Structured Programming

Precedence & associativity

- All operators have two important properties called ***precedence*** and ***associativity***.
  - Both properties affect how operands are attached to operators
- Operators with higher precedence have their operands bound, or grouped, to them before operators of lower precedence, regardless of the order in which they appear.
- In cases where operators have the same precedence, associativity (sometimes called binding) is used to determine the order in which operands grouped with operators.

- 2 + 3 * 4
- 3 * 4 + 2

- a + b – c;
- a = b = c;

Precedence & associativity

Class of operator	Operators in that class	Associativity	Precedence
primary	() [] -> .	Left-to-Right	HIGHEST
unary	**cast operator** **sizeof** **& (address of)** *** (dereference)** - + ~ ++ -- !	Right-to-Left	
multiplicative	* / %	Left-to-Right	
additive	+ -	Left-to-Right	
shift	<< >>	Left-to-Right	
relational	< <= > >=	Left-to-Right	
equality	== !=	Left-to-Right	

Precedence & associativity

Class of operator	**Operators in that class**	**Associativity**	**Precedence**
bitwise AND	**&**	Left-to-Right	
bitwise exclusive OR	^	Left-to-Right	
bitwise inclusive OR	|	Left-to-Right	
logical AND	**&&**	Left-to-Right	
logical OR	||	Left-to-Right	
conditional	**? :**	Right-to-Left	
assignment	= += -= *= /= %= >>= <<= &= ^=	Right-to-Left	
comma	,	Left-to-Right	**LOWEST**

Parenthesis

- The compiler groups operands and operators that appear within the parentheses first, so you can use parentheses to specifiy a particular grouping order.
  - (2 – 3) * 4
  - 2 – (3 * 4)

- The inner most parentheses are evaluated first. The expression (3+1) and (8-4) are at the same depth, so they can be evaluated in either order.

1 + ((3+1) / (8 – 4) – 5)

1 + (4 / (8 – 4) – 5)

1 + (4 / 4 – 5)

1 + (1 – 5)

1 + -4

-3

binary arithmetic operators

Operator	Symbol	Form	Operation
multiplication	*	**x * y**	x times y
division	/	**x / y**	x divided by y
remainder	**%**	**x % y**	remainder of x divided by y
addition	+	**x + y**	x plus y
subtraction	**-**	**x - y**	x minus y

The remainder operator

- Unlike other arithmetic operators, which accept both integer and floating point operands, the remainder operator accepts only integer operands!
- If either operand is negative, the remainder can be negative or positive, depending on the implementation
- The ANSI standard requires the following relationship to exist between the remainder and division operators
  - ***a*** equals ***a*%*b*** + (***a*/*b***) * ***b*** for any integral values of ***a*** and ***b***

arithmetic assignment operators

Operator	Symbol	Form	Operation
assign	=	**a = b**	put the value of ***b*** into ***a***
add-assign	+=	**a += b**	put the value of ***a***+***b*** into ***a***
substract-assign	-=	**a -= b**	put the value of ***a***-***b*** into ***a***
multiply-assign	*=	**a *= b**	put the value of ***a********b*** into ***a***
divide-assign	/=	**a /= b**	put the value of ***a***/***b*** into ***a***
remainder-assign	%=	**a %= b**	put the value of ***a***%***b*** into ***a***

arithmetic assignment operators

int m = 3, n = 4;
float x = 2.5, y = 1.0;

m += n + x – y	m = (m + ((n+x) –y))	(8)
m /= x * n + y	m = (m / ((x*n) + y))	(0)
n %= y + m	n = (n % (y + m))	(invalid operants)
x += y -= m	x = (x + (y = (y – m)))	(0.5)

increment & decrement operators

Operator	Symbol	Form	Operation
postfix increment	++	**a++**	get value of a, then increment a
postfix decrement	--	**a--**	get value of a, then decrement a
prefix increment	++	**++a**	increment a, then get value of a
prefix decrement	--	**--b**	decrement a, then get value of a

increment & decrement operators

```
main () {
        int j=5, k=5;
        printf("j: %d\t k : %d\n", j++, k--);
        printf("j: %d\t k : %d\n", j, k);
        return 0;
}
```

```
main () {
        int j=5, k=5;
        printf("j: %d\t k : %d\n", ++j, --k);
        printf("j: %d\t k : %d\n", j, k);
        return 0;
}
```

increment & decrement operators

int j = 0, m = 1, n = -1,i=5;

m++ - --j	(m++) – (--j)	(1- -1=2)
m += ++j * 2	m = (m + ((++j) * 2)	(1+1*2=3)
x=i* i++	x= j * (j++)	
	(implementation dependent 25 or 30)	

comma operator

- Comma operator allows you to evaluate two or more distinct expressions wherever a single expression allowed!
  - for (j = 0, k = 100; k – j > 0; j++, k--)

relational operators

Operator	Symbol	Form	Result
greater than	>	**a > b**	1 if a is greater than b; else 0
less than	<	**a < b**	1 if a is less than b; else 0
greater than or equal to	>=	**a >= b**	1 if a is greater than or equal to b; else 0
less than or equal to	<=	**a < = b**	1 if a is less than or equal to b; else 0
equal to	==	**a == b**	1 if a is equal to b; else 0
not equal to	!=	**a != b**	1 if a is NOT equal to b; else 0

relational operators

```
int  j=0, m=1, n=-1;
float x=2.5, y=0.0;
```

`j > m`	`j > m`	(0)
`m/n < x`	`(m / n) < x`	(1)
`j <= m >= n`	`( (j <= m) >= n)`	(1)
`++j == m != y * 2`	`((++j) == m) != (y * 2)`	(1)

logical operators

Operator	Symbol	Form	Result
logical AND	**&&**	**a && b**	1 if a and b are non zero; else 0
logical OR	||	**a || b**	1 if a or b is non zero; else 0
logical negation	**!**	**!a**	1 if a is zero; else 0

logical operators

```
int  j=0, m=1, n=-1;
float x=2.5, y=0.0;

j && m                    (j) && (m)                     (0)
j < m && n < m            (j < m) && (n < m)             (1)
x * 5 && 5 || m / n       ((x * 5) && 5) || (m / n)      (1)
!x || !n || m + n         ((!x) || !n) || (m + n)        (0)
```

bit manipulation operators

Operator	Symbol	Form	Result
right shift	>>	**x >> y**	x shifted right by y bits
left shift	<<	**x << y**	x shifted left by y bits
bitwise AND	**&**	**x & y**	x bitwise ANDed with y
bitwise inclusive OR	|	**x | y**	x bitwise ORed with y
bitwise exclusive OR (XOR)	^	**x ^ y**	x bitwise XORed with y
bitwise complement	~	**~x**	bitwise complement of x