Basic Data Types (cont.)

Data Types in C

Four Basic Data Types

- Char (1 Byte = 8 Bits)
- Int (4 Byte)
- Float (single precision – 4 Byte)
- Double (double precision – 8 Byte)

Type Modifiers

- Signedness
  - Unsigned: target type will have unsigned representation
  - Signed: target type will have signed representation (this is the default if omitted)
- Size
  - Short: target type will be optimized for space and will have width of at least 16 bits.
  - Long: target type will have width of at least 32 bits.
  - Long Long: target type will have width of at least 64 bits

Type Comparison

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 *or* 4 bytes	1 wide character

Char vs. Int

- char a = '1';
  - Takes 1 byte in memory Stores byte “011 0001”
  - ASCII printable characters
-

Binary	Oct	Dec	Hex	Glyph	Binary	Oct	Dec	Hex	Glyph	Binary	Oct	Dec	Hex	Glyph
010 1111	057	47	2F	/	100 1111	117	79	4F	O	110 1111	157	111	6F	o
011 0000	060	48	30	0	101 0000	120	80	50	P	111 0000	160	112	70	p
011 0001	061	49	31	1	101 0001	121	81	51	Q	111 0001	161	113	71	q
011 0010	062	50	32	2	101 0010	122	82	52	R	111 0010	162	114	72	r
011 0011	063	51	33	3	101 0011	123	83	53	S	111 0011	163	115	73	s
011 0100	064	52	34	4	101 0100	124	84	54	T	111 0100	164	116	74	t
011 0101	065	53	35	5	101 0101	125	85	55	U	111 0101	165	117	75	u
011 0110	066	54	36	6	101 0110	126	86	56	V	111 0110	166	118	76	v
011 0111	067	55	37	7	101 0111	127	87	57	W	111 0111	167	119	77	w
011 1000	070	56	38	8	101 1000	130	88	58	X	111 1000	170	120	78	x
011 1001	071	57	39	9	101 1001	131	89	59	Y	111 1001	171	121	79	y
011 1010	072	58	3A	:	101 1010	132	90	5A	Z	111 1010	172	122	7A	z
011 1011	073	59	3B	;	101 1011	133	91	5B	[	111 1011	173	123	7B	{
011 1100	074	60	3C	<	101 1100	134	92	5C	\	111 1100	174	124	7C	|

Char vs. Int

- int a = 1;
  - Takes 4 bytes in memory
  - Stores 0000 0000 (first 3 bytes)
  - Stores 0000 0001 (as last byte) in memory
  -
- ASC II characters are also how we store a text file
  - Example: Hexdump

Unsigned vs. Signed (char, int)

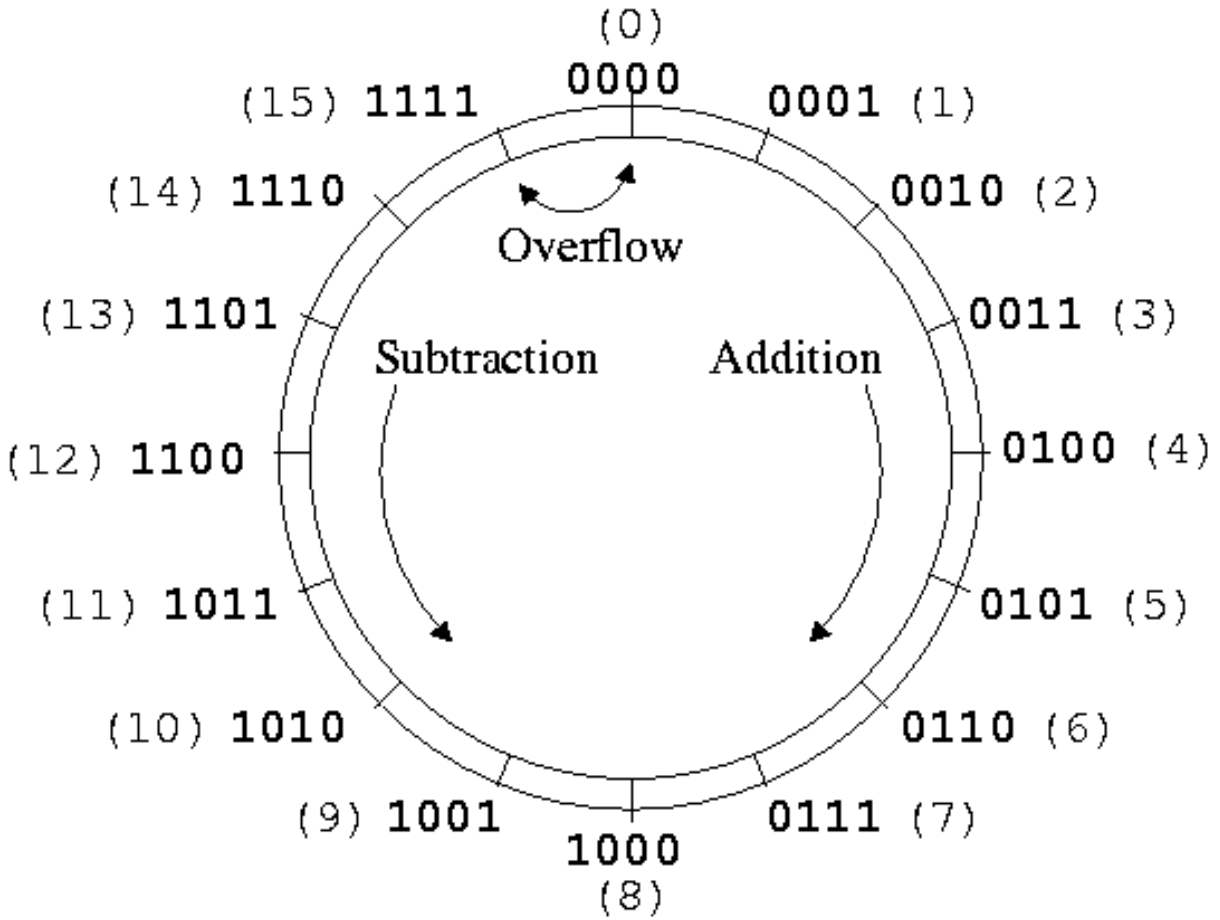
- Unsigned char: 0~255
- Signed char: -128~127

8-bit two's-complement integers

Bits	Unsigned value	2's complement value
0111 1111	127	127
0111 1110	126	126
0000 0010	2	2
0000 0001	1	1
0000 0000	0	0
1111 1111	255	−1
1111 1110	254	−2
1000 0010	130	−126
1000 0001	129	−127
1000 0000	128	−128

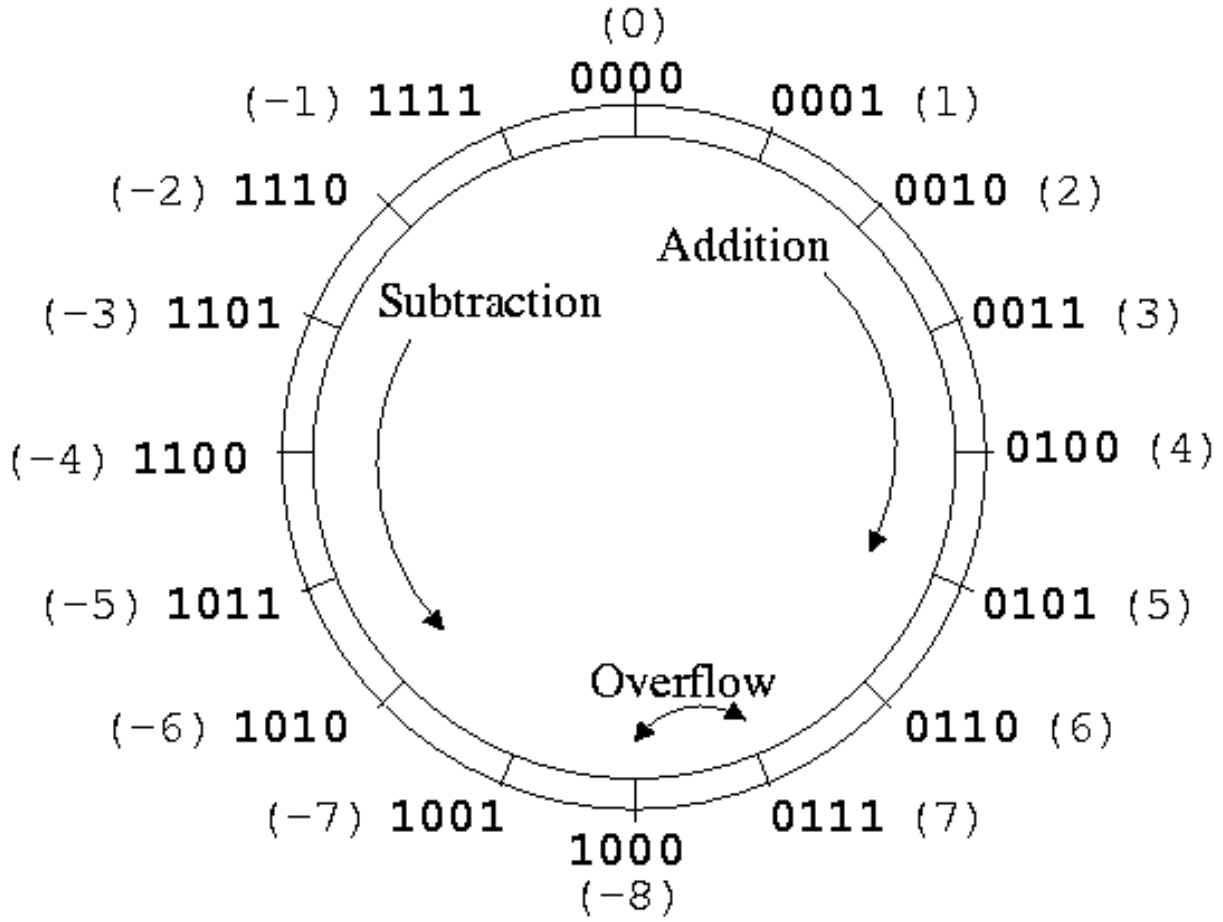
Two complement Arithmetic

- The most common method of representing signed integers on computers
- Unsigned
-



Two complement Arithmetic

- signed



C Integral Data Types

C Declaration	Guaranteed		Typical 32-bit	
	Minimum	Maximum	Minimum	Maximum
`char`	−127	127	−128	127
`unsigned char`	0	255	0	255
`short [int]`	−32,767	32,767	−32,768	32,767
`unsigned short [int]`	0	63,535	0	63,535
`int`	−32,767	32,767	−2,147,483,648	2,147,483,647
`unsigned [int]`	0	65,535	0	4,294,967,295
`long [int]`	−2,147,483,647	2,147,483,647	−2,147,483,648	2, 147, 483, 647
`unsigned long [int]`	0	4,294,967,295	0	4,294,967,295

Figure 2.8: **C Integral Data types.** Text in square brackets is optional.

Overflow

- The max unsigned integer is 2^32-1
  - If add two unsigned integer larger than 2^31, it will overflow, results will be mod by 2^32
- The max signed integer is 2^31-1
  - If add two signed integer larger than 2^31, it will overflow, results will be negative number

Unsigned Overflow

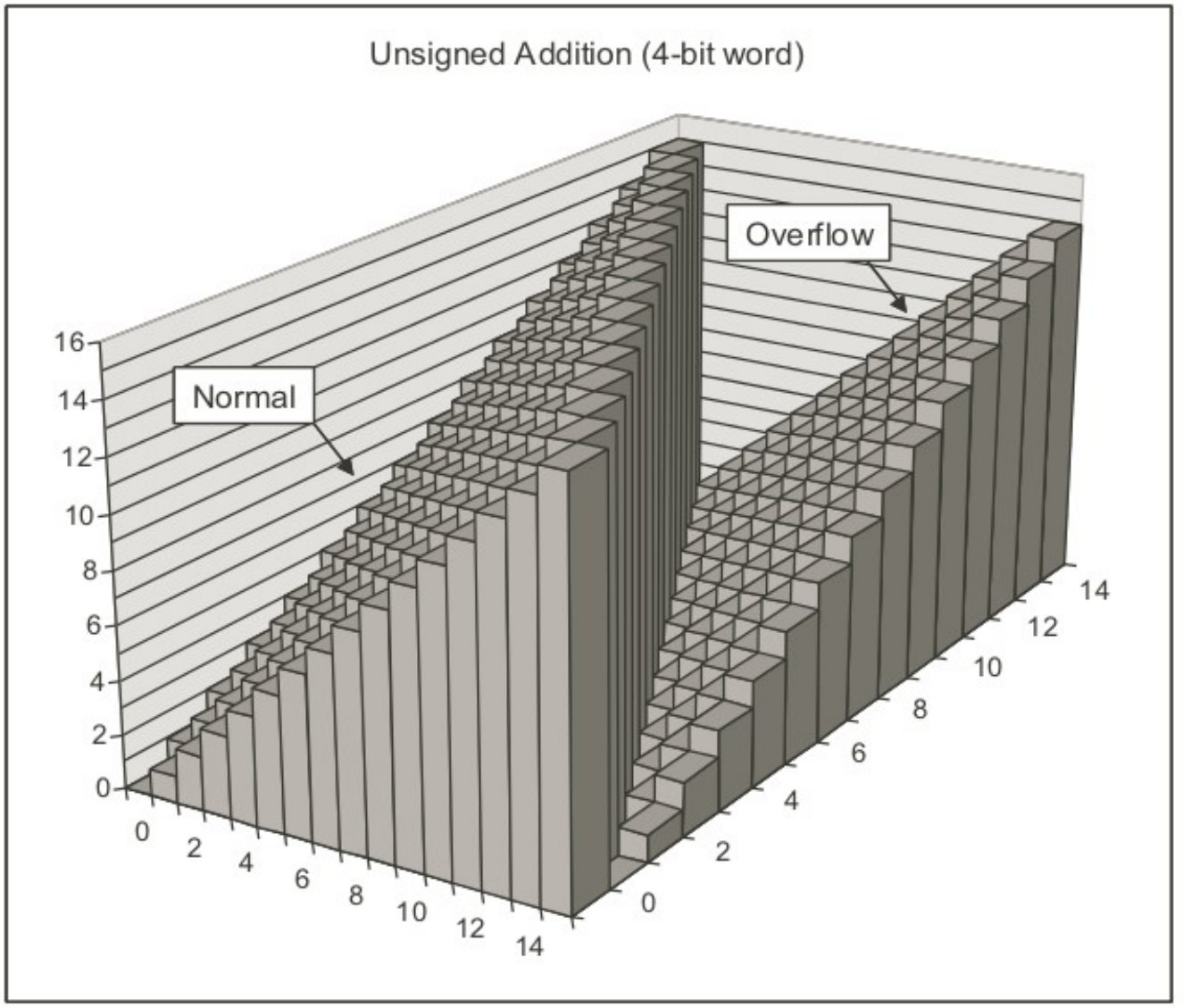


Figure 2.16: **Unsigned Addition.** With a four-bit word size, addition is performed modulo 16.

Signed Overflow

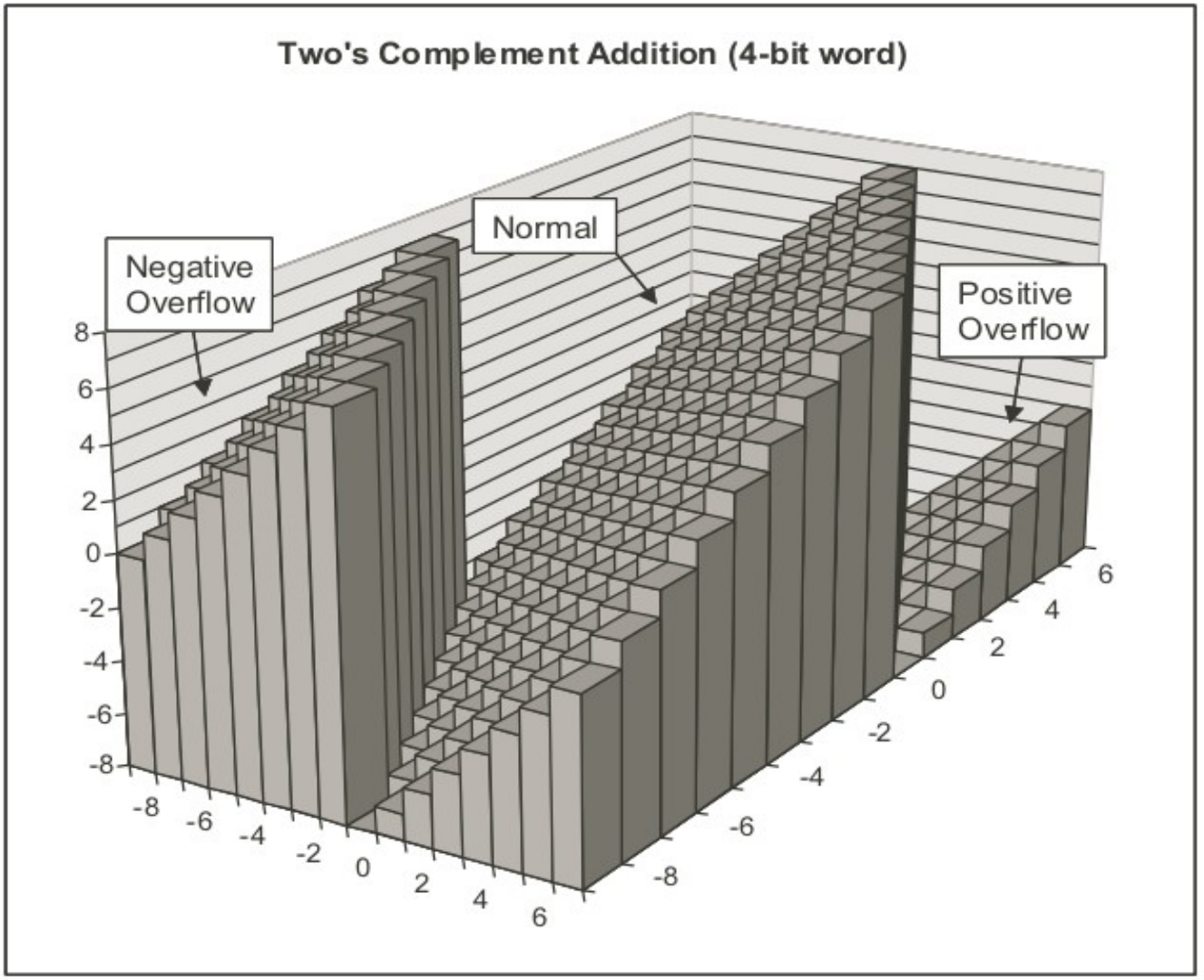
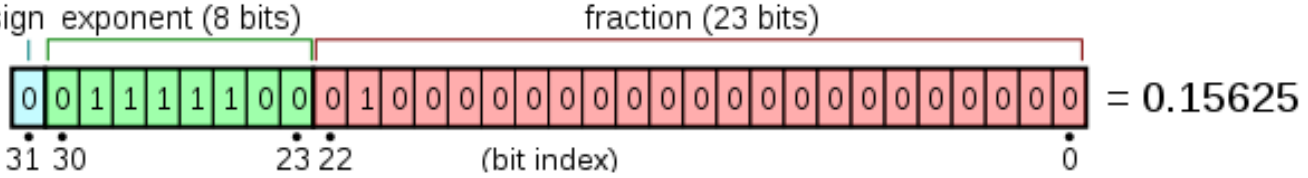


Figure 2.19: **Two's Complement Addition.** With a four-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

Float vs. Double

- Float (single precision 32 bits)

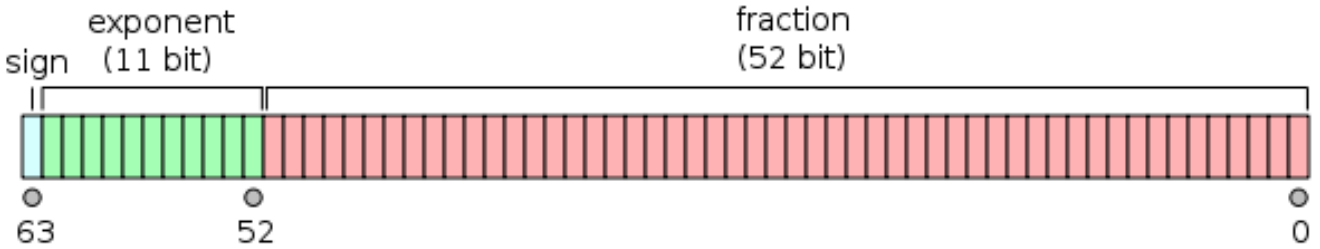
Layout of the IEEE binary 32-bit floating point format:



For example, the 32-bit floating point bit pattern *1 01111101 00101100000000000000000* is interpreted as the following:[3]

$$(-1)^1 \times (1 + 0.34375) \times 2^{(125-127)} = -1.34375 \times 2^{-2} = -0.3359375$$

- Double precision (64 bits)



Why precision is import?

As the name implies, a `double` has 2x the precision of `float` [1]. In general a double has 15 to 16 decimal digits of precision, while `float` only has 7.

This precision loss could lead to truncation errors much easier to float up, e.g.

```
float a = 1.f / 81;
float b = 0;
for (int i = 0; i < 729; ++ i)
        b += a;
printf("%.7g\n", b);   // prints 9.000023
```

while

```
double a = 1.0 / 81;
double b = 0;
for (int i = 0; i < 729; ++ i)
        b += a;
printf("%.15g\n", b);   // prints 8.99999999999996
```

String in C

- C uses “array” of char as a string
  - String must ends with a special character '\0'
    - char array2[] = { 'F', 'o', 'o', 'b', 'a', 'r', '\0' };
  - Alternatively, you can define a string like
    - char array2[] = “Foobar”;
  - Or using a char* “pointer”
    - char *array2 = “Foobar”;
  - In both later ways, the NULL character is hidden

How to read and write

- Examples: (using printf and scanf)

```c
/* printf example */
#include <stdio.h>

int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d %ld\n", 1977, 650000L);
   printf ("Preceding with blanks: %10d \n", 1977);
   printf ("Preceding with zeros: %010d \n", 1977);
   printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
   printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
   printf ("Width trick: %*d \n", 5, 10);
   printf ("%s \n", "A string");
   return 0;
}
```

Output:

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick:    10
A string
```

How to read and write

```c
/* scanf example */
#include <stdio.h>

int main ()
{
  char str [80];
  int i;

  printf ("Enter your family name: ");
  scanf ("%s",str);
  printf ("Enter your age: ");
  scanf ("%d",&i);
  printf ("Mr. %s , %d years old.\n",str,i);
  printf ("Enter a hexadecimal number: ");
  scanf ("%x",&i);
  printf ("You have entered %#x (%d).\n",i,i);

  return 0;
}
```

This example demonstrates some of the types that can be read with `scanf`:

```
Enter your family name: Soulie
Enter your age: 29
Mr. Soulie , 29 years old.
Enter a hexadecimal number: ff
You have entered 0xff (255).
```