Chapter 20

Low-Level Programming

Introduction

- Previous chapters have described C's high-level, machine-independent features.
- However, some kinds of programs 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

Bitwise Operators

- C provides six ***bitwise operators,*** which operate on integer data at the bit level.
- Two of these operators perform shift operations.
- The other four perform bitwise complement, bitwise *and*, bitwise exclusive *or*, and bitwise inclusive *or* operations.

Bitwise Shift Operators

- The bitwise shift operators shift the bits in an integer to the left or right:
  - << left shift
  - >> right shift
- The operands for << and >> may be of any integer type (including `char`).
- The integer promotions are performed on both operands; the result has the type of the left operand after promotion.

Bitwise Shift Operators

- The value of `i << j` is the result when the bits in `i` are shifted left by `j` places.
  - For each bit that is "shifted off" the left end of `i`, a zero bit enters at the right.
- The value of `i >> j` is the result when `i` is shifted right by `j` places.
  - If `i` is of an unsigned type or if the value of `i` is nonnegative, zeros are added at the left as needed.
  - If `i` is negative, the result is implementation-defined. e.g. some implementations add zeros at the left end while others preserve the sign bit by adding ones.

Bitwise Shift Operators

- Examples illustrating the effect of applying the shift operators to the number 13:

```
unsigned short i, j;

i = 13;
  /* i is now 13 (binary 0000000000001101) */
j = i << 2;
  /* j is now 52 (binary 0000000000110100) */
j = i >> 2;
  /* j is now  3 (binary 0000000000000011) */
```

Bitwise Shift Operators

- To modify a variable by shifting its bits, use the compound assignment operators <<= and >>=:

```
i = 13;
  /* i is now 13 (binary 0000000000001101) */
i <<= 2;
  /* i is now 52 (binary 0000000000110100) */
i >>= 2;
  /* i is now 13 (binary 0000000000001101) */
```


Bitwise Shift Operators

- The bitwise shift operators have lower precedence than the arithmetic operators, which can cause surprises:

  `i << 2 + 1` means `i << (2 + 1)`, not `(i << 2) + 1`

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- There are four additional bitwise operators:
  - `~` bitwise complement
  - `&` bitwise *and*
  - `^` bitwise exclusive *or*
  - `|` bitwise inclusive *or*
- The `~` operator is unary; the integer promotions are performed on its operand.
- The other operators are binary; the usual arithmetic conversions are performed on their operands.

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- The `~`, `&`, `^`, and `|` operators perform Boolean operations on all bits in their operands.
- The `^` operator produces 0 whenever both operands have a 1 bit, whereas `|` produces 1.

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- Examples of the `~`, `&`, `^`, and `|` operators:

```
unsigned short i, j, k;
i = 21;
  /* i is now    21 (binary 0000000000010101) */
j = 56;
  /* j is now    56 (binary 0000000000111000) */
k = ~i;
  /* k is now 65514 (binary 1111111111101010) */
k = i & j;
  /* k is now    16 (binary 0000000000010000) */
k = i ^ j;
  /* k is now    45 (binary 0000000000101101) */
k = i | j;
  /* k is now    61 (binary 0000000000111101) */
```


Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- The `~` operator can be used to help make low-level programs more portable.
  - An integer whose bits are all 1: `~0`
  - An integer whose bits are all 1 except for the last five: `~0x1f`

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- Each of the `~`, `&`, `^`, and `|` operators has a different precedence:

  Highest: `~`
  `&`
  `^`
  Lowest: `|`

- Examples:

  `i & ~j | k` means `(i & (~j)) | k`
  `i ^ j & ~k` means `i ^ (j & (~k))`

- Using parentheses helps avoid confusion.

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- The compound assignment operators `&=`, `^=`, and `|=` correspond to the bitwise operators `&`, `^`, and `|`:

```
i = 21;
  /* i is now 21 (binary 0000000000010101) */
j = 56;
  /* j is now 56 (binary 0000000000111000) */
i &= j;
  /* i is now 16 (binary 0000000000010000) */
i ^= j;
  /* i is now 40 (binary 0000000000101000) */
i |= j;
  /* i is now 56 (binary 0000000000111000) */
```


Using the Bitwise Operators to Access Bits

- The bitwise operators can be used to extract or modify data stored in a small number of bits.
- Common single-bit operations:
  - Setting a bit
  - Clearing a bit
  - Testing a bit
- Assumptions:
  - `i` is a 16-bit `unsigned short` variable.
  - The leftmost—or ***most significant***—bit is numbered 15 and the least significant is numbered 0.

Using the Bitwise Operators to Access Bits

- ***Setting a bit.*** The easiest way to set bit 4 of `i` is to *or* the value of `i` with the constant `0x0010`:

```
i = 0x0000;
  /* i is now 0000000000000000 */
i |= 0x0010;
  /* i is now 0000000000010000 */
```

- If the position of the bit is stored in the variable `j`, a shift operator can be used to create the mask:

```
i |= 1 << j;        /* sets bit j */
```

- Example: If `j` has the value 3, then `1 << j` is `0x0008`.

Using the Bitwise Operators to Access Bits

- ***Clearing a bit.*** Clearing bit 4 of `i` requires a mask with a 0 bit in position 4 and 1 bits everywhere else:

```
i = 0x00ff;
  /* i is now 0000000011111111 */
i &= ~0x0010;
  /* i is now 0000000011101111 */
```

- A statement that clears a bit whose position is stored in a variable:

```
i &= ~(1 << j);      /* clears bit j */
```


Using the Bitwise Operators to Access Bits

- ***Testing a bit.*** An `if` statement that tests whether bit 4 of `i` is set:

```
if (i & 0x0010) …   /* tests bit 4 */
```

- A statement that tests whether bit `j` is set:

```
if (i & 1 << j) …   /* tests bit j */
```

Using the Bitwise Operators to Access Bits

- Working with bits is easier if they are given names.
- Suppose that bits 0, 1, and 2 of a number correspond to the colors blue, green, and red, respectively.
- Names that represent the three bit positions:

```
#define BLUE  1
#define GREEN 2
#define RED   4
```


Using the Bitwise Operators to Access Bits

- Examples of setting, clearing, and testing the `BLUE` bit:

```
i |= 1 << BLUE;      /* sets BLUE bit   */
i &= ~(1 << BLUE);   /* clears BLUE bit */
if (i & 1 << BLUE) …/* tests BLUE bit  */
```


Using the Bitwise Operators to Access Bits

- It's also easy to set, clear, or test several bits at time:

```
i |= (1 << BLUE) | (1 << GREEN);
  /* sets BLUE and GREEN bits   */

i &= ~(1 << BLUE | 1 << GREEN);
  /* clears BLUE and GREEN bits */

if (i & (1 << BLUE | 1 << GREEN)) …
  /* tests BLUE and GREEN bits  */
```

- The `if` statement tests whether either the `BLUE` bit or the `GREEN` bit is set.

Using the Bitwise Operators to Access Bit-Fields

- Dealing with a group of several consecutive bits (a ***bit-field***) is slightly more complicated than working with single bits.
- Common bit-field operations:
  - Modifying a bit-field
  - Retrieving a bit-field

Using the Bitwise Operators to Access Bit-Fields

- ***Modifying a bit-field.*** Modifying a bit-field requires two operations:
  - A bitwise *and* (to clear the bit-field)
  - A bitwise *or* (to store new bits in the bit-field)
- Example:

```
i = i & ~0x0070 | 0x0050;
  /* stores 101 in bits 4-6 */
```

- The `&` operator clears bits 4–6 of `i`; the `|` operator then sets bits 6 and 4.

Using the Bitwise Operators to Access Bit-Fields

- To generalize the example, assume that `j` contains the value to be stored in bits 4–6 of `i`.
- `j` will need to be shifted into position before the bitwise *or* is performed:

```
i = (i & ~0x0070) | (j << 4);
  /* stores j in bits 4-6 */
```

- The `|` operator has lower precedence than `&` and <<, so the parentheses can be dropped:

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

Using the Bitwise Operators to Access Bit-Fields

- ***Retrieving a bit-field.*** Fetching a bit-field at the right end of a number (in the least significant bits) is easy:

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

- If the bit-field isn't at the right end of `i`, we can first shift the bit-field to the end before extracting the field using the `&` operator:

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


Program: XOR Encryption

- One of the simplest ways to encrypt data is to exclusive-*or* (XOR) each character with a secret key.
- Suppose that the key is the `&` character.
- XORing this key with the character `z` yields the `\` character:

```
  00100110  (ASCII code for &)
XOR 01111010  (ASCII code for z)
  01011100  (ASCII code for \)
```


Program: XOR Encryption

- Decrypting a message is done by applying the same algorithm:

```
  00100110  (ASCII code for &)
XOR 01011100  (ASCII code for \)
  01111010  (ASCII code for z)
```


Program: XOR Encryption

- The `xor.c` program encrypts a message by XORing each character with the `&` character.
- The original message can be entered by the user or read from a file using input redirection.
- The encrypted message can be viewed on the screen or saved in a file using output redirection.

Program: XOR Encryption

- A sample file named `msg`:

```
Trust not him with your secrets, who, when left
alone in your room, turns over your papers.
            --Johann Kaspar Lavater (1741-1801)
```

- A command that encrypts `msg`, saving the encrypted message in `newmsg`:

```
xor <msg >newmsg
```

- Contents of `newmsg`:

```
rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT _IST VGVCTU.
            --lINGHH mGUVGT jGPGRCT (1741-1801)
```


Program: XOR Encryption

- A command that recovers the original message and displays it on the screen:

```
xor <newmsg
```

Program: XOR Encryption

- The `xor.c` program won't change some characters, including digits.
- XORing these characters with `&` would produce invisible control characters, which could cause problems with some operating systems.
- The program checks whether both the original character and the new (encrypted) character are printing characters.
- If not, the program will write the original character instead of the new character.

xor.c

```
/* Performs XOR encryption */

#include <ctype.h>
#include <stdio.h>

#define KEY '&'

int main(void)
{
  int orig_char, new_char;

  while ((orig_char = getchar()) != EOF) {
    new_char = orig_char ^ KEY;
    if (isprint(orig_char) && isprint(new_char))
      putchar(new_char);
    else
      putchar(orig_char);
  }

  return 0;
}
```