

Operations on Bits

Motivation

- C was developed with systems programming in mind
 - operating systems, device drivers, socket programming, network programming (Only about 10 % of UNIX is assembly code the rest is C!!.)
 - powerful, compiled and run quickly
- C is often used in resource-constrained situations
 - devices without much memory, with slow processors and slow network connection
- It is sometimes necessary to manipulate individual bits of data
 - twiddle with bits
 - bit packing

Binary Representation

- Recall: **ints** are stored as 32-bit (4 byte) integers

- `int x = 42;`

00000000	00000000	00000000	00101010
----------	----------	----------	----------

- The maximum positive **int** value that can be stored is $2^{31}-1$

- `int x = 2147483647; // largest positive value`

01111111	11111111	11111111	11111111
----------	----------	----------	----------

- There exists negative binary numbers too with using left most bit as the “sign bit”.

- All **bitwise operators** discussed here should only be used on **UNSIGNED operands** for portability.

Bitwise Logical Operators

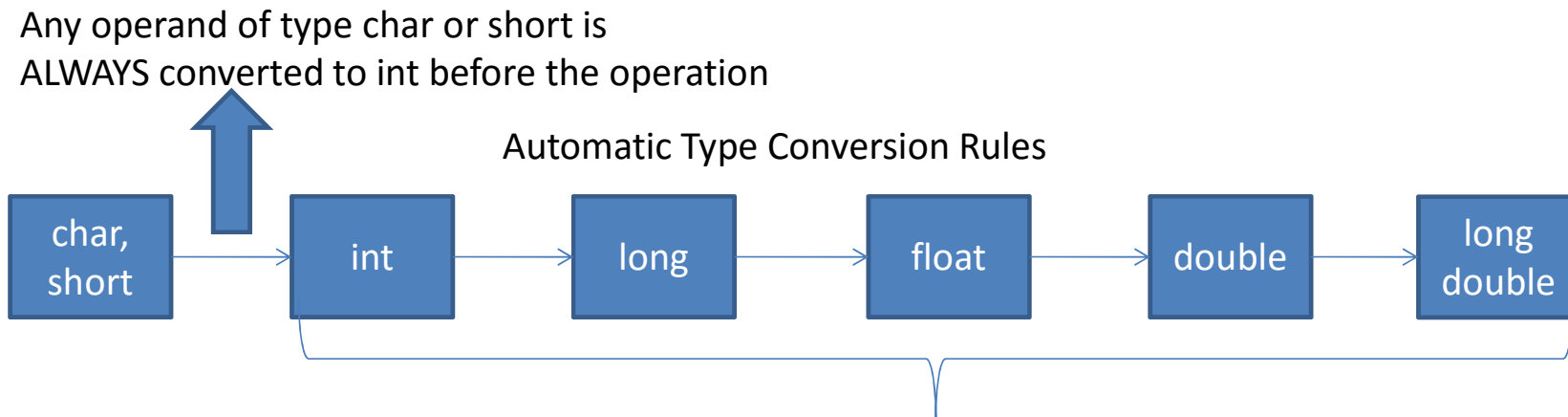
- bitwise complement \sim
- bitwise and $\&$
- bitwise exclusive or \wedge
- bitwise inclusive or $|$

REMARK

- Bitwise logical operators can only be applied to **integral operands**.
- When evaluating expressions, the automatic unary and binary conversions are first performed on the operands. Type of the result is that of the converted operands.

Recap: Automatic Type Conversion

Automatic: C performs all arith. ops with 6 data types: int, unsigned int, long, float, double, long double



Idea: If a binary operator has operands of two different types, lower type is **promoted to** higher type before the op. proceeds, and the **result** is also **higher type**.

Bitwise Logical Operators

- The \sim , $\&$, \wedge , and \mid perform Boolean operations on **all bits in their operands**.
- Bitwise complement is **unary**, others are **binary** operators
- All three binary bitwise operators are both **commutative** and **associative**.
- All three binary bitwise operators can be used with $=$ such as $\&=$, $\wedge=$, $\mid=$ (i.e., **compound assignment** operators)

Bitwise AND

- *intval* **&** *intval*

unsigned int e1 = 0xd;

unsigned int e2 = 0x7;

Expression	Binary Representation (16 bits)	Value
e1	0000 0000 0000 1101	0xd
e2	0000 0000 0000 0111	0x7
e1 & e2	0000 0000 0000 0101	0x5

Remark!

Logical AND (&&) and bitwise AND (&) are **DIFFERENT**:

1) && : result is always 0 or 1

&: depends on the operands (only when both operands are 0-1, result is 0 or 1)

2) For &&, if first operand is 0, second operand is not evaluated (short-circuit)

For &, both operands will be evaluated

Bitwise inclusive OR

- *intval* **|** *intval*

unsigned int e1 = 0xd;

unsigned int e2 = 0x7;

Expression	Binary Representation (16 bits)	Value
e1	0000 0000 0000 1101	0xd
e2	0000 0000 0000 0111	0x7
e1 e2	0000 0000 0000 1111	0xf

Again, logical OR (||) and bitwise OR (|) are different (as shown in previous slide)

Bitwise exclusive OR

- *intval* \wedge *intval*

unsigned int e1 = 0xd;

unsigned int e2 = 0x7;

Expression	Binary Representation (16 bits)	Value
e1	0000 0000 0000 1101	0xD
e2	0000 0000 0000 0111	0x7
e1 \wedge e2	0000 0000 0000 1010	0xA

Exclusive or (\wedge) operator produces 0 whenever both operands have a 1 bit, whereas $|$ produces 1.

Bitwise exclusive OR

- Any value XOR'ed with itself produces 0
 - Often used by assembly programmers to set a value to 0 or to check if two values are equal

Another useful property of this operator is that if the result of XORing a value with another value is again XORed with the second value, the result is the first value. Thus, we have

- $(e1 \wedge e2) \wedge e2$ is equal to $e1$
 - Used for cryptography, or
 - two swap two values without using a temp variable

Swap with XOR

Expression	Binary Representation (16 bits)	Value
e1	0000 0000 0000 1101	0xD
e2	0000 0000 0000 0111	0x7
e1 ^= e2	0000 0000 0000 1010	0xA
e2 ^= e1	0000 0000 0000 1101	0xD
e1 ^= e2	0000 0000 0000 0111	0x7

- e1 = e1 XOR e2
- e2 = e2 XOR **e1** --> e2 XOR (e1 XOR e2) --> e1
- e1 = e1 XOR **e2** --> (e1 XOR e2) XOR e2 --> e2

Bitwise Complement

\sim intval

yields 1s complement, i.e., converts 1 to 0, and 0 to 1

Expression	Binary Representation (16 bits)	Value
e1	0000 0000 0000 1101	0xD
\sim e1	1111 1111 1111 0010	0xFFF2

Bitwise Complement

- **Do not confuse** \sim with arithmetic unary minus: **-** or logical negation: **!**
- Assume an int $e = 0$;
 - $-e$ --> still 0
 - $!e$ --> true, so 1
 - $\sim e$ all 1s, which for instance represents -1 in 2s complement form
- Complement is good for following things: set the last 8 bits to 0 on any machine
 $e \&= \sim 0xFF$; // sets the last 8 bits to 0 regardless of size of e

Operator	Type	Associativity
Function call: () Array subscript: [] Dot operator: . Arrow operator: →		Left to right
(type) + - ++ -- ! & * sizeof ~ (complement)	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
< <= > >=	Binary	Left to right
== !=	Binary	Left to right
& (bitwise AND)		Left to right
^ (bitwise XOR)		Left to right
 (bitwise OR)		Left to right
&&	Binary	Left to right
	Binary	Left to right
= *= /= %= += -=	Binary	Right to left
,		Left to right

Precedence and associativity

- $01 \mid \sim 01 \wedge 01 \& 01$
 $(01 \mid ((\sim 01) \wedge (01 \& 01)))$
- $i \& 01 == 0$
- Correct usage $(i \& 01) == 0$

Bitwise Shift Operators

- Bitwise Shift Operators
 - << left shift
 - >> right shift
- Both are binary operators
- Left operand is the integral data (whose bits are to be shifted)
- Right operand (**shift count**) specifies **number of positions** by which **bits will shift**
 - Must be non-negative,
 - Less than the number of bits required to represent the left operand
- Automatic **unary conversions** are performed on both the operands. Type of the result is that of the promoted LEFT operand (i.e., right operand does not promote the result)

Bitwise Shift Operators

- Bitwise Shift Operators
 - << left shift
 - >> right shift
- All **bitwise operators** discussed here should only be used on **UNSIGNED operands** for portability.
 - Applying to signed operands is implementation dependent
- These operators can also form **compound assignments** operators >>= <<=

Left Shift Operator

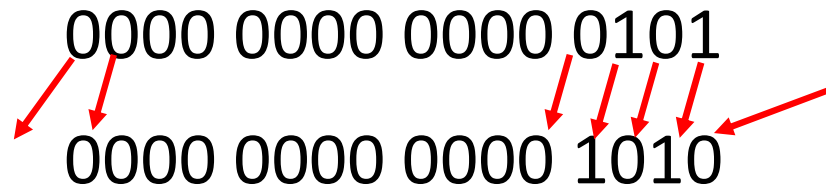
- *intval* << *intval*
- Shifts bits to left, 0 bits enter to low-order positions; bits shifted out through high-order positions are lost

unsigned int i = 5;

i << 1

is

0000 0000 0000 0101
0000 0000 0000 1010



0 bit enters to low order position

i << 15

is

1000 0000 0000 0000 (which is 32768)

- Left shift has the effect of **multiplying by 2**
 - unless an overflow occurs due to a 1 falling from the high-order position

Right Shift Operator

- *intval* **>>** *intval*
- Shifts bits to right, 0 bits enter to high-order positions (for unsigned data); bits shifted out through low-order positions are lost

unsigned int i = 40960; 1010 0000 0000 0000
 0 bit enters to
 high order position 0101 0000 0000 0000 falls...
 i >> 1 (which is 20480)

i >> 15 is 0000 0000 0000 0001 (which is 1)

- Right shift has the effect of **dividing by 2**
 - Provided that original value is non-negative

Operator	Type	Associativity
Function call: () Array subscript: [] Dot operator: . Arrow operator: →		Left to right
(type) + - ++ -- ! & * sizeof ~ (complement)	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
<< >>	Binary	Left to right
< <= > >=	Binary	Left to right
== !=	Binary	Left to right
& (bitwise AND)		Left to right
^ (bitwise XOR)		Left to right
 (bitwise OR)		Left to right
&&	Binary	Left to right
	Binary	Left to right
= *= /= %= += -=	Binary	Right to left
		Left to right

Setting bits

- **Setting a bit.**

```
i = 0x0000; // 0000 0000 0000 0000
```

```
i |= 0x0010; // 0000 0000 0001 0000
```

 0000 0000 0001 0000

- 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 $1 \ll j$ is 0x0008.

Clearing bits

- **Clearing a bit.**

```
i = 0x00ff; // 0000 0000 1111 1111
```

```
i &= ~0x0010; // 0000 0000 1110 1111
```


0000 0000 0001 0000 ^{complement} → 1111 1111 1110 1111

- A statement that clears a bit whose position is stored in a variable:

- `i &= ~(1 << j); // clears bit j`

Testing bits

- **Testing a bit.**

- An if statement that tests whether bit 4 of i is set:

- `if (i & 0x0010) ; // tests bit 4` (lowest order bit is bit-0)

0000 0000 0001 0000



- A statement that tests whether bit j is set:

- `if (i & 1 << j) ; // tests bit j`

Do I need extra parentheses???

Information Packing

- Information can be PACKED by representing several data items within a single machine word using non-overlapping adjacent groups of bits

For PC-dos, the list of installed equipment (in the AX register):

bit 0:	1 if diskettes present
bit 1:	not used
bits 2,3:	system board RAM, 11 => 64K
bits 4,5:	video mode, 01 => 40-column color, 10 => 80-column color, 11 => monochrome
bits 6,7:	number of disk drives
bit 8:	DMA chip, 0 => installed
bits 9,10,11:	number of RS-232 ports
bit 12:	1 => game adapter installed
bit 13:	1 => serial printer (PC <i>jr</i> only)
bit 14,15:	number of printers

1111 1111 1110 1100

Bit 0: diskette



Information Packing

- Let check whether the video mode is 3 (means monochrome)

Bit 0: diskette



1111 1111 1110 1100

0000 1111 1111 1110

```
if ( (ax >> 4) & ~ (~0 << 2) ) == 3;
```

1111 1111 1111 1111

1111 1111 1111 1100

0000 0000 0000 0011

bit 0: 1 if diskettes present
 bit 1: not used
 bits 2,3: system board RAM, 11 => 64K
 bits 4,5: video mode, 01 => 40-column color,
 10 => 80-column color, 11 => monochrome
 bits 6,7: number of disk drives
 bit 8: DMA chip, 0 => installed
 bits 9,10,11: number of RS-232 ports
 bit 12: 1 => game adapter installed
 bit 13: 1 => serial printer (PC *jr* only)
 bit 14,15: number of printers

Bit-fields

- C provides a more convenient method for defining and accessing fields within a word than the use of the bit-manipulation operators.
- A **bit-field** is a set of **adjacent bits** within an implementation dependent storage unit, called a "word".
- C allows a special syntax in the **structure definition** to define **bit-fields** and assign names to them.
- The syntax for defining a bit-field is

struct name

```
{ type field-name : bit-length;
```

...

```
type field-name : bit-length; }
```

Type: Must be type int, signed or unsigned



Bit fields

```
struct
```

```
{  
    unsigned diskette      : 1;  
    unsigned unused        : 1;  
    unsigned sysboard_ram  : 2;  
    unsigned video         : 2;  
    unsigned disks         : 2;  
    unsigned dma_chip      : 1;  
    unsigned rs232_ports   : 3;  
    unsigned game_adapter  : 1;  
    unsigned serial_printer : 1;  
    unsigned printers      : 2;  
} ax;
```

```
if (ax.video == 3) ...
```

Bit fields

- declares a field that occupies exactly **bit-length** bits
- can be declared only inside a struct
- exact ordering of bits is compiler-dependent
- can't make pointers to them; not directly addressable Can't apply addressOf (&) operator to bit-fields
- Not dimensional, i.e. array of bit fields is not possible
- Structure may also contain non-bit-field members
- Almost everything is implementation dependent
 - Not portable
 - Use in cases in which memory is scarce or externally defined data structures must be matched exactly