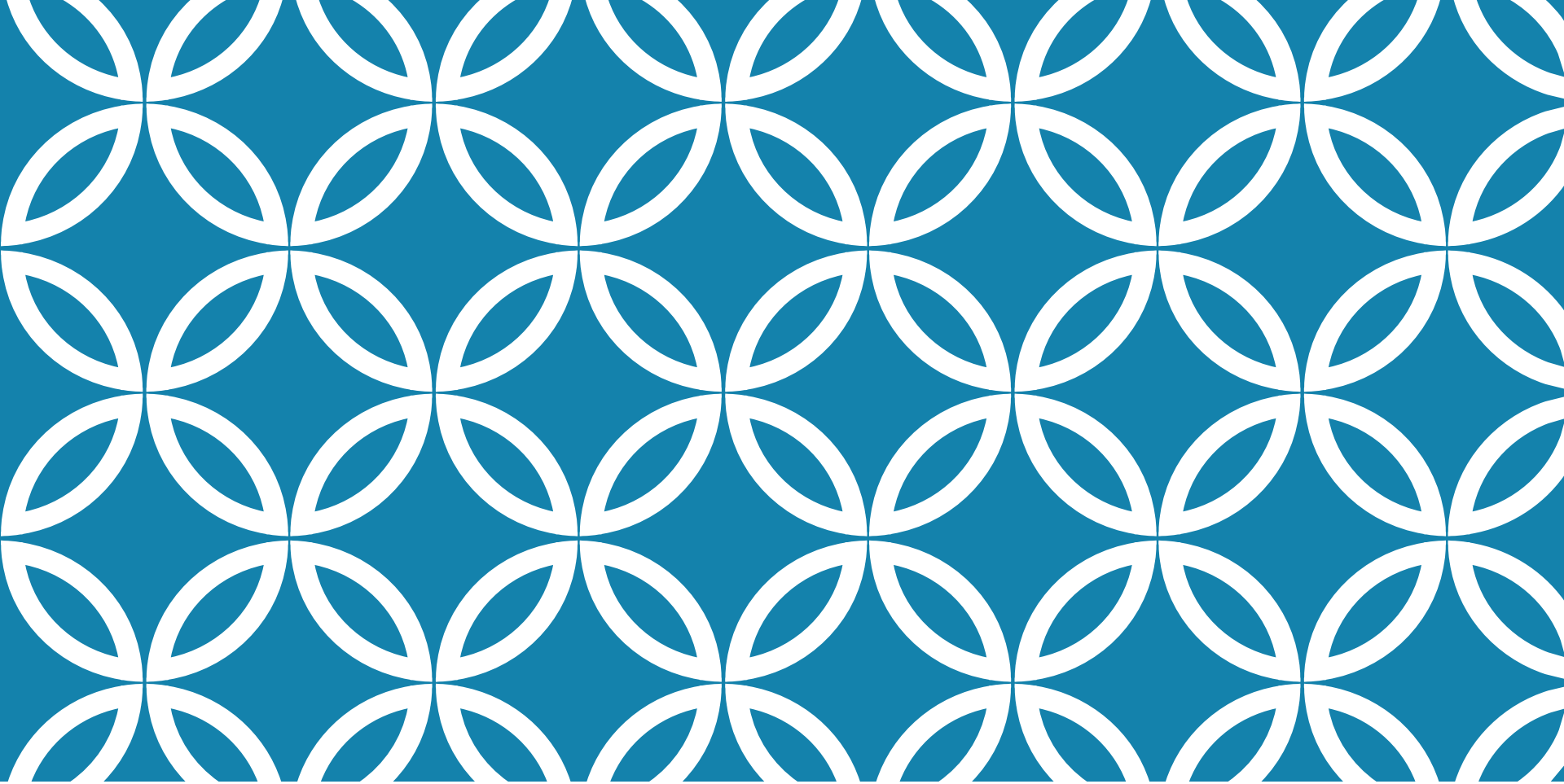


C OPERATORS, OPERANDS, EXPRESSIONS & STATEMENTS

C OPERATORS

Unary Operators		
+	Unary plus maintains the value of the operand. Any plus sign in front of a constant is not part of the constant.	`+aNumber`
–	Unary minus operator negates the value of the operand. For example, if `num` variable has the value 200, `-num` has the value -200. Any minus sign in front of a constant is not part of the constant.	`–342`

C OPERATORS

You can put the ++/-- before or after the operand. If it appears before the operand, the operand is incremented/decremented. The incremented value is then used in the expression. If you put the ++/-- after the operand, the value of the operand is used in the expression before the operand is incremented/decremented.		
++	Post-increment. After the result is obtained, the value of the operand is incremented by 1.	`aNumber++`
--	Post-decrement. After the result is obtained, the value of the operand is decremented by 1.	`aNumber--`
++	Pre-increment. The operand is incremented by 1 and its new value is the result of the expression.	`++yourNumber`
--	Pre-decrement. The operand is decremented by 1 and its new value is the result of the expression.	`--yourNumber`

PRECEDENCE OF INCREMENT OPERATORS

The precedence of **post increment** is more than precedence of pre increment, and their associativity is also different. The associativity of pre increment is right to left, of post increment is left to right.

C OPERATORS

&	The address-of operator (`&`) gives the address of its operand. The operand of the address-of operator can be either a function designator or an l-value that designates an object that is not a bit field and is not declared with the register storage-class specifier. The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.	`&addressOfData`

C OPERATORS

`sizeof`	sizeof operator for expressions	**`sizeof`** `723`
`sizeof``()`	sizeof operator for types	**`sizeof`**`(char)`

C OPERATORS

Arithmetic Operators		
*		`p = q * r;`
/		`a = b / c;`
%		`x = y % z;`
+		X = a+b
-		Y=a-b

C OPERATORS

- For relational expression, `0` is `FALSE`, `1` is `TRUE`.
- Any numeric value is interpreted as either `TRUE` or `FALSE` when it is used in a C / C++ expression or statement that is expecting a logical (true or false) value. The rules are:

1. A value of `0` represents `FALSE`.
2. Any non-zero (including negative numbers) value represents `TRUE`.

C OPERATORS

Relational Inequality Operators		
<	Specifies whether the value of the left operand is less than the value of the right operand.	`i < 7`
>	Specifies whether the value of the left operand is greater than the value of the right operand.	`j > 5`
<=	Specifies whether the value of the left operand is less than or equal to the value of the right operand.	`k <= 4`
>=	Specifies whether the value of the left operand is greater than or equal to the value of the right operand.	`p >= 3`

C OPERATORS

Relational Equality Operators		
==	Indicates whether the value of the left operand is equal to the value of the right operand.	`nChoice == 'Y'`
!=	Indicates whether the value of the left operand is not equal to the value of the right operand.	`nChoice != 'Y'`

C OPERATORS

Expressions	Evaluates as
`(3 == 3) && (4 != 3)`	True (1) because both operands are true
`(4 > 2) || (7 < 11)`	True (1) because (either) one operand/expression is true
`(3 == 2) && (7 == 7)`	False (0) because one operand is false
`! (4 == 3)`	True (1) because the expression is false
`NOT(FALSE) = TRUE`	
`NOT(TRUE) = FALSE`	

Program example: less-than, greater-than, less-than and equal-to, greater-than and equal-to, not-equal, equal and assignment operators

C OPERATORS

Logical NOT Operator (unary)		
!	Logical not operator. Produces value 0 if its operand or expression is true (nonzero) and the value 1 if its operand or expression is false (0). The result has an `int` type. The operand must be an integral, floating, or pointer value.	`!(4 == 2)` `!x`

Operand (or expression)	Output
`!0`	`1 ( T )`
`!1`	`0 ( F )`

C OPERATORS

Logical AND Operator		
&&	Indicates whether both operands are true. If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0. `1 && 4` evaluates to `1` (`True && True = True`) while `1 & 4` (`0001 & 0100 = 0000`) evaluates to `0`	`x && y`

Operand1	Operand2	Output
0	0	0 (F)
0	1	0 (F)
1	0	0 (F)
1	1	1 (T)

C OPERATORS

Logical OR Operator		
`||`	Indicates whether either operand is true. If either of the operands has a nonzero value, the result has the value 1. Otherwise, the result has the value 0. The logical `OR` (`||`) should not be confused with the bitwise `OR` (`|`) operator. For example: `1 || 4` evaluates to `1` (or `True || True = True`) while `1 | 4` (`0001 | 0100 = 0101`) evaluates to `5`	`x || y`

Operand1	Operand2	Output
0	0	0 (F)
0	1	1 (T)
1	0	1 (T)
1	1	1 (T)

Program example: Logical-AND, logical-OR, logical-NOT, XOR operators

C OPERATORS

Conditional Operator (ternary)		
`?:`	The first operand/expression is evaluated, and its value determines whether the second or third operand/expression is evaluated: If the value is true, the second operand/expression is evaluated. If the value is false, the third operand/expression is evaluated. The result is the value of the second or third operand/expression. The syntax is: `First operand ? second operand : third operand`	`size != 0 ? size : 0`

C OPERATORS

- The compound assignment operators consist of a binary operator and the simple assignment operator.
- They perform the operation of the binary operator on both operands and store the result of that operation into the left operand.
- The following table lists the simple and compound assignment operators and expression examples:

C OPERATORS

Simple Assignment Operator		
=	▪ The simple assignment operator has the following form: *lvalue* = *expr* ▪ The operator stores the value of the right operand *expr* in the object	`i = 5 + x;`

C OPERATORS

Bitwise (complement) NOT Operators	
~	The ~ (bitwise negation) operator yields the bitwise (one) complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand.

C OPERATORS

Bitwise Shift Operators		
<<	Left shift operator, shift their first operand left (<<) by the number of positions specified by the second operand.	`nbits << nshiftSize`
>>	Right shift operator, shift their first operand right (>>) by the number of positions specified by the second operand.	`nbits >> nshiftSize`

OPERATOR PRECEDENCE

- Consider the following arithmetic operation:
  - left to right

```
6 / 2 * 1 + 2 = 5
```

  - right to left

```
6/2 * 1 + 2 = 1
```

  - using parentheses

```
= 6 / (2 * 1) + 2
= (6 / 2) + 2
= 3 + 2
= 5
```

Inconsistent answers!

OPERATOR PRECEDENCE

- Operator precedence: a rule used to clarify unambiguously which operations (operator and operands) should be performed first in the given (mathematical) expression.
- Use precedence levels that conform to the order *commonly* used in mathematics.
- However, parentheses take the highest precedence and operation performed from the innermost to the outermost parentheses.

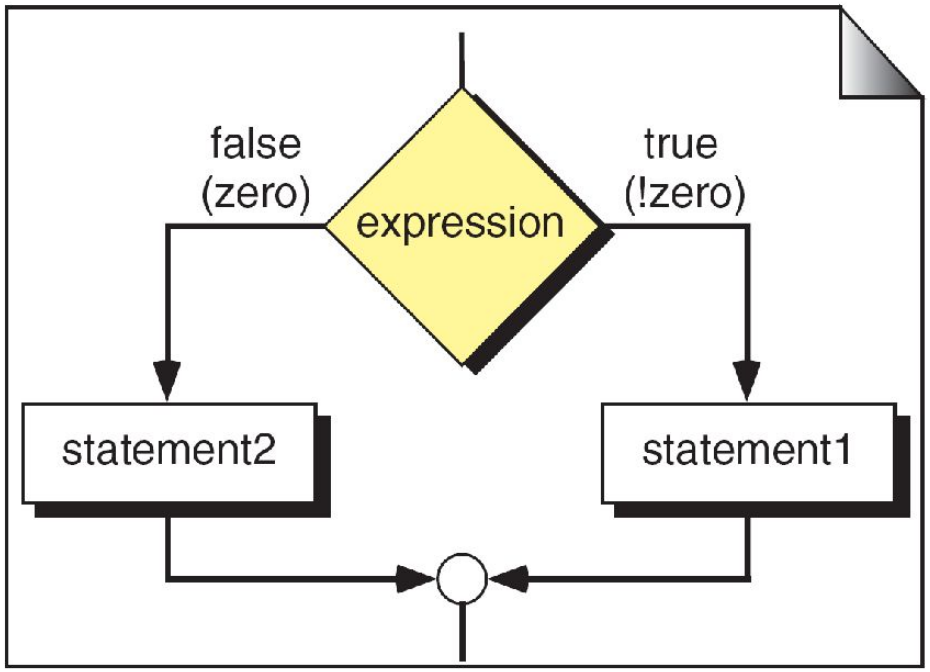
OPERATOR PRECEDENCE

- Precedence and associativity of C operators affect the grouping and evaluation of operands in expressions.
- Is meaningful only if other operators with higher or lower precedence are present.
- Expressions with higher-precedence operators are evaluated first.

OPERATOR PRECEDENCE

Precedence and Associativity of C Operators		
Symbol	**Type of Operation**	**Associativity**
`[ ] ( ) . ->` postfix ++ and postfix --	Expression	Left to right
prefix ++ and prefix -- `sizeof` **& * + - ~ !**	Unary	Right to left
typecasts	Unary	Right to left
*** / %**	Multiplicative	Left to right
+ -	Additive	Left to right
<< >>	Bitwise shift	Left to right
< > <= >=	Relational	Left to right
== !=	Equality	Left to right
&	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-expression	Right to left
= *= /= %= += -= <<= >>= &= ^= |=	Simple and compound assignment	Right to left
,	Sequential evaluation	Left to right

CONDITIONAL STATEMENTS

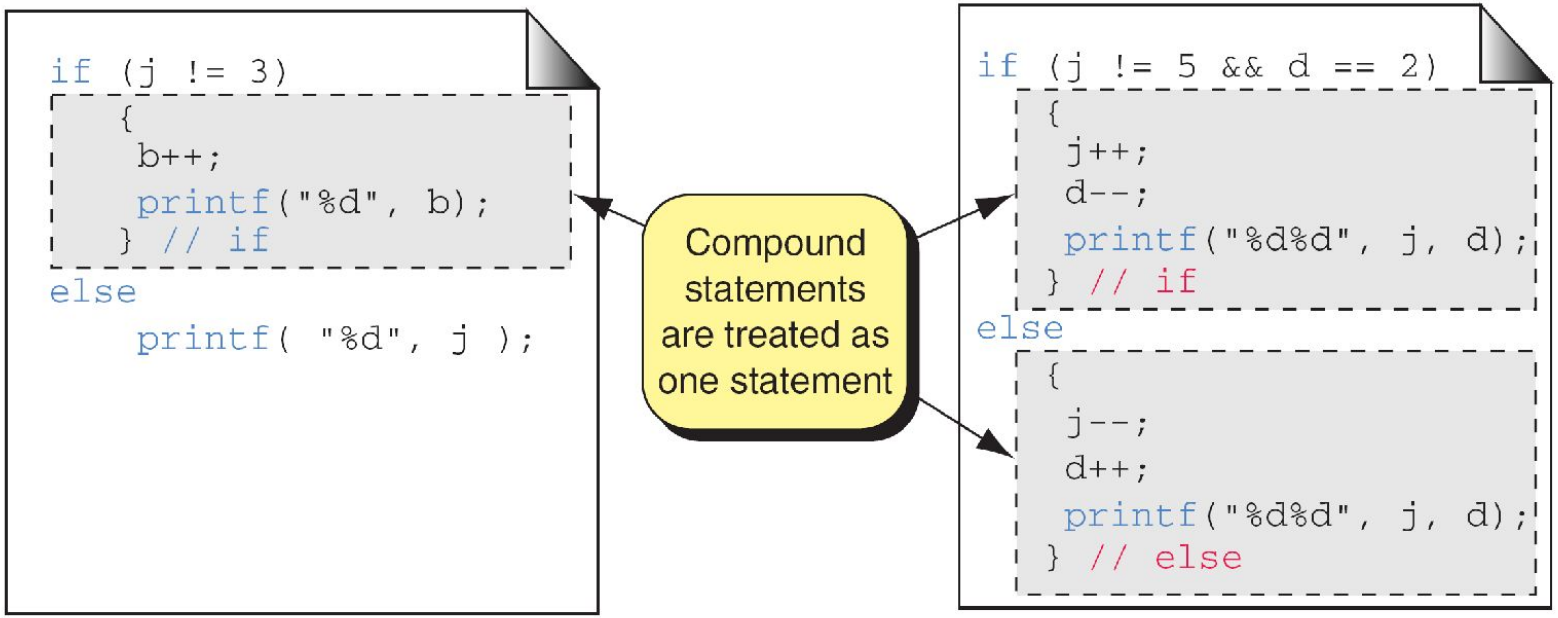


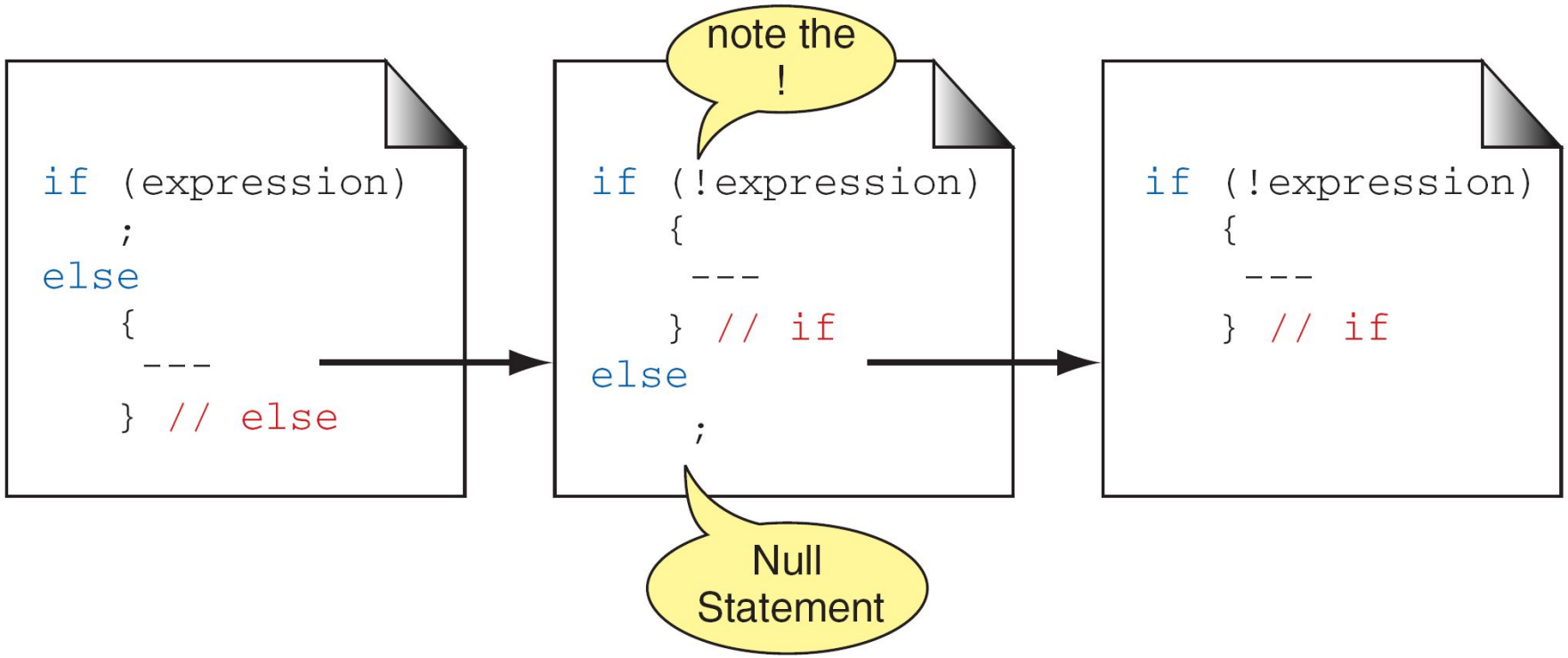
(a) Logical Flow

```
if (expression)
     statement1
else
     statement2
```

(b) Code

1. The expression must be enclosed in parentheses.
2. No semicolon (;) is needed for an *if...else* statement; statement 1 and statement 2 may have a semicolon as required by their types.
3. The expression can have a side effect.
4. Both the true and the false statements can be any statement (even another *if...else* statement) or they can be a null statement.
5. Both statement 1 and statement 2 must be one and only one statement. Remember, however, that multiple statements can be combined into a compound statement through the use of braces.
6. We can swap the position of statement 1 and statement 2 if we use the complement of the original expression.


if (j != 3)
{
b++;
printf("%d", b);
} // if
else
printf("%d", j);
Compound statements are treated as one statement
if (j != 5 && d == 2)
{
j++;
d--;
printf("%d%d", j, d);
} // if
else
{
j--;
d++;
printf("%d%d", j, d);
} // else


if (expression)
;
else
{

} // else
note the
!
if (!expression)
{

} // if
else
;
Null
Statement
if (!expression)
{

} // if