

Chapter 2 primitive data type and operator

Speaker: Lung-Sheng Chien

OutLine

- Basic data type
 - integral
 - character and string
 - floating point
- Operator
- Type conversion

Fundamental type [1]

Page 36, section 2.2 in textbook

type	description
char	A single byte capable of holding one character in the local character set
int	An integer, typically reflecting the natural size of integers on host machine
float	Single-precision floating point
double	Double-precision floating point

Fundamental type [2]

Page 36, section 2.2 in textbook

qualifier	description
short	Length of “short int” is less than “int”
long	Length of “long int” is larger than “int”
signed	Signed number can be positive, zero or negative
unsigned	Unsigned number is always positive or zero

Search “fundamental type” in MSDN Library [1]

Category	Type	Contents
Integral	char	Type char is an integral type that usually contains members of the execution character set — in Microsoft C++, this is ASCII.
		The C++ compiler treats variables of type char , signed char , and unsigned char as having different types. Variables of type char are promoted to int as if they are type signed char by default, unless the <code>/J</code> compilation option is used. In this case they are treated as type unsigned char and are promoted to int without sign extension.
	bool	Type bool is an integral type that can have one of the two values true or false . Its size is unspecified.
	short	Type short int (or simply short) is an integral type that is larger than or equal to the size of type char , and shorter than or equal to the size of type int .
		Objects of type short can be declared as signed short or unsigned short . Signed short is a synonym for short .
	int	Type int is an integral type that is larger than or equal to the size of type short int , and shorter than or equal to the size of type long .
		Objects of type int can be declared as signed int or unsigned int . Signed int is a synonym for int .
	__intn	Sized integer, where <i>n</i> is the size, in bits, of the integer variable. The value of <i>n</i> can be 8, 16, 32, or 64. (__intn is a Microsoft-specific keyword.)
	long	Type long (or long int) is an integral type that is larger than or equal to the size of type int .
		Objects of type long can be declared as signed long or unsigned long . Signed long is a synonym for long .
	long long	Larger than an unsigned long .
		Objects of type long long can be declared as signed long long or unsigned long long . Signed long long is a synonym for long long .

Search “fundamental type” in MSDN Library [2]

Floating	float	Type float is the smallest floating type.
	double	Type double is a floating type that is larger than or equal to type float , but shorter than or equal to the size of type long double . ¹
	long double ¹	Type long double is a floating type that is equal to type double .
Wide-character	__wchar_t	A variable of __wchar_t designates a wide-character or multibyte character type. By default, wchar_t is a native type but you can use /Zc:wchar_t- to make wchar_t a typedef for unsigned short . Use the L prefix before a character or string constant to designate the wide-character-type constant.

The following table lists the amount of storage required for fundamental types in Microsoft C++.

Sizes of Fundamental Types

Type	Size
bool	1 byte
char, unsigned char, signed char	1 byte
short, unsigned short	2 bytes
int, unsigned int	4 bytes
__intn	8, 16, 32, 64, or 128 bits depending on the value of <i>n</i> . __intn is Microsoft-specific.
long, unsigned long	4 bytes
float	4 bytes
double	8 bytes
long double ¹	8 bytes
long long	Equivalent to __int64 .

Search “integer limit” in MSDN Library

The limits for integer types are listed in the following table. These limits are also defined in the standard header file `LIMITS.H`.

Limits on Integer Constants

Constant	Meaning	Value
CHAR_BIT	Number of bits in the smallest variable that is not a bit field.	8
SCHAR_MIN	Minimum value for a variable of type signed char .	- 128
SCHAR_MAX	Maximum value for a variable of type signed char .	127
UCHAR_MAX	Maximum value for a variable of type unsigned char .	255 (0xff)
CHAR_MIN	Minimum value for a variable of type char .	- 128; 0 if /J option used
CHAR_MAX	Maximum value for a variable of type char .	127; 255 if /J option used
MB_LEN_MAX	Maximum number of bytes in a multicharacter constant.	5
SHRT_MIN	Minimum value for a variable of type short .	- 32768
SHRT_MAX	Maximum value for a variable of type short .	32767
USHRT_MAX	Maximum value for a variable of type unsigned short .	65535 (0xffff)
INT_MIN	Minimum value for a variable of type int .	- 2147483648
INT_MAX	Maximum value for a variable of type int .	2147483647
UINT_MAX	Maximum value for a variable of type unsigned int .	4294967295 (0xffffffff)
LONG_MIN	Minimum value for a variable of type long .	- 2147483648

Compare table in MSDN with numbers in page 257 of textbook

Search “float limit” in MSDN Library

The following table lists the limits on the values of floating-point constants. These limits are also defined in the standard header file `FLOAT.H`.

Limits on Floating-Point Constants

Constant	Meaning	Value
FLT_DIG DBL_DIG LDBL_DIG	Number of digits, q , such that a floating-point number with q decimal digits can be rounded into a floating-point representation and back without loss of precision.	6 15 15
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	Smallest positive number x , such that $x + 1.0$ is not equal to 1.0.	1.192092896e-07F 2.2204460492503131e-016 2.2204460492503131e-016
FLT_GUARD		0
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	Number of digits in the radix specified by FLT_RADIX in the floating-point significand. The radix is 2; hence these values specify bits.	24 53 53
FLT_MAX DBL_MAX LDBL_MAX	Maximum representable floating-point number.	3.402823466e+38F 1.7976931348623158e+308 1.7976931348623158e+308
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	Maximum integer such that 10 raised to that number is a representable floating-point number.	38 308 308
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	Maximum integer such that FLT_RADIX raised to that number is a representable floating-point number.	128 1024 1024
FLT_MIN DBL_MIN LDBL_MIN	Minimum positive value.	1.175494351e-38F 2.2250738585072014e-308 2.2250738585072014e-308
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	Minimum negative integer such that 10 raised to that number is a representable floating-point number.	- 37 - 307 - 307

Compare table in MSDN with numbers in page 257 of textbook

OutLine

- Basic data type
 - integral
 - character and string
 - floating point
- Operator
- Type conversion

What is integer

In real world, we use integer based 10 = $\pm(a_0 10^0 + a_1 10^1 + a_2 10^2 + \dots + a_n 10^n)$

In computer, we use integer based 2 = $\pm(b_0 2^0 + b_1 2^1 + b_2 2^2 + \dots + b_n 2^n)$

Question 1: why choose base 2, not base 10 in computer?

Question 2: how to transform between base 2 and base 10?

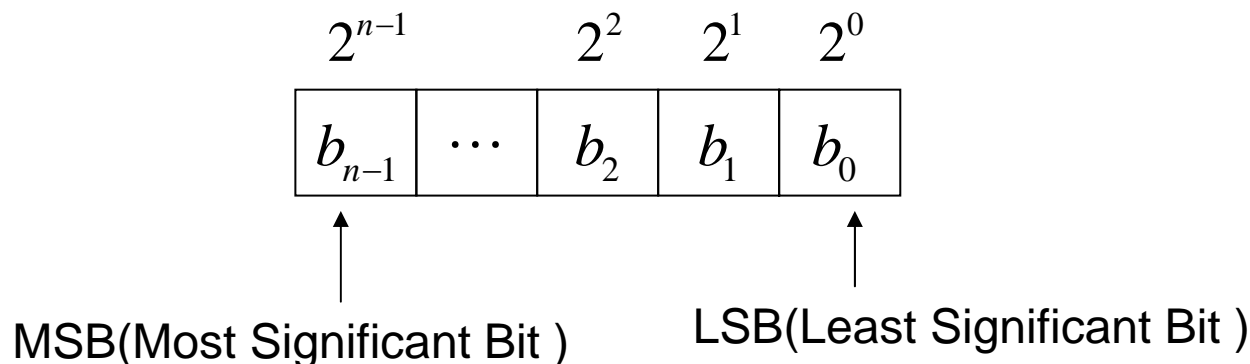
Question 3: the range of exponent n ?

Question 4: How to represent negative number in base 2 (2's complement)?
do we need to represent sign in computer?

Question 1: why choose base 2?

- Basic unit in computer in bit = $\{0,1\}$ (位元). We use electronic device to represent a bit, say 0:discharge(放電), 1:charge(充電). Such representation has high tolerance during fluctuation of voltage.
- Simple to evaluation, since $0+0=0$, $0+1=1$, $1+0=1$, $1+1=0$ (a carry), we can use boolean algebra to implement adder (加法器).

Format of unsigned integer



Question 2: how to transform between base 2 and base 10?

For simplicity, we choose $n = 4$, consider $7 = b_0 2^0 + b_1 2^1 + b_2 2^2 + b_3 2^3$

$$b_0 = 7 \bmod 2 = 1$$

$$b_1 = \frac{7 - b_0 2^0}{2} \bmod 2 = \frac{6}{2} \bmod 2 = 3 \bmod 2 = 1$$

$$b_2 = \frac{7 - (b_0 2^0 + b_1 2^1)}{4} \bmod 2 = \frac{4}{4} \bmod 2 = 1 \bmod 2 = 1$$

$$b_3 = \frac{7 - (b_0 2^0 + b_1 2^1 + b_2 2^2)}{8} \bmod 2 = 0 \bmod 2 = 0$$

$$7 = 0111_2 = \begin{array}{c} 2^3 \quad 2^2 \quad 2^1 \quad 2^0 \\ \boxed{0} \quad \boxed{1} \quad \boxed{1} \quad \boxed{1} \end{array}$$

Question 3: the range of n ?

Consider unsigned integer, maximum occurs when $b_j = 1 \quad \forall j$

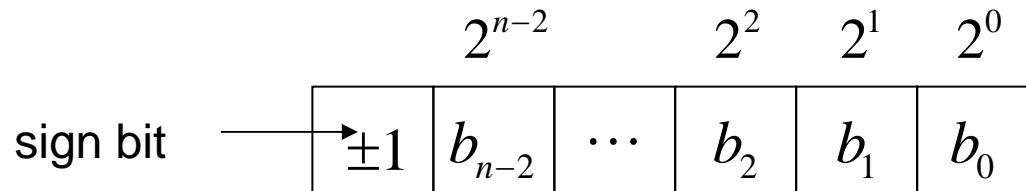
$$\text{maximum} = 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1} = 2^n - 1$$

n	Bytes	Maximum value
8	1	255
16	2	65535
32	4	4294967295

Byte (位元組) = 8 bits is unit of type where bit is basic unit in computation

Question 4: How to represent negative number in base 2?

$$\text{Magnitude representation} = \text{sign} + \text{number} = \pm \sum_{k=0}^{n-2} b_k 2^k$$



$$+7 = 0111_2 =$$

0	1	1	1
---	---	---	---

Weights: + 2^2 2^1 2^0

$$-6 = 1101_2 =$$

1	1	0	1
---	---	---	---

Weights: - 2^2 2^1 2^0

Drawback: we have two representations for zero

$$+0 = 0000_2 =$$

0	0	0	0
---	---	---	---

Weights: + 2^2 2^1 2^0

$$-0 = 1000_2 =$$

1	0	0	0
---	---	---	---

Weights: - 2^2 2^1 2^0

Not good

2's complement representation

Magnitude representation

$$x = \begin{array}{c} 2^{n-2} \quad 2^2 \quad 2^1 \quad 2^0 \\ \boxed{+} \quad \boxed{b_{n-2}} \quad \boxed{\cdots} \quad \boxed{b_2} \quad \boxed{b_1} \quad \boxed{b_0} \end{array}$$

2's complement representation

$$x_{2's} = \begin{array}{c} 2^{n-2} \quad 2^2 \quad 2^1 \quad 2^0 \\ \boxed{0} \quad \boxed{b_{n-2}} \quad \boxed{\cdots} \quad \boxed{b_2} \quad \boxed{b_1} \quad \boxed{b_0} \end{array}$$

$x = \begin{array}{c} 2^{n-2} \quad 2^2 \quad 2^1 \quad 2^0 \\ \boxed{-} \quad \boxed{b_{n-2}} \quad \boxed{\cdots} \quad \boxed{b_2} \quad \boxed{b_1} \quad \boxed{b_0} \end{array}$
→

$\begin{array}{c} 2^n \quad 2^{n-1} \quad 2^{n-2} \quad 2^2 \quad 2^1 \quad 2^0 \\ \boxed{1} \quad \boxed{0} \quad \boxed{0} \quad \boxed{\cdots} \quad \boxed{0} \quad \boxed{0} \quad \boxed{0} \end{array}$
 $-$
 $\begin{array}{c} \boxed{0} \quad \boxed{b_{n-2}} \quad \boxed{\cdots} \quad \boxed{b_2} \quad \boxed{b_1} \quad \boxed{b_0} \end{array}$

 $x_{2's} = \begin{array}{c} \boxed{1} \quad \boxed{c_{n-2}} \quad \boxed{\cdots} \quad \boxed{c_2} \quad \boxed{c_1} \quad \boxed{c_0} \end{array}$

if $x < 0$, then $x_{2's} = 2^n - |x|$

Another computation of 2's complement

$$x_{2's} = 2^n - |x| = (2^n - 1 - |x|) + 1 = x_{1's} + 1$$

$$\bar{b}_j = \begin{cases} 1 & \text{if } b_j = 0 \\ 0 & \text{if } b_j = 1 \end{cases}$$

is complement of b_j

$$2^n - 1 = \begin{array}{cccccc} 2^{n-1} & 2^{n-2} & & 2^2 & 2^1 & 2^0 \\ \boxed{1} & \boxed{1} & \boxed{\cdots} & \boxed{1} & \boxed{1} & \boxed{1} \end{array}$$

$$- \begin{array}{cccccc} \boxed{0} & \boxed{b_{n-2}} & \boxed{\cdots} & \boxed{b_2} & \boxed{b_1} & \boxed{b_0} \end{array}$$

$$x_{1's} = \begin{array}{cccccc} \boxed{1} & \boxed{\bar{b}_{n-2}} & \boxed{\cdots} & \boxed{\bar{b}_2} & \boxed{\bar{b}_1} & \boxed{\bar{b}_0} \end{array}$$

1's complement of x

$$+ \begin{array}{cccccc} & & & & & \boxed{1} \end{array}$$

$$x_{2's} = \begin{array}{cccccc} \boxed{1} & \boxed{c_{n-2}} & \boxed{\cdots} & \boxed{c_2} & \boxed{c_1} & \boxed{c_0} \end{array}$$

Example: 2's complement of -5

$$\begin{array}{r} \\ 2^3 2^2 2^1 2^0 \\ 2^4 - 1 = \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 1 \\ \hline \end{array} \\ - \begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 1 \\ \hline \end{array} = 5 \end{array}$$

$$\begin{array}{r} (-5)_{1's} = \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \end{array} \\ + \phantom{(-5)_{1's} =} \phantom{(-5)_{1's} =} \phantom{(-5)_{1's} =} \begin{array}{|c|} \hline 1 \\ \hline \end{array} \end{array}$$

$$(-5)_{2's} = \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 1 \\ \hline \end{array}$$

Exercise : under 2's complement, representation of 0 is unique

Table of 2's complement versus decimal

signed integer of 2's complement ranges from -8 to 7

decimal	2's (binary)	decimal	2's (binary)
0	0000	-8	1000
1	0001	-7	1001
2	0010	-6	1010
3	0011	-5	1011
4	0100	-4	1100
5	0101	-3	1101
6	0110	-2	1110
7	0111	-1	1111

Exercise: signed integer of 2's complement ranges from -2^{n-1} to $2^{n-1} - 1$

Integer limit from MSDN Library, how can we confirm it?

Type	Bytes	Minimum value	Maximum value
(signed) short (int)	1	SHRT_MIN	SHRT_MAX
(signed) int	2	INT_MIN	INT_MAX
(signed) long (int)	4	LONG_MIN	LONG_MAX

Question 1: how to determine size of data type for different machines?

Question 2: how to determine limit of range of the data type?

Question 1: how to determine size of data type?

- C provide a compile-time unary operator, **sizeof**, that can be used to compute size of any object (data type)
format: **sizeof** (type name)
example: **sizeof** (int)
- Primitive types in C and C++ are implementation defined (that is, not precisely defined by the standard), so we need compiler to determine actual size of data type.
- “**sizeof**” is followed by a **type name**, **variable**, or expression and return unsigned value which is equal to size of data type in bytes.
see page 204 in textbook

Question 2: how to determine limit of the data type?

signed integer of 2's complement under 4 bits range from -8 to 7

However if we add 7 by 1, then it becomes -8, not +8, so 7 is maximum

$$\begin{array}{r} \begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ 7 = & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} \\ + & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} \\ \hline (-8)_{2's} = & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} \end{array} \end{array}$$

Show size of type “short” and its limit

```
#include <stdio.h>
#include <limits.h>

int main( int argc, char *argv[] )
{
    short    x_sint ; // x_sint is signed short integer

    printf("size of short = %d bytes\n", sizeof( short ) );    '%d' means to print integer

    x_sint = SHRT_MAX ; // SHRT_MAX is defined in file limits.h
    printf("maximum of short = %d\n", x_sint );

    x_sint = SHRT_MAX + 1 ;

    printf("maximum(short) + 1 = %d\n", x_sint );

    printf("maximum(short) + 1 = %d\n", SHRT_MAX + 1 );

    return 0 ;
}
```

```
size of short = 2 bytes
maximum of short = 32767
maximum(short) + 1 = -32768
maximum(short) + 1 = 32768
Press any key to continue
```

This shows SHRT_MAX is maximum of short

Why this number is not -32768

Beyond integer

- If someone want to compute prime number, for example, up to 100 decimal representation of integer, we cannot use type `int`, why?
- How can we do to compute large prime number? (if someone is interested in this topic, take it as a project)

$$\text{Limits of type int: } 2^{32} = 2^3 \cdot (2^{10})^3 \sim 8 \cdot (10^3)^3 \sim 10^{10}$$

OutLine

- Basic data type
 - integral
 - character and string
 - floating point
- Operator
- Type conversion

ASCII code

- American Standard Code for Information Interchange, 美國信息互換標準代碼
- Map integers to symbols since computer only stores 0/1, symbols are just interpretation by human being.
- 定義128個字元 (7 bits), 其中33個字元無法顯示(在DOS下為笑臉, 撲克牌花式等等), 另外95個為可顯示字元, 代表英文字母或數字或符號 (鍵盤上的符號)
- EASCII (Extended ASCII) 將ASCII code 由7 bits 擴充為8 bits, 包括表格符號, 計算符號, 希臘字母和特殊的拉丁符號

ASCII code, Hexadecimal arrangement

From <http://www.jimprice.com/jim-asc.shtml>

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

A-Z occupy 0x41 ~ 0x5A, monotone increasing

a-z occupy 0x61 ~ 0x7A, monotone increasing

0-9 occupy 0x30 ~ 0x39, monotone increasing

ASCII code Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	:	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Extended ASCII code

128	Ç	144	É	161	í	177	⌘	193	⊥	209	〒	225	β	241	±
129	ü	145	æ	162	ó	178	⌘	194	⊥	210	π	226	Γ	242	≥
130	é	146	Æ	163	ú	179		195	⊥	211	ℓ	227	π	243	≤
131	â	147	ô	164	ñ	180	⊥	196	—	212	ℓ	228	Σ	244	∫
132	ä	148	ö	165	Ñ	181	⊥	197	+	213	ƒ	229	σ	245	∫
133	à	149	ò	166	²	182	⊥	198	⊥	214	π	230	μ	246	÷
134	â	150	û	167	°	183	π	199	⊥	215	⊥	231	τ	247	≈
135	ç	151	ù	168	¿	184	⊥	200	ℓ	216	⊥	232	Φ	248	°
136	ê	152	—	169	—	185	⊥	201	π	217	⊥	233	⊕	249	.
137	ë	153	Ö	170	⊥	186	⊥	202	⊥	218	⊥	234	Ω	250	.
138	è	154	Û	171	½	187	π	203	π	219	■	235	δ	251	√
139	ï	156	£	172	¼	188	⊥	204	⊥	220	■	236	∞	252	—
140	î	157	¥	173	¡	189	⊥	205	=	221	■	237	φ	253	²
141	ï	158	—	174	«	190	⊥	206	⊥	222	■	238	ε	254	■
142	Ä	159	ƒ	175	»	191	⊥	207	⊥	223	■	239	∧	255	
143	Å	160	á	176	⌘	192	⊥	208	⊥	224	α	240	≡		

Source: www.LookupTables.com

Escape sequence

character	description	character	Description
\a	Alert (bell) character	\\	Backslash
\b	Backspace	\?	Question mark
\f	Formfeed (page break)	\'	Single quote
\n	Newline	\"	Double quote
\r	Carriage return	\ooo	Octal number
\t	Horizontal tab	\xhh	Hexadecimal number
\v	Vertical tab	\0	0

Question: what is corresponding integral value of escape sequence?

Exercise: How to find integral value of escape sequence

```
#include <stdio.h>

#define NUM_ESCAPE_CHAR 12    Symbolic constant

int main( int argc, char* argv[])
{
    int i ;
    char word[ NUM_ESCAPE_CHAR ] ; Declaration (宣告), for type checking

    word[0] = '\a' ; word[1] = '\b' ; word[2] = '\f' ; word[3] = '\n' ;
    word[4] = '\r' ; word[5] = '\t' ; word[6] = '\v' ; word[7] = '\\ ' ;
    word[8] = '\?' ; word[9] = '\'' ; word[10] = '\"' ; word[11] = '\0' ;

    for ( i = 0 ; i < NUM_ESCAPE_CHAR ; i++){
        printf("%c = 0x%x\n", word[i] , word[i] );
    }

    return 0 ;
}
```

1. Array “word” has 12 elements, each element is a character
2. We use single quote to define a character
3. “for-loop” executes $i=0,1,2,\dots,11$
4. **%c**: print character, **%x**: print hexadecimal value

#include <ctype.h>

Read page 248 ~ 249 and page 166 in textbook

isalnum(c)	isalpha(c) or isdigit(c) is true
isalpha(c)	isupper(c) or islower(c) is true
iscntrl(c)	Nonzero if c is control character, 0 if not
isdigit(c)	Nonzero if c is digit, 0 if not
isgraph(c)	printing character except space
islower(c)	Nonzero if c is lower case, 0 if not
isprint(c)	printing character including space
ispunct(c)	printing character except space or letter or digit
isspace(c)	space, formfeed, newline, carriage return, tab, vertical tab
isupper(c)	Nonzero if c is upper case, 0 if not
int tolower(int c)	convert c to lower case
int toupper(int c)	convert c to upper case

Character array (字元陣列)

Array index												
	0	1	2	3	4	5	6	7	8	9	10	11
word[12] =	\a	\b	\f	\n	\r	\t	\v	\\	\?	\'	\"	\0

- Array index in C starts from 0, array “word” has 12 elements, labeled as word[0], word[1], ..., word[11], value 11 is called array bound.
- Don't access excess of array bound, or either memory fault may happen and result is invalid. A common fault of programmers is to use word[12]

String constant (string literal) versus Character array

“hello, world” is called string constant, the quotes are not part of the string, just to delimit the string.



Corresponding character array

Array index	0	1	2	3	4	5	6	7	8	9	10	11	12
	h	e	l	l	o	,		w	o	r	l	d	\0

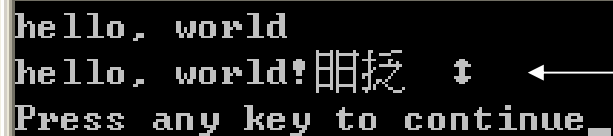
null character, to
terminate the string

Question: what happens if we remove ‘\0’ from the string?

Exercise: test string constant

```
#include <stdio.h>

int main( int argc, char* argv[])
{
    char p1[] = "hello, world" ; /* compiler would decide size of p1 */
    printf("%s\n", p1 ); /* %s : print string */
    p1[12] = '!' ; /* remove \0 of p1 */
    printf("%s\n", p1 );
    return 0 ;
}
```



```
hello, world
hello, world!
Press any key to continue_
```

It should be “hello, world!”, why?

#include <string.h>

Read page 249 ~ 250 and page 166 in textbook

char* <i>strcpy</i> (s,ct)	copy string ct to string s, including '\0', return s
char* <i>strcat</i> (s,ct)	concatenate string ct to end of string s, return s
int <i>strcmp</i> (cs,ct)	compare string cs to string ct, return <0 if cs < ct, return 0 if cs ==ct, or return >0 if cs>ct
size_t <i>strlen</i> (cs)	return length of cs
void * <i>memcpy</i> (s,ct,n)	copy n characters from ct to s, return s
void* <i>memset</i> (s,c,n)	place character c into first n characters of s, return s

Example of strcat in MSDN Library

strcat, wcsat Search

URL: ms-help://MS.MSDN.QTR.v80/en/MS.MSDN.v80/MS.WINCE.v50/en/wceappdev5/html/wce50hfstcatcmawcsat.htm

☒ Platform Builder for Microsoft Windows CE 5.0

search for strcat

strcat, wcsat

Append a string.

```
char *strcat( char *strDestination, const char *strSource );  
wchar_t *wcsat( wchar_t *strDestination, const wchar_t *strSource );
```

Example

```
/* STCPY.C: This program uses strcpy  
 * and strcat to build a phrase.  
 */  
  
#include <string.h>  
#include <stdio.h>  
  
void main( void )  
{  
    char string[80];  
    strcpy( string, "Hello world from " );  
    strcat( string, "strcpy " );  
    strcat( string, "and " );  
    strcat( string, "strcat!" );  
    printf( "String = %s\n", string );  
}
```

Why declare character array with 80 elements?

Output

String = Hello world from strcpy and strcat!

OutLine

- Basic data type
 - integral
 - character and string
 - floating point
- Operator
- Type conversion

Fixed point versus floating point [1]

- Suppose fixed-point representation has 8 decimal digits, with radix point (小數點) positioned after 6-th digit.
Example: 123456.78, 8765.43, 123.00
- Under 8 decimal digits, floating point representation can be 1.2345678, 1234567.8, 0.000012345678 or 1234567800000
- Floating point representation supports wider range of values than fixed-point representation, however it needs more storage to encode the position of radix point.
- Floating point representation is in scientific notation.

Fixed point versus floating point [2]

Consider a decimal floating-point system with 4 digits and 3 digits after radix point.

Theoretical: $0.12 \times 0.12 = 0.0144$ requires 4 digits after radix point to keep accuracy

Fixed-point: $0.120 \times 0.120 = 0.014$ losses one-digit accuracy

Floating-point: $(1.2 \times 10^{-1}) \times (1.2 \times 10^{-1}) = 1.44 \times 10^{-2}$

maintains the same accuracy and only use 2-digits after radix point

IEEE 754: standard for binary floating-point arithmetic

- Single-precision (32-bit)
- Double-precision (64-bit)
- Single-extended precision (> 42-bit, not commonly used)
- Double-extended precision (80-bit), used in Intel CPU



$$v = s \times 2^E \times m$$

$$s = \pm : \text{sign bit} \quad E = \text{exp} - N$$



Excess-N biased

$$m = 1.b_1b_2 \cdots \quad \text{normalized}$$

$$m = 0.b_1b_2 \cdots \quad \text{denormalized}$$

Excess-N biased

The exponent is biased by $N = 2^M - 1$

where M is number of bits in exponent field

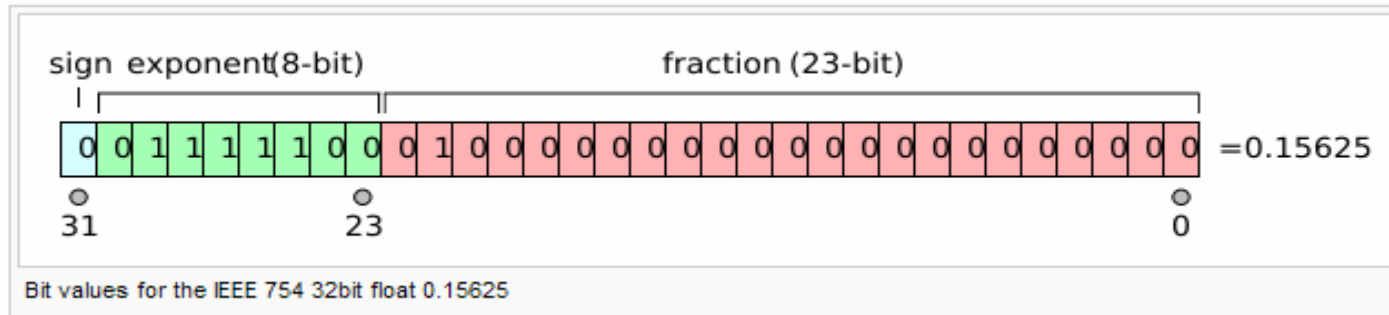
Single precision: $M = 8$, $N = 2^7 - 1 = 127$

double precision: $M = 11$, $N = 2^{10} - 1 = 1023$

8 bit excess-127		
Binary value	Excess-127 interpretation	Unsigned interpretation
00000000	-127	0
00000001	-126	1
...
01111111	0	127
10000000	+1	128
...
11111111	+128	255

This is different from 2's complement

Single precision (32-bit, 4 bytes)



$$s = 0 : v > 0$$

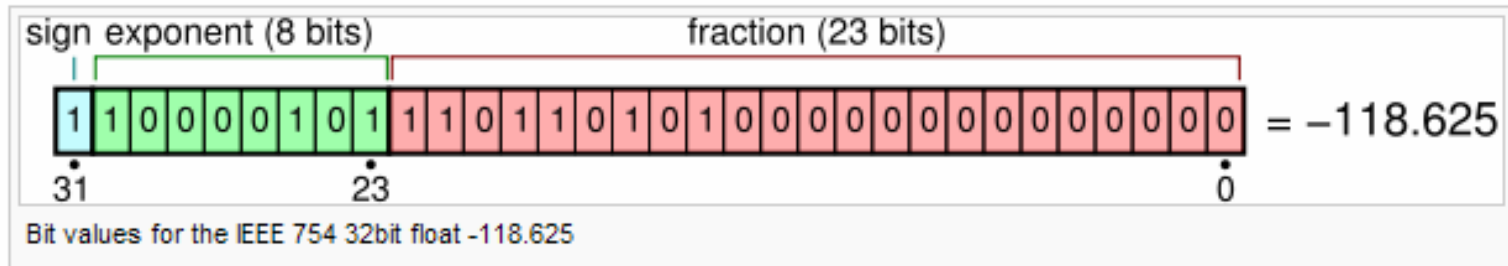
$$\text{exp} = 01111100 = 0x7C = 7 \times 16 + 12 = 124 \quad N = \text{exp} - 127 = -3$$

$$m = 1.01 = 1 + \frac{1}{4} = 1.25$$

$$\text{Normalized value} \quad v = s \times 2^E \times m = +1.25 \times 2^{-3} = +0.15625$$

Exercise: example of single precision

Check the configuration has normalized decimal value -118.625



Limits of single precision

Type	Sign	Exp	Exp+Bias	Exponent	Significand	Value
Zero	0	-127	0	0000 0000	000 0000 0000 0000 0000 0000	0.0
One	0	0	127	0111 1111	000 0000 0000 0000 0000 0000	1.0
Minus One	1	0	127	0111 1111	000 0000 0000 0000 0000 0000	-1.0
Smallest denormalized number	*	-127	0	0000 0000	000 0000 0000 0000 0000 0001	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	-127	0	0000 0000	100 0000 0000 0000 0000 0000	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	-127	0	0000 0000	111 1111 1111 1111 1111 1111	$\pm (1 - 2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	-126	1	0000 0001	000 0000 0000 0000 0000 0000	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Largest normalized number	*	127	254	1111 1110	111 1111 1111 1111 1111 1111	$\pm (2 - 2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	128	255	1111 1111	000 0000 0000 0000 0000 0000	$+\infty$
Negative infinity	1	128	255	1111 1111	000 0000 0000 0000 0000 0000	$-\infty$
Not a number	*	128	255	1111 1111	non zero	NaN

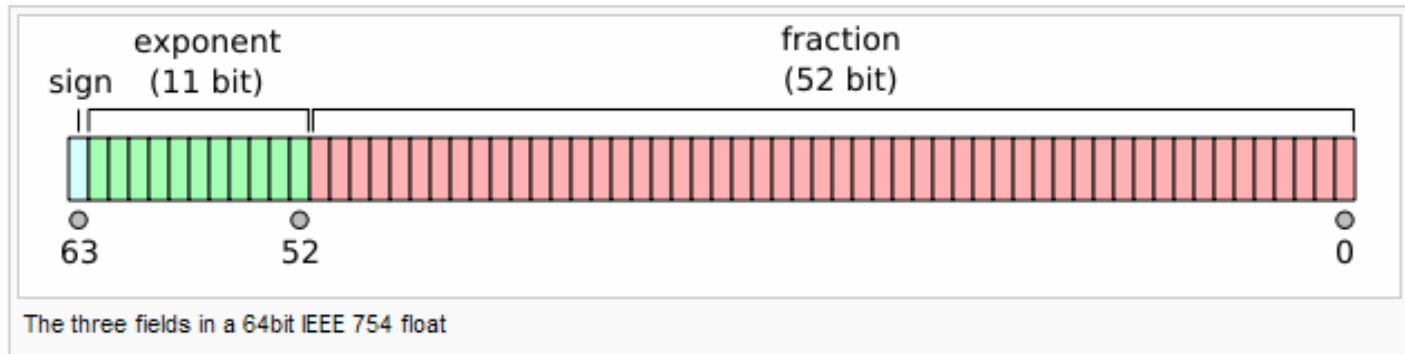
* Sign bit can be either 0 or 1 .

Exponent ranges from -126 to +127

Extreme of exponents are used to represent 0, NaN (not a number) and infinity

$$NaN = \frac{0}{0}, \frac{\infty}{\infty}, \frac{\infty}{-\infty}, \frac{-\infty}{\infty}, \frac{-\infty}{-\infty}, \infty + (-\infty), \infty - \infty, \sqrt{-1}, \log(-2), \sin^{-1}(1.1)$$

double precision (64-bit, 8 bytes)



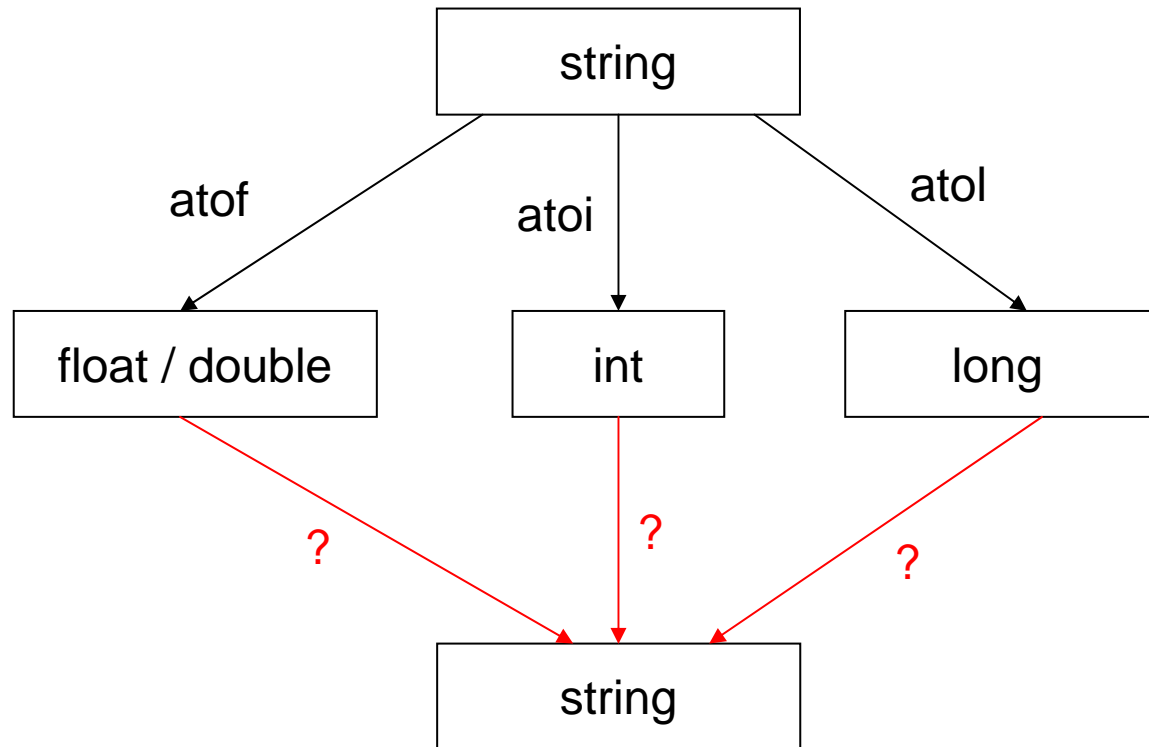
Question: How to represent 12.456 ?

Question: what's extreme value of double ?

Question

- How does compiler convert decimal value into binary value
- How does function printf show decimal value
- Size and limits of floating point
- Distribution of floating number
- Rounding error

Conversion between string and integral/floating



stdlib.h

```
double atof ( const char* s)
```

```
int atoi ( const char *s)
```

```
long atol (const char *s )
```

`#include <math.h>`

Read page 250,251 in textbook

<code>sin(x)</code>	<code>cos(x)</code>	<code>tan(x)</code>	<code>asin(x)</code>
<code>acos(x)</code>	<code>atan(x)</code>	<code>atan2(x)</code>	<code>sinh(x)</code>
<code>cosh(x)</code>	<code>tanh(x)</code>	<code>exp(x)</code>	<code>log(x)</code>
<code>log10(x)</code>	<code>pow(x,y)</code>	<code>sqrt(x)</code>	<code>ceil(x)</code>
<code>floor(x)</code>	<code>fabs(x)</code>		

Beyond double

- double-double 四精度: 32 decimal digit accuracy
- quadruple-double: 八精度: 64 decimal digit accuracy
- arbitrary precision: 任意精度: up to 1000 decimal digit accuracy.

<http://crd.lbl.gov/~dhbailey/mpdist/>

- **ARPREC** (C++/Fortran-90 arbitrary precision package)
Unix-based systems (including Apple Macintosh systems): [arprec-2.2.1.tar.gz \(version date 2008-01-23\)](#)
Windows systems: [arprec-2.2.1-windll.zip \(version date 2008-01-23\)](#)
Before installing either version, please see the "release notes" in the README file.

This package supports a flexible, arbitrarily high level of numeric precision -- the equivalent of hundreds or even thousands of decimal digits. Special routines are provided for extra-high precision (above 1000 digits). The entire library is written in C++. For Fortran-90 translation modules are also provided that permit one to convert an existing C++ or Fortran program to arbitrary precision. In most cases only the type statements and (in the case of Fortran-90 programs) read/write statements need be changed. A large number of common transcendental and combinatorial functions, multi-precision PSLQ (one-, two- or three-level versions), high precision sine and cosine, and summation of series are included, as well as three high-precision quadrature programs. New users are encouraged to use the test programs.

This version of the ARPREC package now includes "The Experimental Mathematician's Toolkit", which is available as a separate package. It provides an interactive high-precision arithmetic computing environment. One enters expressions in a Mathematica-style syntax. The precision of arithmetic can be set from 100 to 1000 decimal digit accuracy. Variables and vector arrays can be defined. A large number of common transcendental and combinatorial functions, multi-precision PSLQ (one-, two- or three-level versions), high precision sine and cosine, and summation of series are included, as well as three high-precision quadrature programs. New users are encouraged to use the test programs.

- **QD** (C++/Fortran-90 double-double and quad-double package)
Unix-based systems (including Apple Macintosh systems): [qd-2.3.6.tar.gz \(version date 2008-03-20\)](#)
Windows systems: [qd-2.3.4-windll.zip \(version date 2008-01-23\)](#)
Before installing either version, please see "release notes" in the README file.

OutLine

- Basic data type
 - integral
 - character and string
 - floating point
- Operator
- Type conversion

Operator

- Relational operator
> (greater than) >= (greater or equal)
< (less than) <= (less or equal)
== (equal) != (not equal)
- Bitwise operator
& (and) | (or) ^ (exclusive or)
<< (left shift) >> (right shift)
~ (1's complement)
- Arithmetic operator
+ - * / % (modulus)
- Assignment operator
+= -= *= /= %=
<<= >>= &= ^= |=
- Increment / decrement operator
++ --

Assignment operator

$\text{expr1} \ += \ \text{expr2} \quad \quad \quad \text{expr1} = (\text{expr1}) + (\text{expr2})$

$x \ *= \ y + 1$

$x = (x) * (y + 1)$

This form is awkward since we write **x** twice, note that in Matlab we cannot write $x *= y + 1$

$i \ += \ 2$

increment **i** by 2

$i = i + 2$

take **i**, add 2, then put the result back in **i**

add 2 to **i**

Increment / decrement operator

- An operand of integral or floating type is incremented or decremented by the integer value 1.
- The operand must have **integral**, floating, or pointer type.
- The unary operators (**++** and **--**) are called "prefix" ("postfix") increment or decrement operators when the increment or decrement operators appear before (after) the operand.
- Postfix: The increment or decrement operation occurs **after** the operand is evaluated.

```
#include <stdio.h>
#define NUM_ESCAPE_CHAR 6
int main( int argc, char* argv[])
{
    int i ;
    char word[ NUM_ESCAPE_CHAR ] ;
    word[0] = '\a' ; word[1] = '\b' ; word[2] = '\f' ; word[3] = '\n' ;
    word[4] = '\r' ; word[5] = '\t' ; word[6] = '\v' ; word[7] = '\\ ' ;
    word[8] = '\?' ; word[9] = '\\' ; word[10] = '\"' ; word[11] = '\0' ;

    for ( i = 0 ; i < 12 ; i++ ){ postfix increment

        printf("%c = 0x%x\n", word[i] , word[i] );
    }
    return 0 ;
}
```

Potential bug of equality operator

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int x = 5 ;

    if ( 1 == x ){
        printf("x (=%d) is equal to 1\n", x );
    }else{
        printf("x (=%d) is NOT equal to 1\n", x );
    }


    return 0 ;
}
```

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int x = 5 ;

    if ( x == 1 ){
        printf("x (=%d) is equal to 1\n", x );
    }else{
        printf("x (=%d) is NOT equal to 1\n", x );
    }

    return 0 ;
}
```



```
x <=5> is NOT equal to 1
Press any key to continue_
```

Question: which one (coding style) is better ?

When typing error occurs,

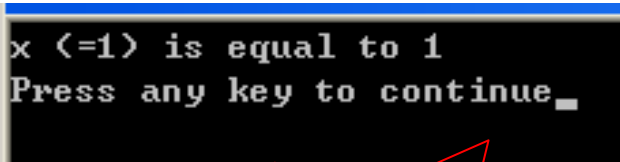
```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int x = 5 ;

    if ( x = 1 ){
        printf("x (=%d) is equal to 1\n", x );
    }else{
        printf("x (=%d) is NOT equal to 1\n", x );
    }

    return 0 ;
}
```

Good! Compiler help us to detect typing error



```
x <=1> is equal to 1
Press any key to continue.
```

Wrong ! Why?

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int x = 5 ;

    if ( 1 = x ){
        printf("x (=%d) is equal to 1\n", x );
    }else{
        printf("x (=%d) is NOT equal to 1\n", x );
    }

    return 0 ;
}
```

```
-----Configuration: equality - Win32 Debug-----
Compiling...
main.cpp
F:\course\2008summer\c_lang\example\chap2\equality\main.cpp(9) : error C2106: '=' : left operand must be l-value
Error executing cl.exe.

main.obj - 1 error(s), 0 warning(s)
```

Why `0 == x` is better than `x == 0`

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int x = 5 ;

    if ( x = 1 ){
        printf("x (=%d) is equal to 1\n", x );
    }else{
        printf("x (=%d) is NOT equal to 1\n", x );
    }

    return 0 ;
}
```

==

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int x = 5 ;

    x = 1 ;
    if ( x != 0 ){
        printf("x (=%d) is equal to 1\n", x );
    }else{
        printf("x (=%d) is NOT equal to 1\n", x );
    }

    return 0 ;
}
```

pseudo-logical bug, this kind of bug is very difficult to find out, so write `0 == x` always

Bitwise operator: AND

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    char  a, b ;
    char  a_and_b ;

    a = 'a' ; // 'a' = 0x61
    b = '1' ; // '1' = 0x31
    a_and_b = a & b ;

    printf("    a = %p\n", a ) ;
    printf("    b = %p\n", '1' ) ;
    printf("a & b = %p\n", a_and_b ) ;

    return 0 ;
}
```

True table (真值表)

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

'a' = 0x61

0	1	1	0	0	0	0	1
---	---	---	---	---	---	---	---

&

'1' = 0x31

0	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

'a' & '1' = 0x21

0	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---

Bitwise operator: OR

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    char a, b ;
    char a_or_b ;

    a = 'a' ; // 'a' = 0x61
    b = '1' ; // '1' = 0x31
    a_or_b = a | b ;

    printf("    a = %p\n", a ) ;
    printf("    b = %p\n", '1' ) ;
    printf("a | b = %p\n", a_or_b ) ;

    return 0 ;
}
```

True table (真値表)

x	y	x y
0	0	0
0	1	1
1	0	1
1	1	1

'a' = 0x61

0	1	1	0	0	0	0	1
---	---	---	---	---	---	---	---

'1' = 0x31

0	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

'a' | '1' = 0x71

0	1	1	1	0	0	0	1
---	---	---	---	---	---	---	---

Bitwise operator: exclusive OR

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
    char a, b ;
    char a_ex_b ;

    a = 'a' ; // 'a' = 0x61
    b = '1' ; // '1' = 0x31
    a_ex_b = a ^ b ;

    printf("    a = %p\n", a ) ;
    printf("    b = %p\n", '1' ) ;
    printf("a ^ b = %p\n", a_ex_b ) ;

    return 0 ;
}
```

True table (真值表)

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

'a' = 0x61

0	1	1	0	0	0	0	1
---	---	---	---	---	---	---	---

'1' = 0x31

0	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

&

'a' ^ '1' = 0x50

0	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

shift operator

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    char a = 'a' ;
    char a_right_shift_1 = a >> 1 ;
    int a_left_shift_1 = a << 1 ;

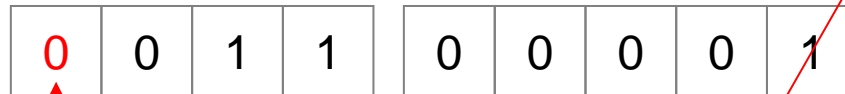
    printf("    a = %p\n", a ) ;
    printf(" a >> 1 = %p\n", a_right_shift_1 ) ;
    printf(" a << 1 = %p\n", a_left_shift_1 ) ;

    return 0 ;
}
```

'a' = 0x61



'a' >> 1 = 0x30



truncate

Fill zero

'a' << 1 = 0xc2



OutLine

- Basic data type
 - integral
 - character and string
 - floating point
- Operator
- Type conversion

Type conversion: done by compiler automatically

Conditions for Type Conversion

Conditions Met	Conversion
Either operand is of type long double .	Other operand is converted to type long double .
Preceding condition not met and either operand is of type double .	Other operand is converted to type double .
Preceding conditions not met and either operand is of type float .	Other operand is converted to type float .
Preceding conditions not met (none of the operands are of floating types).	<p>Integral promotions are performed on the operands as follows:</p> <ul style="list-style-type: none">• If either operand is of type unsigned long, the other operand is converted to type unsigned long.• If preceding condition not met, and if either operand is of type long and the other of type unsigned int, both operands are converted to type unsigned long.• If the preceding two conditions are not met, and if either operand is of type long, the other operand is converted to type long.• If the preceding three conditions are not met, and if either operand is of type unsigned int, the other operand is converted to type unsigned int.• If none of the preceding conditions are met, both operands are converted to type int.

```
double dVal;
float fVal;
int iVal;
unsigned long ulVal;

int main( int argc, char* argv[] ) {
    // iVal converted to unsigned long
    // result of multiplication converted to double
    dVal = iVal * ulVal;

    // ulVal converted to float
    // result of addition converted to double
    dVal = ulVal + fVal;

    return 0 ;
}
```

casting (coercion) 強制型別轉換 done by programmer

```
double dVal;  
float fVal;  
int iVal;  
unsigned long ulVal;  
  
int main( int argc, char* argv[] ) {  
    // iVal converted to unsigned long  
    // result of multiplication converted to double  
    dVal = ( (double) iVal ) * ( (double) ulVal ) ;  
  
    // ulVal converted to float  
    // result of addition converted to double  
    dVal = ( (double) ulVal ) + ( (double) fVal ) ;  
  
    return 0 ;  
}
```

Do coercion as possible as you can.