CSE 230 Intermediate Programming in C and C++

Bitwise Operators and Enumeration Types

Fall 2017

Stony Brook University

Instructor: **Shebuti Rayana**

http://www3.cs.stonybrook.edu/~cse230/

Overview

- Bitwise Operators
  - The bitwise operators act on integral expressions represented as binary digits.
  - Expressions with bitwise operators are explicitly system-dependent
  - Useful in packing and unpacking data
- Enumeration Types
  - User defined types
  - Allow the programmer to name a finite set together with its elements, which are called enumerators

Bitwise Operators

Types of Bitwise Operators		
Logical Operators	(unary) bitwise complement	~
	Bitwise AND	&
	Bitwise inclusive OR	|
	Bitwise exclusive OR	^
Shift Operators	Left shift	<<
	Right shift	>>

Precedence and Associativity

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td>() [] ++(postfix) --(postfix)</td><td>Left to right</td></tr>
<tr><td>++ -- (prefix) ! ~ sizeof() + - (unary) &(address) *(pointer)</td><td>Right to left</td></tr>
<tr><td>* / %</td><td>Left to right</td></tr>
<tr><td>+ -</td><td>Left to right</td></tr>
<tr><td><< >></td><td>Left to right</td></tr>
<tr><td>< <= > >=</td><td>Left to right</td></tr>
<tr><td>== !=</td><td>Left to right</td></tr>
<tr><td>&</td><td>Left to right</td></tr>
<tr><td>^</td><td>Left to right</td></tr>
<tr><td>|</td><td>Left to right</td></tr>
<tr><td>&&</td><td>Left to right</td></tr>
<tr><td>||</td><td>Left to right</td></tr>
<tr><td>?:</td><td>Right to left</td></tr>
<tr><td>= += -= *= /= %= <<= >>= &= ^= |=</td><td>Right to left</td></tr>
<tr><td>, (comma)</td><td>Left to right</td></tr>
</table>

Bitwise Complement

- ~ is called one's complement
  - Inverts all the bits, (0's become 1's and 1's become 0's)
  - Example: `int a = 70707;` in binary 00000000 00000001 00010100 00110011
  - `~a` is one's complement for a 11111111 11111110 11101011 11001100
  - So `~a` becomes `-70708`

Two's Complement

- The two's complement representation of a nonnegative integer `n` is the bit string obtained by writing `n` in base 2.
- If we take the bitwise complement of the bit string and add 1 to it, we obtain the two's complement representation of `-n`

Value of **n**	Binary Representation	Bitwise Complement	Two's Complement Representation of -n	Value of **-n**
7	00000000 00000111	11111111 11111000	11111111 11111001	-7
8	00000000 00001000	11111111 11110111	11111111 11111000	-8
9	00000000 00001001	11111111 11110110	11111111 11110111	-9
-7	11111111 11111001	00000000 00000110	00000000 00000111	7

*Two lower order bytes in 4 bytes machine

*A machine which uses this representation is called a two's complement machine

Two’s Complement (cont.)

- 0 : all bits off, -1: all bits on
- if a binary string is added to its bitwise complement the result has all bits on, which is the two's complement representation of -1.
- Negative numbers are characterized by having the high bit on.
- On a two's complement machine, the hardware that does addition and bitwise complementation can be used to implement subtraction. The operation a - b is the same as a + (-b), and -b is obtained by taking the bitwise complement of b and adding 1.

Bitwise Binary Logical Operators

Single bit Operations				
a	b	a&b	a^b	a|b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

*Operated on bit position by bit position

Examples: Bitwise Operators

Declaration and Initialization		
`int a = 33333; int b = -77777;`		
Expression	Representation	Value
`a`	00000000 00000000 10000010 00110101	33333
`b`	11111111 11111110 11010000 00101111	-77777
`a&b`	00000000 00000000 10000000 00100101	32805
`a^b`	11111111 11111110 01010010 00011010	-110054
`a|b`	11111111 11111110 11010010 00111111	-77249
`~(a|b)`	00000000 00000001 00101101 11000000	77248
`(~a&~b)`	00000000 00000001 00101101 11000000	77248

De Morgan's Law: `~(a|b) = (~a&~b), ~(a&b) = (~a|~b)`

Left Shift Operator

- The two operands of a left shift operator must be integral expressions.
- Example: `expr1 << expr2`, the bit representation of `expr1` is shifted to the left by `expr2` positions.
  - On the low-order end, 0's are shifted in.
  - Both the operands are promoted to integral types before shifting
  - The resulting type is the type of left operand

Example: Left shift

Declaration and Initialization		
Char c = ‘Z’;		
Expression	Representation	Action
c	00000000 00000000 00000000 01011010	unshifted
c << 1	00000000 00000000 00000000 10110100	Left shifted 1
c << 4	00000000 00000000 00000101 10100000	Left shifted 4
c << 31	00000000 00000000 00000000 00000000	Left shifted 31

Right Shift Operator

- The right shift operator is not similar to the left shift operator
- For unsigned expressions shifted positions are filled with 0’s
- But for signed expressions: (i) some machines shift in 0’s, and (ii) some shift in the sign bit (left most bit or high order bit)
  - Sign bit is 0 for nonnegative integers and 1 for negative integers

Example: Right Shift

Declaration and Initialization		

`int a = 1 << 31; // shift 1 to the high bit`
`unsigned b = 1 << 31;`		

Expression	Representation	Action
a	10000000 00000000 00000000 00000000	unshifted
a >> 3	11110000 00000000 00000000 00000000	Right shifted 3
b	10000000 00000000 00000000 00000000	unshifted
b >> 3	00010000 00000000 00000000 00000000	Right shifted 3

If the right operand of a shift operator is negative or has a value that equals or exceeds the number of bits used to represent the left operand, then the behavior is undefined.

Precedence and Associativity

Declaration and Assignments			
`unsigned a = 1, b = 2;`			
Expression	Equivalent Expression	Representation	Value
a << b >> 1	(a << b) >> 1	00000000 00000010	2
a << 1 + 2 << 3	(a << (1 + 2)) << 3	00000000 01000000	64
a+b << 12 * a >> b	((a+b) << (12 * a)) >> b	00001100 00000000	3072

*two low order bytes are shown only
*in C++, the two shift operators are overloaded and used for input/output. Overload -ing in C++ is a method of giving existing operators and functions additional meanings.

Masks

- A mask is a constant or variable, that is used to extract desired bits from another variable or expression.
- if we wish to find the value of a particular bit in an expression, we can use a mask that is 1 in that position and 0 elsewhere.
- Example: 00000000 00000000 00000000 00000001

```
int i, mask = 1;
for(i=0; i<10; i++);
printf("%d ", i & mask);
```

- This code prints the right most bit of every number in the range [0,9]

More Example: Mask

- 1 << 2, can be used as a mask for third bit
- `(v & (1 << 2)) ? 1 : 0`
- Another mask is 255 = $2^8 - 1$,

  00000000 00000000 00000000 11111111

  – v & 255 will give only the low order byte, as such, 255 is called mask for low-order byte

Printing an Integer Bitwise

```
#include <limits.h>
void bit_print(int a){
    int i;
    int n = sizeof(int) * CHAR_BIT;
    int mask = 1 << (n - 1); // mask 100…0

    for(i=1; i < n; i++){
        putchar(((a & mask) == 0) ? '0':'1');
        a <<= 1;
        if(i % CHARBIT == 0 && i < n)
            putchar(' ');
    }
}
```

Packing

- Bitwise expressions help in data compression
  - Saving both time and space
- Example: pack 4 `char` into an `int`

```
#include <limits.h>
int pack(char a, char b, char c, char d){
    int p = a;
    p = (p << CHAR_BIT) | b;
    p = (p << CHAR_BIT) | c;
    p = (p << CHAR_BIT) | d;
    return p;
}
```

Packing (cont.)

```
printf("abcd == ");
bit_print(pack('a', 'b', 'c', 'd'));
putchar(' \n');
```

- Output:

  97 98

  ```
  abed = 01100001 01100010
  01100011 01100100
  ```

  99 100

Unpacking

```
#include <limits.h>
int unpack(int p, int k){ //k=0,1,2,3
    int n = k*CHAR_BIT; //n=0,8,16,24
    unsigned mask = 255;
    mask <= n;
    return ((p & mask) >> n);
}
```

Unpacking (cont.)

Expression	Binary Representation	Value
`p`	11111111 11001001 01100000 10010111	-3579753
`mask`	00000000 11111111 00000000 00000000	16711680
`p & mask`	00000000 11001001 00000000 00000000	13172736
`(p & mask) >> n`	00000000 00000000 00000000 11001001	201

Enumeration Types

- User defined types
- Provides a means of naming a finite set, and declaring identifiers as elements of the set.
- Keyword: `enum`
- Example:

```
enum day {sun, mon, tue, wed, thu,
fri, sat}
```

- `day` is a user defined enumeration type
- The identifiers `sun, …, sat` are constants of type `int`
- By default, the first one is 0, and each succeeding one has the next integer value.

Enumeration Types (cont.)

- This declaration is an example of a type specifier, which we also think of as a template.
- Declaration of a variable of type `enum`:
  `enum day d1, d2;`
  - `d1` and `d2` can only take values from the set `day`
- Initialization: `d1 = fri;`
- Condition check:
  `if(d1 == d2){/*do something*/}`
  - `enum day` is a type, `enum` by itself is not a type

Enumeration Types (cont.)

- The enumerators can be initialized
- Variables can be declared along with the template
- `enum suit {clubs = 1, diamonds, hearts, spades} a, b, c;`
  - As clubs is initialized to 1, diamonds, hearts, and spades have the values 2,3, and 4, respectively.
- `enum fruit {apple = 7, pear, orange = 3, lemon} frt;`
  - As apple is initialized to 7, pear has value 8. Similarly, because orange has value 3, lemon has value 4.
- Valid types:

```
enum veg {beet = 17, carrot = 17, corn = 17}  vege1, vege2;
enum {fir, pine} tree;
```

Example: enum

```
/* compute the next day */
enum day {sun, mon, tue, wed, thu,
fri, sat}
typedef enum day day;
day find_next_day(day d){
   if((int) d >= 0 && (int) d < 7)
       return ((day)(((int)d+1)));
}
```