CS 1037
Computer Science Fundamentals II

Part Three: Binary Representation

SIMPLE C PROGRAM

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
   char a;     /* 1 byte   */
   int b;      /* 4 bytes  */
   float c;    /* 4 bytes  */
   double d;   /* 8 bytes  */


   a = 'K';
   b = 37;
   c = 2.5;
   d = 75.3;

   printf( "1st value of a is : %c \n" , a );
   printf( "2nd value of b is : %d \n" , b );
   printf( "3rd value of c is : %f \n" , c );
   printf( "4rd value of d is : %lf \n" , d );

   return 0 ;
}
```

OUTPUT:

1st value of a is : K
2nd value of b is : 37
3rd value of c is : 2.50000000
4th value of d is : 75.50000000

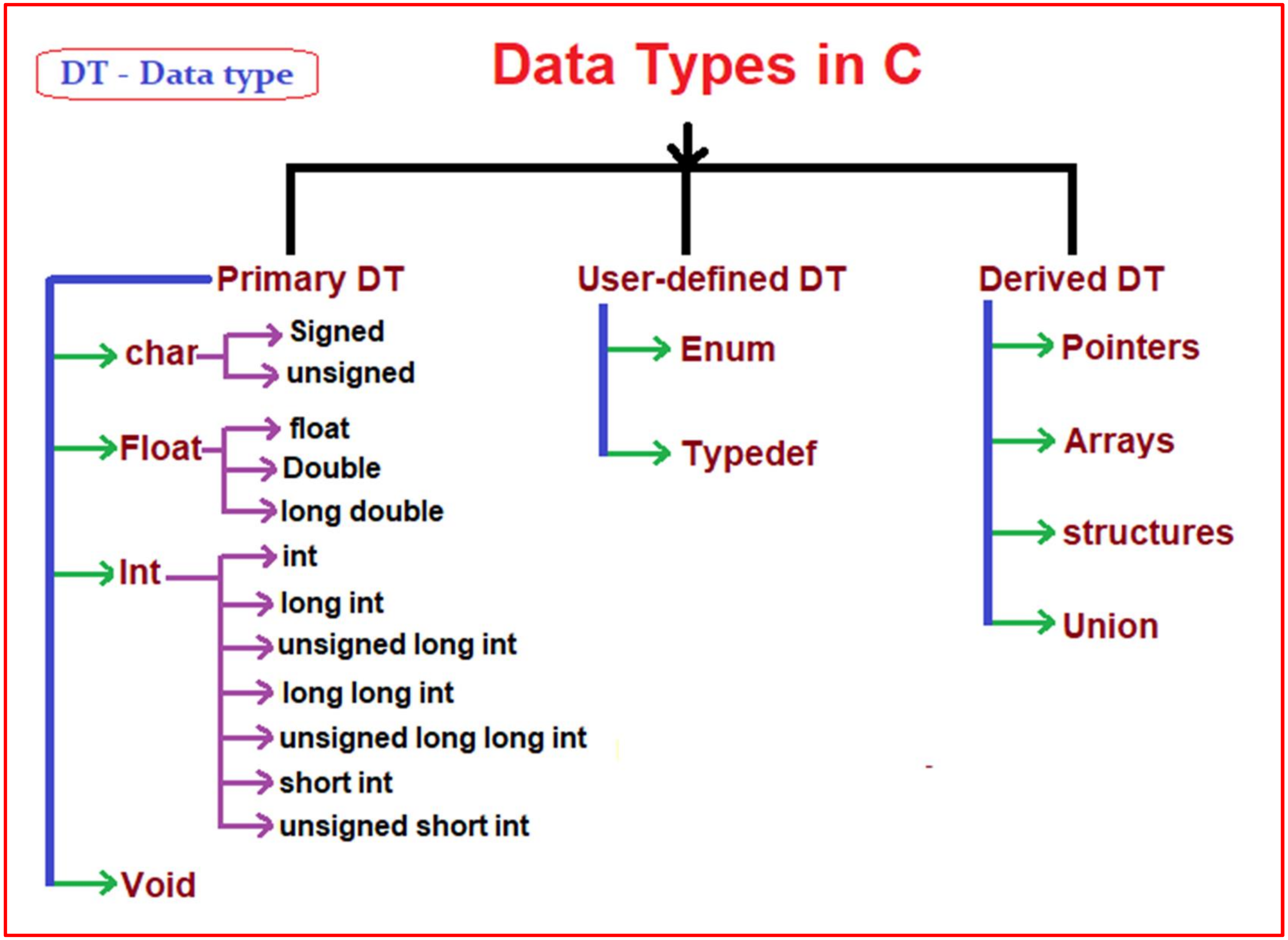
C's built-in data types that are similar to ones in Java:

Syntax	Name	Java's counterpart	use
`char`	character	`byte`	Stores an ASCII code (character) or it can also store a very short integer (−128..127)
`short`	short integer	`short`	uses 2 byte memory, value between −32768 and 32767
`int`	ordinary integer	`int`	uses 4 byte memory, value between −2147483648 and 2147483647
`long`	long integer	`long`	uses 8 bytes memory, value between −9223372036854775808 and 9223372036854775807
`float`	single precision float	`float`	uses 4 byte memory, absolute value between 1.4E−45 and 3.4E38
`double`	double precision float	`double`	uses 8 byte memory, absolute value between 4.9E−324 and 1.8E308
`_Bool`	boolean	`boolean`	true (1) or false (0)

C's built-in data types that do not have an equivalent in Java:

Syntax	Name	use
`unsigned char`	Unsigned character	Very short positive integer (0..255)
`unsigned short`	Unsigned short integer	uses 2 byte memory, value between 0 and 65535
`unsigned int`	Unsigned ordinary integer	uses 4 byte memory, value between 0 and 4294967295
`unsigned long`	Unsigned long integer	uses 8 bytes memory, value between 0 and 18446744073709551615
`*`	*Reference type*	Contains a memory address (usually 4 bytes, but 64 bits machines will use 8 bytes)

BREAKDOWN of DATA TYPES



American Standard Code for Information Interchange (ASCII)

Binary representation of characters:

Dec	Symbol	Binary	Dec	Symbol	Binary
65	A	0100 0001	83	S	0101 0011
66	B	0100 0010	84	T	0101 0100
67	C	0100 0011	85	U	0101 0101
68	D	0100 0100	86	V	0101 0110
69	E	0100 0101	87	W	0101 0111
70	F	0100 0110	88	X	0101 1000
71	G	0100 0111	89	Y	0101 1001
72	H	0100 1000	90	Z	0101 1010
73	I	0100 1001	91	[	0101 1011
74	J	0100 1010	92	\	0101 1100
75	K	0100 1011	93	]	0101 1101
76	L	0100 1100	94	^	0101 1110
77	M	0100 1101	95	_	0101 1111
78	N	0100 1110	96	`	0110 0000
79	O	0100 1111	97	a	0110 0001
80	P	0101 0000	98	b	0110 0010
81	Q	0101 0001	99	c	0110 0011
82	R	0101 0010	100	d	0110 0100

UNICODE: **UTF-8** 8 bits = 2^7= 128 possible characters.

UNICODE (UTF-8 UTF-16 UTF-32)

Binary representation of characters:

Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char
0	0000 0000	00	[NUL]	32	0010 0000	20	space	64	0100 0000	40	@	96	0110 0000	60	`
1	0000 0001	01	[SOH]	33	0010 0001	21	!	65	0100 0001	41	A	97	0110 0001	61	a
2	0000 0010	02	[STX]	34	0010 0010	22	"	66	0100 0010	42	B	98	0110 0010	62	b
3	0000 0011	03	[ETX]	35	0010 0011	23	#	67	0100 0011	43	C	99	0110 0011	63	c
4	0000 0100	04	[EOT]	36	0010 0100	24	$	68	0100 0100	44	D	100	0110 0100	64	d
5	0000 0101	05	[ENQ]	37	0010 0101	25	%	69	0100 0101	45	E	101	0110 0101	65	e
6	0000 0110	06	[ACK]	38	0010 0110	26	&	70	0100 0110	46	F	102	0110 0110	66	f
7	0000 0111	07	[BEL]	39	0010 0111	27	'	71	0100 0111	47	G	103	0110 0111	67	g
8	0000 1000	08	[BS]	40	0010 1000	28	(	72	0100 1000	48	H	104	0110 1000	68	h
9	0000 1001	09	[TAB]	41	0010 1001	29	)	73	0100 1001	49	I	105	0110 1001	69	i
10	0000 1010	0A	[LF]	42	0010 1010	2A	*	74	0100 1010	4A	J	106	0110 1010	6A	j
11	0000 1011	0B	[VT]	43	0010 1011	2B	+	75	0100 1011	4B	K	107	0110 1011	6B	k
12	0000 1100	0C	[FF]	44	0010 1100	2C	,	76	0100 1100	4C	L	108	0110 1100	6C	l
13	0000 1101	0D	[CR]	45	0010 1101	2D	-	77	0100 1101	4D	M	109	0110 1101	6D	m
14	0000 1110	0E	[SO]	46	0010 1110	2E	.	78	0100 1110	4E	N	110	0110 1110	6E	n
15	0000 1111	0F	[SI]	47	0010 1111	2F	/	79	0100 1111	4F	O	111	0110 1111	6F	o
16	0001 0000	10	[DLE]	48	0011 0000	30	0	80	0101 0000	50	P	112	0111 0000	70	p
17	0001 0001	11	[DC1]	49	0011 0001	31	1	81	0101 0001	51	Q	113	0111 0001	71	q
18	0001 0010	12	[DC2]	50	0011 0010	32	2	82	0101 0010	52	R	114	0111 0010	72	r
19	0001 0011	13	[DC3]	51	0011 0011	33	3	83	0101 0011	53	S	115	0111 0011	73	s
20	0001 0100	14	[DC4]	52	0011 0100	34	4	84	0101 0100	54	T	116	0111 0100	74	t
21	0001 0101	15	[NAK]	53	0011 0101	35	5	85	0101 0101	55	U	117	0111 0101	75	u
22	0001 0110	16	[SYN]	54	0011 0110	36	6	86	0101 0110	56	V	118	0111 0110	76	v
23	0001 0111	17	[ETB]	55	0011 0111	37	7	87	0101 0111	57	W	119	0111 0111	77	w
24	0001 1000	18	[CAN]	56	0011 1000	38	8	88	0101 1000	58	X	120	0111 1000	78	x
25	0001 1001	19	[EM]	57	0011 1001	39	9	89	0101 1001	59	Y	121	0111 1001	79	y
26	0001 1010	1A	[SUB]	58	0011 1010	3A	:	90	0101 1010	5A	Z	122	0111 1010	7A	z
27	0001 1011	1B	[ESC]	59	0011 1011	3B	;	91	0101 1011	5B	[	123	0111 1011	7B	{
28	0001 1100	1C	[FS]	60	0011 1100	3C	<	92	0101 1100	5C	\	124	0111 1100	7C	|
29	0001 1101	1D	[GS]	61	0011 1101	3D	=	93	0101 1101	5D	]	125	0111 1101	7D	}
30	0001 1110	1E	[RS]	62	0011 1110	3E	>	94	0101 1110	5E	^	126	0111 1110	7E	~
31	0001 1111	1F	[US]	63	0011 1111	3F	?	95	0101 1111	5F	_	127	0111 1111	7F	[DEL]

UNICODE is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.

UNICODE: **UTF-16** 16 bits = 2^{15} = 1,112,064 possible characters

UTF-32 32 bits = 2^{32} = 2,147,483,648 possible characters

BINARY (base 2) NUMBERIC SYSTEM

Direct translation of systems:

DECIMAL (BASE 10)	BINARY (BASE 2)
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
1000	1111101000

BINARY (base 2) NUMBERIC SYSTEM

Magnitude Form
- unsigned (positive only) whole numbers

Sign-Magnitude Form
- signed (positive and negative) whole numbers

One's Compliment Form (1's compliment)
- negative represented by the compliment (flipping bits) of the positive number

Two's Compliment Form (2's compliment)
- negative represented by adding the values of 1 to the compliment of the positive number

Single Precision Floating Point Form
- fixed point model using Scientific Notation to represent values (32 bit form)

Double Precision Floating Point Form
- fixed point model using Scientific Notation to represent values (64 bit form)

BINARY (base 2) NUMBERIC SYSTEM

Positive 75

0	1	0	0	1	0	1	1

Sign-bit -75

1	1	0	0	1	0	1	1

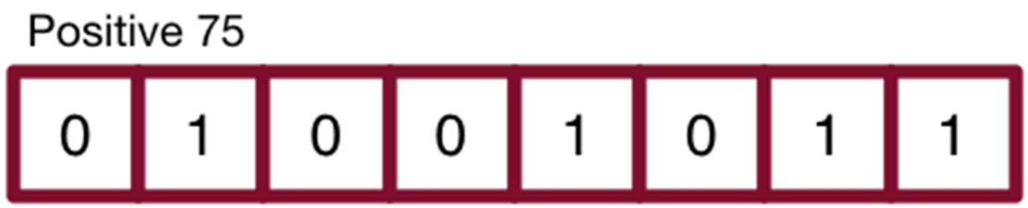
1s Complement -75

1	0	1	1	0	1	0	0

2s Complement -75

1	0	1	1	0	1	0	1

BINARY (base 2) NUMBERIC SYSTEM



BINARY (base 2) NUMBERIC SYSTEM

Positive 75

0	1	0	0	1	0	1	1

Sign-bit -75

1	1	0	0	1	0	1	1

BINARY (base 2) NUMBERIC SYSTEM

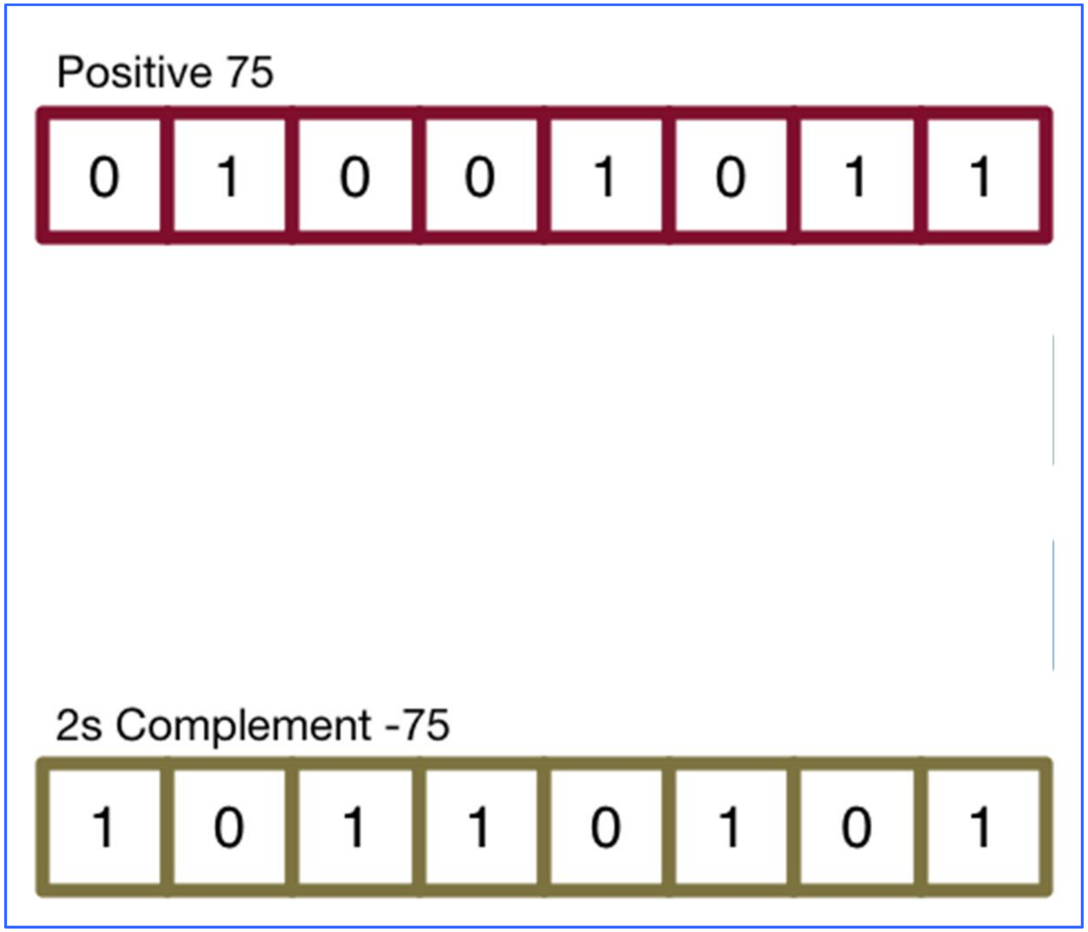
Positive 75

0	1	0	0	1	0	1	1

1s Complement -75

1	0	1	1	0	1	0	0

BINARY (base 2) NUMBERIC SYSTEM



BINARY (base 2) NUMBERIC SYSTEM

Floating Point Form

method for storing real (fractional) numbers

for example: **18.375**

18 -> whole number

375 -> fractional part

- can use negative powers of two

remember: 2^3 -> 8

2^{-3} -> 0.125

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	
16	8	4	2	1	0.5	0.25	0.125	
1	**0**	**0**	**1**	**0**	**0**	**1**	**1**	-> 18.375

can be written as: **10010.011** (this is FIXED POINT MODEL)

limited range of values due to **forced** position

BINARY (base 2) NUMBERIC SYSTEM

Floating Point Form

remember (in Decimal (base 10))

18.375 = 1.8375×10^1 -> move it one (1) place

123.456 = 1.23456×10^2 -> move it two (2) places

we can do the same thing in Binary:

10010.011 = 1.0010011×2^4 -> move it four (4) places

this allows for **FLOATING POINT** representation

=> ignore the **2** (always present – assumed)

=> ignore the **leading mantissa** (number to the left of decimal)

just need to store the sign (- or +), the fractional part and the exponent

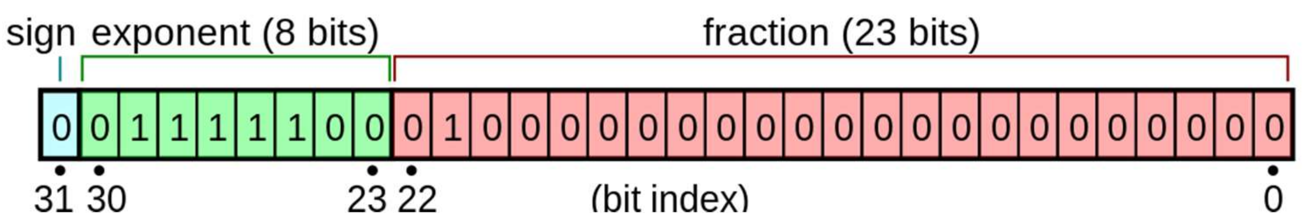
8 bits does not give enough to play with: assume 32 bits (4 bytes)

- store exponent using 8 bits - store fraction using 23 bits

BINARY (base 2) NUMBERIC SYSTEM

Single Precision Floating Point Form

bit:	1	8	23
	sign	exponent (e)	fraction (f)
position	31	30 … 23	22 … 0



BUT: the exponent stores numbers from -127 to +128
(NOT 0 to 255)

BUT: it stores it as **biased** towards 127
i.e. take the exponent and add 127 to it (do not have to worry about sign bit…)

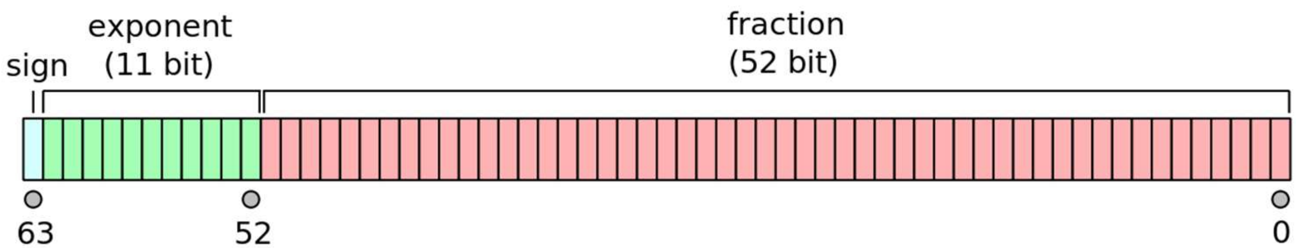
(eg) 6 = 6 + 127 = 133 (10000101)

(eg) -12 = -12 + 127 = 115 (0111 0011)

BINARY (base 2) NUMBERIC SYSTEM

Double Precision Floating Point Form

bit:	1	11	53
	sign	exponent (e)	fraction (f)
position	63	62 … 53	52 … 0



BUT: the exponent stores numbers from -1023 to +1024
(NOT 0 to 2047)

BUT: it stores it as **biased** towards 1023
i.e. take the exponent and add 1023 to it (do not have to worry about sign bit…)

(ex) 36 = 36 + 1023 = 133 (0100 0010 0011)

(e) -182 = -182 + 1023 = 115 (0011 0100 1001)

TYPES in C

ALL DATA IS STORED AS A TYPE

characters	- 1 byte
short	- 2 bytes
int	- 4 bytes
float	- 4 bytes
double	-16 bytes

For a computer to perform an arithmetic operation:

- the operands must usually be of the same size
  i.e. (the same number of bits)

and

- be stored in the same way.
  i.e. (int and float are both 4 bytes, but different usage of the 1's and 0's)

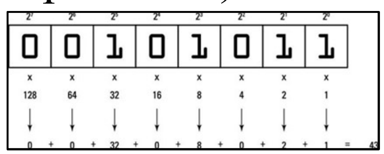
TYPES in C

ALL DATA IS STORED AS A TYPE

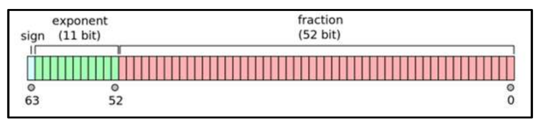
MIXED USE:

for example: adding an

int (4 bytes – double precision)



double (8 bytes floating point)



computer MUST convert to one type of data representation.

in C, the computer will always convert to the more complex representation (variable promotion)

TYPES in C

ALL DATA IS STORED AS A TYPE

When operands of different types are mixed in expressions,
the C compiler may have to generate instructions that
change the types of some operands
so that hardware will be able to evaluate the expression.

IMPLICIT CONVERSION

If we add a 16-bit short and a 32-bit int,
the compiler will arrange for the short value to be converted to 32 bits.

If we add an int and a float,
the compiler will arrange for the int to be converted to float format.

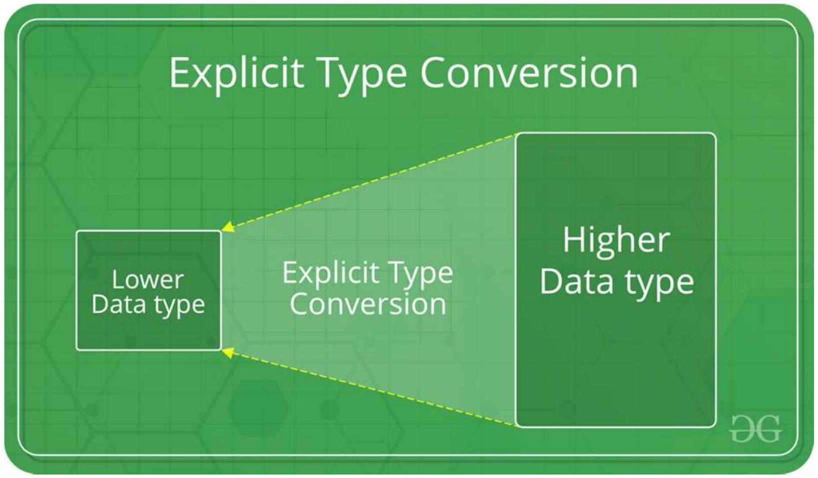
TYPES in C

IMPLICIT CONVERSION

Generally takes place when in an expression more than one data type is present.

In such condition type conversion (type promotion) takes place to avoid lose of data.

All the data types of the variables are upgraded to the data type of
the variable with largest data type.



TYPES in C

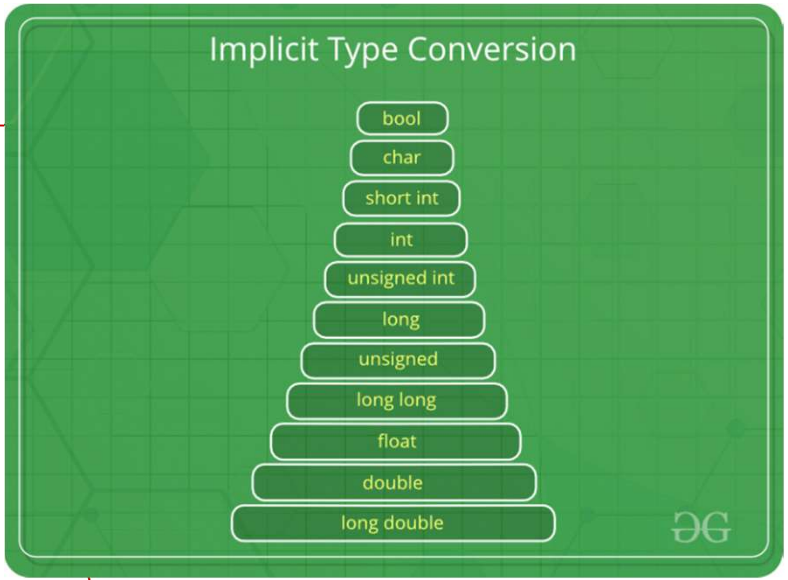
IMPLICIT CONVERSION

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
   int x = 10; // integer x
   char y = 'a'; // character c

   // y implicitly converted to int.
   // value of 'a' is 97
   x = x + y;

   // x is implicitly converted to float
   float z = x + 1.0;

   printf("x = %d, z = %f", x, z);
   return 0;
}
```



implicit conversion is **automatic**
it requires no user intervention

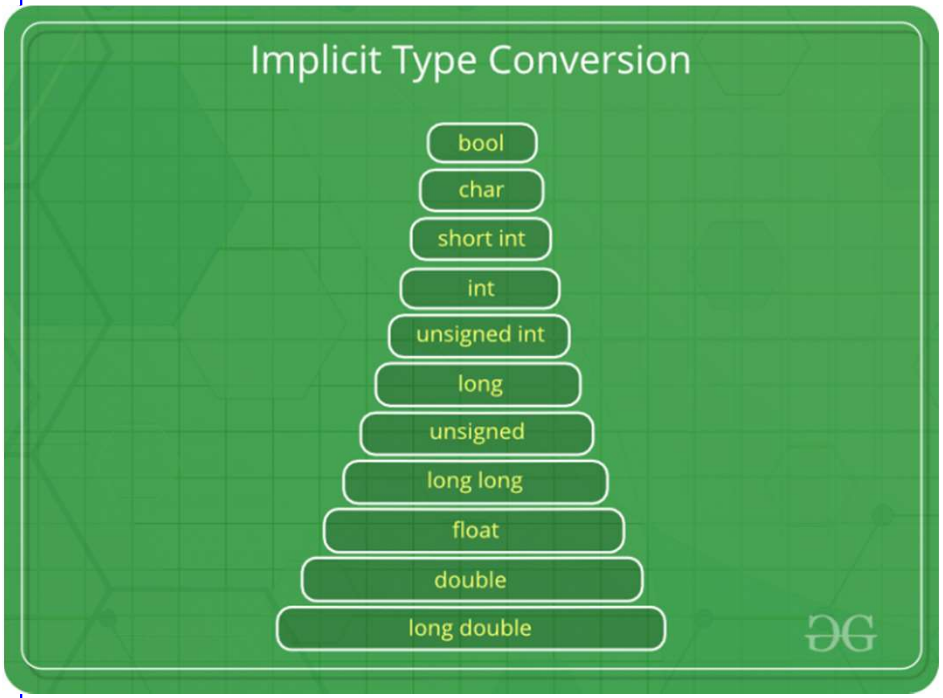
TYPES in C

```
char ta = 'b';
long tb = 343437;
long tc;
double td;

tc = ta + tb;

td = tc / ta;

printf("value of ta: %c\n",ta);
printf("value of ta: %d\n",ta);
printf("value of tb: %d\n",tb);
printf("value of tc: %d\n",tc);
printf("value of td: %lf\n",td);
printf("\n\n");
```




```
value of ta: b
value of ta: 98
value of tb: 343437
value of tc: 343535
value of td: 3505.00000000000
```

IMPLICIT CONVERSION

TOP to BOTTOM
(anything above will converted to the type below it)

TYPES in C

IMPLICIT CONVERSION

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;      /* c is converted to int                */
i = i + s;      /* s is converted to int                */
u = u + i;      /* i is converted to unsigned int       */
l = l + u;      /* u is converted to long int           */
ul = ul + l;    /* l is converted to unsigned long int */
f = f + ul;     /* ul is converted to float             */
d = d + f;      /* f is converted to double             */
ld = ld + d;    /* d is converted to long double        */
```

TYPES in C

ALL DATA IS STORED AS A TYPE

When operands of different types are mixed in expressions, the C compiler may have to generate instructions that change the types of some operands so that hardware will be able to evaluate the expression.

EXPLICIT CONVERSION

Basically, the programmer forces an expression to be of a specific type.

Explicit type conversion is also called **type casting**.

The general format of explicit type conversion is as follows:

(data_type)(expression);

TYPES in C

```
// C program to demonstrate explicit type casting
#include<stdio.h>

int main()
{
        double x = 1.2;

        // Explicit conversion from double to int
        int sum = (int)x + 1;

        printf("sum = %d", sum);

        return 0;
}
```

EXPLICIT CONVERSION

notice we are ***forcing*** the computer to cast
a higher complexity type (double)
to a lower complexity (int)

TYPES in C

EXPLICIT CONVERSION

C regards (*type-name*) as a **unary operator**.
Unary operators have higher **precedence** than binary operators,
so the compiler interprets

```
(float) dividend / divisor
```

as

```
((float) dividend) / divisor
```

Casts are sometimes necessary to avoid overflow:

```
long i;
int j = 1000;

i = j * j;    /* overflow may occur */
```

Using a cast avoids the problem:

```
i = (long) j * j;
```

The statement

```
i = (long) (j * j);    /*** WRONG ***/
```

wouldn’t work, since the overflow would already have occurred by the time of the cast.

TYPE DEFINITIONS in C

The `#define` directive can be used to create a "Boolean type" macro:

```
#define BOOL int
```

There's a better way using a feature known as a ***type definition:***

```
typedef int Bool;
```

`Bool` can now be used in the same way as the built-in type names.
Example:

```
Bool flag;   /* same as int flag; */
```

`typedef` *`(known data type)`* *`(alias to be used instead of)`*

is nothing more than creating an 'alias'

```
typedef float Dollars // Dollars now is an alias for float
Dollars cash_in, cash_out;
// is more informative than float cash_in, cash_out;
```

TYPE DEFINITIONS in C

Type definitions can also make a program easier to modify.
To redefine `Dollars` as `double`, only the type definition need be changed:

```
typedef double Dollars;
```

Without the type definition, we would need to locate all `float` variables that store dollar amounts and change their declarations.

Type definitions are an important tool for writing portable programs.
One of the problems with moving a program from one computer to another is that types may have different ranges on different machines.

If `i` is an `int` variable, an assignment like

```
i = 100000;
```

is fine on a machine with 32-bit integers, but fails on a machine with 16-bit integers.

BINARY (base 2) NUMBERIC SYSTEM

Size DOES matter …

Kilobyte (KB)	*1,000 bytes OR 10^3 bytes* 2 Kilobytes: A typewritten page 100 Kilobytes: A low-resolution photograph
Megabyte (MB)	*1,000,000 bytes OR 10^6 bytes* 1 Megabyte: A small novel OR a 3.5-inch floppy disk 2 Megabytes: A high-resolution photograph 5 Megabytes: The complete works of Shakespeare 10 Megabytes: A minute of high-fidelity sound 100 Megabytes: One meter of shelved books 500 Megabytes: A CD-ROM
Gigabyte (GB)	*1,000,000,000 bytes OR 10^9 bytes* 1 Gigabyte: A pickup truck filled with books 20 Gigabytes: A good collection of the works of Beethoven 100 Gigabytes: A library floor of academic journals
Terabyte (TB)	*1,000,000,000,000 bytes OR 10^{12} bytes* 1 Terabyte: 50,000 trees made into paper and printed 2 Terabytes: An academic research library 10 Terabytes: The print collections of the U.S. Library of Congress 400 Terabytes: National Climactic Data Center (NOAA) database
Petabyte (PB)	*1,000,000,000,000,000 bytes OR 10^{15} bytes* 1 Petabyte: Three years of EOS data (2001) 2 Petabytes: All U.S. academic research libraries 20 Petabytes: Production of hard-disk drives in 1995 200 Petabytes: All printed material
Exabyte (EB)	*1,000,000,000,000,000,000 bytes OR 10^{18} bytes* 2 Exabytes: Total volume of information generated in 1999 5 Exabytes: All words ever spoken by human beings

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

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

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

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

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

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

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

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

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

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

|________________|________________|________________|

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

|________________|________________|________________|

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

|________________|________________|________________|

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

|________________|________________|________________|

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

|________________|________________|________________|

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

|________________|________________|________________|

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

|________________|________________|________________|

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

|________________|________________|________________|

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

|________________|________________|________________|

397 -	398 -	399 -
1 0 1 1 1 0 0 1	1 0 1 1 1 0 0 1	1 0 0 0 1 1 0 1
400 -	401 -	402 -
0 0 1 0 0 0 1 1	0 1 1 0 0 1 1 1	0 0 1 1 0 1 1 1
403 -	404 -	405 -
0 1 0 1 0 1 1 1	1 0 1 0 1 0 1 1	0 1 0 0 0 1 1 0
406 -	407 -	408 -
1 1 1 0 0 1 1 0	1 1 1 0 0 1 1 0	0 0 1 1 0 0 1 0
409 -	410 -	411 -
0 0 1 0 0 0 1 1	0 1 1 0 0 1 1 1	0 0 1 1 0 1 1 1
412 -	413 -	414 -
1 0 1 1 1 0 0 1	1 0 1 1 1 0 0 1	1 0 0 0 1 1 0 1
415 -	416 -	417 -
1 1 1 0 0 1 1 0	1 1 1 0 0 1 1 0	0 0 1 1 0 0 1 0
418 -	419 -	420 -
0 1 0 1 0 1 1 1	1 0 1 0 1 0 1 1	0 1 0 0 0 1 1 0
421 -	422 -	423 -
1 0 1 1 1 0 0 1	1 0 1 1 1 0 0 1	1 0 0 0 1 1 0 1