# Low-Level Programming (2)

## *Program Design (II)*

*2022 Spring*

*Fu-Yin Cherng*
*Dept. CSIE, National Chung Cheng University*

# Recap: Bitwise Shift Operators

- The **bitwise shift operators** shift the **bits** in an integer to the **left** or **right**:

  << left shift

  >> right shift

```
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) */
```

# Recap:
# Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

<table>
<tr><th>Symbol</th><th>Meaning</th><th>Example</th><th>Result of Example</th></tr>
<tr><td>~</td><td>bitwise complement</td><td><code>~(001)</code></td><td><code>110</code></td></tr>
<tr><td>&</td><td>bitwise and</td><td><code>(011) & (110)</code></td><td><code>0</code><code>1</code><code>0</code></td></tr>
<tr><td>^</td><td>bitwise exclusive or</td><td><code>(011) ^ (110)</code></td><td><code>101</code></td></tr>
<tr><td>|</td><td>bitwise inclusive or</td><td><code>(011) | (110)</code></td><td><code>111</code></td></tr>
</table>

# Outline

- Using the Bitwise Operators to Access Bits
- Using the Bitwise Operators to Access Bit-Fields
- Bit-Fields in Structures

# 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

# Using the Bitwise Operators to Access Bits

- In the following example, we will use the assumptions that:
- `i` is a 16-bit `unsigned short` variable.
- The leftmost—or ***most significant***—bit is numbered 15 and the least significant is numbered 0.

```
unsigned short i;

/* binary of i 0000000000000000) */
```

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

# Using the Bitwise Operators to Access Bits

- If the position of the bit is stored in the variable `j`, a shift operator can be used to create the mask
- Example: If `j` has the value **3**, then `1 << j` is `0x0008`.
  - `1` in binary: `0000000000000001`
  - `0x0008` in binary: `0000000000001000`

```
i = 0x0000;   /* i is now 0000000000000000 */
i |= 0x0010; /* i is now  0000000000010000 */
             /* 1 << j is 0000000000001000 */
i |= 1 << j;  /* i is now 0000000000011000 */
              /* sets bit j (3) to 1*/
```

# 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 */
              /* ~0x0010 is 1111111111101111 */
i &= ~0x0010; /* i is now 0000000011101111 */
```

# Using the Bitwise Operators to Access Bits

- A statement that clears a bit whose position is stored in a variable
  - the position of the bit is stored in the variable `j`

```
i = 0x00ff;   /* i is now 0000000011111111 */
i &= ~0x0010; /* i is now 0000000011101111 */
i &= ~(1 << j);     /* clears bit j */
```

# Using the Bitwise Operators to Access Bits

- A statement that clears a bit whose position is stored in a variable
  - the position of the bit is stored in the variable `j`

```
i = 0x00ff;   /* i is now 0000000011111111 */
i &= ~0x0010; /* i is now 0000000011101111 */
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:
  - A statement that tests whether bit `j` is set:

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

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

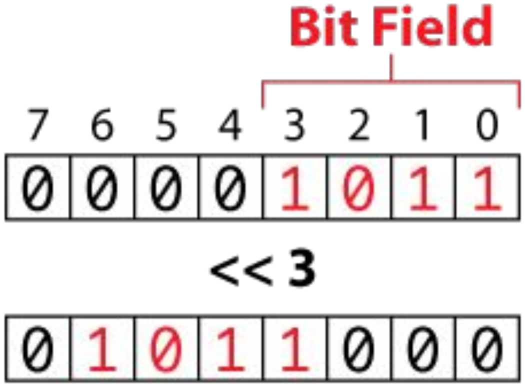
# Using the Bitwise Operators to Access Bits

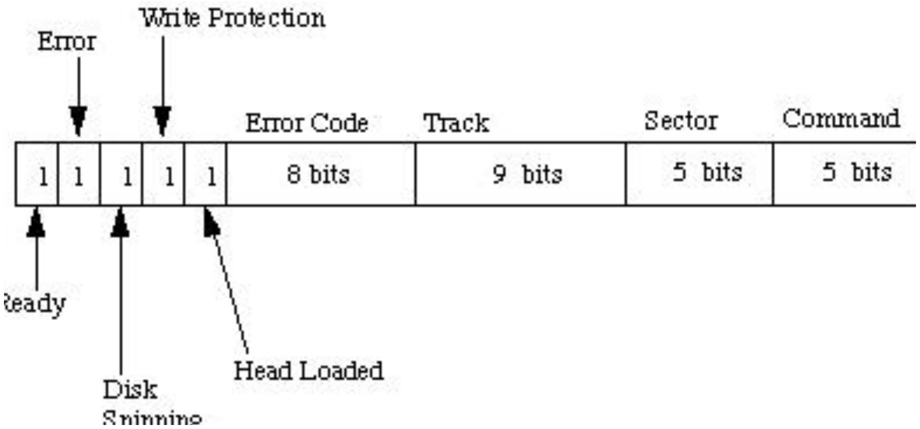
- It’s also easy to set, clear, or test several bits at time

```
i |= BLUE | GREEN;          /* sets BLUE and GREEN bits   */
i &= ~(BLUE | GREEN);       /* clears BLUE and GREEN bits */
```

# 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.
- Using a 16-bits intager to store a small number is wasting. Sometimes, we can store information in certain bit field.



# Using the Bitwise Operators to Access Bit-Fields

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

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

# 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 */
```

# 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
- `0x0007` in binary: `0000000000000111`

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

# Using the Bitwise Operators to Access Bit-Fields

- 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 of i*/
```

# 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
- XORing `\` with `&` (key) and we can get `z` back

```
    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. Why?
  - XORing these characters with `&` would produce invisible control characters, which could cause problems with some operating systems.
- So, the program checks whether both the original character and the new (encrypted) character are printing characters by using `isprint()` function
- If not, the program will write the original character instead of the new character.

# xor.c

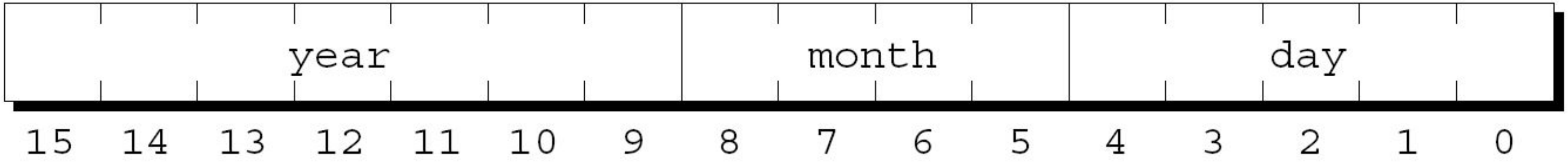
```
#include <ctype.h>
#include <stdio.h>
#define KEY '&'
int main(void){
  int orig_char, new_char;
  while ((orig_char = getchar()) != EOF) {
    new_char = orig_char ^ KEY;
    if (isprint(orig_char) && isprint(new_char))
      putchar(new_char);
    else
      putchar(orig_char);
  }
  return 0;
}
```

# Bit-Fields in Structures

- The bit-field techniques discussed previously can be tricky to use and potentially confusing.
- Fortunately, C provides an alternative: declaring structures whose members represent bit-fields.

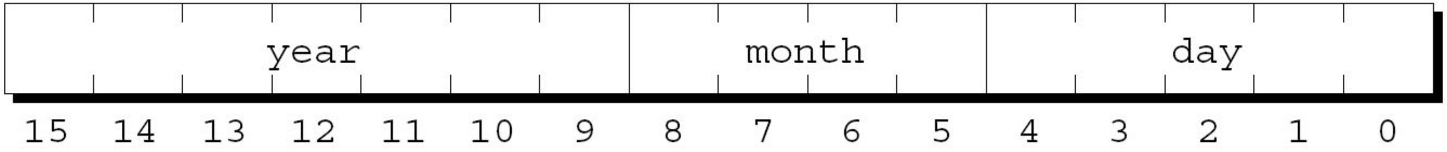
# Bit-Fields in Structures

- Example: How MS-DOS stores the date at which a file was created or last modified.
- Since days, months, and years are small numbers, storing them as normal integers would waste space.
- Instead, DOS allocates only 16 bits for a date, with 5 bits for the day, 4 bits for the month, and 7 bits for the year:



# Bit-Fields in Structures

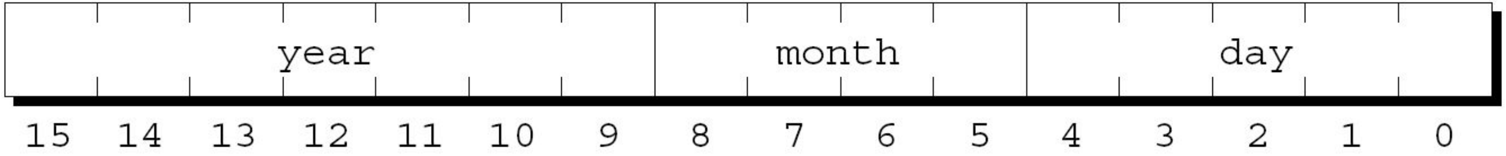
- A C structure that uses bit-fields to create an identical layout
- The type of a bit-field must be either `unsigned int`, or `signed int`.




```
struct file_date {
     unsigned int day: 5;
     unsigned int month: 4;
     unsigned int year: 7;
};
```

# Bit-Fields in Structures

- A bit-field can be used in the same way as any other member of a structure
  - year is stored relative to 1980, the beginning of the world for Microsoft




```
struct file_date fd;
fd.day = 28;
fd.month = 12;
fd.year = 8;      /* represents 1988 */
```

# Bit-Fields in Structures

- Appearance of the `fd` variable after these assignments:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

```
struct file_date fd;
fd.day = 28;
fd.month = 12;
fd.year = 8;     /* represents 1988 */
```

# Bit-Fields in Structures

- One restriction of Bit-field members is that they don’t have usual address.
- So, the address operator (`&`) can’t be applied to a bit-field.
- Because of this rule, functions such as `scanf` can’t store data directly in a bit-field

```
scanf("%d", &fd.day);   /*** WRONG ***/
```

# Bit-Fields in Structures

- One restriction of Bit-field members is that they don’t have usual address.
- So, the address operator (`&`) can’t be applied to a bit-field.
- Because of this rule, functions such as `scanf` can’t store data directly in a bit-field
- We can still use `scanf` to read input into an ordinary variable and then assign it to `fd.day`.

```
scanf("%d", &fd.day);   /*** WRONG ***/
int n;
scanf("%d", &n);
fd.day = n;
```

# Summary

- Using the Bitwise Operators to Access Bits
  - Common single-bit operations:
    - Setting a bit
    - Clearing a bit
    - Testing a bit
- Using the Bitwise Operators to Access Bit-Fields
  - Modifying a bit-field
  - Retrieving a bit-field
  - Program: XOR Encryption
- Bit-Fields in Structures
  - example of MS-DOS stores the date