stdio.h>
/* program to show associativity of the “/” operator */
int main() {
    float num1;
    float num2;
    float num3;
    num1 = 2.0 / 1.0 / 2.0;
    num2 = (2.0 / 1.0) / 2.0;
    num3 = 2.0 / (1.0 / 2.0);
    printf("num1 is: %f \n", num1);     /* num1 is: 1.000000        */
    printf("num2 is: %f \n", num2);
    printf("num3 is: %f \n", num3);
}
```

Associativity

```
#include <stdio.h>
/* program to show associativity of the "/" operator */
int main() {
    float num1;
    float num2;
    float num3;
    num1 = 2.0 / 1.0 / 2.0;
    num2 = (2.0 / 1.0) / 2.0;
    num3 = 2.0 / (1.0 / 2.0);
    printf("num1 is: %f \n", num1);    /*   num1 is: 1.000000       */
    printf("num2 is: %f \n", num2);    /*   num2 is: 1.000000
                                                Result with L-R associativity */

    printf("num3 is: %f \n", num3);
}
```

Associativity

```
#include <stdio.h>
/* program to show associativity of the "/" operator */
int main() {
    float num1;
    float num2;
    float num3;
    num1 = 2.0 / 1.0 / 2.0;
    num2 = (2.0 / 1.0) / 2.0;
    num3 = 2.0 / (1.0 / 2.0);
    printf("num1 is: %f \n", num1);    /*   num1 is: 1.000000       */
    printf("num2 is: %f \n", num2);    /*   num2 is: 1.000000
                                              Result with L-R associativity */

    printf("num3 is: %f \n", num3);    /*   num3 is: 4.000000
                                              Result with R-L associativity    */
}
```

Example for Precedence

- Let's see how an expression is evaluated, using the precedence and associativity in C.
- Suppose, before this statement is executed,
  - a = 1, b = 3, and c = 5:

```
d = ++a * c + b++;
```

- How does the compiler determine the order of operations?

Beware of unsequenced side effects

- Given these declarations:
  int f(int x, int y, int z);
  int a = 2;

- What will be the values of x, y, and z?
  a = f(++a, a++, a=5);

Example for Precedence

- How does the compiler determine the order of operations?
- We can take the view that the compiler does the binary operation with the highest precedence first, then next highest, but expressions with unary operators are not evaluated until they need to be, in order to evaluate a larger expression of which they are a part.
- We will also suppose that operands of binary operators are evaluated left to right (this is true for most compilers, and it is true for ours).
- A good practice is to use parentheses to show the order of evaluation, starting with the binary operator which has the highest precedence, and going to the lowest, considering associativity where necessary.
- So, let's try to parenthesize the binary operators in the expression above, after parenthesizing all unary operator expressions (unary operators have higher precedence than all binary operators generally).

Example for Precedence

Parenthesize unary operators:

```
d = (++a) * c + (b++);
```

Precedence of binary operators: * first, then +, then =

Now we can add the rest of the parentheses:

```
d = (((++a) * c) + (b++));
```

What is d after execution of this statement (Remember, before this statement, a = 1, b = 3, and c = 5)?

Example for Precedence (continued)

Value of expression

```
                2       5         3
(d = (((++a) * c) + (b++)));
```

d = (2 * 5) + 3

So, d has the value 13 after the statement is executed (a = 2, b = 4, and c = 5)

Assignment Operator

- In C, assignments are **expressions**. This means that an assignment expression, just as any expression, has **a value**, which is the value of the rightmost expression.
- Lowest precedence (except for the comma operator)
- Embedded assignments – legal anywhere an expression is legal.
  - This allows multiple assignment: a = b = 1; /*R-L associativity */
  - We'll see how these are used in C a bit later.
  - Other assignment operators (compound assignment operators) – same associativity – R-L
    - +=, -= , *=, /=, %=
- **NOTE**: Using an assignment operator (=) is legal anywhere it is legal to compare for equality (==), so it is not a syntax error (some compilers may give a warning, although **our compiler will not!!!**).

The $20 Million Bug

Expert C Programming:Deep C Secrets, Peter Van Der Linden

In Spring 1993, in the Operating System development group at SunSoft, we had a "priority-one" bug report come in describing a problem in the asynchronous I/O library. The bug was holding up the sale of $20 million worth of hardware to a customer who specifically needed the library functionality, so we were extremely motivated to find it. After some intensive debugging sessions, the problem was finally traced to a statement that read:

```
x==2;
```

It was a typo for what was intended to be an assignment statement. The programmer's finger had bounced on the "equals' key, accidentally pressing it twice instead of once. The statement as written compared x to 2, generated true or false, and discarded the result.

C is enough of an expression language that the compiler did not complain about a statement which evaluated an expression, had no side-effects, and simply threw away the result. We didn't know whether to bless our good fortune at locating the problem, or cry with frustration at such a common typing error causing such an expensive problem

= vs ==

- We can get help from the compiler in some situations
- Desired code:
  - If(tokens ==6)
- What if we accidentally drop an = sign?
  - If(tokens = 6)
- The above code compiles and looks good at a glance but is broken. But we can code defensively
  - If(6 == tokens)
- If we forget an = we get code that *doesn't* compile
  - If(6 = tokens)
- This only works with constants that can't be the target of assignment

Arithmetic Operators

- Mathematical Symbols
  - + - * / %
  - addition, subtraction, multiplication, division, modulus
- Works for both integers and float
  - + - * /
  - / operator performs integer division if both operands are integer, i.e., *truncates*; otherwise, if at least one operand is float, performs floating point division (i.e., *casting* is used – more on this later).
- % operator divides two integer operands and gives integer result of the remainder
- Associativity – L-R.
- Precedence:

  1. Anything inside ()
  2. * / %
  3. + -

Unary Operators

- ++a and a++ have the behavior of a = a + 1
- --a and a-- have the behavior of a = a - 1

[Postfix operators have higher precedence than prefix operators]

- HOWEVER…
  - ++a → a is incremented BEFORE a is evaluated in the expression
  - --a → a is decremented BEFORE a is evaluated in the expression
  - a++ → a is incremented AFTER a is evaluated in the expression
  - a-- → a is decremented AFTER a is evaluated in the expression

In both examples below, the final value of a is 2

  - ++a-- → compiler will accept, but behavior is undefined/inexact

```
int main()
{
    int a = 1;
    printf (" a is %d", ++a);
    return 0;
}
/* 2 will be printed  */
```

```
int main()
{
    int a = 1;
    printf (" a is %d", a++);
    return 0;
 }
/* 1 will be printed */
```