YORK UNIVERSITÉ UNIVERSITY

Introduction to C

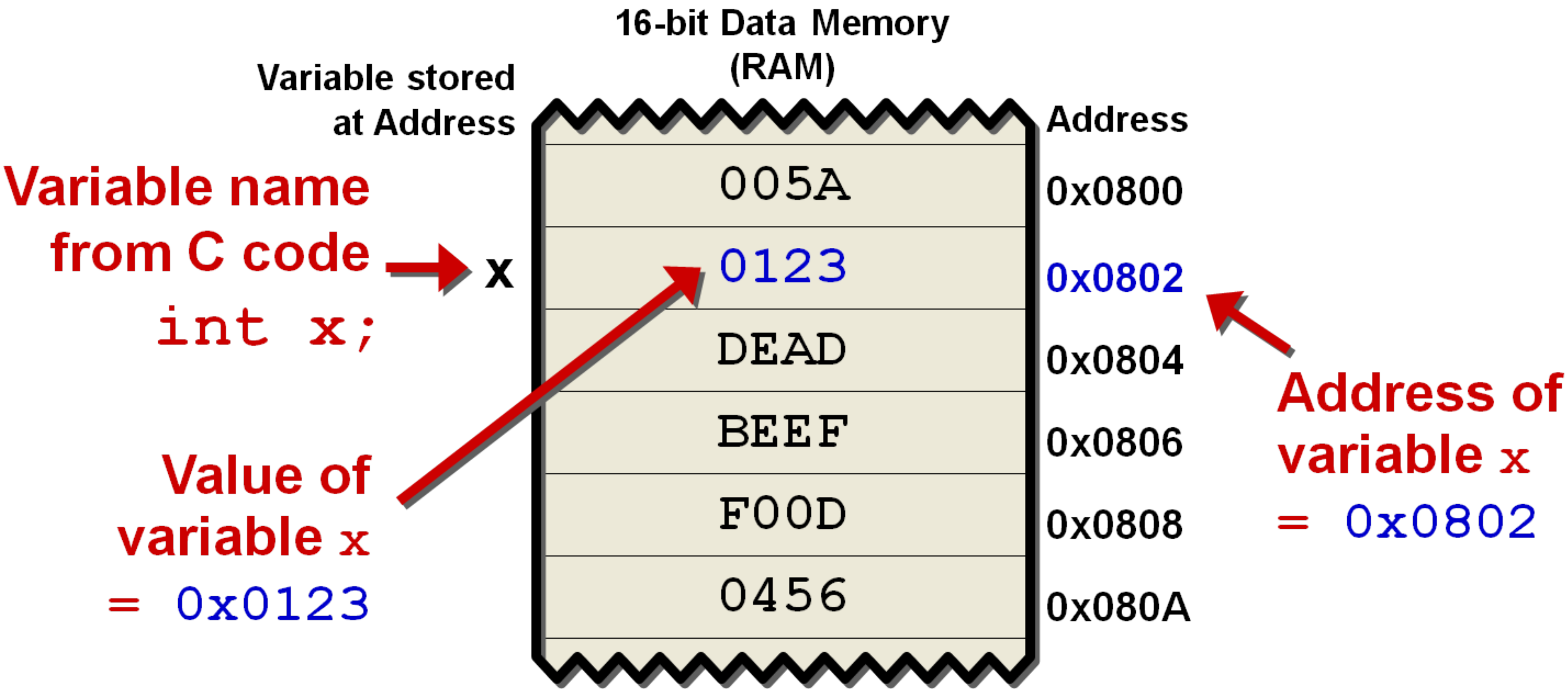
EECS 2031

Song Wang
wangsong@eecs.yorku.ca
eecs.yorku.ca/~wangsong/

Acknowledgement

- Some of the covered materials are based on previous EECS2031 offerings:
  - Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

printf() and scanf()



Variable stored at Address	16-bit Data Memory (RAM)	Address
	005A	0x0800
x	0123	0x0802
	DEAD	0x0804
	BEEF	0x0806
	F00D	0x0808
	0456	0x080A

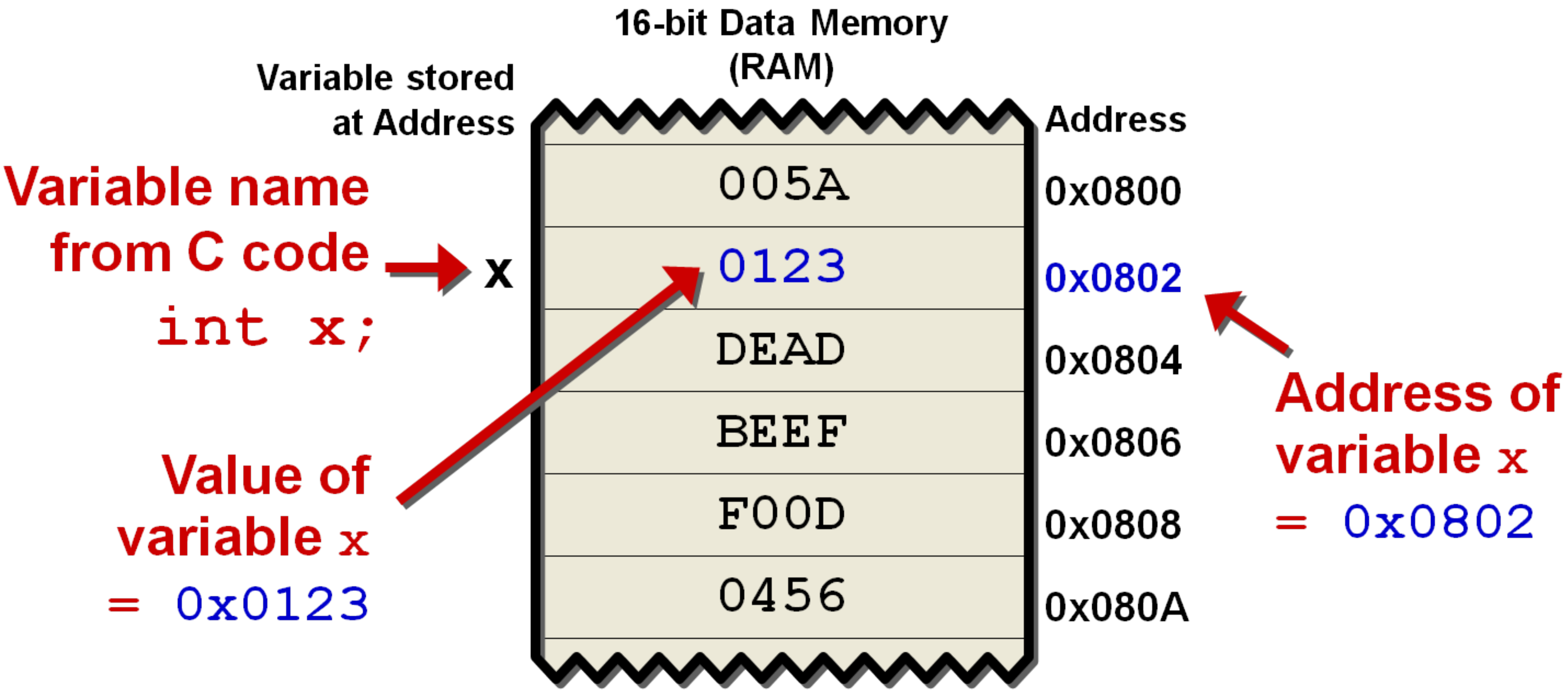
Variable name from C code

`int x;`

Value of variable x = 0x0123

Address of variable x = 0x0802

printf() and scanf()



Variable stored at Address	16-bit Data Memory (RAM)	Address
	005A	0x0800
x	0123	0x0802
	DEAD	0x0804
	BEEF	0x0806
	F00D	0x0808
	0456	0x080A

Variable name from C code

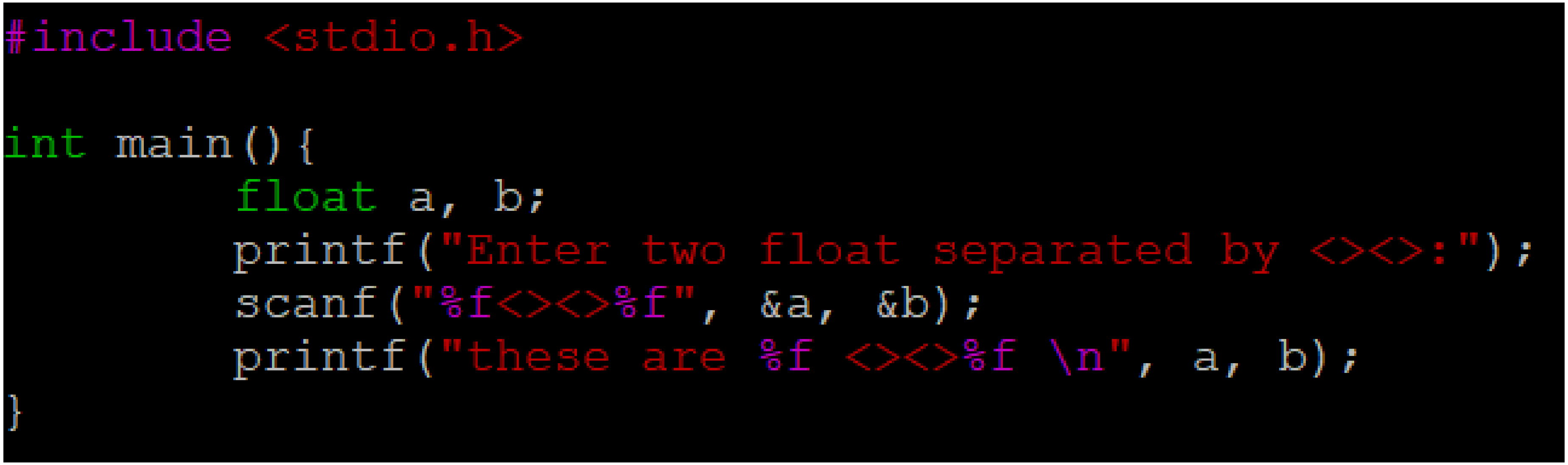
`int x;`

Value of variable x = 0x0123

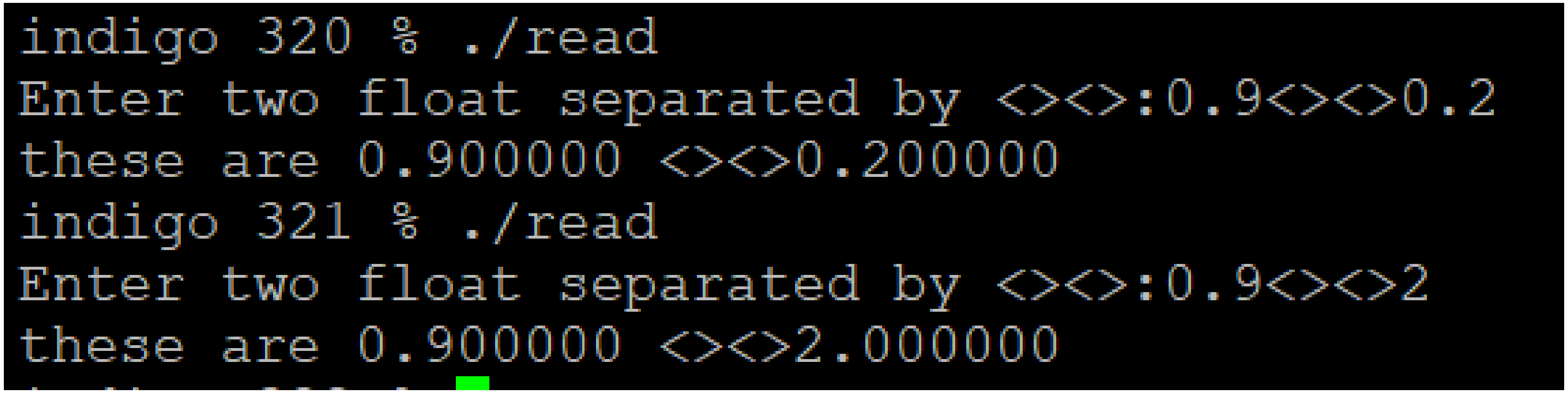
Address of variable x = 0x0802

```
printf(a) : values of variables
scanf(&a): addresses of variables
```

printf() and scanf()



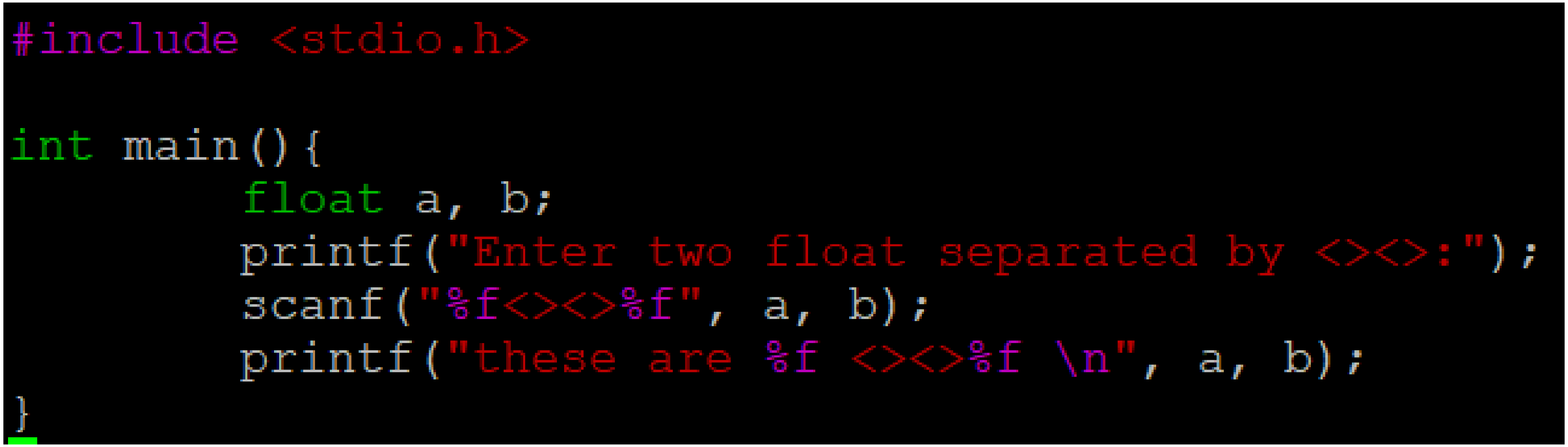
```c
#include <stdio.h>

int main(){
        float a, b;
        printf("Enter two float separated by <><>:");
        scanf("%f<><>%f", &a, &b);
        printf("these are %f <><>%f \n", a, b);
}
```



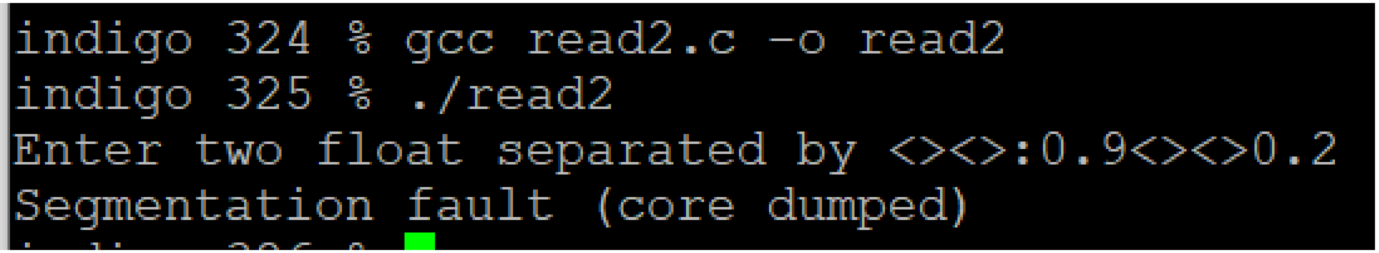
```
indigo 320 % ./read
Enter two float separated by <><>:0.9<><>0.2
these are 0.900000 <><>0.200000
indigo 321 % ./read
Enter two float separated by <><>:0.9<><>2
these are 0.900000 <><>2.000000
```

printf() and scanf()

If we use “a” and “b” as the input for scanf():scanf(“%f<><>%f”, a, b);



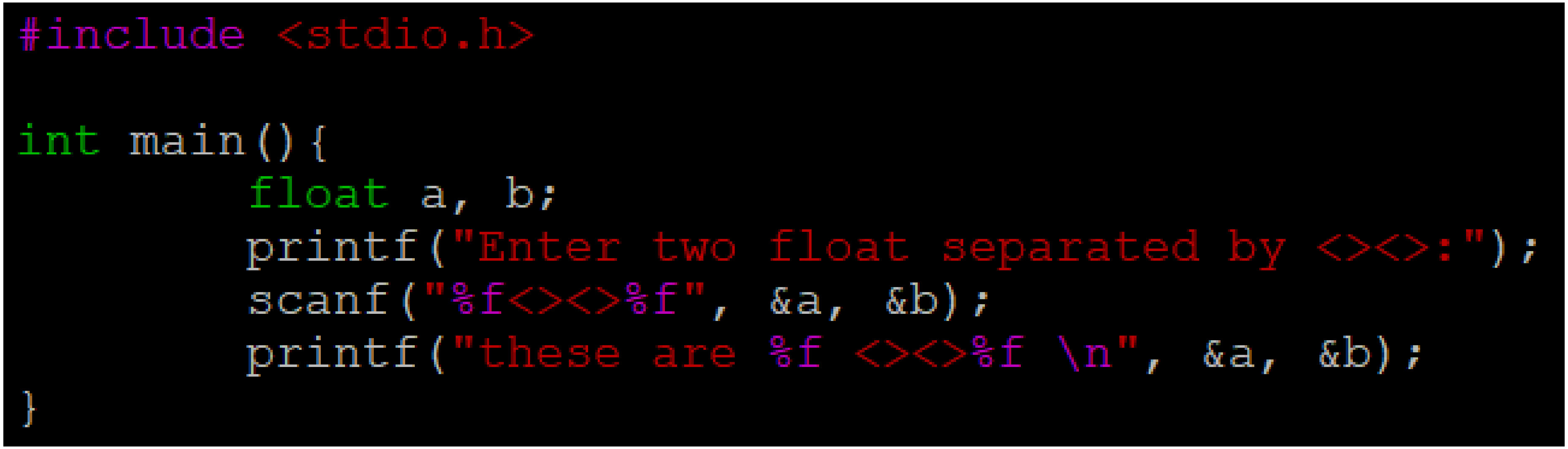
```
#include <stdio.h>

int main(){
        float a, b;
        printf("Enter two float separated by <><>:");
        scanf("%f<><>%f", a, b);
        printf("these are %f <><>%f \n", a, b);
}
```



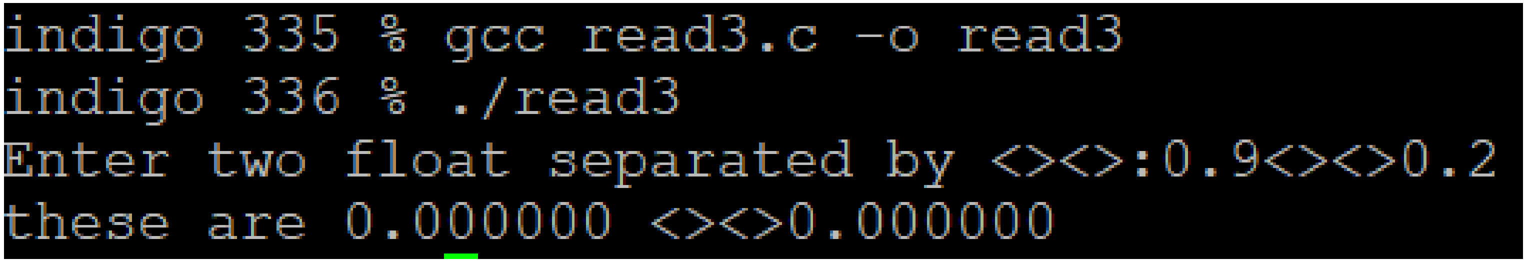
```
indigo 324 % gcc read2.c -o read2
indigo 325 % ./read2
Enter two float separated by <><>:0.9<><>0.2
Segmentation fault (core dumped)
indigo 326 %
```

printf() and scanf()

If we use “&a” and “&b” as the input for printf():printf(“these are %f<><>%f”, &a, &b);



```c
#include <stdio.h>

int main(){
        float a, b;
        printf("Enter two float separated by <><>:");
        scanf("%f<><>%f", &a, &b);
        printf("these are %f <><>%f \n", &a, &b);
}
```



```
indigo 335 % gcc read3.c -o read3
indigo 336 % ./read3
Enter two float separated by <><>:0.9<><>0.2
these are 0.000000 <><>0.000000
```

getchar, putchar

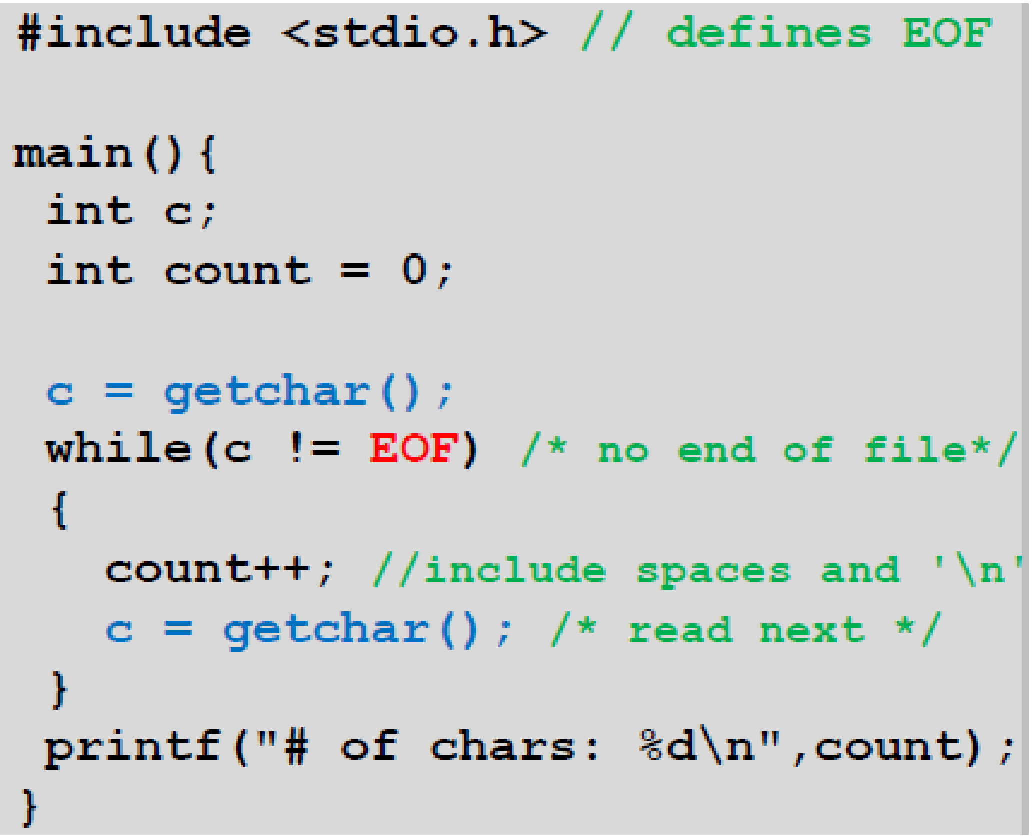
- **`int getchar(void)`**
  - To read one character at a time from the standard input
  - Returns the next input char each time it is called;
  - Returns EOFwhen it encounters end of file.oend of file; Using < : end of input file
  - keyboard: Ctrl-D (Unix) or Ctrl-Z (Windows).   “Keyboard is a file”

- EOF: an **`int`** constant defined in <**`stdio.h`**>, value is -1.

- •**`int putchar(int c)`**
  - Puts the character c on the *standard output*
  - Returns the character written (usually ignored);
  - Like **`printf("%c", c);`**

getchar, putchar

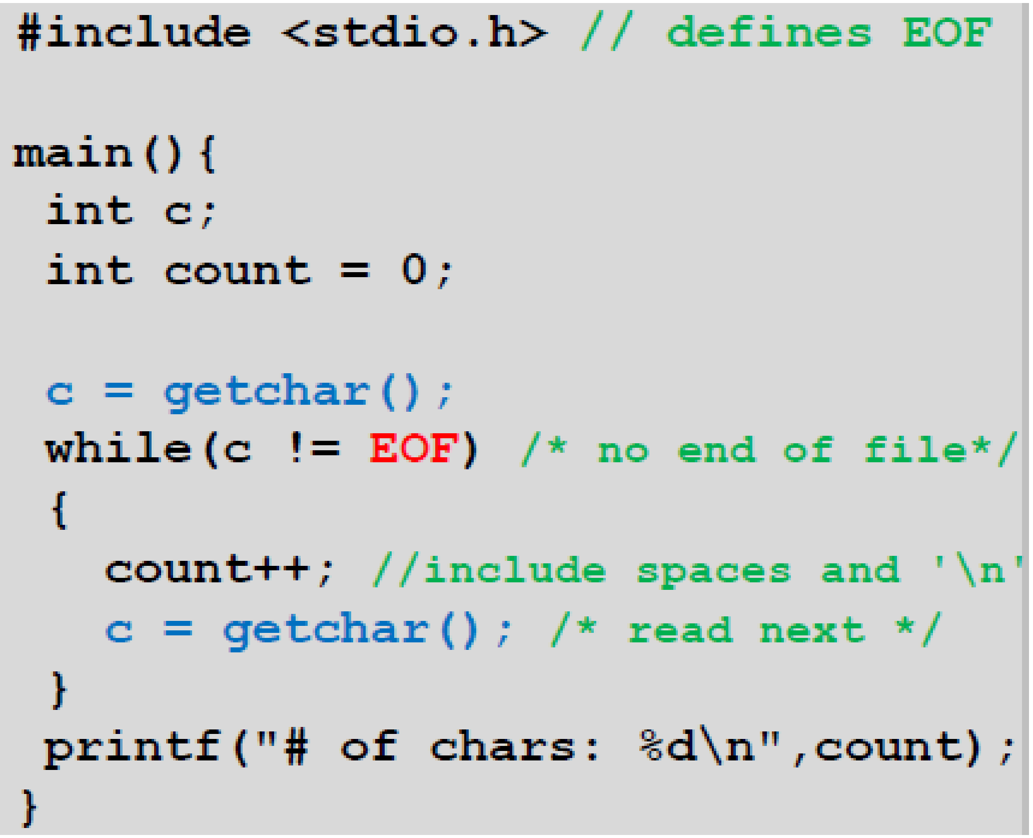
- **countChar.c**



```
#include <stdio.h> // defines EOF

main(){
 int c;
 int count = 0;

 c = getchar();
 while(c != EOF) /* no end of file*/
 {
   count++; //include spaces and '\n'
   c = getchar(); /* read next */
 }
 printf("# of chars: %d\n",count);
}
```

getchar, putchar

- **countChar.c**

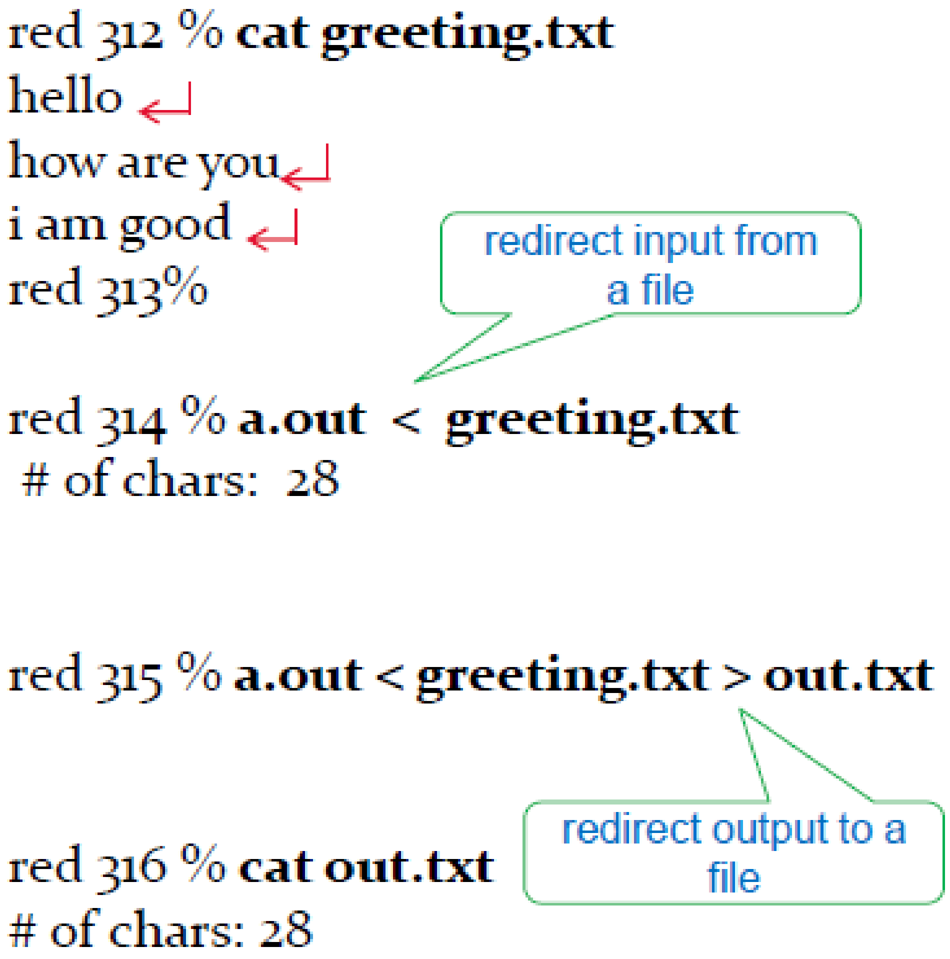


```
#include <stdio.h> // defines EOF

main(){
 int c;
 int count = 0;

 c = getchar();
 while(c != EOF) /* no end of file*/
 {
   count++; //include spaces and '\n'
   c = getchar(); /* read next */
 }
 printf("# of chars: %d\n",count);
}
```

```
red 309 % a.out
hello↵
how are yo↵
iam goo↵
Ctrl + D (end of the input)

# of chars: 28
```

getchar, putchar

- Redirected from file



```
red 312 % cat greeting.txt
hello ↵
how are you↵
i am good ↵
red 313%

red 314 % a.out  <  greeting.txt
 # of chars:  28


red 315 % a.out < greeting.txt > out.txt


red 316 % cat out.txt
# of chars: 28
```

redirect input from a file

redirect output to a file

Statements

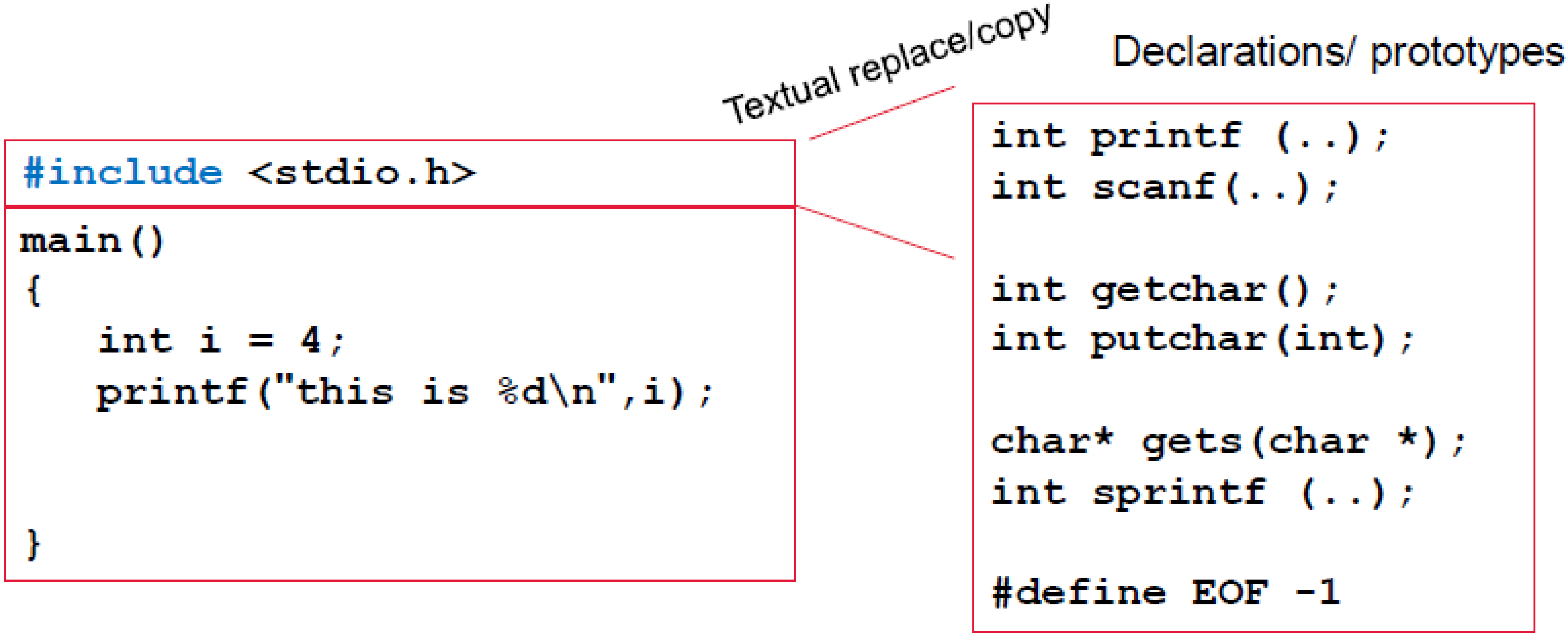
- Expression statement
  - y = i+1; i++; x = 4;
- Function call statement
  - printf("the result is %d");
- Control flow statement
  - if else, for(), while(), do while, case switch

Expression

- Formed by combining **operands**(variable, constants and function calls) using **operators (+ -* % > < == !=**)

- Has return values
  - `x+1`
  - `i < 20`   false: 0   true: 1   `printf("%d", i<20);`
  - `sum (i+j)`
  - `x = 5    = is an operator in C (and Java)! Return value 5`
  - `x = k + sum(i,j)`   `printf("%d", x=5);`

Preprocessing: # include, #define



Textual replace/copy

Declarations/ prototypes

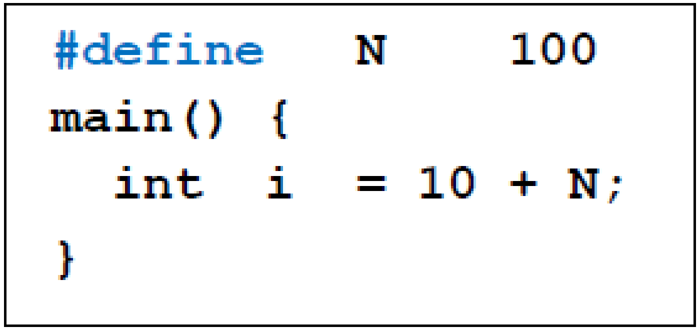
```
#include <stdio.h>
```

```
main()
{
    int i = 4;
    printf("this is %d\n",i);


}
```

```
int printf (..);
int scanf(..);

int getchar();
int putchar(int);

char* gets(char *);
int sprintf (..);

#define EOF -1
```

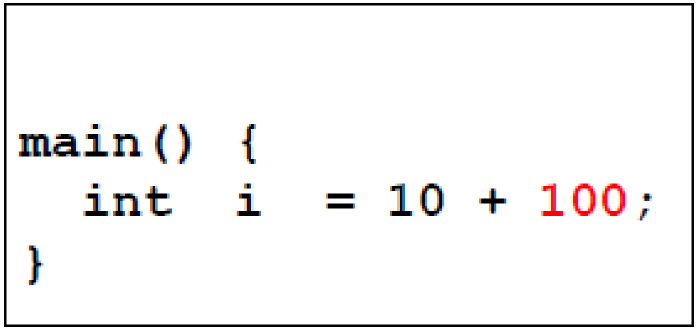
#define directive

- Syntax **`#define name value`** No type;
  - Name called symbolic **constant**, conventionally written in upper case
  - Value can be any sequence of characters



```
#define    N     100
main() {
   int  i  = 10 + N;
}
```

→

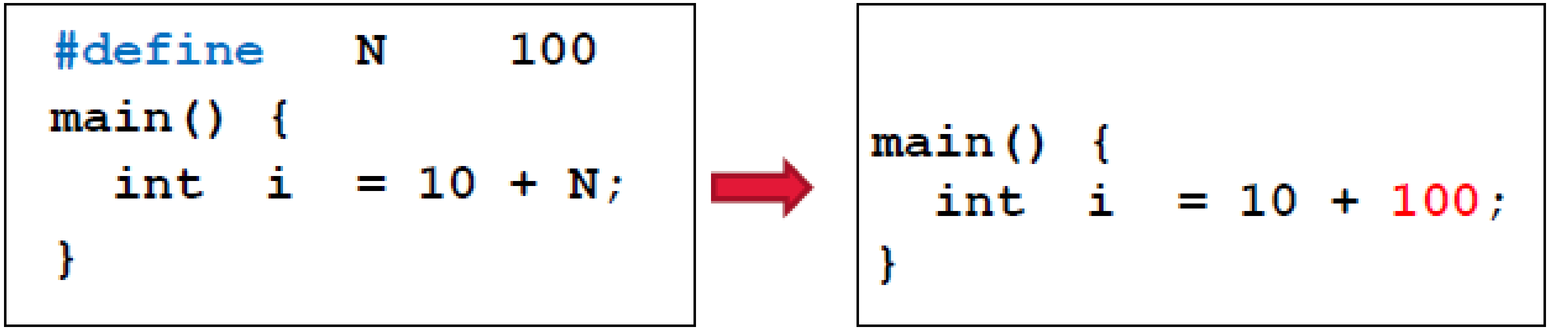


```

main() {
   int  i  = 10 + 100;
}
```

- Use as constant **`N = x + 2;`** ✗

#define directive

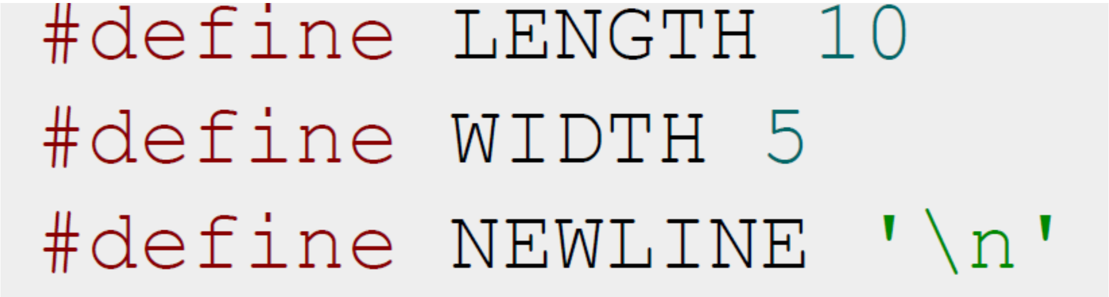
- Syntax **`#define name value`** No type;
  - Name called symbolic **constant**, conventionally written in upper case
  - Value can be any sequence of characters



```
#define    N     100
main() {
   int  i  = 10 + N;
}
```

```

main() {
   int  i  = 10 + 100;
}
```

- Use as constant **`N = x + 2;`** ✗



```
#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'
```

YORK U
UNIVERSITÉ
UNIVERSITY

C-Types, Operators, Expressions

- [Primitive/scalar] Types and sizes
  - Primitive Types
  - Constant values (literals)

- [Structured/aggregated] Array and "strings“

- Expressions
  - Basic operators
  - Type promotion and conversion
  - Other operators
  - Precedence of operators

C Primitive Types

- Variables and values have types

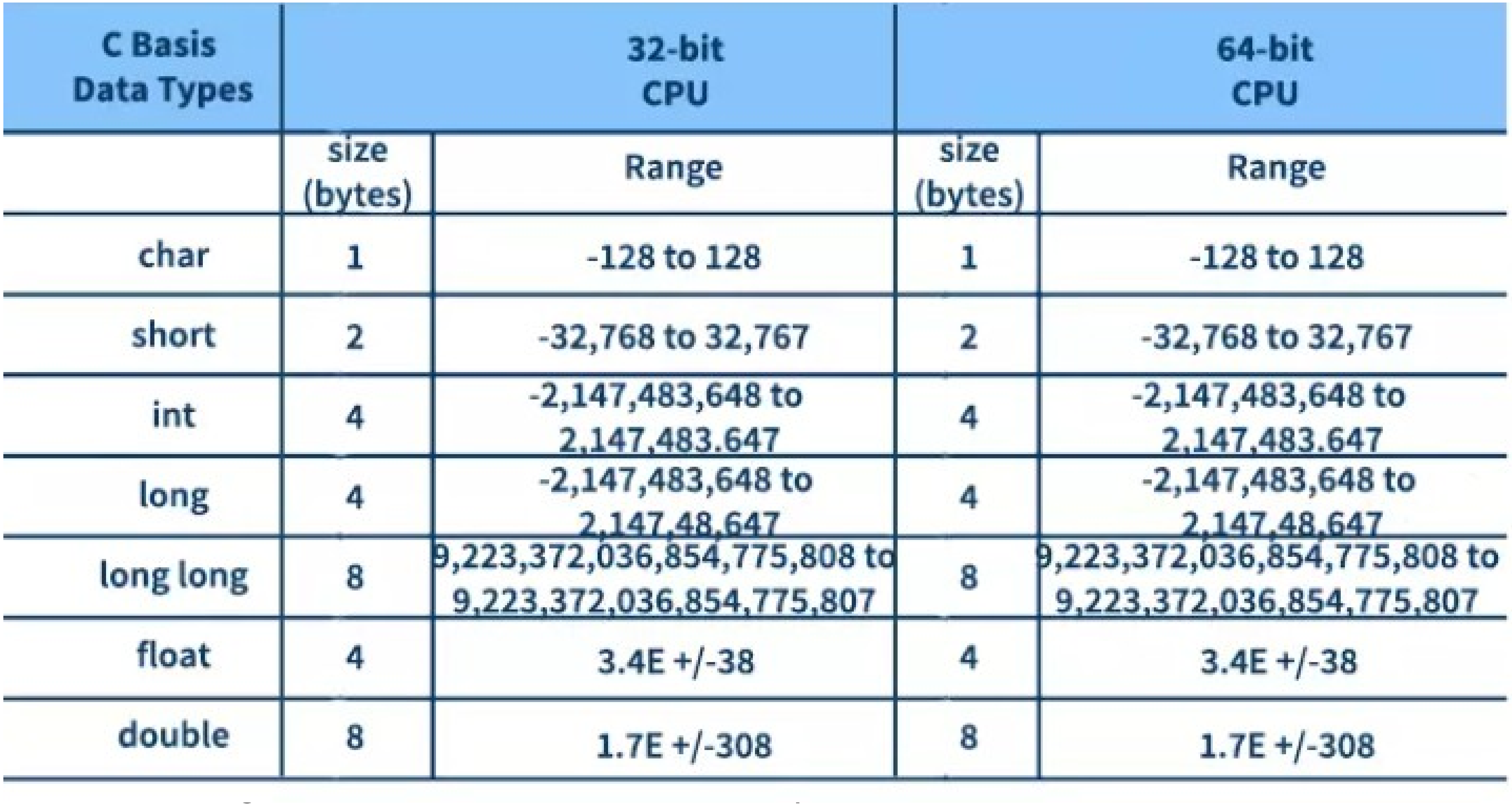
- There are two basic types in ANSI-C: integer, and floating point

- **Integer type**
  - **char** -character, 1 byte (8 bits)
  - **short [int]** -short integer, usually2 bytes (16 bits)
  - **int** -integer, usually2 or 4 bytes (16 or 32 bits)
  - **long [int]** -long integer, usually4 or 8 bytes (32 or 64 bits)

- **Floating point**
  - **float** -single-precision, usually4 bytes (32 bits)
  - **double** -double-precision, usually8 bytes (64 bits)
  - **long double** -extended-precision

C Primitive Types and Sizes

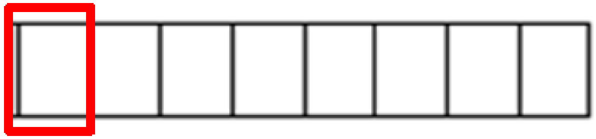
- Variables and values have types



C Basis Data Types	32-bit CPU		64-bit CPU	
	size (bytes)	Range	size (bytes)	Range
char	1	-128 to 128	1	-128 to 128
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,48,647	4	-2,147,483,648 to 2,147,48,647
long long	8	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	3.4E +/-38	4	3.4E +/-38
double	8	1.7E +/-308	8	1.7E +/-308

Qualifiers (modifiers) for integer type

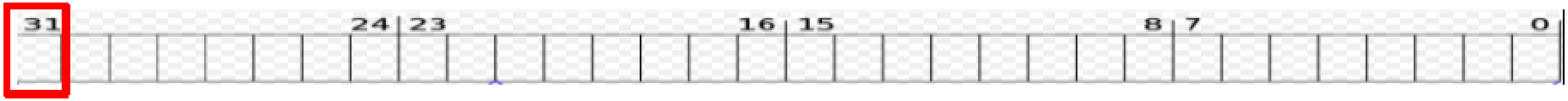
- **signed**, **unsigned** qualifiers can be applied to integer type
  - Signed: default. Left most bit signifies sign 0: positive 1: negative
  - Unsigned: positive. Left most bit contributes to magnitude too



- `[signed] char`
- `[signed] int`
- `[signed] short int`
- `[signed] long  int`

Java: no direct support for unsigned int -- always signed

- `unsigned char`
- `unsigned int`
- `unsigned short int`
- `unsigned long  int`



`unsigned int`	$0 \sim 2^{32}-1$	2^{32} values	Max: 1111111....|11111
`(signed) int`	$-2^{31} \sim 2^{31}-1$	2^{32} values	Max: 0111111....11111

Qualifiers for floating points

- “**`long`**” can be used with double:
  - **`long double`**

- Thus, there are three types of floating points:
  - **`float       /* single-precision floating point */`**
  - **`double      /* double-precision floating point */`**
  - **`long double /* extended-precision floating point */`**

- More bits, more precise: 3.1415926535….

- printf/scanf("**%f**") for float, ("**%lf**") for double, ("**%Lf**") for long double

- Storage of floating point is complicated.
  - **`float x=4.8,  float y = 6.4/2+1.6; x == y`** may not always true.

Qualifiers for floating points

- "**`long`**" can be used with double:
  - **`long double`**

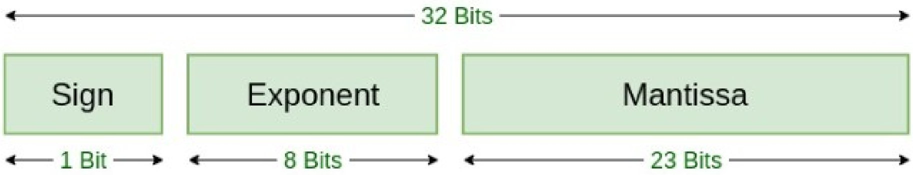
- Thus, there are three types of floating points:
  - **`float       /* single-precision floating point */`**
  - **`double      /* double-precision floating point */`**
  - **`long double /* extended-precision floating point */`**

- More bits, more precise: 3.1415926535….

- printf/scanf("%f") for float, ("%lf") for double, ("%Lf") for long double

- Storage of floating point is complicated.
  - **`float x=4.8,  float y = 6.4/2+1.6; x == y`** may not always true.

- No unsigned. All signed



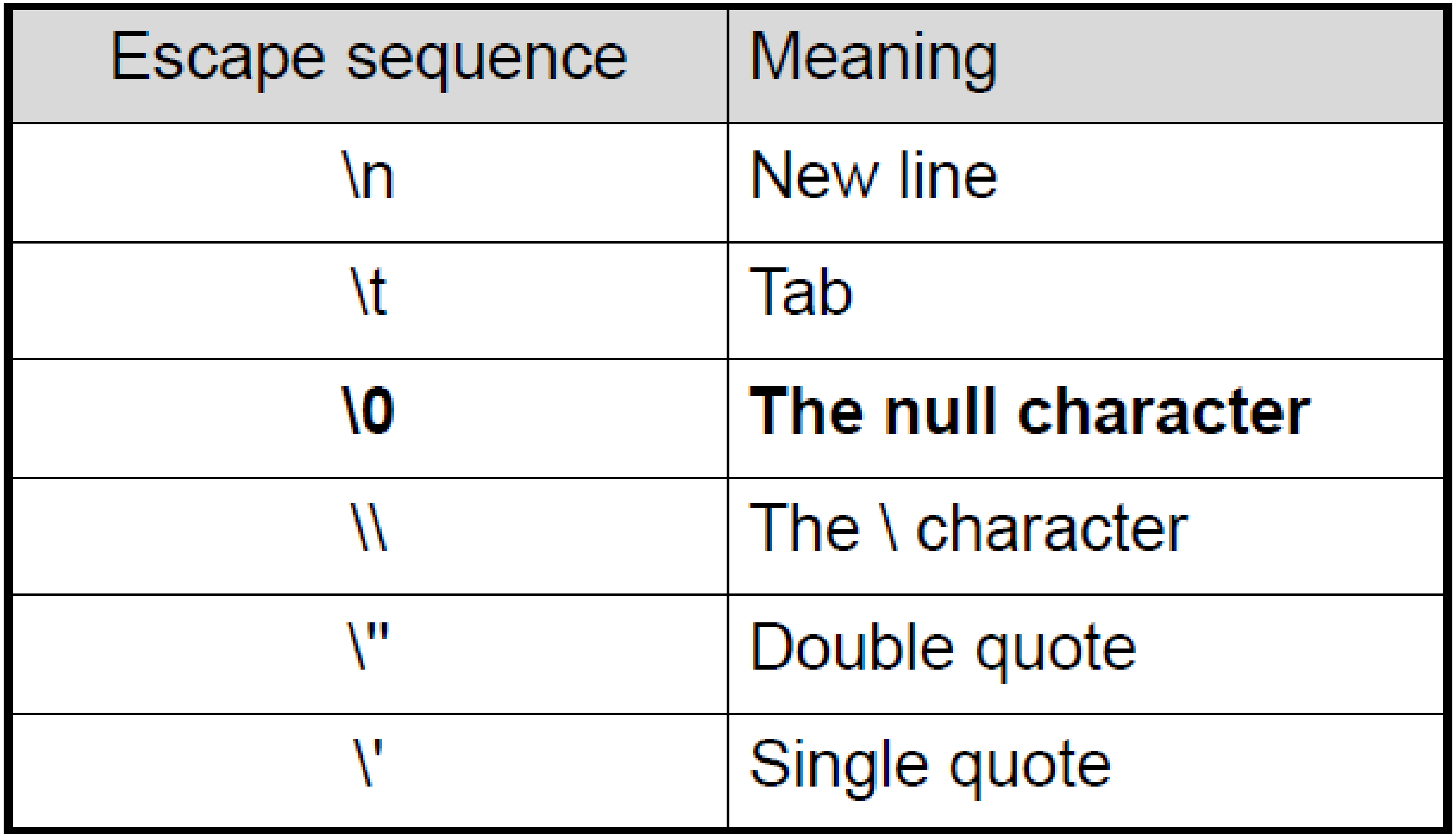
Character Constants

- A **`char`** in C is one byte (8-bit) in size (16-bit in Java)
  - Will elaborate why 8 bits,16 bits

- A constant char is specified with single quotes:
  - Regular characters: **`'A', 'C', 'z', '0', '#', '$’,…`**
  - **`char x = 'A';`**

- Special characters: invisible or control chars
  - New line, tab, del ….
  - Use escape sequence to represent

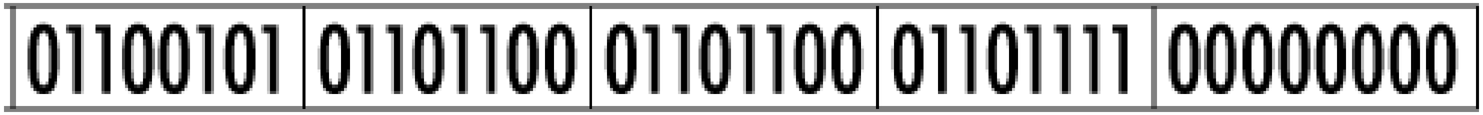
Special Characters



Escape sequence	Meaning
\n	New line
\t	Tab
\0	**The null character**
\\	The \ character
\"	Double quote
\'	Single quote

```
char c  = '\t';
char c2 = '\n';
```

Internal Representation of characters



01100101	01101100	01101100	01101111	00000000

- characters as 1/0 bits. So they are stored as (small) integer values, interpreted according to the **character set encoding** (usually ASCII, 7 bits for **128 characters**),
  - **`'a'`** has encoding **`97,`** **`'0'`** has **`48,`** **`'9'`** has **`57`**

- •Escape sequences are integers too
  - e.g. **`'\n'`** has**`10`** (newline character)
  - **`'\t'`** has**`9`** (horizontal tab)
- Special escape: **`'\0'`**
  - has encoding 0 -the null character

Internal Representation of Characters

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

Source: www.LookupTables.com

Characters

- **`chars`** are treated in C (and Java) as small integers, **`char`** variables and constants are identical to **`int`** in arithmetic expressions:
  - **char c** is converted to its encoding (index in the character set table)

-
  ```
  char aChar= 'E';   // encoding 69
   'E' + 8      //expression with value 69+8 = 77
   'E' + 'B'      //expression with value 69+66 = 135
   'E' -'B'      //expression with value 69-66 = 3
  ```

- Same for other expressions. In relational/logical expression, characters can be compared directly, comparing indexes/encodings

-
  ```
  aChar== EOF    //index == -1?
  ```

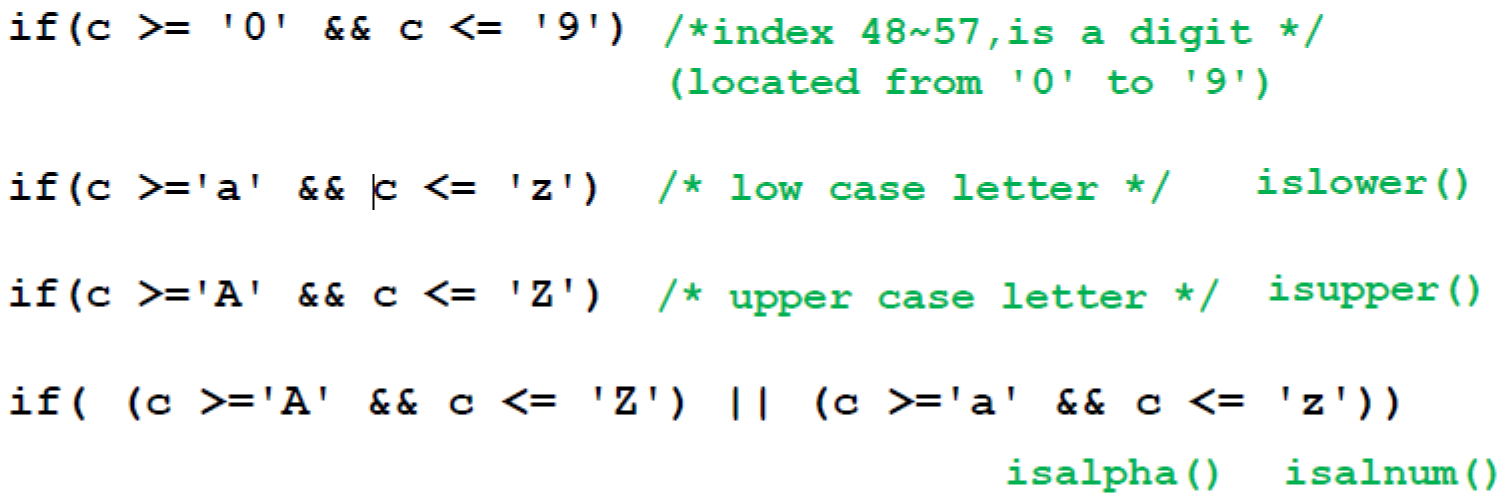
-
  ```
  aChar== 'H'    //index == 72?
  ```

-
  ```
  aChar== '\n'   // index == 10?
  ```

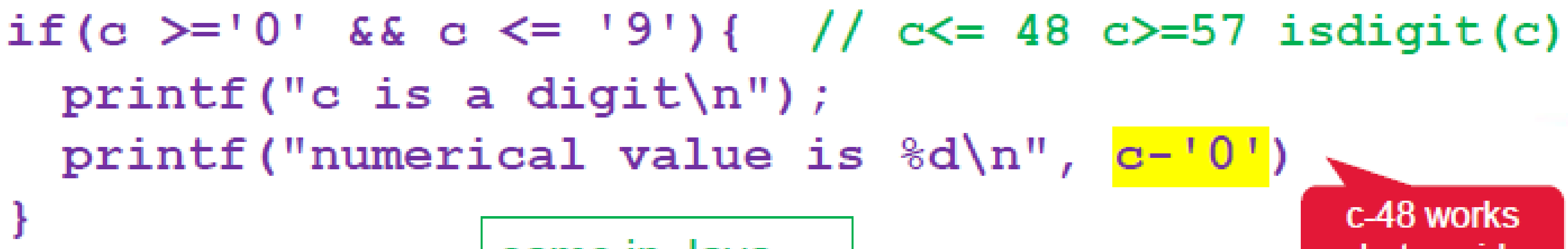
-
  ```
  aChar< 'H' //69 < 72?
  ```

Characters

- Since **`chars`** are just small integers, **`char`** variables and constants are identical to **`int`** in arithmetic expressions. Some programming idioms that take advantage of this:



```
if(c >= '0' && c <= '9') /*index 48~57,is a digit */
                          (located from '0' to '9')

if(c >='a' && c <= 'z')  /* low case letter */   islower()

if(c >='A' && c <= 'Z')  /* upper case letter */  isupper()

if( (c >='A' && c <= 'Z') || (c >='a' && c <= 'z'))
                                          isalpha()  isalnum()
```

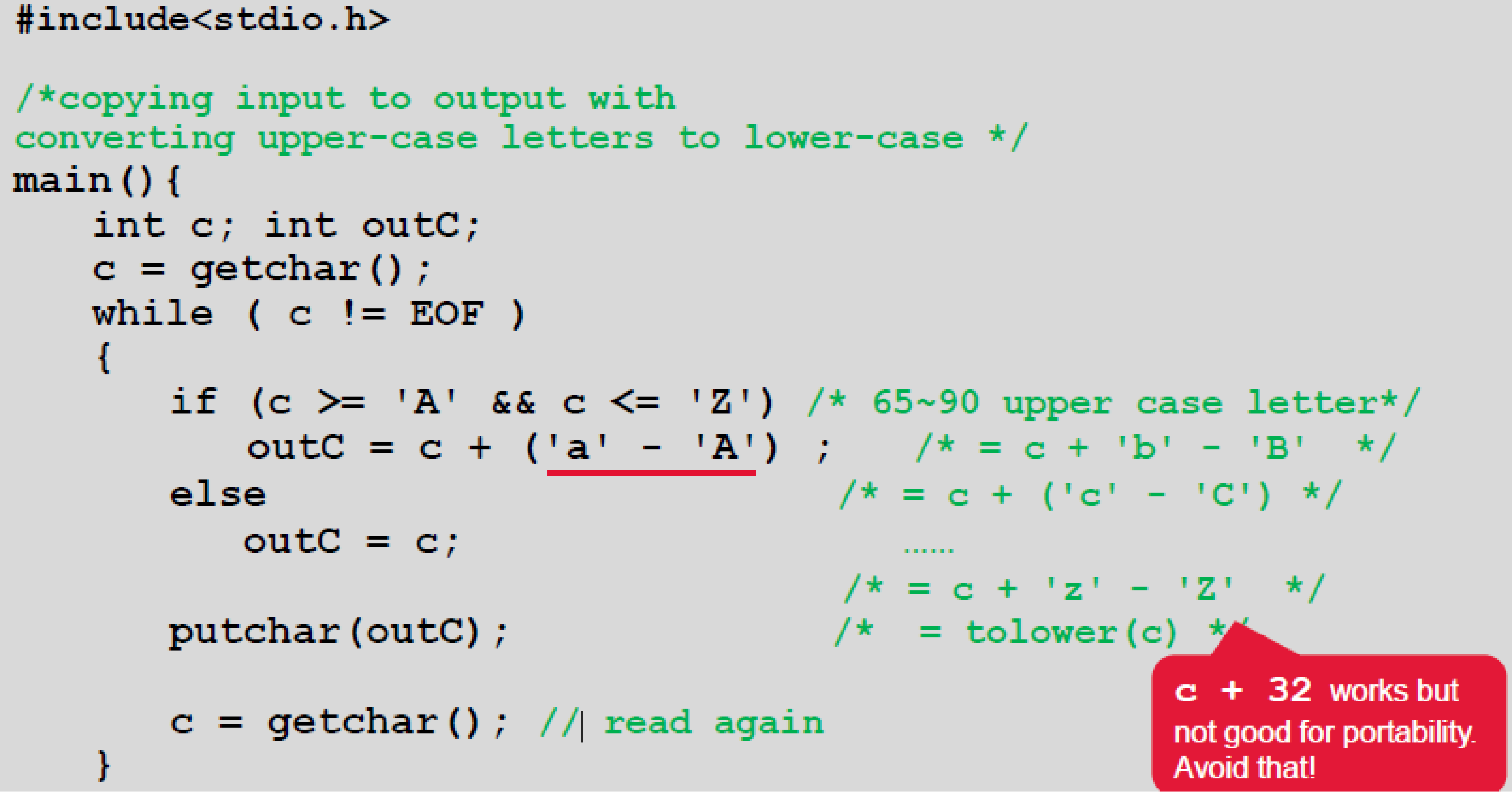


```
if(c >='0' && c <= '9'){  // c<= 48 c>=57 isdigit(c)
  printf("c is a digit\n");
  printf("numerical value is %d\n", c-'0')
}
```

same in Java

c-48 works
but avoid

Example



```c
#include<stdio.h>

/*copying input to output with
converting upper-case letters to lower-case */
main(){
    int c; int outC;
    c = getchar();
    while ( c != EOF )
    {
        if (c >= 'A' && c <= 'Z') /* 65~90 upper case letter*/
            outC = c + ('a' - 'A') ;    /* = c + 'b' - 'B'  */
        else                          /* = c + ('c' - 'C') */
            outC = c;                      ......
                                      /* = c + 'z' - 'Z'  */
        putchar(outC);                /*  = tolower(c) */

        c = getchar(); // read again
    }
```

c + 32 works but not good for portability. Avoid that!