Outline

• Binary representation of integer numbers
• Operations on bits
  – The Bitwise AND Operator
  – The Bitwise Inclusive-OR Operator
  – The Bitwise Exclusive-OR Operator
  – The Ones Complement Operator
  – The Left Shift Operator
  – The Right Shift Operator
– Binary representation of floating-point numbers
• Review: Operators: The complete table of operator's precedence
Binary representation of integers

- Positive integers:

```
0 0 0 0 0 0 0 0 1 0 1
```

1\*2^0 + 0\*2^1 + 1\*2^2 = 5

Sign bit

most significant or high-order bit

least significant or low-order bit
Binary representation of integers

- Negative integers:

Steps to convert a negative number from decimal to binary: Ex: -5
1. add 1 to the value: -5+1=-4
2. express the absolute value of the result in binary: 4=00000100
3. then “complement” all the bits: 11111011=-5

Steps to convert a negative number from binary to decimally: Ex: 1111011
1. complement all of the bits: 00000100
2. convert the result to decimal: 4
3. change the sign of the result, and then subtract 1: -4-1=-5
Bit operators

- &  Bitwise AND
- |  Bitwise Inclusive-OR
- ^  Bitwise Exclusive-OR
- ~  Ones complement
- << Left shift
- >> Right shift

All the operators, with the exception of the ones complement operator ~, are binary operators and as such take two operands.

Bit operations can be performed on any type of integer value in C—be it short, long, long long, and signed or unsigned—and on characters, but cannot be performed on floating-point values.
Bitwise AND

<table>
<thead>
<tr>
<th>b1</th>
<th>b2</th>
<th>b1&amp;b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

int a=25;
int b=77;

printf("%x %x %x", a, b, a&b);
Exercise: even or odd

- testing if an integer is even or odd, using bitwise operations:
- the rightmost bit of any odd integer is 1 and of any even integer is 0.

```c
int n;
if ( n & 1 )
    printf("odd");
else
    printf("even");
```
Bitwise Inclusive-OR (OR)

<table>
<thead>
<tr>
<th>b1</th>
<th>b2</th>
<th>b1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

int a = 25;
int b = 77;

printf("%x %x %x", a, b, a|b);

a = 0x19
b = 0x4d
a|b = 0x5d
Bitwise vs boolean!

- **AND**: bitwise: &; boolean: &&
- **OR**: bitwise: |; boolean: ||

```c
int a, b;
a = 1;
b = 2;
printf("a:%d b:%d &:%d &&:%d |:%d ||:%d \n",
        a, b, a & b, a && b, a | b, a || b);
```
Bitwise Exclusive-OR (XOR)

<table>
<thead>
<tr>
<th>b1</th>
<th>b2</th>
<th>b1^b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

int a=25;
int b=77;

printf("\%x %x %x", a, b, a^b);

a=0x19
b=0x4d
a^b=0x54
The Ones Complement Operator

<table>
<thead>
<tr>
<th>b1</th>
<th>~b1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

int a=25;

printf("%x %x",a,~a);

a=0x19

~a=0xfffffffffe6
Usage: Masking bits

- **A mask** is a bit pattern with some bits set to on (1) and some bits to off (0).
- Example:
  - `int MASK = 2;  // 00000010, with only bit number 1 nonzero.
- Masking bits:
  - `flags = flags & MASK;`
  - all the bits of `flags` (except bit 1) are set to 0 because any bit combined with 0 using the & operator yields 0. Bit number 1 will be left unchanged. (If the bit is 1, 1 & 1 is 1; if the bit is 0, 0 & 1 is 0.) This process is called "using a mask" because the zeros in the mask hide the corresponding bits in `flags`.
- you can think of the 0s in the mask as being opaque and the 1s as being transparent. The expression `flags & MASK` is like covering the flags bit pattern with the mask; only the bits under MASK's 1s are visible
Usage: Turning bits on

- Turn on (set to 1) particular bits in a value while leaving the remaining bits unchanged.

- Example: an IBM PC controls hardware through values sent to ports. To turn on a device (i.e., the speaker), the driver program has to turn on the 1 bit while leaving the others unchanged.

- This can be done with the bitwise OR operator.

- The MASK has bit 1 set to 1, other bits 0

  ```
  int MASK = 2;  // 00000010, with only bit number 1 nonzero.
  ```

- The statement

  ```
  flags = flags | MASK;
  ```

  sets bit number 1 in flags to 1 and leaves all the other bits unchanged. This follows because any bit combined with 0 by using the | operator is itself, and any bit combined with 1 by using the | operator is 1.
Usage: Turning bits off

- Turn off (set to 0) particular bits in a value while leaving the remaining bits unchanged.
- Suppose you want to turn off bit 1 in the variable `flags`.
- The `MASK` has bit 1 set to 1, other bits 0
- `int MASK = 2;  // 00000010, with only bit number 1 nonzero.
- The statement:
- `flags = flags & ~MASK;`
- Because `MASK` is all 0s except for bit 1, `~MASK` is all 1s except for bit 1. A 1 combined with any bit using `&` is that bit, so the statement leaves all the bits other than bit 1 unchanged. Also, a 0 combined with any bit using `&` is 0, so bit 1 is set to 0 regardless of its original value.
Usage: Toggling bits

- Toggling a bit means turning it off if it is on, and turning it on if it is off.
- To toggle bit 1 in flag:
  - The MASK has bit 1 set to 1, other bits 0
  - int MASK = 2;  // 00000010, with only bit number 1 nonzero.
  - flag = flag ^ MASK;
- You can use the bitwise EXCLUSIVE OR operator to toggle a bit.
  - The idea is that if b is a bit setting (1 or 0), then 1 ^ b is 0 if b is 1 and is
    1 if b is 0. Also 0 ^ b is b, regardless of its value.
  - Therefore, if you combine a value with a mask by using ^, values
    corresponding to 1s in the mask are toggled, and values corresponding to
    0s in the mask are unaltered.
Usage: Checking value of a bit

• For example: does flag has bit 1 set to 1?
• you must first mask the other bits in flag so that you compare only bit 1 of flag with MASK:
  
  if ((flag & MASK) == MASK)
The Left Shift Operator

- the bits contained in the value are shifted to the left a number of places (bits)
- Bits that are shifted out through the high-order bit of the data item are lost,
- 0s are always shifted in through the low-order bit of the value.

```c
int a=25;
printf("%x %x",a,a<<1);
```

```plaintext
a : 0x19
a<<1 : 0x32
```
Exercise: multiply with 2

- Left shifting has the effect of multiplying the value that is shifted by two.
- Multiplication of a value by a power of two: left shifting the value the appropriate number of places

```c
int a;
scanf("%i", &a);
printf("a multiplied with 2 is %i \n", a<<1);
printf("a multiplied with 4 is %i \n", a<<2);
```
The Right Shift Operator

- Shifts the bits of a value to the right.
- Bits shifted out of the low-order bit of the value are lost.
- Right shifting an unsigned value or a signed positive value always results in 0s being shifted in on the left.

```c
int a = 25;
printf("%x %x", a, a >> 1);
```

```
00000000 000011001  
Bit lost
```

```
00000000 000001100  
a : 0x19
```

```
00000000 000001100  
a >>1 : 0xc
```
Right shift of negative integers

• If the sign bit is 1, on some machines 1s are shifted in, and on others 0s are shifted in.
• This former type of operation is known as an arithmetic right shift, whereas the latter is known as a logical right shift.
• Never make any assumptions about whether a system implements an arithmetic or a logical right shift. A program that shifts signed values right might work correctly on one system but fail on another due to this type of assumption.
Example: extract groups of bits

Example: an unsigned long value is used to represent color values, with the low-order byte holding the red intensity, the next byte holding the green intensity, and the third byte holding the blue intensity.

You then want to store the intensity of each color in its own unsigned char variable.

```c
unsigned char BYTE_MASK = 0xff;
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
printf("%x %x %x\n", red, green, blue);
```
Exercise: getbits

getbits(x, p, n): returns the (right adjusted) n-bit field of \( x \) that begins at position \( p \). We assume that bit position 0 is at the right end and that \( n \) and \( p \) are positive values.

/* getbits: get n bits from position p */
unsigned getbits(unsigned x, int p, int n)
{
  return (x>>(p+1-n))&~(~0<<n);
}
Exercise: printbits

printbits(x, p, n) : prints the n-bit field of x that begins at position p.
   We assume that bit position 0 is at the right end and that n and p are positive values.

void printbits(unsigned x, int p, int n)
{
   int i;
   unsigned mask=1<<p;
   for (i=0; i<n; i++) {
      if (x&mask) printf("1");
      else printf("0");
      mask=mask>>1;
   }
}
Exercise: IEEE float representation

- Standards: single precision / double precision
- IEEE Standard 754 Floating Point Numbers in Single precision

Layout:

```
  31  30  23  22  2  1  0

  sign | biased exponent | fraction
    1 bit |         8 bits   |      23 bits
```
IEEE float (cont)

- Sign: 0=positive, 1=negative
- Exponent: a *bias* is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127
- Examples:
  - *actual exponent* =0: *biased exponent field stored* =127.
  - *biased exponent field stored* = 200: *actual exponent* =(200-127)= 73.
- The exponents are powers of 2 in binary!
- Mantissa: there is an assumed leading 1 in the *actual mantissa* that is not stored in memory, so the mantissas are actually 24 bits, even though only 23 bits are stored in the *fraction field*.
- Examples:
  - Stored fraction =11000…0: actual mantissa=1.11000…0(binary)= 1+1/2+1/4
- Represented value:

\[
\text{value} = 2^{\text{biased exponent}-127} \cdot 1.\text{fraction}
\]
IEEE float examples

- Float representation:
  00111111111000000000000000000000
  Sign=0
  Biased exponent=0111111=127  => Actual exponent=127-127=0
  Fraction=11000000000000000000000000
  Mantissa=1.f=1.11=1+1/2+1/4=1.75
  Value of number is $2^0 \times 1.75 = 1.75$

- Float representation:
  01000000011000000000000000000000
  Sign=0
  Biased exponent=1000000=128  => Actual exponent=128-127=1
  Fraction=11000000000000000000000000
  Mantissa=1.f=1.11=1+1/2+1/4=1.75
  Value of number is $2^1 \times 1.75 = 3.5$
Exercise: print bits of a float

```c
void printbits(unsigned x, int p, int n);

int main(void)
{
    float f=3.5;
    unsigned int *ip=&f;

    printf("\n sign: "); printbits(*ip, 31, 1);
    printf("\n biased exponent: "); printbits(*ip, 30, 8);
    printf("\n fraction: "); printbits(*ip, 22, 23);
    return 1;
}
```
# Review: Operators in C

## Table A.5  Summary of C Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Function call</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>Array element reference</td>
<td></td>
</tr>
<tr>
<td>-&gt;</td>
<td>Pointer to structure member reference</td>
<td>Left to right</td>
</tr>
<tr>
<td>.</td>
<td>Structure member reference</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Logical negation</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>Ones complement</td>
<td>Right to left</td>
</tr>
<tr>
<td>*</td>
<td>Pointer reference (indirection)</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Address</td>
<td></td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td>Size of an object</td>
<td></td>
</tr>
<tr>
<td>(type)</td>
<td>Type cast (conversion)</td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td>Operation</td>
<td>Direction</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Left to right</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>Left to right</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left shift</td>
<td>Left to right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>Left to right</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td></td>
</tr>
<tr>
<td>=&gt;</td>
<td>Greater than or equal to</td>
<td></td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Associativity</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
<td>Left to right</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality</td>
<td>Left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>Left to right</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Conditional</td>
<td>Right to left</td>
</tr>
<tr>
<td>=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*= /= %=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+= -= &amp;=</td>
<td>Assignment operators</td>
<td>Right to left</td>
</tr>
<tr>
<td>^=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;= &gt;&gt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>Comma operator</td>
<td>Right to left</td>
</tr>
</tbody>
</table>