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.

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 |= 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 */
if (i & (BLUE | 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

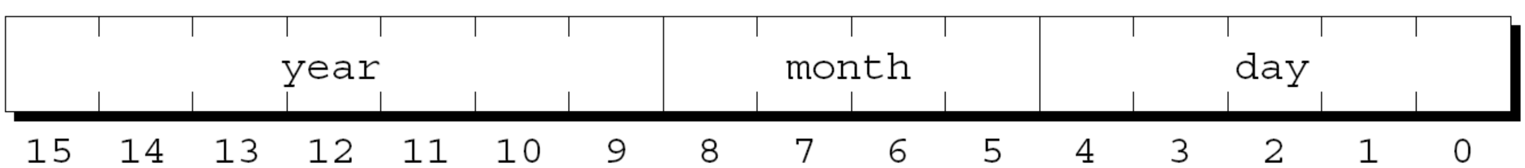
```
/* Performs XOR encryption */

#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;
}
```

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 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:

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

- A condensed version:

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

Bit-Fields in Structures

- The type of a bit-field must be either `int`, `unsigned int`, or `signed int`.
- Using `int` is ambiguous; some compilers treat the field's high-order bit as a sign bit, but others don't.
- In C99, bit-fields may also have type `_Bool`.
- C99 compilers may allow additional bit-field types.

Bit-Fields in Structures

- A bit-field can be used in the same way as any other member of a structure:

```
struct file_date fd;

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

- 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

Bit-Fields in Structures

- 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 ***/
```

- We can still use `scanf` to read input into an ordinary variable and then assign it to `fd.day`.

How Bit-Fields Are Stored

- The C standard allows the compiler considerable latitude in choosing how it stores bit-fields.
- The rules for handling bit-fields depend on the notion of "storage units."
- The size of a storage unit is implementation-defined.
  - Typical values are 8 bits, 16 bits, and 32 bits.

How Bit-Fields Are Stored

- The compiler packs bit-fields one by one into a storage unit, with no gaps between the fields, until there's not enough room for the next field.
- At that point, some compilers skip to the beginning of the next storage unit, while others split the bit-field across the storage units.
- The order in which bit-fields are allocated (left to right or right to left) is also implementation-defined.

How Bit-Fields Are Stored

- Assumptions in the `file_date` example:
  - Storage units are 16 bits long.
  - Bit-fields are allocated from right to left (the first bit-field occupies the low-order bits).
- An 8-bit storage unit is also acceptable if the compiler splits the `month` field across two storage units.

How Bit-Fields Are Stored

- The name of a bit-field can be omitted.
- Unnamed bit-fields are useful as "padding" to ensure that other bit-fields are properly positioned.
- A structure that stores the time associated with a DOS file:

```
struct file_time {
  unsigned int seconds: 5;
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

How Bit-Fields Are Stored

- The same structure with the name of the `seconds` field omitted:

```
struct file_time {
  unsigned int : 5;        /* not used */
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

- The remaining bit-fields will be aligned as if `seconds` were still present.

How Bit-Fields Are Stored

- The length of an unnamed bit-field can be 0:

```
struct s {
  unsigned int a: 4;
  unsigned int : 0;     /* 0-length bit-field */
  unsigned int b: 8;
};
```

- A 0-length bit-field tells the compiler to align the following bit-field at the beginning of a storage unit.
  - If storage units are 8 bits long, the compiler will allocate 4 bits for `a`, skip 4 bits to the next storage unit, and then allocate 8 bits for `b`.
  - If storage units are 16 bits long, the compiler will allocate 4 bits for `a`, skip 12 bits, and then allocate 8 bits for `b`.

Other Low-Level Techniques

- Some features covered in previous chapters are used often in low-level programming.
- Examples:
  – Defining types that represent units of storage
  – Using unions to bypass normal type-checking
  – Using pointers as addresses
- The `volatile` type qualifier was mentioned in Chapter 18 but not discussed because of its low-level nature.

Defining Machine-Dependent Types

- The `char` type occupies one byte, so characters can be treated as bytes.
- It's a good idea to define a `BYTE` type:

```
typedef unsigned char BYTE;
```

- Depending on the machine, additional types may be needed.
- A useful type for the x86 platform:

```
typedef unsigned short WORD;
```

Using Unions to Provide Multiple Views of Data

- Unions can be used in a portable way, as shown in Chapter 16.
- However, they're often used in C for an entirely different purpose: viewing a block of memory in two or more different ways.
- Consider the `file_date` structure described earlier.
- A `file_date` structure fits into two bytes, so any two-byte value can be thought of as a `file_date` structure.

Using Unions to Provide Multiple Views of Data

- In particular, an `unsigned short` value can be viewed as a `file_date` structure.
- A union that can be used to convert a short integer to a file date or vice versa:

```
union int_date {
  unsigned short i;
  struct file_date fd;
};
```

Using Unions to Provide Multiple Views of Data

- A function that prints an `unsigned short` argument as a file date:

```
void print_date(unsigned short n)
{
  union int_date u;

  u.i = n;
  printf("%d/%d/%d\n", u.fd.month,
         u.fd.day, u.fd.year + 1980);
}
```

Using Unions to Provide Multiple Views of Data

- Using unions to allow multiple views of data is especially useful when working with registers, which are often divided into smaller units.
- x86 processors have 16-bit registers named AX, BX, CX, and DX.
- Each register can be treated as two 8-bit registers.
  - AX is divided into registers named AH and AL.

Using Unions to Provide Multiple Views of Data

- Writing low-level applications for x86-based computers may require variables that represent AX, BX, CX, and DX.
- The goal is to access both the 16- and 8-bit registers, taking their relationships into account.
  - A change to AX affects both AH and AL; changing AH or AL modifies AX.
- The solution is to set up two structures:
  - The members of one correspond to the 16-bit registers.
  - The members of the other match the 8-bit registers.

Using Unions to Provide Multiple Views of Data

- A union that encloses the two structures:

```
union {
  struct {
    WORD ax, bx, cx, dx;
  } word;
  struct {
    BYTE al, ah, bl, bh, cl, ch, dl, dh;
  } byte;
} regs;
```

Using Unions to Provide Multiple Views of Data

- The members of the `word` structure will be overlaid with the members of the `byte` structure.
  - `ax` will occupy the same memory as `al` and `ah`.
- An example showing how the `regs` union might be used:

```
regs.byte.ah = 0x12;
regs.byte.al = 0x34;
printf("AX: %hx\n", regs.word.ax);
```

- Output:

```
AX: 1234
```

Using Unions to Provide Multiple Views of Data

- Note that the `byte` structure lists `al` before `ah`.
- When a data item consists of more than one byte, there are two logical ways to store it in memory:
  - ***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.

Using Unions to Provide Multiple Views of Data

- 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.
- It’s also relevant when working with files that contain non-character data.

Using Pointers as Addresses

- An address often has the same number of bits as an integer (or long integer).
- Creating a pointer that represents a specific address is done by casting an integer to a pointer:

```
BYTE *p;

p = (BYTE *) 0x1000;
  /* p contains address 0x1000 */
```

Program: Viewing Memory Locations

- The `viewmemory.c` program allows the user to view segments of computer memory.
- The program first displays the address of its own `main` function as well as the address of one of its variables.
- The program next prompts the user to enter an address (as a hexadecimal integer) plus the number of bytes to view.
- The program then displays a block of bytes of the chosen length, starting at the specified address.

Program: Viewing Memory Locations

- Bytes are displayed in groups of 10 (except for the last group).
- Bytes are shown both as hexadecimal numbers and as characters.
- Only printing characters are displayed; other characters are shown as periods.
- The program assumes that `int` values and addresses are stored using 32 bits.
- Addresses are displayed in hexadecimal.

viewmemory.c

```
/* Allows the user to view regions of computer memory */

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

typedef unsigned char BYTE;

int main(void)
{
  unsigned int addr;
  int i, n;
  BYTE *ptr;

  printf("Address of main function: %x\n", (unsigned int) main);
  printf("Address of addr variable: %x\n", (unsigned int)
   &addr);
```

```
printf("\nEnter a (hex) address: ");
scanf("%x", &addr);
printf("Enter number of bytes to view: ");
scanf("%d", &n);

printf("\n");
printf(" Address              Bytes
 Characters\n");
printf(" ------- -----------------------------
 ----------\n");
```

```
  ptr = (BYTE *) addr;
  for (; n > 0; n -= 10) {
    printf("%8X  ", (unsigned int) ptr);
    for (i = 0; i < 10 && i < n; i++)
      printf("%.2X ", *(ptr + i));
    for (; i < 10; i++)
      printf("   ");
    printf(" ");
    for (i = 0;  i < 10 && i < n; i++) {
      BYTE ch = *(ptr + i);
      if (!isprint(ch))
        ch = '.';
      printf("%c", ch);
    }
    printf("\n");
    ptr += 10;
  }

  return 0;
}
```

Program: Viewing Memory Locations

- Sample output using GCC on an x86 system running Linux:

```
Address of main function: 804847c
Address of addr variable: bff41154

Enter a (hex) address: 8048000
Enter number of bytes to view: 40

 Address               Bytes              Characters
 -------  -----------------------------  ----------
 8048000  7F 45 4C 46 01 01 01 00 00 00  .ELF......
 804800A  00 00 00 00 00 00 02 00 03 00  ..........
 8048014  01 00 00 00 C0 83 04 08 34 00  ........4.
 804801E  00 00 C0 0A 00 00 00 00 00 00  ..........
```

- The 7F byte followed by the letters E, L, and F identify the format (ELF) in which the executable file was stored.

Program: Viewing Memory Locations

- A sample that displays bytes starting at the address of `addr`:

```
Address of main function: 804847c
Address of addr variable: bfec5484

Enter a (hex) address: bfec5484
Enter number of bytes to view: 64

 Address              Bytes              Characters
 -------  -----------------------------  ----------
BFEC5484  84 54 EC BF B0 54 EC BF F4 6F  .T...T...o
BFEC548E  68 00 34 55 EC BF C0 54 EC BF  h.4U...T..
BFEC5498  08 55 EC BF E3 3D 57 00 00 00  .U...=W...
BFEC54A2  00 00 A0 BC 55 00 08 55 EC BF  ....U..U..
BFEC54AC  E3 3D 57 00 01 00 00 00 34 55  .=W.....4U
BFEC54B6  EC BF 3C 55 EC BF 56 11 55 00  ..<U..V.U.
BFEC54C0  F4 6F 68 00                    .oh.
```

- When reversed, the first four bytes form the number BFEC5484, the address entered by the user.

The **`volatile`** Type Qualifier

- On some computers, certain memory locations are "volatile."
- The value stored at such a location can change as a program is running, even though the program itself isn't storing new values there.
- For example, some memory locations might hold data coming directly from input devices.

The `volatile` Type Qualifier

- The `volatile` type qualifier allows us to inform the compiler if any of the data used in a program is volatile.
- `volatile` typically appears in the declaration of a pointer variable that will point to a volatile memory location:

```
volatile BYTE *p;
  /* p will point to a volatile byte */
```

The `volatile` Type Qualifier

- Suppose that `p` points to a memory location that contains the most recent character typed at the user's keyboard.
- A loop that obtains characters from the keyboard and stores them in a buffer array:

```
while (buffer not full) {
  wait for input;
  buffer[i] = *p;
  if (buffer[i++] == '\n')
    break;
}
```

The **volatile** Type Qualifier

- A sophisticated compiler might notice that this loop changes neither `p` nor `*p`.
- It could optimize the program by altering it so that `*p` is fetched just once:

```
store *p in a register;
while (buffer not full) {
  wait for input;
  buffer[i] = value stored in register;
  if (buffer[i++] == '\n')
    break;
}
```

The `volatile` Type Qualifier

- The optimized program will fill the buffer with many copies of the same character.
- Declaring that `p` points to volatile data avoids this problem by telling the compiler that `*p` must be fetched from memory each time it's needed.