Today's Goals

- Bitwise operators
  - shift
  - complement
  - *and*
  - exclusive and inclusive *or*
- Bit fields
- Bit fields and **`struct`**s

Section 1

Bitwise Operators

- These operators work at bit-level.
- Bit manipulation and other low-level operations are extremely useful for system programming.
  - OS
  - compilers
  - graphics
- Fast execution and efficient use of memory.
- Bits are indexed from **`0`** starting from the right

Shift

- **`<<`** – left shift, **`>>`** – right shift
  - **`i << j`**
  - shifts the bits in **`i`** by **`j`** places to the left
  - for each "shifted off" bit to the left, a **`0`** bit appears on the right

```
unsigned int i, j;
i = 13;        0000000000001101
j = i << 2;    0000000000110100
j = i >> 2;    0000000000000011
```

- Operands may be of any integer type, but use **`unsigned`** for portability

More Bitwise Ops

- **`~`** – complement
- **`&`** – *and*, **`^`** – exclusive *or*, **`|`** – inclusive *or*

```
unsigned int i, j, k;
i = 21;       0000000000010101
j = 56;       0000000000111000
k = ~i;       1111111111101010
k = i & j;    0000000000010000
k = i | j;    0000000000111101
k = i ^ j;    0000000000101101
```


Precedence

- Shift operators have lower precedence than arithmetic operators
  - **`i << 2 + 1`** ==> **`i << (2+1)`**
- **`~`** > shift > **`&`** > **`^`** > **`|`**
  - **`i & ~j | k`**
- **`~`**, **`&`**, **`^`** and **`|`** have lower precedence than relational and equality operators:
  - **`if (status & 0x4000 != 0) ==> if (status & (0x4000 != 0))`**
- **`&=`**, **`^=`** and **`|=`** work as expected.

Machine Dependency

- The result of bitwise operators is often machine dependent, that is, it depends on the size of integers on the local machine.
- However, **`~`** can often be used to initialize integers machine-independently.
  - **`~0`** – an integer whose bits are all **`1`**
  - **`~0x00f`** – an integer whose bits are all **`1`** except for the last **`4`**

Using Bitwise Ops

- Setting a bit (on, set to `1`)
  - `i = 0x0000;     0000000000000000`
    `i |= 0x0010;    0000000000010000`
- The constant used to set a bit is known as a mask.
- Shift operators can be used to create a mask
  - `i |= 1 << 3`
  - `i`'s bit `3` will be set to `1`
  - `i |= 1 << j`

Using Bitwise Ops, ctd

- Clearing a bit (turn off, set to `0`)
  - `i = 0x00ff;     0000000011111111`
    `i &= ~0x0010;   0000000011101111`
- Clearing a bit whose position is stored in a variable `j`
  - `i &= ~(1 << j)`
- Testing a bit
  - `if (i & 0x0010)` – tests the bit `4`
  - `if (i & 1 << j)` – tests the `j` bit

`enum` and bit masks

- Use enum to give names to masks
  - `enum{BLUE = 1, GREEN = 2, RED = 4};`
  - `i |= BLUE;` – sets the `BLUE` bit
  - `i &= ~BLUE;` – clears the `BLUE` bit
  - `if (i & BLUE)` – tests the `BLUE` bit
- Several bits
  - `i |= BLUE|GREEN` – sets the `BLUE` and `GREEN` bits
  - `i &= ~(BLUE|GREEN)` – clears `BLUE` and `GREEN`
  - `if (i&(BLUE|GREEN))` – tests `BLUE` and `GREEN`

Section 2

Bit Fields

- A group of consecutive bits is a bit field.
- Modifying a bit field:
  - Storing binary `101` in bits `4-6` of variable `i`
  - `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`
- Storing `j` into a particular bit field
  - `i = i & ~0x0070 | j << 4`

Bit Fields in Structures

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

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

```
struct file_date fd;

fd.day = 19;
fd.month = 4;
fd.year = 24;   /* 2004 */
```

- Year dates from 1980. Being 7-bits, all DOS programs have a year 2108 problem.

0	0	1	1	0	0	0	0	1	0	0	1	0	0	1	1

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.

Summary

- Bitwise ops and masks occur more often than you would expect.
- Bit and bit fields manipulations are highly efficient, but very machine dependent.
- At the very least, one needs to find out the size of a storage unit.

CS246 13 Lec17