Lecture 19: Low-level Programming

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro.
Some content taken from K.N. King's slides based on course text book]

Agenda

- Bitwise operators
- Bitwise operator precedence
- Accessing individual bits

Readings

- Textbook Chapter 20.1

Why Low-level Programming?

- Throughout the course, we have discussed C's high-level, machine-independent features

- However, there are some kinds of programs that need to perform operations at the bit level:
  - Systems programs (including compilers and operating systems)
  - Encryption programs
  - Graphics programs
  - Programs for which fast execution and/or efficient use of space is critical

Hexadecimal to Binary Relationship

Decimal	Binary	Hexadecimal
0	0000 0000	0x0
1	0000 0001	0x1
2	0000 0010	0x2
...	...	...
12	0000 1100	0xC
...	...	...
15	0000 1111	0xF
16	0001 0000	0x10
17	0001 0001	0x11

Every 4 bits in binary correspond to a “digit/letter” in Hex

Bitwise Operators

- C provides six bitwise operators, which operate on integer data at the bit level
  - Shift right (>>) and Shift left (<<)
  - bitwise complement (~)
  - bitwise and (&)
  - bitwise inclusive or (|)
  - bitwise exclusive or (^)

demo: binaryop.c

Bitwise Shift Operators

- They shift bits in an integer to the left or to the right

- The operands for << and >> may be of any integer type (including char)

- The integer promotions are performed on both operands, and the result has the type of the left operand after promotion

Examples of Shift Operators

```
unsigned short i, j; //for simplicity, using 16 bits)
i =13; /* (i is now 0000 0000 0000 1101) */
j = i << 2; /* j is now 0000 0000 0011 0100 */
j = i >> 2; /* j is now 0000 0000 0000 0011 */
```

Examples of Shift Operators

```
unsigned short i, j; //for simplicity, using 16 bits)
i =13; /* (i is now 0000 0000 0000 1101) */
j = i << 2; /* j is now 0000 0000 0011 0100 */
j = i >> 2; /* j is now 0000 0000 0000 0011 */
```

- Important Notes:

Examples of Shift Operators

```
unsigned short i, j; //for simplicity, using 16 bits)
i =13; /* (i is now 0000 0000 0000 1101) */
j = i << 2; /* j is now 0000 0000 0011 0100 */
j = i >> 2; /* j is now 0000 0000 0000 0011 */
```

- Important Notes:
  - For each bit “shifted off” one end of a number, a 0 enters on the other end.

Examples of Shift Operators

```
unsigned short i, j; //for simplicity, using 16 bits)
i =13; /* (i is now 0000 0000 0000 1101) */
j = i << 2; /* j is now 0000 0000 0011 0100 */
j = i >> 2; /* j is now 0000 0000 0000 0011 */
```

- Important Notes:
  - For each bit “shifted off” one end of a number, a 0 enters on the other end.
  - If the number is signed, the implementation of >> may vary: some compilers keep the sign by adding 1s when the number is negative while others add 0.

Examples of Shift Operators

```
unsigned short i, j; //for simplicity, using 16 bits)
i =13; /* (i is now 0000 0000 0000 1101) */
j = i << 2; /* j is now 0000 0000 0011 0100 */
j = i >> 2; /* j is now 0000 0000 0000 0011 */
```

- Important Notes:
  - For each bit “shifted off” one end of a number, a 0 enters on the other end.
  - If the number is signed, the implementation of >> may vary: some compilers keep the sign by adding 1s when the number is negative while others add 0.
  - **For portability, use shift operators only on unsigned numbers**

Examples of Shift Operators

```
unsigned short i, j; //for simplicity, using 16 bits)
i =13; /* (i is now 0000 0000 0000 1101) */
j = i << 2; /* j is now 0000 0000 0011 0100 */
j = i >> 2; /* j is now 0000 0000 0000 0011 */
```

- Important Notes:
  - For each bit “shifted off” one end of a number, a 0 enters on the other end.
  - If the number is signed, the implementation of >> may vary: some compilers keep the sign by adding 1s when the number is negative while others add 0.
  - **For portability, use shift operators only on unsigned numbers**
  - Neither operator modifies its operands. To modify a variable by shifting its bits, you can use <<= or >>=

What do the Shift Operators Really Mean?

- Shift left by one is equivalent to multiplication by 2

- Shift right by one is equivalent to division by 2

Operator Precedence

- Bitwise shift operators have lower precedence than arithmetic operators so `i << 2 + 1` is equivalent to `i << (2 + 1)`

Bitwise Complement ~

- It is a unary operator
- Replaces 0's with 1's and 1's with 0's

```
unsigned short i, j; //for simplicity, using 16 bits)
i = 13; /* (i is now 0000 0000 0000 1101) */
j = ~i; /* (j is now 1111 1111 1111 0010) */
printf("complement of I=0x%x\n”, j); // prints 0xfff2
```

- Useful when we want to create portable integers (where we may not necessarily know the size of the int on that machine). For example, to get an integer whose bits are all 1s: ~0. If we want an integer whose bits are 1 except for the last five, we can write ~0x1f

Bitwise And Operator (&)

- performs a boolean-and operation on each pair of bits

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

```
unsigned short i, j; //for simplicity, using 16 bits)
i =13; /* (i is now 0000 0000 0000 1101) */
j = 129; /* (j is now 0000 0000 1000 0001) */
printf("i & j = 0x%x”, i & j); // prints 0x1
```

Bitwise Or

- Performs a Boolean-or on each pair of bits

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

```
unsigned short i, j; //for simplicity, using 16 bits)
i = 13; /* (i is now 0000 0000 0000 1101) */
j = 129; /* (j is now 0000 0000 1000 0001) */
printf("i & j = 0x%x", i | j); // prints 0x8d
```

Bitwise xor (^)

- Performs a Boolean-xor on each pair of bits

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

```
unsigned short i, j, and; //for simplicity, using 16 bits)
i =13; /* (i is now 0000 0000 0000 1101) */
j = 129; /* (j is now 0000 0000 1000 0001) */
printf("i & j = 0x%x”, i | j); // prints 0x8c
```

Binary Operator Precedence

Highest ~

&

^

Lowest |

precedence of all bitwise operators is lower than relational and equality operators

Using Bitwise Operators to Access Bits

- Setting a bit
- Cleaning a bit
- Testing a bit

Setting a Bit

Setting a Bit

- You may want to set a particular bit in an integer (e.g., to indicate that this bit is “true” where each bit represents some error code)

Setting a Bit

- You may want to set a particular bit in an integer (e.g., to indicate that this bit is “true” where each bit represents some error code)

- The easiest way to set bit `i` in integer `j` is to *or* the value of `j` with a constant that contains only 1 in position `i`

Setting a Bit

- You may want to set a particular bit in an integer (e.g., to indicate that this bit is “true” where each bit represents some error code)

- The easiest way to set bit `i` in integer `j` is to *or* the value of `j` with a constant that contains only 1 in position `i`

- Assume we want to set bit 4 (i.e., the bit at index 4) in a 16-bit integer i = 0x0001

  - We need to create a *mask* j that has a 1 bit in index 4 (0 based): 0x0010

  - If we do `i |= j;` then i becomes 0x0011 and we know for sure that the bit at index 4 is 1

Creating Masks

- To create a mask that contains 1 in position j, we can just shift the integer 1 by j positions to the right:

```
1 << j; //shifts the 1 by j positions
i |= 1 << j; //this sets bit j in i to 1
```

- There are idioms that allow you to create portable masks. Suppose I want to create a mask with the high bit set (i.e., most significant bit):

```
~((~0U >> 1))
```

Clearing a bit

- If we want to clear a bit (i.e., set it to 0), then we can “and” it with a number that has 1’s in all bits except in the bit we want to clear

```
i &= ~(1 << j); //this sets clears bit j in i
```

Testing a bit

- If we want to test that a given bit is set (i.e., 1), we can “and” the number with a mask that has 1 in the bit we want to test and 0’s everywhere else

```
if ( i & 0x0010 ) … //tests if bit 4 is set
if (i & 1 << j ) …  //tests if bit j is set
```

Using Macros to Define Masks

- Suppose we are using a given integer to denote colors where bits 0, 1, and 2 correspond to colors blue, green, and red respectively (i.e., RGB)

```
#define BLUE 1 //1 means that bit at position 0 is set
#define GREEN 2 //2 means that bit at position 1 is set
#define RED 4 //4 means that bit at position 2 is set

i |= BLUE; // sets the BLUE bit
i &= ~BLUE; //clears the BLUE bit
if ( i & BLUE )  //tests if BLUE bit is set

i | = BLUE | GREEN; //sets BLUE and GREEN bits
i &= ~(BLUE | GREEN); //clears BLUE and GREEN bits
if (i & (BLUE | GREEN) …  //tests if BLUE or GREEN bits are set
```

Using Bitwise Operators to Access Bit-Fields

- Bit-fields refer to a group of several consecutive bits

Modifying a Bit-field

- Assume we want to store 101 in bits 4-6 of i:
  - First, we need to clear the bits we want to manipulate using and

```
i = i & ~0x0070; //the mask has bits 000 in bits 4-6
```

  - Then, we can set the bits we want using or

```
i |= 0x0050; //the mask has bits 101 in bits 4-6
```

- We can combine both

```
i = i & ~0x0070 | 0x0050;
```

- To generalize, assume j has the value you want to set in bits 4-6 of i:

```
i = (i & ~0x0070) | (j << 4);
```

Retrieving a Bit-field

- Retrieving a bit-field from the right end

```
j = i & 0x0007; // retrieves bits 0-2
```

- To retrieve a bit-field from somewhere else, first shift the bit-field to the right end before extracting it

```
j = (i >> 4) & 0x0007; // retrieves bits 4-6

j = (i >> 6) & 0x000f; // retrieves bits 6-9
```

XOR Encryption

- You can use exclusive-or to encrypt data
- You can xor each character with a secret key. Using xor allows you to apply the same algorithm for encryption and decryption

demo: xor.c