Some Boolean Identities

- A && B && C $\Leftrightarrow$ (A && B) && C $\Leftrightarrow$ A && (B && C)
- A || B || C $\Leftrightarrow$ (A || B) || C $\Leftrightarrow$ A || (B || C)
- A && B $\Leftrightarrow$ B && A
- A && A $\Leftrightarrow$ A
- A || A $\Leftrightarrow$ A
- A && TRUE $\Leftrightarrow$ A

Some Boolean Identities

- A && FALSE $\Leftrightarrow$ FALSE
- A || TRUE $\Leftrightarrow$ TRUE
- A || FALSE $\Leftrightarrow$ A
- A && (B || C) $\Leftrightarrow$ (A && B) || (A && C)
- !(!A) $\Leftrightarrow$ A
- !A || A $\Leftrightarrow$ TRUE
- !A && A $\Leftrightarrow$ FALSE

De Morgan's Complimentary

- !(A && B && C) ⇔ (!A) || (!B) || (!C)
- !(A || B || C) ⇔ (!A) && (!B) && (!C)

A || B ⇔ B || A

valid from Boolean algebra

(WARNING: may not work in C)

Storage

- C is an exception as it has no in-built data type for storing TRUE and FALSE.
- Can use integer data type as a Boolean type.
- **An integer value of 0 is considered FALSE.**
- **Any non-zero integer is considered TRUE. The internal representation of TRUE uses a value of 1**

Example

```
printf("\nInternal value of TRUE is  %d",4>3);
```

Internal value of TRUE is 1

```
printf("\nInternal value of FALSE is %d",4<3);
```

Internal value of FALSE is 0

Evaluation Logical Expressions

- In C, the evaluation of a logical expression is **terminated** as soon as its **truth value** has been **ascertained**.
- This process, known as **shortcut evaluation**,
- can occasionally lead to some **unusual side effects**

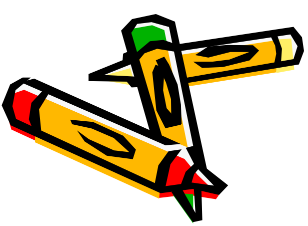
Logical Expression #1

m=3, n=2, and p=1 before expressions are evaluated.

(m++ == 3) || (n++ == 2) || (p++ == 1)

TRUE

m=4, n=2, and p=1 after this expression has been evaluated.



Logical Expression #2

m=3, n=2, and p=1 before expressions are evaluated.

```
(m++ == 5) || (n++ == 2) || ( p++ == 1)
                  TRUE
```

m=4, **n=3**, and p=1 after this expression has been evaluated.

Logical Expression #3

m=3, n=2, and p=1 before expressions are evaluated.

(m++ == 5) || (n++ == 7) || (p++ == 1)

TRUE

m=4, n=3, and **p=2** after this expression has been evaluated.

Points To Remember

- We must watch out for the side effects of short cut evaluation of logical expressions.
- Any integral value can be used to represent a logical value. A value of 0 represents FALSE and any non-zero value represents TRUE.
- The logical operators allow us to create and combine logical expressions.

Bit Wise Operators

Bitwise Operators

- & bitwise AND
- | bitwise OR
- ^ bitwise EX-OR (exclusive OR)
- ~ bitwise NEGATION (1’s complement) (unary)

Resultant for bitwise operators

a	b	a&b	a|b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Example for bitwise operators

- **A = 10110101**
- **B = 10100011**

A | B = 10110111

A & B = 10100001

A ^ B = 00010110

~ A = 01001010

Bitwise Shifting

- Bitwise left shift

“<< “

syntax :

var << nbit

number of bits shift

Variable on which bit shift is performed

Bitwise Shifting

- Bitwise right shift

“>>“

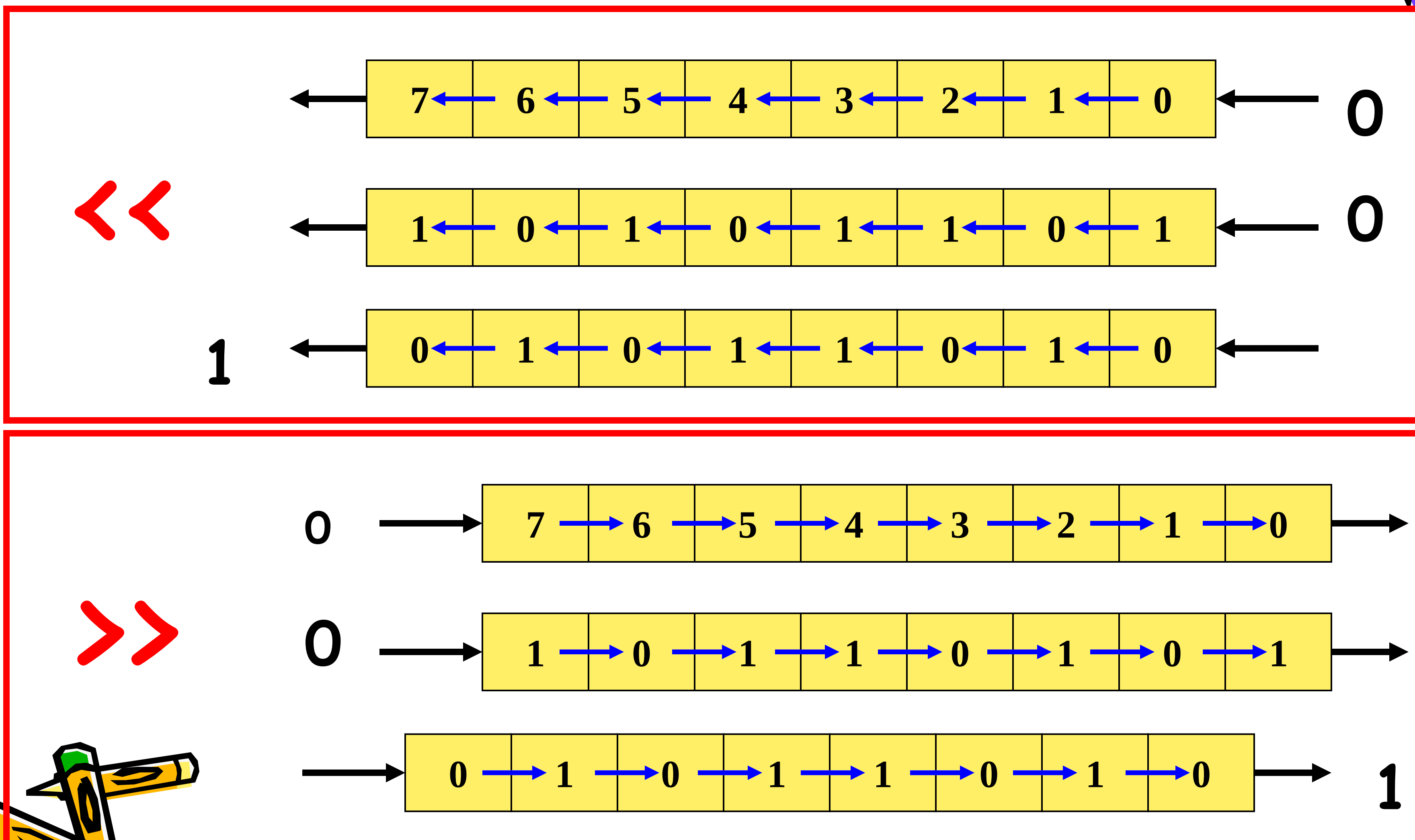
syntax :

var >> nbit

number of bits shift

Variable on which bit shift is performed

Bitwise Shifting-Example



The Bitwise Operators

- **Bit manipulation operators lend the C a great language for interfacing with hardware and digital circuits.**
- **The second use of bitwise operations is for achieving data compression, i.e., storage of more data in fewer bytes.**
- **The availability of bit level operators gives it the flexibility required for system software development.**

Points to Remember

- **The bitwise AND operator (&) works differently from the logical AND operator (&&).**
- **The bitwise OR operator (|) works differently from the logical OR operator (||).**
- **We can use the shift operators for quick multiplication and division with powers of 2.**
- **We can set, reset, and toggle individual bits of an integer value using bitwise operators**

Conditional Operator

Sometimes, we need to choose between two expressions based on the truth value of a third logical expression. The conditional operator can be used in such cases.

logexpr ? exprT : exprF

If the logical expression logexpr evaluates to TRUE, the result is exprT.
If the logical expression logexpr evaluates to FALSE, the result is exprF.

Example

m = (n==4)? 2 : 3;

Here, **m** is assigned a value of **2** if **n** equals **4**. Otherwise, it is assigned a value of **3**.

Associative

The conditional operator is **right associative** with respect to its first and third operands. Therefore,

a ? b : c ? d : e

is interpreted as

a ? b : (c ? d : e)

The Comma Operator

- The comma operator combines two expressions into a single expression. It appears as a pair of expressions separated by a comma.

left_expr, right_expr

Sequence Of Steps

- The comma operator evaluates from left to right, i.e., left_expr is evaluated first.
- All side effects of evaluating left_expr are completed.
- The resultant value of left_expr is discarded.
- right_expr is evaluated.
- The value and type of the final result is the value and type obtained by evaluating right_expr.

Example

Let us look at an example where k=4 is an int and x is a float.

$$x = k{+}{+}, k+3$$

5

x=8

Typecast Operator (Explicit Conversion-Typecasting)

- Most of the time, you do not have to worry about C's automatic conversion of data types.
- You can override C's default conversions by specifying your own temporary type change using the format:

(data type) expression

- (data type) can be any valid C data type and expression is any variable, constant or a combination of both, e.g.
- This code type casts the integer variable age into a double floating-point variable temporarily, so it can be multiplied by the double floating-point factor

```
age_factor = (double)age * factor;
```

Example

```
int k=5;
double rootval;
rootval = sqrt( (double) k);
```

The typecast operator **(double)** converts the integer value of k into the type double. Note that the data type of k remains unchanged as int

Example

```
float x = 7.8;
int krem;
krem = ( (int) x) %2;
```

we **cannot** use the **remainder operator** with a float value, we type cast the value into an int. Note that the value of x remains unchanged as 7.8 by the type cast operation. The type cast operator yields an int value of 7 which leads to a value of 1 for krem.

Other Operators

`sizeof`	sizeof operator
`*`	indirection operator
`&`	address operator
`.`	structure member operator
`->`	structure pointer operator
`()`	function call operator
`[]`	array Index operator

We Will Study These operators In Detail In Their Related Topics

Precedence & Associativity

Symbol	Operator Precedence Group	Associativity from
( )	function call	left
[ ]	array index	left
.	structure member operator	left
->	structure pointer operator	left

Precedence & Associativity

!	logical NOT operator	right
~	one's complement operator	right
++	increment operator	right
--	decrement operator	right
+	unary plus operator	right
-	unary minus operator	right
*	indirection operator	right
&	address operator	right
(type)	typecast operator	right
sizeof	sizeof operator	right

Precedence & Associativity

*	multiplication operator	left
/	division operator	left
%	remainder operator	left
+	binary addition operator	left
-	binary subtraction operator	left
<<	bitwise left shift operator	left
>>	bitwise right shift operator	left

Precedence & Associativity

<	less than operator	left
<=	less than or equal to operator	left
>	greater than operator	left
>=	greater than or equal to operator	left
==	equality operator	left
!=	inequality operator	left

Precedence & Associativity

<table>
<tr><td>&</td><td>bitwise AND operator</td><td>left</td></tr>
<tr><td>^</td><td>bitwise exclusive OR operator</td><td>left</td></tr>
<tr><td>|</td><td>bitwise inclusive OR operator</td><td>left</td></tr>
<tr><td>&&</td><td>logical AND operator</td><td>left</td></tr>
<tr><td>||</td><td>logical OR operator</td><td>left</td></tr>
<tr><td>?:</td><td>conditional operator</td><td>right</td></tr>
<tr><td>=</td><td>assignment operator</td><td>right</td></tr>
<tr><td>x =</td><td>various compound assignment operators
+= -= *= /= %= &= ^= |= <<= >>=</td><td>right</td></tr>
<tr><td>,</td><td>comma operator</td><td>left</td></tr>
</table>

Evaluation Sequence

Sequence of evaluation is well defined for

&&, ||, ? :, and ' , '

But for z = foo1(x) + foo2(x);

Which will perform first ???????

CAN'T SAY !!!!!!

SOLUTION:

```
temp = foo2(x);
z = foo1(x) + temp;
```

Points To Remember

- The evaluation sequence of operators is defined by their precedence and associativity.
- Some types of expressions might have an implementation dependent evaluation sequence. Such expressions must be avoided for improving portability of our programs.

THANK YOU