Operators and Expressions

Arithmetic Operators

- The binary arithmetic operators are :

  + - * / %

- The expression *x % y* produces the remainder when *x* is divided by *y*
- The operands of *%* should be unsigned integers.
- Some unary operators are: + -

  There are many other operators, discussion is deferred.

Relational and Logical Operators

- The relational operators are

  >   >=   <   <=   ==   !=

- Logical operators are

  && || !

Expressions

- Expressions combine operands (variables, constants) and operators to produce new values.
  - Eg: 3 + count * (i+j)
- A constant expression is an expression that involves only constants.
  - A variable can be initialized using a constant expression. Eg: int total = 2+3*4;
  - Is the value of *3 * 4 + 5 = 17* , Or

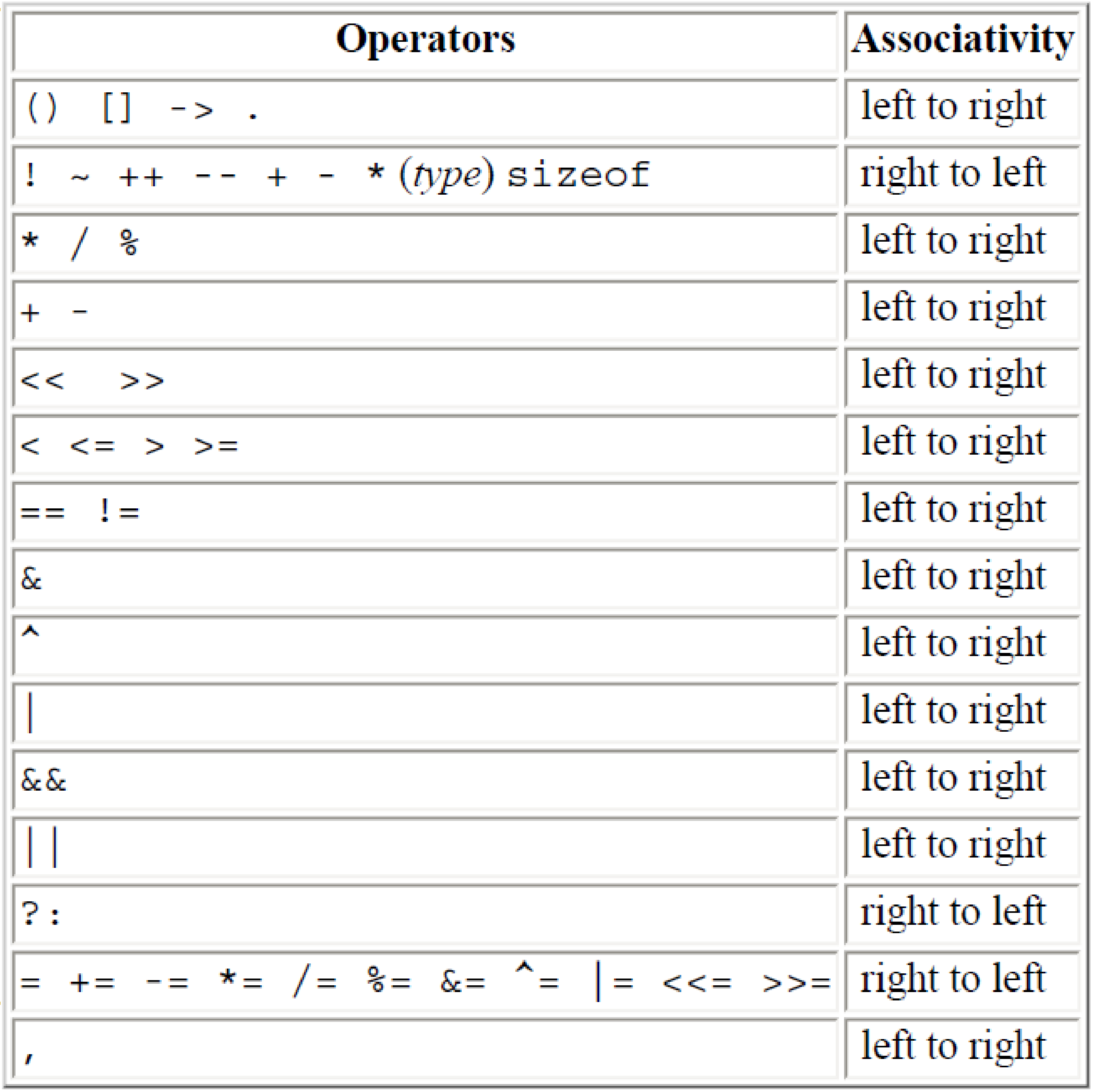
    = 27

Precedence and Associativity Rules

- *3 * 4 + 5* is *((3 * 4) + 5),* but not *(3 * (4 + 5))*
- It is because * is at higher precedence than +
- *3 / 4 / 5 = ((3 / 4) / 5),* but not *(3 / (4 / 5))*
- It is because *associativity* of / is from left to right.
- *- - 4 = (- (-4)). This is because associativity of - (unary minus operator) is from right to left.*

Precedence and Associativity

() [] -> .	left to right
! - ++ -- + - & (type) sizeof (unary)	right to left
* / % (binary)	left to right
+ - (binary)	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
=	right to left



Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - *` (*type*) `sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Precedence and Associativity

- For a complete list of rules, refer Table 2-1, in Page 53 of Kernighan and Ritchie book.
- What is the value of

  ```
  4 + 3 * 2  ==  9
  ```
- TRUE is represented with 1 and FALSE with 0
- The unary negation operator ! Converts a non-zero operand into 0, and zero operand into 1
  - So, what is the value of !(2+3 == 4)
  - int i;   i = !5;   /*what is the value of i */
  - int i = 15;   printf("%d", i = 10); /* what will get on the screen */

Expressions

- Check the following
  - int count;   count = 2 + 3 == 4;
  - Is the above syntactically valid?
  - What is the value of count ?
- The operands used in an expression should be ideally of same type. The result of the expression will be of same type as operands type.
  - int i;   i = 3/4; /* what will be value of i */
- Automatic type conversion is done some times when the operands are of different types.

Automatic Type Conversions

- A narrower type is converted to wider type.
  - In `3 + 4.0` 3 is converted to float 3.0
  - But

```
int  sum[5];
sum[3.0] = 100;
```

/* wrong, 3.0 is not allowed, because it doesnot make sense */

- Expressions that might lose information, like assigning a longer integer type to a shorter, may draw a warning, but they are not illegal.

Expressions

- What will be the values of i, j and k

```
float i, j;  int k;
i = 3/2;
j = 3.0/2;
k = 3.0/2;
```

- Conversions take place across assignments; the value of the right side is converted to the type of the left.

Explicit type conversions

- You can force the type to be converted.
  - (float) 3; /* has value 3.0 */
- Syntax : (type-name) expression
  - float f; f = (float)3/2;

    /* 3 → 3.0 because of explicit type conversion

    3.0/2 → 3.0/2.0 because of automatic conversion

    So, f gets value 1.5 */
- (type-name) is actually an unary operator.
  - double d = 3.5;

    int i;

    i = (int) d; /* value of d itself is not changed */

    /* the value of (int) d is 3 */

Assignment Operators and Expressions

- i = i+2 ; can be written in a compressed form as i += 2;
- Most binary operators of the form

  variable = variable op expression can be written like variable op= expression
- x *= y+1; means x = x * (y+1); and

  not x = x * y + 1;

Conditional Expression

- expr1 ? expr2 : expr3 is an expression and has value expr2 if expr1 is non-zero(true), otherwise has value expr3
- z = (a > b) ? a : b; /* z = max(a, b) */
- x = 5 ? 0:1; /* what is value of x */

Increment and Decrement operators

- ++ (increment unary operator)
- -- (decrement unary operator)
- x ++ means x = x+1
  - So, 5++ is not allowed because it means 5 = 5+1
- ++ x also means x = x+1 But there is a subtle difference.
  - y = x++ ; is same as y = x; x = x+1;
    - Post increment : Use and then increment
  - y = ++x ; means x = x+1; y = x;
    - Pre increment : increment and then use
- Same rules for --

Increment and Decrement Operators

- Check

```
int j, k, m;
j = 5;
k = j++;

m =  ++(j + k);  /*This is illegal */
```

- Check

```
int a[5], j = 0;
a[++j] = 4;   /* j = j+1;

                  a[j] = 4; */

               /*So,  a[1] = 4  */
```