S&T2024

Computer Programming

(Part 2 – Advanced C Programming Language)

Chapter 2

Lecturer
A/Prof Tay Seng Chuan

E-mail: pvotaysc@nus.edu.sg

Office of the Provost
National University of Singapore

Ground Rules

- Switch off your handphone and pager
- No talking while lecture is going on
- No gossiping while the lecture is going on
- Raise your hand if you have question to ask
- Be on time for lecture
- Be on time to come back from the recess break to continue the lecture
- Bring your printed lecture notes to lecture or use a laptop for the e-copy. Do not use handphone to read the pdf file.
- Do not open any app, except this pdf file and an editing tool (to take notes).

Chapter 2
Bit Operations

1. Decimal System

Example : N = 152_{10}.

- Subscript "10" means base 10.
- Each character is a digit with a corresponding weight.

$N = 1 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 = 152_{10}$

2. Binary System

Computers are built from devices which behave like switches having only two possible states (and not ten possible states), i.e. OFF or ON, which may be represented as either 0 or 1. Thus computers use the binary system (base 2) to represent numbers.

- Weights are powers of two instead of powers of ten.

Example :

$$\begin{aligned} N &= 110_2 \\ &= 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 4 + 2 + 0 \\ &= 6_{10} \end{aligned}$$

Decimal Sum

carry digits: **11**

carry: 11

```
  137
+  69
-----
  206
```

Binary Sum

Addition Rules:

$0_2 + 0_2 = 0_2$,

$1_2 + 0_2 = 1_2$,

$0_2 + 1_2 = 1_2$,

$1_2 + 1_2 = 10_2$

Binary Addition:

carry digits: 11

carry: 11

```
   1110
 +  101
 ------
  10011
```

Subtraction in base 2 involves borrowing *twos* :

(Actually, we borrow one, but its weight is 2)

carry: 22

carry digits: -1-1

```
  100
 - 11
 ----
  001
```

Data Representation used in Computer

- Computers use the binary system to represent numbers internally.
- Binary digits are also called **BIT**s. Bits may exist in the computer as *electrical voltages*, for example +5 Volts might be used to represent a binary 1, and -5 Volts may be used to represent a binary 0. Alternatively they may exist as *charge on capacitors*, a charged capacitor representing a binary 1, and an uncharged capacitor representing a binary 0.
- Bits are stored in groups of 8, called Byte.
- Word varies from compiler/computer to the others.
  In DEV C and Visual C, 1 word = 4 bytes. In Turbo C, 1 word = 2 bytes.
- By our convention the computer uses the leftmost binary digit to determine the sign (+ or -) of a number: the number being positive if the leftmost bit is zero and negative if the leftmost bit is a one.
- **Suppose** our computer uses 8 bits (i.e., 1 byte) to represent integers. In this course, **the 8 bits are numbered 0 through 7 from right to left in our convention**! Bit number 7 is the Most Significant Bit (MSB) and bit number 0 is the Least Significant Bit (LSB).

MSB LSB

7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	1

The number represented here is positive (because bit-7 is 0), and has the value:

MSB LSB

7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	1

$1 \times 2^6 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0$
$= 64 + 8 + 4 + 1$
$= 77_{10}$

- The biggest positive integer that may be represented using 8-bit two's complement notation is $+127_{10}$ (01111111_2).

- What is the most negative number represented by this notation ? We will see.

How to represent -77_{10} as a binary byte?

We use two's complement notation for -ve numbers.

- First, take the binary representation of 77 and toggle all the 1s to 0s, and all the 0s to 1s to obtain the one's complement notation.

  So,

  $+77_{10} = 01001101_2$

  becomes,

  $10110010_{(1's)}$ (after toggling all bits)

- Next, add 1 to the one's complement notation to obtain the two's complement notation.

  Now, the number becomes

MSB							LSB
7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	1

So,

$-77_{10} = 10110011_{(2's)}$ in two's complement notation.

How does computer calculate +77 + (-77) ?

carry: 11111111

```
   01001101 (+77)
+  10110011 (-77)
  100000000
```

Binary digit is carried over into position number 8, but it doesn't matter because we only use 8 bits (bits 0 to 7) !!!

Decimal Value of a Two's Complement Notations

For a two's complement 8-bit integer, the decimal value can be computed as follows :

In general, for a n-bit integer,

N = [number represented by bits 0 to (n-2)] - [bit (n-l) x 2^{n-1}]

MSB							LSB
n-1	n-2	..	..	3	2	1	0
1	0	1	1	0	0	1	1

$10110011_2 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^1 + 1 \times 2^0 - 1 \times 2^7$

$= 32_{10} + 16_{10} + 2_{10} + 1_{10} - 128_{10}$

$= 51_{10} - 128_{10} = -77_{10}$

N = (number represented by bits 0 to 6) - (bit 7 x 128)

n-1	n-2	..	..	3	2	1	0
1	0	1	1	0	0	1	1

So, the most negative number we can represent is -128 (-2^7) , i.e., 10000000_2 for a 8-bit integer.

Unsigned Integer

If we assumed that numbers were always positive, we could use the MSB to represent value and

our biggest number would be 11111111_2 = 255_{10} (i.e., $2^8 - 1$),

the smallest value is 00000000_2 = 0, and

the number is called an **unsigned integer** because the MSB is no longer used as a sign bit. Now the **MSB carries a weight for unsigned integer**.

By default, integer is signed, unless stated otherwise.

Overflow Problems in Arithmetic for Signed Integer

Assume 8-bit integer:

```
carry :   111
          01110000 (112₁₀)
        + 00110000 (+48₁₀)
          10100000
          ↖Sign bit is set!!!
```

According to our two's complement notation,
the result is negative !! *Why and what is the cause ???*

Another way to run into trouble is to add together two negative numbers whose sum is less than -128_{10} .

For instance $(-111_{10}) + (-39_{10})$:

```
carry : 1   1    1
          10010001 (-111₁₀ )
         +11011001 (- 39₁₀ )
          01101010
          ↖Sign bit is not set
```

The result is positive !!!
CARRY bit and **OVERFLOW** bit are use to monitor addition.

- The carry bit acts as bit number 8 and catches any carry over from bit number 7.
- In the last example, after the computer had performed the addition the carry bit would be set (1). If there is no carry out from bit 7 the carry bit will be clear (0).
- The overflow bit is there to tell the computer when an addition has gone wrong i.e., when the result cannot fit in the byte it overflows.

The following two digits are used to set the overflow bit :
(i) the carry digit (C_7) from bit 7 (sign bit) to the carry bit
(ii) the carry digit (C_6) from bit 6 to bit 7

C_7 from Bit-7	C_6 from Bit-6	Overflow Bit
0	0	0
0	1	1
1	0	1
1	1	0

The relation is known as Exclusive-OR (XOR), meaning "one ***and only*** one". The overflow bit is the XOR of C_6 and C_7. If the overflow bit is set after an addition, the sum cannot be represented using only 8 bits.
To understand, consider the 2-bit two's complement notations with small numbers: 00 (0), 01 (1), 10 (-2), 11 (-1), ie, $-2_{(10)}$ to $1_{(10)}$.

Carry Bit:	0 0	0 1	1 0	1 1
	0 0	0 1	1 0	1 1
	0 1	0 1	1 1	1 1
	0 1	1 0	0 1	1 0

Example for case 1 :

$64_{10} + 32_{10} = 96_{10}$ (no overflow)

$C_6 = 0$

$C_7 = 0$

$$
\begin{array}{rl}
01000000 & (\ 64_{10}) \\
+\ 00100000 & (\ 32_{10}) \\
\hline
01100000 & (+96_{10})
\end{array}
$$

Example for case 2 :

$64_{10} + 64_{10} = 128_{10} > 127_{10}$ (overflow)

$C_6 = 1$

$C_7 = 0$

$$
\begin{array}{rl}
01000000 & (\ 64_{10}) \\
+\ 01000000 & (\ 64_{10}) \\
\hline
10000000 & (\ 128_{10})
\end{array}
$$

sign bit is set

Example for case 3 :

$(-64_{10}) + (-65_{10}) = -129_{10} < -128_{10}$ (overflow)

$C_6 = 0$

$C_7 = 1$

$$
\begin{array}{rl}
11000000 & (-\ 64_{10}) \\
+\ 10111111 & (-\ 65_{10}) \\
\hline
01111111 & (-\ 129_{10})
\end{array}
$$

sign bit is not set

Example for case 4 :

$(-2_{10}) + (-2_{10}) = -4_{10}$ (no overflow)

$C_6 = 1$

$C_7 = 1$

$$
\begin{array}{rl}
11111110 & (\ -2_{10}) \\
+\ 11111110 & (\ -2_{10}) \\
\hline
11111100 & (\ -4_{10})
\end{array}
$$

3. Octal System

$135_8 = 1 \times 8^2 + 3 \times 8^1 + 5 \times 8^0$

$= 64_{10} + 24_{10} + 5_{10}$

$= 93_{10}$

Convert from Binary to Octal

Group of 3 bits !!

$011\,010\,101_2 = 325_8$

4. Hexadecimal System

Digits : 0 1 2 3 4 5 6 7 8 9 A B C D E F

(Small letter a b c d e f can be used.)

$AB6_{16} = 10 \times 16^2 + 11 \times 16^1 + 6 \times 16^0$

$= 2560_{10} + 176_{10} + 6_{10}$

$= 2742_{10}$

Conversion of Decimal Numbers to Hexadecimal Numbers

Example : $106_{10} = ??_{16}$

106 / 16 = 6 remainder 10_{10} (A_{16})

6 / 16 = 0 remainder 6_{10} (6_{16})

i.e. $106_{10} = 6A_{16}$

Binary to Hexadecimal

Group of 4 bits

$1010\,1110\,1111\,0011_2 = AEF3_{16}$

If the number of digits in the binary number is not a multiple of four, we must append the leading zeros to the left hand side. Thus

$11011_2 = \underline{000}1\,1011_2 = 0001\ 1011_2 = 1B_{16}$

When adding and subtracting hexadecimal numbers remember to carry or borrow sixteens!

ASCII Codes

. ASCII is only a 7 bit code, for 128 characters, the 8th bit (bit-7) is used for error checking during data transmission.
. IBM have extended the ASCII codes to include a special set of characters.

	MSBs							
LSBs	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC_1	!	1	A	Q	a	q
0010	STX	DC_2	“	2	B	R	b	r
0011	ETX	DC_3	#	3	C	S	c	s
0100	EOT	DC_4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	O	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Character	ASCII Code
0	0110000
1	0110001
. . .	. . .
9	0111001
:	0111010
A	1000001
B	1000010
. . .	. . .
Z	1011010
[	1011011
\	1011100

The first 32 characters are known as control characters. These are used to control some action of the computer and are not printable characters like the remaining 224 characters. Some points to notice about the table are:

- The decimal digits 0 to 9 have ASCII codes in the range 48_{10} to 57_{10} or 30_{16} to 39_{16} which may be easier to remember because the last digit of the code in hexadecimal is the same as the character represented).

- Uppercase (capital) letters have codes in the range 65_{10} to 90_{10} (i.e. 41_{16} to $5A_{16}$). Lowercase (small) letters have codes in the range 97_{10} to $l22_{10}$ (i.e. 61_{16} to $7A_{16}$). To convert an uppercase character to a lower case character we need to add 32_{10} (20_{16}) to the ASCII code. To convert lowercase to uppercase we would subtract 32_{10} (20_{16}) from the ASCII code, i.e., 65+32 = 97.

‘A’ = 65, ‘a’=97

65 66 97 98

A B ? a b ?*