Low-Level Programming

Based on slides from K. N. King and Dianna Xu

Bryn Mawr College
CS246 Programming Paradigm

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
- Bits are indexed from `0` starting from the right

Bitwise Operators

- Bitwise operators operate on integer data at the bit level.
  - shift
    - `<<` left shift
    - `>>` right shift
  - bitwise complement `~`
  - bitwise *and* `&`
  - exclusive *or* `^`
  - inclusive *or* `|`

Integer Promotion

- If an `int` can represent all values of the original type, the value is converted to an `int` ; otherwise, it is converted to an `unsigned int`. These are called the integer promotions. All other types are unchanged by the integer promotions.

Bitwise Shift Operators

- `<<` left shift `>>` right shift
  - shift the bits in an integer to the left or right
  - operands may be of any integer type (including `char`).
  - the result has the type of the left operand after promotion.

Bitwise Shift Operators

- `i << j`
  - shifts the bits in `i` to the left by `j` places
  - for each bit that is "shifted off" the left end of `i`, a `0` bit enters at the right.
- `i >> j`
  - shifts the bits in `i` to the right by `j` places
  - if `i` is of an unsigned type or if the value of `i` is nonnegative, `0s` are added at the left as needed.
  - if `i` is negative, the result is implementation-defined.
- Operands may be of any integer type, but use **`unsigned`** for portability

Bitwise Shift Operators

```
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) */
```

- 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 Complement, And, Exclusive Or, and Inclusive Or

- There are four additional bitwise operators:
  - `~` bitwise complement : unary
  - `&` bitwise *and* : binary
  - `^` bitwise exclusive *or* : binary
  - `|` bitwise inclusive *or* : binary
- 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 compound assignment 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) */
```

Precedence

- 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`
- 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.

Machine Dependency

- The result of bitwise operators is often machine dependent, that is, it depends on the size of integers on the local machine.
- The `~` operator can be used to help make low-level programs more portable.
  - o An integer whose bits are all 1: `~0`
  - o An integer whose bits are all 1 except for the last five: `~0x1f`

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.**

```
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 */
```

- The constant used to set a bit is known as a mask.
- Example:
  - If `j` has the value 3, then `1 << j` is `0x0008`.

Using the Bitwise Operators to Access Bits

- **Clearing a bit.**

```
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 */
```

`enum` and Bit Masks

- 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:
  - **`enum{BLUE = 1, GREEN = 2, RED = 4};`**
- Examples of setting, clearing, and testing the `BLUE` bit:
  - **`i |= BLUE;`** – sets the **`BLUE`** bit
  - **`i &= ~BLUE;`** – clears the **`BLUE`** bit
  - **`if (i & BLUE)`** – tests the **`BLUE`** bit
- It's also easy to set, clear, or test several bits at time:
  - **`i |= BLUE|GREEN`** – sets the **`BLUE`** and **`GREEN`** bits
  - **`i &= ~(BLUE|GREEN)`** – clears **`BLUE`** and **`GREEN`**
  - **`if (i&(BLUE|GREEN))`** – tests **`BLUE`** and **`GREEN`**
    - The `if` statement tests whether either the `BLUE` bit or the `GREEN` bit is set.

Bit Fields

- A group of several consecutive bits is a bit-field.
- Common bit-field operations:
  - Modifying a bit-field
  - Retrieving a bit-field

Bit Fields

- Modifying a bit-field
  - A bitwise *and* (to clear the bit-field)
  - A bitwise *or* (to store new bits in the bit-field)
  - Example: stores 101 in bits 4-6

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

  - The **&** clears bits **4-6** and the **|** sets bits **4** and **6**
  - Just using **|** will not always work, as it doesn't clear bit **5**
  - Assume that j contains the value to be stored in bits 4–6 of i. To store j into position 4-6 of i:

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

Bit Fields

- Retrieving a bit-field
  - Fetching a bit-field at the right end of a number (in the least significant bits)
    - Example: retrieve bits 0-2 of `i`

```
j = i & 0x0007;
```

  - What if the bit-field isn't at the right end of `i`?
    - Example: retrieve bits 4-6 of `i`
      - First shift the bit-field to the end
      - Then extracting the field using the `&` operator:

```
j = (i >> 4) & 0x0007;
```

Program: XOR Encryption

- 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 \)
```

- 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

- 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)
```

- 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;
}
```

Bit-Fields in Structures

- C allows structure declarations whose members are bit-fields.
- 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:

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

```
struct file_date fd;

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

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

Bit-Fields and Memory

- Bit Fields do not have addresses
  - `scanf("%d", &fd.day); /*wrong*/`
- How bit fields are stored is highly machine and implementation dependent. The example in the previous slide assumes 16-bit units.
- When bit fields do not fit a storage unit precisely, what happens is compiler dependent.

Big-endian and Little-endian

- When a data item consists of more than one byte, there are two logical ways to store it in memory (the order of storing bytes):
  - ***Big-endian:*** Bytes are stored in "natural" order (the leftmost byte comes first).
  - ***Little-endian:*** Bytes are stored in reverse order (the leftmost byte comes last).
- x86 processors use little-endian order.
- We don't normally need to worry about byte ordering.
- However, programs that deal with memory at a low level must be aware of the order in which bytes are stored.

Big-endian and Little-endian

- A way to determine endianness of your machine

```
#include <stdio.h>
int main()
{
   unsigned int i = 1;
   char *c = (char*)&i;
   if (*c)
       printf("Little endian");
   else
       printf("Big endian");
   getchar();
   return 0;
}
```