**Bitwise Operators**

Before proceeding to Bitwise operators lets us recall the logical operators

**Logical AND operators(&&) Logical OR operators(||)**

|  |  |  |
| --- | --- | --- |
| A | B | Result |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

|  |  |  |
| --- | --- | --- |
| A | B | Result |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Logical XOR operators(|) Logical Not(!)**

|  |  |  |
| --- | --- | --- |
| A | B | Result |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

!right;

right == 0;

Examples of Expressions Using the Logical Operators

| **Given the following declarations:**  **int j = 0, m = 1, n = -1;[LINEBREAK] float x = 2.5, y = 0.0;[LINEBREAK]** | | |
| --- | --- | --- |
| **Expression** | **Equivalent Expression** | **Result** |
| j && m | (j) && (m) | 0 |
| j < m && n < m | (j < m) && (n < m) | 1 |
| m + n || ! j | (m + n) || (!j) | 1 |
| x \* 5 && 5 || m / n | ((x \* 5) && 5) || (m / n) | 1 |
| j <= 10 && x >= 1 && m | ((j <= 10) && (x >= 1)) && m | 1 |
| !x || !n || m+n | ((!x) || (!n)) || (m+n) | 0 |
| x \* y < j + m || n | ((x \* y) < (j + m)) || n | 1 |
| (x > y) + !j || n++ | ((x > y) + (!j)) || (n++) | 1 |
| (j || m) + (x || ++n) | (j || m) + (x || (++n)) | 2 |

**Bitwise**

* Bitwise AND(&)
* Bitwise OR(|)
* Bitwise NOT(~)
* Bitwise XOR(^)
* Shift left(<<)
* Shift Right(>>)

&  Bitwise AND

The & operator compares two values with a bitwise AND function. You use the & operator with two operands, like this:

Evaluate a & b

Each of the operands a and b is converted to a binary number, and each of their bits is compared; that is, bit 1 of a is compared with bit 1 of b, and so on. When two corresponding bits are both 1, bitwise AND combines them to produce 1. When one or both of the corresponding bits are 0, bitwise AND combines them to produce 0.

For example, if you enter

Evaluate 25 & 77

25 and 77 are combined with a bitwise AND, and the result is 9.

|  Bitwise OR

The | operator compares two values with a bitwise OR function. You use the | operator with two operands, like this:

Evaluate a | b

Each of the operands a and b is converted to a binary number, and each of their bits   
is compared; that is, bit 1 of a is compared with bit 1 of b, and so on. When two corresponding bits are both 0, bitwise OR combines them to produce 0. When either of the corresponding bits is 1, bitwise OR combines them to produce 1.

For example, if you enter

Evaluate 25 | 77

25 and 77 are combined with a bitwise OR, and the result is 93.

~  Bitwise NOT

In a logical expression, the ~ operator computes the binary negative of a number. The binary negative is calculated by converting a character's 0 bits to 1 bits and its 1 bits to   
0 bits.

For example, to find the binary negative of 5, you could enter

Evaluate ~5

The result of this command is -6.

^  Bitwise XOR

The ^ operator compares two values with a bitwise XOR (that is, a bitwise exclusive OR) function. You use the ^ operator with two operands, like this:

Evaluate a ^ b

Each of the operands a and b is converted to a binary number, and each of their bits is compared; that is, bit 1 of a is compared with bit 1 of b, and so on. When one of the bits--but not both--is 1, bitwise XOR combines them to produce 1. When both bits are 0 or when both bits are 1, bitwise XOR combines them to produce 0.

For example, if you enter

Evaluate 25 ^ 77

25 and 77 are combined with a bitwise XOR, and the result is 84.

<<  Shift left

The << operator performs a shift left function on a number. A shift left function shifts a value's bits *n* characters to the left. The bits that are shifted out through the high-order bit are lost. For example, if you enter

Evaluate 3 << 1

the bit pattern in the number 3 is shifted left by one place. In other words, the bit pattern

... 0000 0011  #the value 3

becomes

... 0000 0110  #the value 6

shifting left one place actually has the effect of multiplying a value by 2.

>>  Shift right

The >> operator performs a shift right function on a number. A shift right function shifts a value's bits *n* characters to the right. The bits moved out of the low-order bit are lost, and the bits shifted in to the high-order bits on the left have a value of 0. For example, if you enter

Evaluate 3 >> 1

the bit pattern

... 0000 0011

becomes

... 0000 0001

which has a value of 1. The right shift is a *logical* right shift, because bits with a value of 0 are moved into the high-order bits.

**Important programs using Bitwise**

**To set a bit in a number To set a bit in a number**

int num,pos;

int ch;

printf("\nPlease Enter a Number:\n");

scanf("%d",&num);

printf("\nEnter the position you want to set a bit\n");

scanf("%d",&pos);

num = num | (1 << pos);

printf("\nAfter setting at position %d,the number is %d\n",pos,num);

int num,pos;

int ch;

printf("\nPlease Enter a Number:\n");

scanf("%d",&num);

printf("\nEnter the position you want to clear a bit\n");

scanf("%d",&pos);

num = num & ~(1 << pos);

printf("\nAfter clearing at position %d,the number is %d\n",pos,num);

**Check Even/odd**

int n = 0;

printf("\nEnter the number:\n");

scanf("%d",&n);

n & 1?printf("\nNumber is odd\n"):printf("\nNumber is Even\n");

**Count Number of bits set in a Number Power of two**

main()

{

int num;

int cnt = 0; //Holds the count

printf("\nPlease Enter the desired Number:\n");

scanf("%d",&num);

while(num)

{

cnt++;

num = num & (num - 1); // This clears the least significant bit set.

}

printf("\n Number of bits set in the above number is %d\n",cnt);

}

int n = 0;

printf("\nPlease Enter the Number to Check:\n");

scanf("%d",&n);

if(n & (n-1))

printf("\nSorry Boss This is not Power of Two\n");

else

printf("\nYess Its Power of Two\n");

**String Reverse**

main()

{

char i,j;

char str[10];

printf("\nPlease Enter a String:\n");

scanf("%s",str);

for(i=0,j=(strlen(str)-1);j>i;i++,j--)

{

str[i] = str[i] ^ str[j];

str[j] = str[i] ^ str[j];

str[i] = str[i] ^ str[j];

}

printf("\nstring is %s\n",str);

}

**Swapping of variables**

main()

{

int i,k;

printf("\nPlease Enter the Numbers You want to Exchange\n");

scanf("%d %d",&i,&k);

printf("\nValue of i=%d k=%d before swapping\n", i, k);

i = i ^ k;

k = i ^ k;

i = k ^ i;

printf("\nValue of i=%d k=%d after swapping\n", i, k);

}

**Little to Big Endianess**

int myreversefunc(int);

main()

{

int num = 12;

myreversefunc(num);

}

int myreversefunc(int num)

{

int byte0, byte1, byte2, byte3,i;

byte0 = (num & 0x000000FF) >> 0;

byte1 = (num & 0x0000FF00) >> 8 ;

byte2 = (num & 0x00FF0000) >> 16 ;

byte3 = (num & 0xFF000000) >> 24 ;

printf("\nbyte0 = %d byte1 = %d byte2 = %d byte3 = %d\n",byte0,byte1,byte2,byte3);

printf("\ni = %d\n",i);

}

**Little to Big Endianess**

main()

{

int word = 1;

char \*byte = (char \*)&word;

if(byte[0] == 1)

printf("\nLittile Endian\n");

else

printf("\nBig Endian\n");

}

**Add without using + operator Program for Modifying/Retrieving a value at particular Position**

main()

{

int x,y,z;

printf("\nEnter Values to be added:\n");

scanf("%d %d",&x,&y);

while(y)

{

z = x^y;

y = (x & y) << 1;

x = z;

}

printf("\n%d\n",z);

}

Rever

main()

{

int num = 0x123456;

num = num & 0x103456;

printf("%x\n",num);

num = num | (9 << 14);

printf("%x\n",num);

}

**Reverse Binary Bits Swapping Nibbles of a number**

unsigned int swap\_nibbles(unsigned int);

int main(void)

{

unsigned int ch = 1234;

printf("\nThe Exchanged value is %x\n",swap\_nibbles(ch));

return 0;

}

unsigned int swap\_nibbles(unsigned int c)

{

unsigned int temp1, temp2;

temp1 = c & 0x000F;

temp2 = c & 0xF000;

printf("\ntemp1 = %d temp2 = %d\n",temp1,temp2);

temp1=temp1 << 4;

temp2=temp2 >> 4;

return(temp2|temp1); //adding the bits

}

main()

{

unsigned char num = 0x1;

unsigned int temp = num;

unsigned char temp = num;

int i;

for (i = (sizeof(num)\*8-1); i; i--)

{

temp = temp | (num & 1);

temp <<= 1;

num >>= 1;

}

temp = temp | (num & 1);

printf("\ntemp = %x\n",temp);

}

**Multiplication of No.s**

**Notes:**

* Bitwise AND operator is suitable for checking whether a particular bit is ON or OFF
* Bitwise AND operator (&), one's complement operator(~)is suitable for turning OFF a particular bit in a number

int a,b,result;

printf("nEnter the numbers to be multiplied :");

scanf("%d%d",&a,&b);

result=0;

while(b != 0)

{

if (b&01)

result=result+a;

a<<=1;

b>>=1;

}

printf("nResult:%d",result);