Basic Data Types & Memory & Representation

Basic data types

Primitive data types are similar to JAVA:

- char
- int
- short
- long

- float
- double

Unlike in JAVA, All can be **signed/unsigned**

Default: signed

Unlike in JAVA, the **types sizes are machine dependant**!

Memory – few definitions

Bit – a binary digit - zero or one

Byte – 8 bits

Not C specific

Basic data types

The types sizes must obey the following rules.

1. Size of char = 1 byte
2. Size of short ≤ size of int ≤ size of long

<u>Undetermined type sizes</u>

Advantage:

- hardware support for arithmetic operations

Disadvantage:

- problems with porting code from one machine to another

signed VS. unsigned

Each type could be signed or unsigned

```
int negNum = -3;
int posNum = 3;
unsinged int posNum = 3;
unsinged int posNum = -3;
```

Integers binary representation

Bit Pattern	Unsigned	2's Complement
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
•	•	•
•	•	•
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
•	•	•
•	•	•
1111 1110	254	-2
1111 1111	255	-1

Integers binary representation

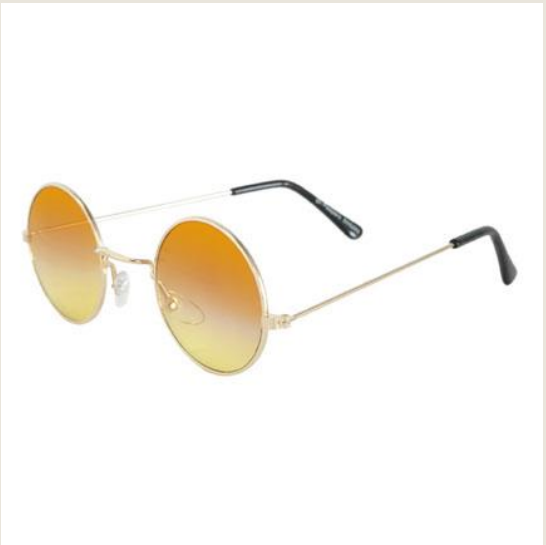
- **unsigned** integers types are represented with the binary representation of the number they stand for.
- The representation range therefore is…$[0,2^x-1]$

  Where x is the size in bits of the type

- The exact representation of **signed** integers types is machine dependant.
- The most popular representation is, the two-complement representation
- The representation range is: $[-2^{x-1},2^{x-1}-1]$

Integers binary representation

```
short num = -1;

printf("%u",num);
```



The output on some machines is: 65535

What happened?

-1 -> 11111111 11111111(signed short representation).

1111111...1->65535 when interpreted as **unsigned** short.

Integers binary representation

Bit Pattern	Unsigned	2's Complement
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
•	•	•
•	•	•
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
•	•	•
•	•	•
1111 1110	254	-2
1111 1111	255	-1

The sizeof() operator

The operator sizeof (type) returns the size of the type:

```
printf("%lu",sizeof(char)); //charSize == 1
```

Primitive types & sizeof

```
int main()
{
  //Basic primitive types
  printf("sizeof(char)   = %lu\n",sizeof(char));
  printf("sizeof(int)    = %lu\n",sizeof(int));
  printf("sizeof(float)  = %lu\n",sizeof(float));
  printf("sizeof(double) = %lu\n",sizeof(double));

  //Other types:
  printf("sizeof(void*)  = %lu\n",sizeof(void*));
  printf("sizeof(long double)= %lu\n",sizeof(long double));
  return 0;
}
```

Integral types - characters

chars can represent small integers or a character code.

Examples:

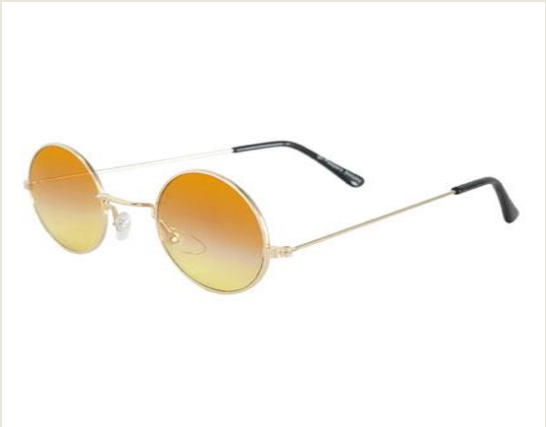
- `char c = ‘A’;`
- `char c = 65;`

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	&#127;	DEL

Source: www.LookupTables.com

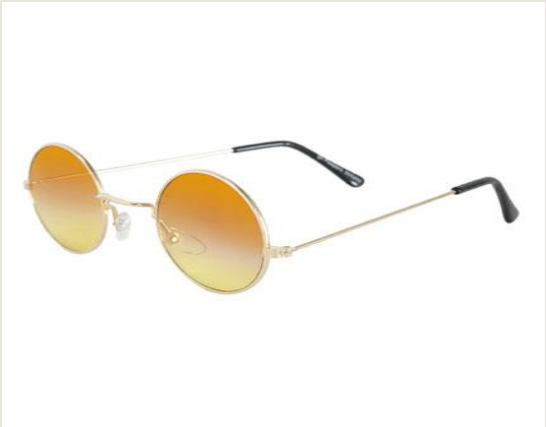
Arithmetic with character variables

```
char ch = 'A';

printf("The character %c has the

ASCII code %u.\n", ch, ch);
```



Arithmetic with character variables

```
for (char ch= 'A'; ch <= 'Z'; ++ch)
{
    printf("%c", ch);
}
```



General Input/Output

```c
#include <stdio.h>
int main()
{
   int n;
   float q;
   double w;
   printf("Please enter an int, a float and a double\n");
   scanf("%d %f %lf",&n,&q,&w);
   printf("ok, I got: n=%d, q=%f, w=%lf",n,q,w);
   return 0;
}
```

& "%d %f %lf

Characters input and output

```c
#include <stdio.h>
int main ()
{
   char c;
   printf("Enter character:");
   c=getchar();
   printf("You entered:");
   putchar(c);
   return 0;
}
```

Printing MAX_LINES

```
#include <stdio.h>
#define MAX_LINES 10
int main()
{
    int n = 0;
    int c;
    while(((c=getchar())!=EOF) && (n<MAX_LINES) )
    {
        putchar(c);
        if( c == '\n' )
          n++;
    }
    return 0;
}
```

Boolean types

Boolean type **doesn't exist in C**!

Use *char/int instead (there is also a possibility to work on bits)*

zero = false

non-zero = true

Examples:

```
while (1)
{
}

infinite
loop
```

```
if (-1974)
{
}

true
statement
```

```
i = (3==4);

i equals
zero
```

Boolean types

Boolean type **doesn't exist in C**!

Use *char/int instead (there is also a possibility to work on bits)*

zero = false

non-zero = true

Examples:

```
#define TRUE 1
while (TRUE)
{
}

infinite loop
```

```
while (1)
{
}

infinite
loop
```

```
if (-1974)
{
}

true
statement
```

```
i = (3==4);

i equals
zero
```

Casting and Type Conversion

Operands with different types get converted when you do arithmetic.

Everything is converted to the type of the **floatiest, longest operand, signed if possible without losing bits**

Casting possible between all primitive types.

Casting up - usually automatic:

- float => double
- short => int => long
  etc.

```
int i;
short s;
long l;
i=s;  // no problem
l=i;  // no problem
s=l;  // might lose info,
      // warning not
guaranteed
```

Casting and Type Conversion

Operands with different types get converted when you do arithmetic.

Everything is converted to the type of the **floatiest, longest operand, signed if possible without losing bits**

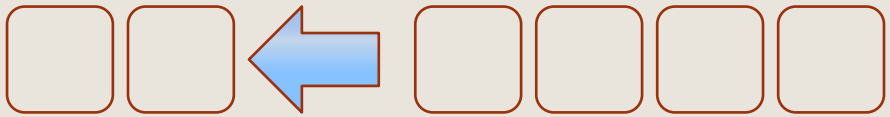
Casting possible between all primitive types.

Casting up - usually automatic:

- float => double
- short => int => long
  etc.

Casting down – warning !

```
int i;
short s;
long l;
i=s;  // no problem
l=i;  // no problem
s=l;  // might lose info,
      // warning not
guaranteed
```



Integer division

Mathematical operators on only int operands

- The result is int
- Integer numbers are treated as “int”

```
float f=1/3;            //0
float f=(float)1/3;     //0.3333..
float f=1/3.0;          //0.3333..
```

Expressions as Values

&& - logical “and”

- & - bitwise “and”

|| - logical “or”

- | - bitwise “or”

bitwise – end of the year

Evaluation:

```
int i=0;
if (i==1 && x=isValid())
{
    …
}
```

Might not be evaluated

Functions declarations and definitions

Declarations and definitions

- Function declaration:
  - Specification of the function prototype.
  - No specification of the function operations.

- Function definition
  - Specification of the exact operations the function performs.

Declarations and definitions

- "One definition rule": many declarations, one definition.
- Examples: many function declarations (no body, just "signature", one definition – with body. Legal but bad style

```
int f(),f(int);
int f(int);
int f(int a);
int f(int a)
{
   return a*2;
}
```

  - Later on we will see the same with `structs`, and see cases where for a variable a declaration isn't the same as definition (=storage allocation).

Function signature

Syntactically you declare a function exactly the same as you do a variable:

<type> <name>

With the added “modifier” of “()” to the name.

```
int g(), f, x,i, k();
```

The above line declares 3 int’s and two int functions.

Function signature

In **C** (not c++) function signature is composed only from it's name and return type:

```
int f(int a,int b);
```

Has the same type as

```
int f(float c);
```

Why? Once upon a time (80') you did not write parameter names in function declarations or definitions, so all functions where like this:

int f(), g();

And you would need to remember what arguments to pass…

Function signature

Today it is still legal but it's a very bad style and error prone.

This leads to a side effect that:

```
int f(int a,char b);
int f(float a)
{
   return a*2;
}
int main()
{
   f(5,'a');
}
```

Will **compile and run**, and call the second f! But, If you will try to **define** also the first signature – you will get an error.

Function signature

Today it is still legal but it's a very bad style and error prone.

This leads to a side effect that:

```
int f(int
int f(flo
{
    return
}
int main(
{
    f(5,'a
}
```

No overloads.

If you use high warning level, above code will give you some warning. Pay attention.

To get these warnings with functions that has no arguments declare:

```
int f(void);
```

Will **compile and run**, and call the second f! But, If you will try to **define** also the first signature – you will get an error.