Introduction to Computer Science
Lecture 6

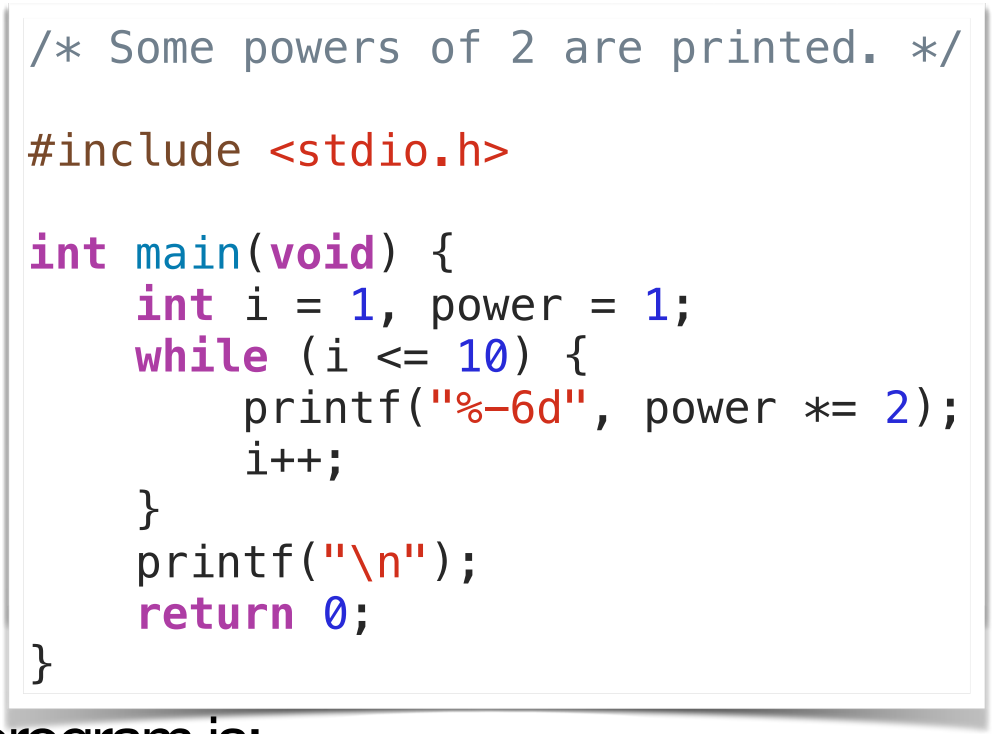
Nazareno Aguirre

(based on material by Guillaume Hoffmann)

Today's Topics

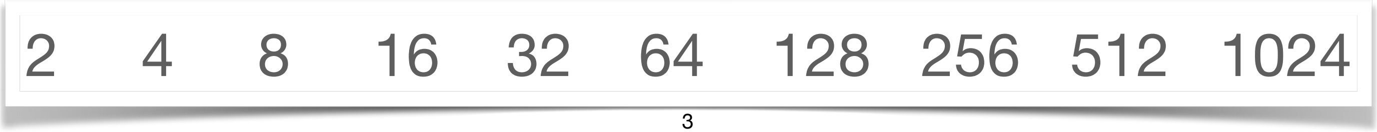
- printf() formatting
- getchar() and putchar()
- ASCII code
- Input/Output from/to the command line
- End-Of-File (EOF) signal

Example: Computing Powers of 2



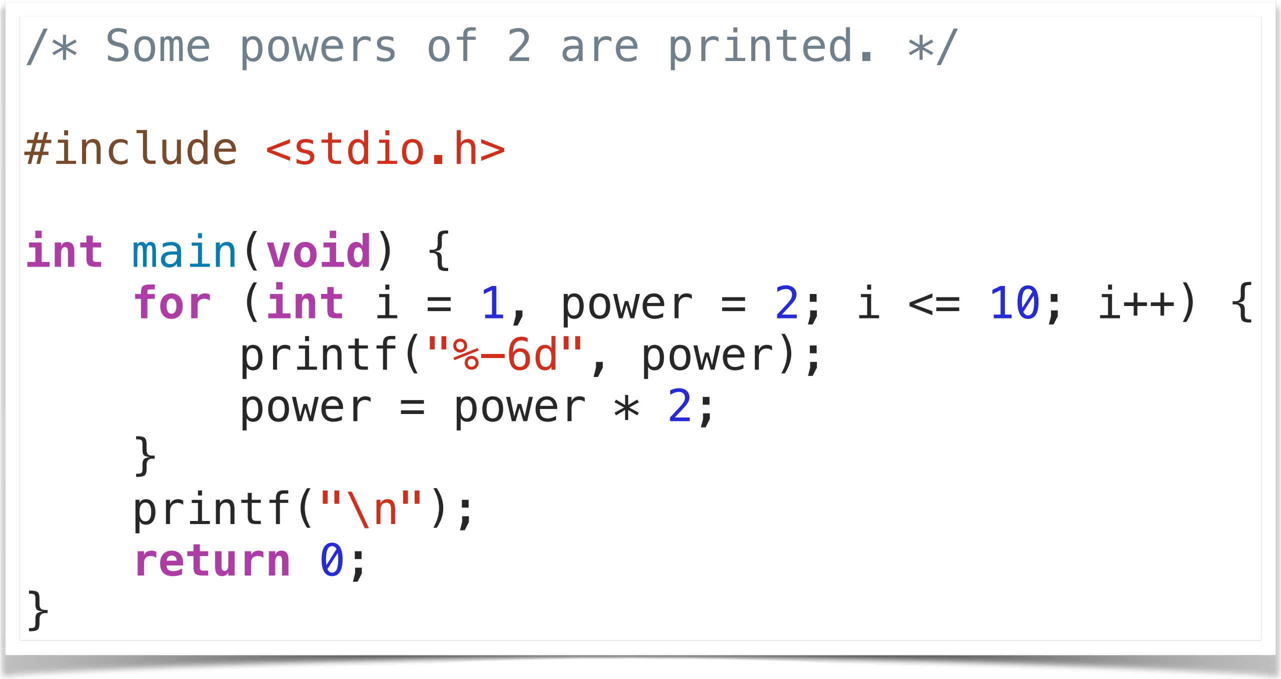
```c
/* Some powers of 2 are printed. */

#include <stdio.h>

int main(void) {
    int i = 1, power = 1;
    while (i <= 10) {
        printf("%-6d", power *= 2);
        i++;
    }
    printf("\n");
    return 0;
}
```

The output of the program is:



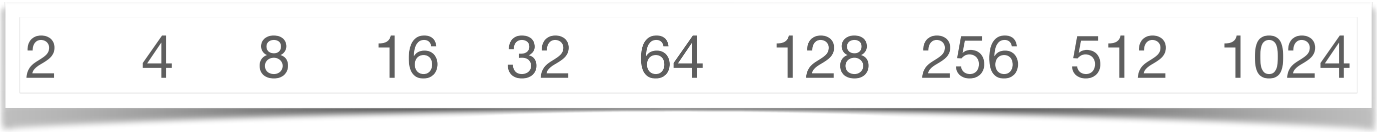
```
2     4     8     16    32    64    128   256   512   1024
```

Example: Computing Powers of 2



```c
/* Some powers of 2 are printed. */

#include <stdio.h>

int main(void) {
    for (int i = 1, power = 2; i <= 10; i++) {
        printf("%-6d", power);
        power = power * 2;
    }
    printf("\n");
    return 0;
}
```

The output of the program is:



```
2     4     8     16    32    64    128   256   512   1024
```

About printf() Formatting

```
printf("%-6d", power *= 2);
```

The placeholder `%-6d` indicates that the value is to be printed as a decimal integer with field width 6. The minus sign indicates that the value is to be left-adjusted in its field.

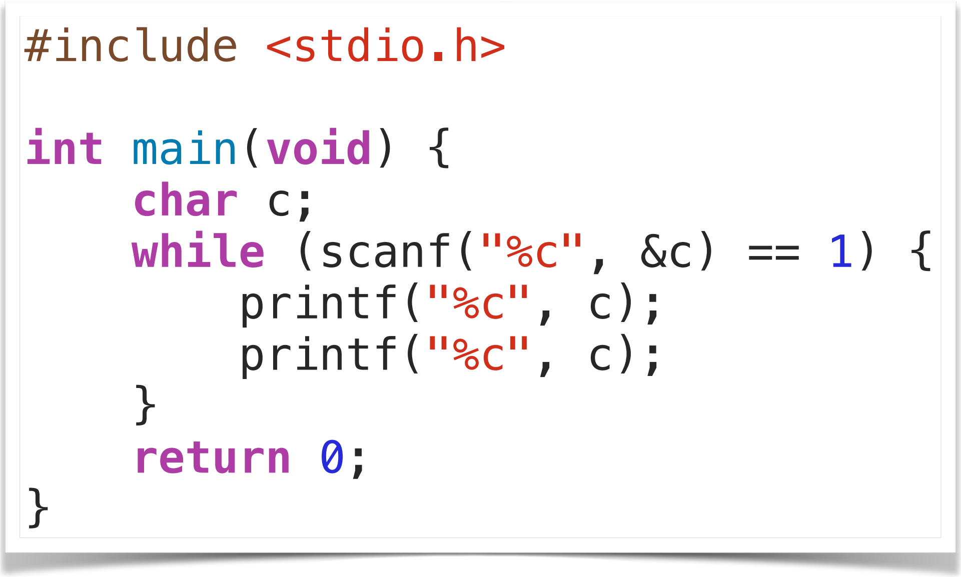
Try without the minus sign to align values to the right: `%6d`

Complete information about formatting is available in the manpage of `printf()`:

```
$ man 3 printf
```

Standard Input and Standard Output

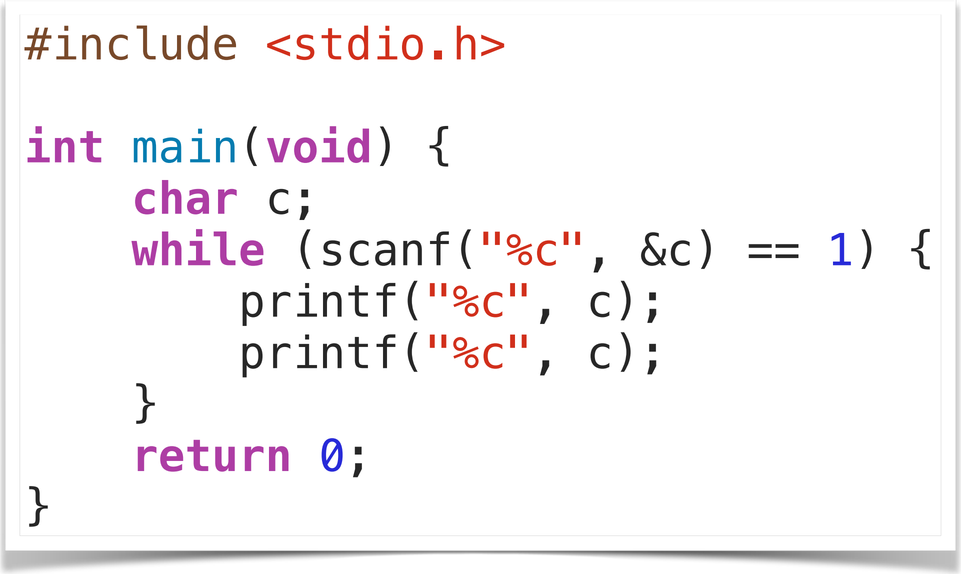
- Consider the following program
  - It reads characters from the standard input (normally the keyboard) with `scanf()`
  - It writes each character twice to the standard output (normally the terminal screen) with `printf()`
  - %c is the placeholder to read and print a single character



```c
#include <stdio.h>

int main(void) {
    char c;
    while (scanf("%c", &c) == 1) {
        printf("%c", c);
        printf("%c", c);
    }
    return 0;
}
```

Return Value of `scanf()`

- When `scanf()` is successful, it returns the number of input items successfully matched and assigned; this can be fewer than provided for, or even zero, in the event of an early matching failure.
  - Here, `scanf("%c", &c)` has a single placeholder (`%c`) so while `scanf("%c", &c)` returns value 1, the reading of input is correct.



```c
#include <stdio.h>

int main(void) {
    char c;
    while (scanf("%c", &c) == 1) {
        printf("%c", c);
        printf("%c", c);
    }
    return 0;
}
```

Redirection of Input and Output

- Suppose we compile the program into an executable `dbl_out:`

```
$ tcc -w dbl_out.c -o dbl_out
```

- We can use redirection to allow the the executable to receive input and produce output in different ways:

```
$ ./dbl_out
$ ./dbl_out < infile
$ ./dbl_out > outfile
$ ./dbl_out < infile > outfile
```

Redirection of Input and Ouput

`dbl_out:` input from keyboard (stdin), output to screen (stdout)

`dbl_out < infile:` input from file "infile", output to screen (stdout)

`dbl_out > outfile:` input from keyboard (stdin), output to file "outfile"

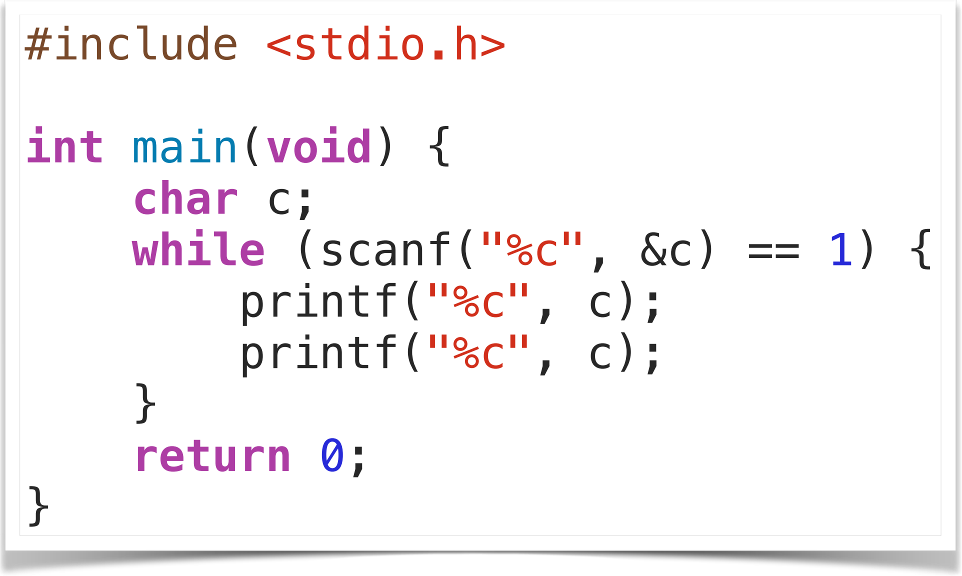
`dbl_out < infile > outfile:` input from file "infile", output to file "outfile"

Why does the loop end?

- When using this program with a standard input redirection:

```
$ ./dbl_out < infile
```

the input file is consumed. When it is completely consumed, an **End-of-File signal** is sent to the program, making `scanf()` return a special value (not 1).



```c
#include <stdio.h>

int main(void) {
    char c;
    while (scanf("%c", &c) == 1) {
        printf("%c", c);
        printf("%c", c);
    }
    return 0;
}
```

The End-of-File Signal

- When the input is taken from a file, then the end-of-file signal is automatically generated when the input file is done being fed to the program.
- When a program takes its input from the keyboard, it is necessary to generate an end-of-file signal manually.
- In Linux, control+d is the typical way to generate an end-of-file signal.

Control+c and End-of-File Are Not The Same

- The following command is of special interest:

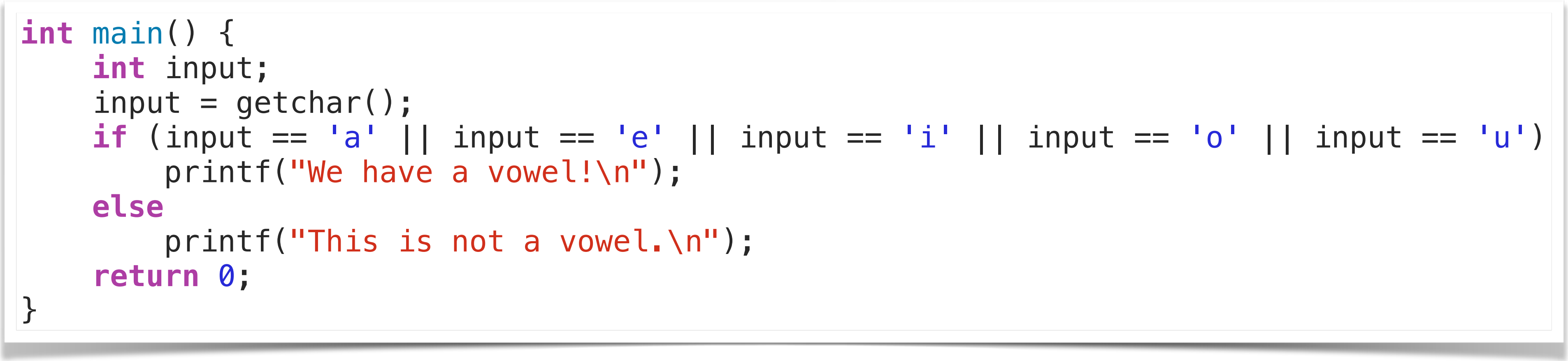
```
    $ ./dbl_out > outfile
```

- This command causes `dbl_out` to take its input from the keyboard (standard input) and to write its output in the file `outfile`, provided that you issue an end-of-file signal when you are finished.
- But if instead of typing control+d, you type control+c to kill the program, nothing gets written into `outfile`!

getchar() and putchar()

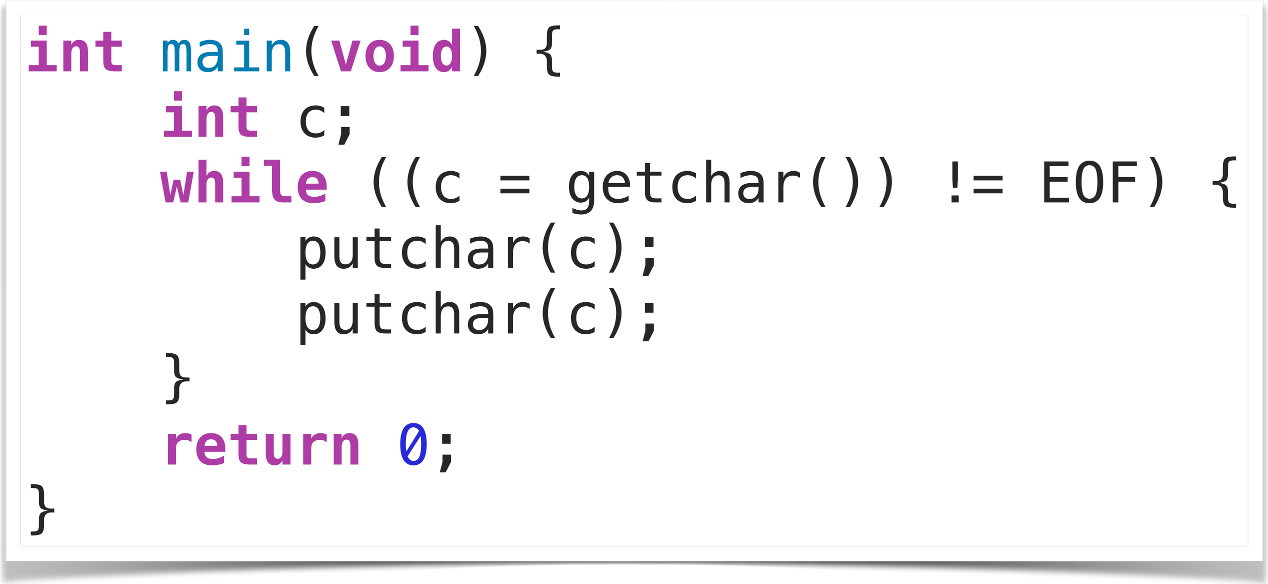
- Functions `getchar()` and `putchar()` are defined in `stdio.h`.
- They are used to read a single character from the keyboard and to write a single character to the screen, respectively.
- They are typically used to manipulate character data.
- They are sometimes more convenient to use than `scanf()` and `printf()`.

getchar() Example



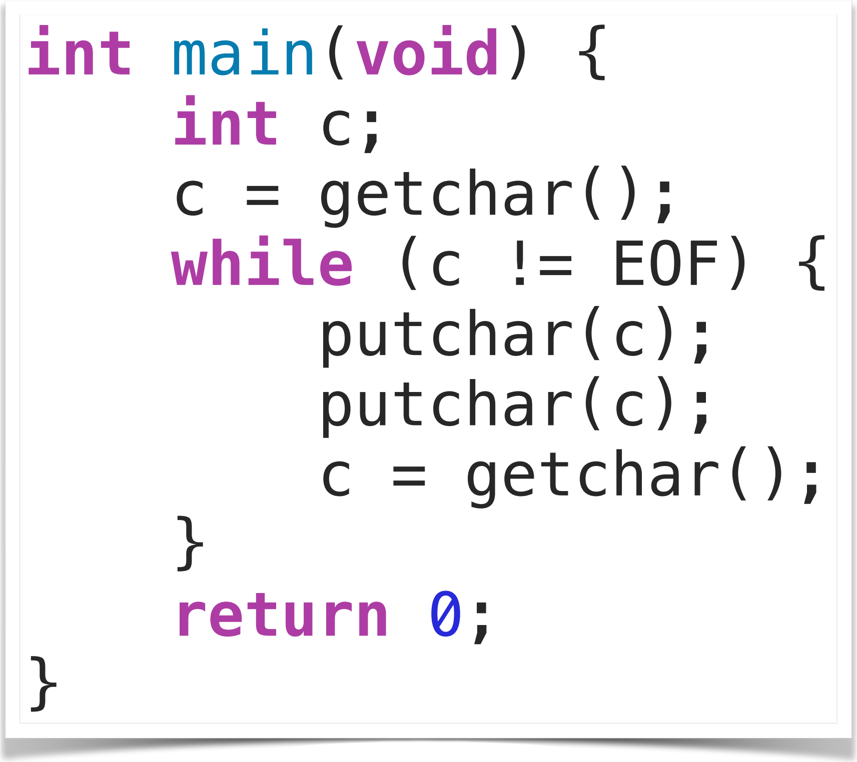
```c
int main() {
    int input;
    input = getchar();
    if (input == 'a' || input == 'e' || input == 'i' || input == 'o' || input == 'u')
        printf("We have a vowel!\n");
    else
        printf("This is not a vowel.\n");
    return 0;
}
```

Example with getchar() and putchar()



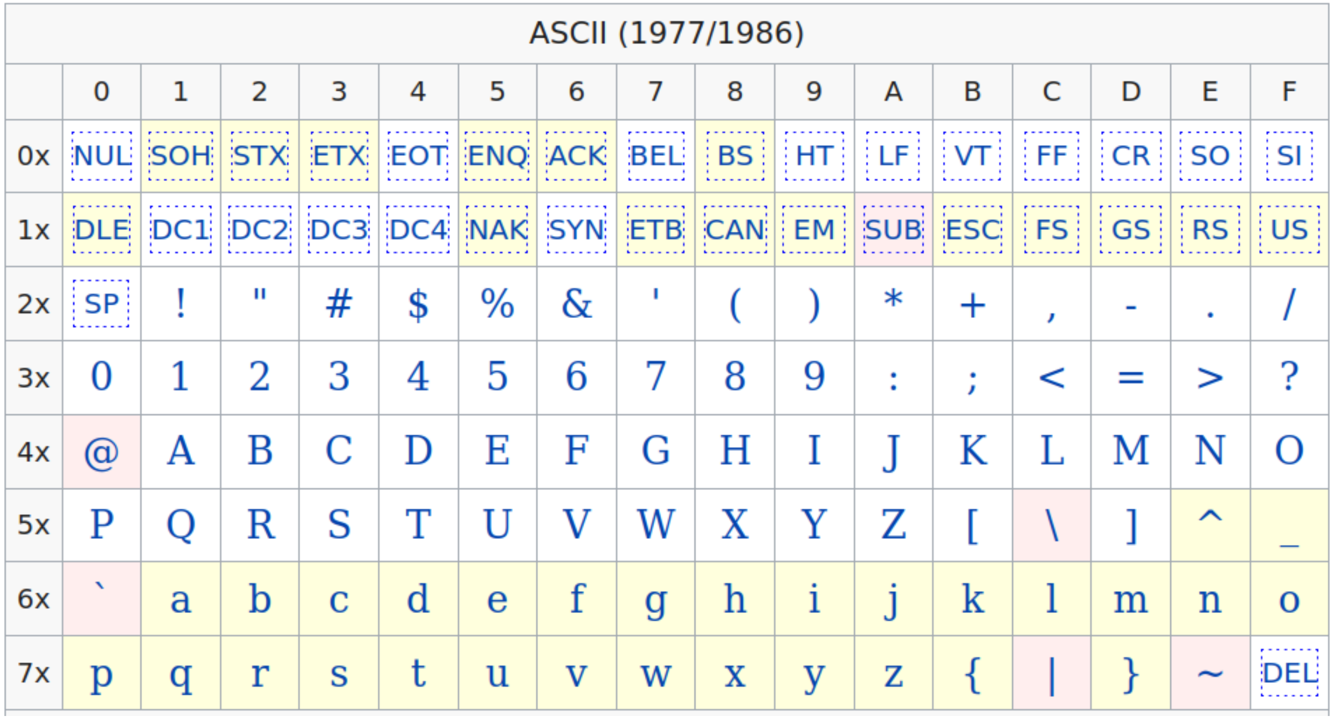
```c
int main(void) {
    int c;
    while ((c = getchar()) != EOF) {
        putchar(c);
        putchar(c);
    }
    return 0;
}
```

Example with getchar() and putchar()



```c
int main(void) {
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        putchar(c);
        c = getchar();
    }
    return 0;
}
```

ASCII



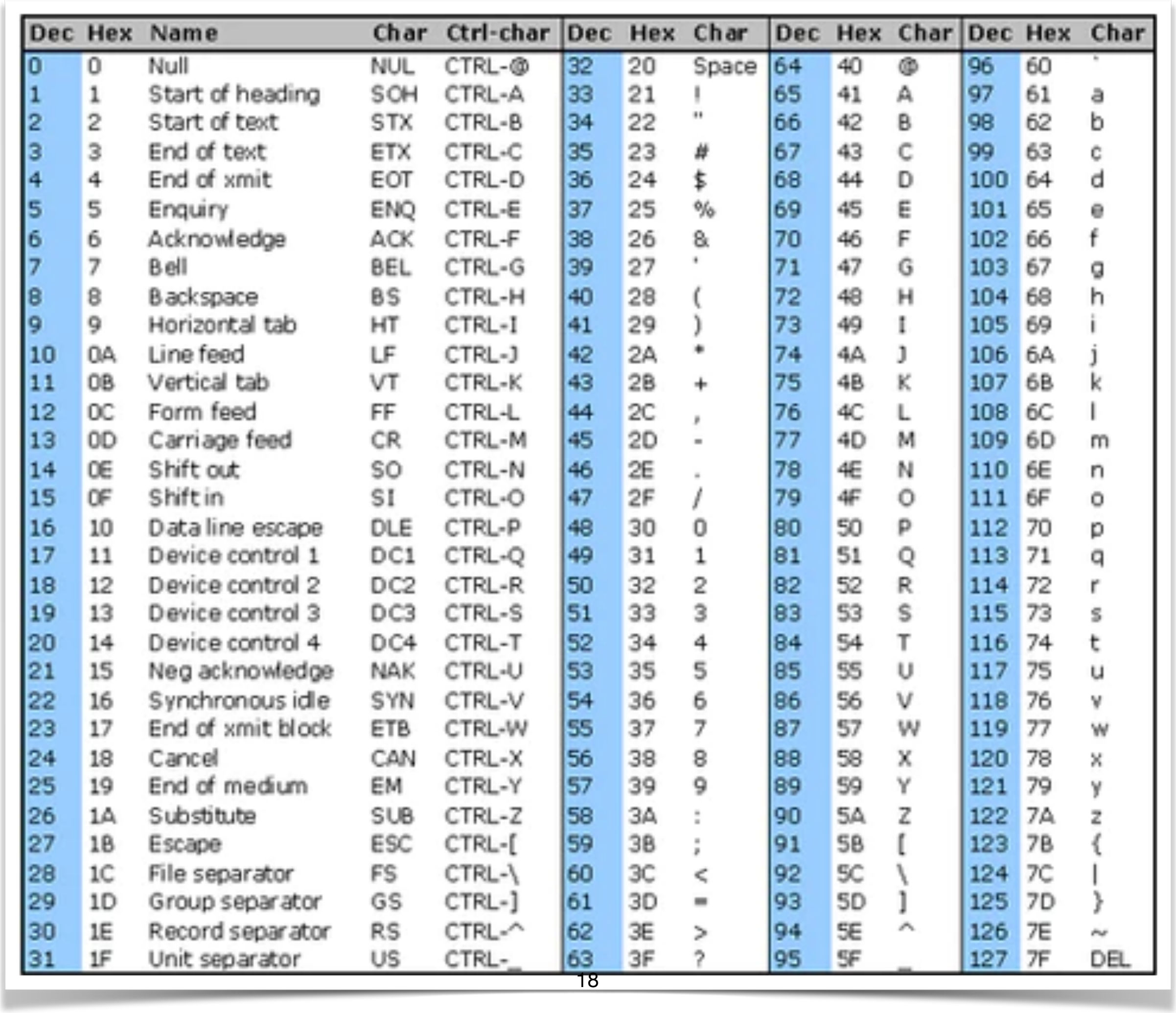
ASCII (1977/1986)

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

The American Standard Code for Information Interchange (ASCII) is a standard that sets how to interpret integers from 0 to 127 (0x00 to 0x7F in hexa) as printable characters and control codes.

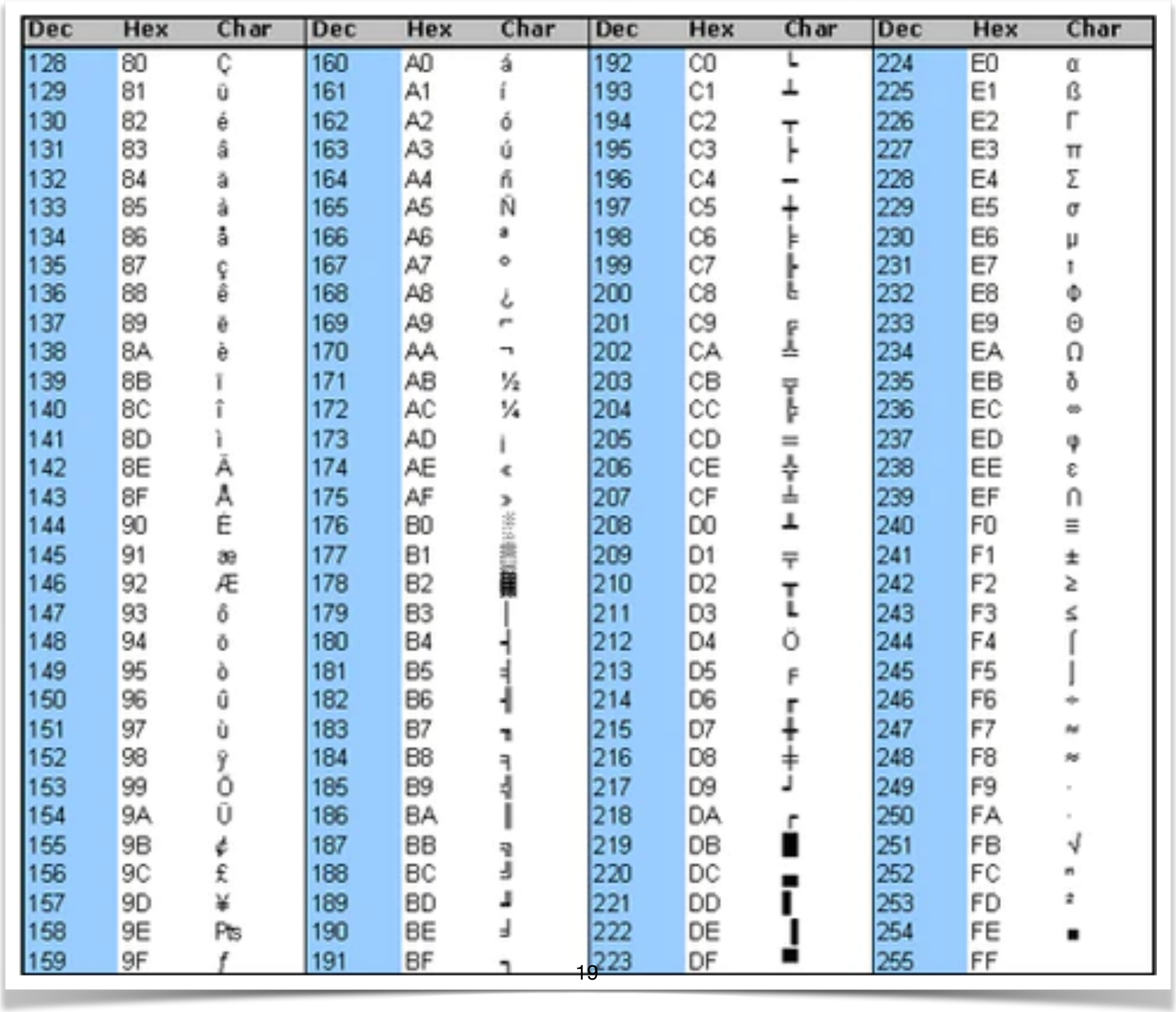
from https://en.wikipedia.org/wiki/ASCII

Standard ASCII (0 to 127)



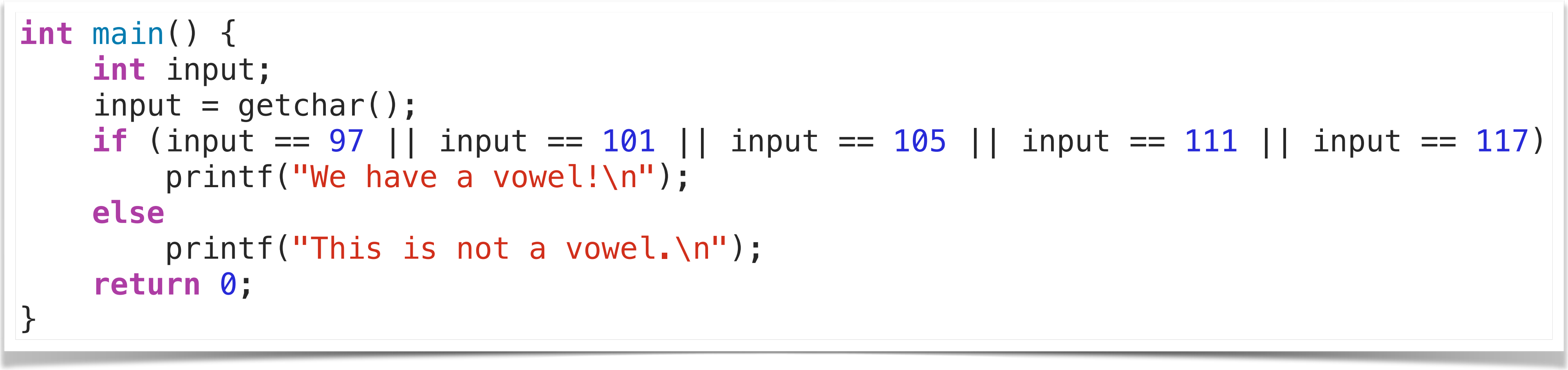
Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

Extended ASCII (128 to 255)



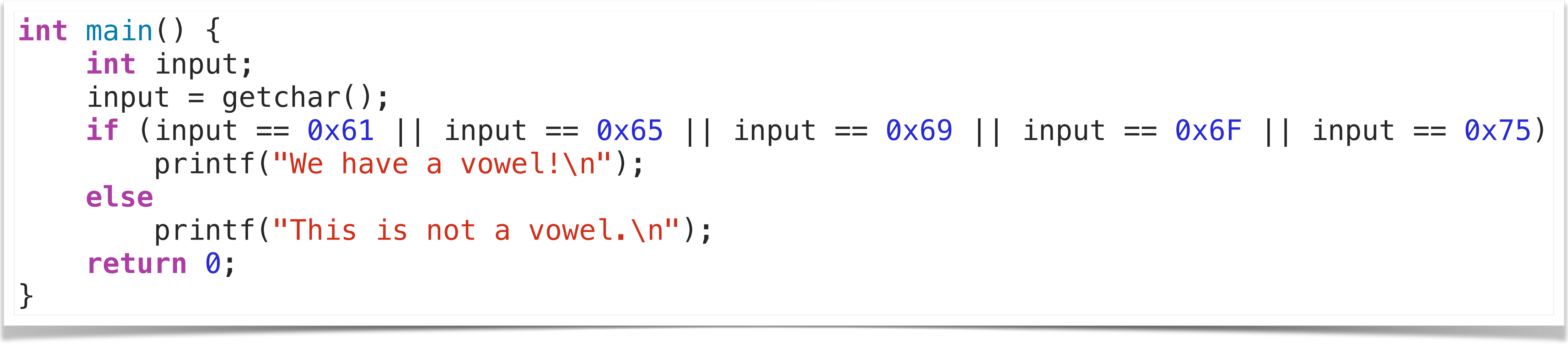
Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	Ö	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	≈
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	·
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

getchar() and integer constants according to ASCII



```c
int main() {
    int input;
    input = getchar();
    if (input == 97 || input == 101 || input == 105 || input == 111 || input == 117)
        printf("We have a vowel!\n");
    else
        printf("This is not a vowel.\n");
    return 0;
}
```

getchar() and integer constants (in hexadecimal)



```
int main() {
    int input;
    input = getchar();
    if (input == 0x61 || input == 0x65 || input == 0x69 || input == 0x6F || input == 0x75)
        printf("We have a vowel!\n");
    else
        printf("This is not a vowel.\n");
    return 0;
}
```

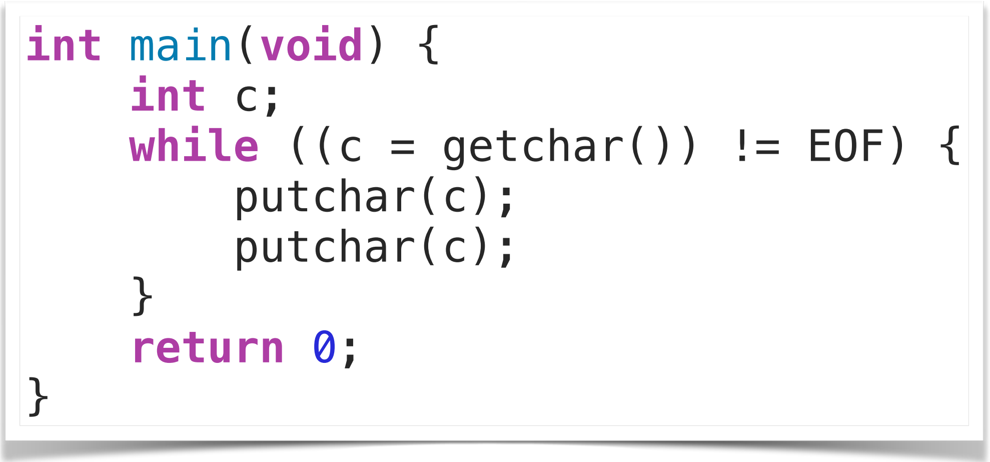
Why is c of type int and not char?

`EOF` is defined in `stdio.h` as `-1`

- The actual value of `EOF` is system-dependent.
- Value `-1` is often used, but it is better to use `EOF` and let the file `stdio.h` define its concrete value.

`getchar()` evaluates to an int value, not char.

- The value used to signal the end of file cannot be a character value (e.g., -1).
- Because `c` is an `int`, it can hold all possible character values **and** the special value `EOF`.

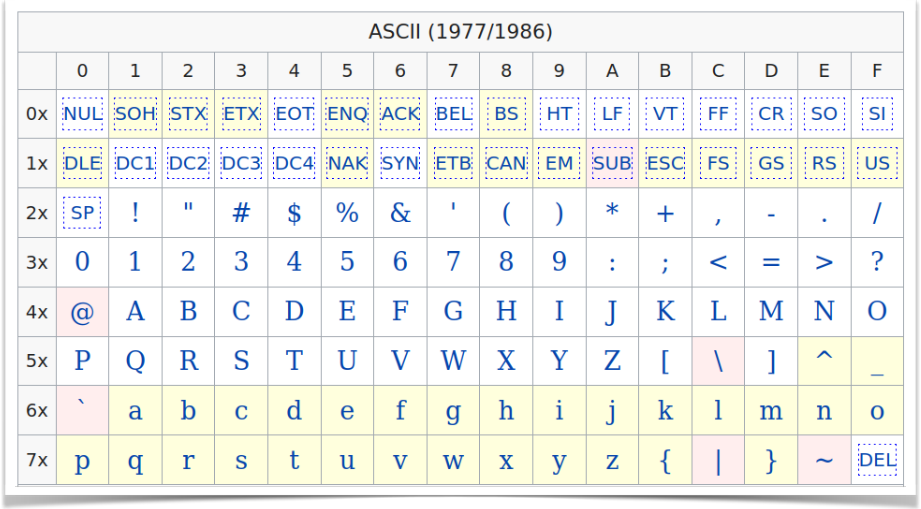


```c
int main(void) {
    int c;
    while ((c = getchar()) != EOF) {
        putchar(c);
        putchar(c);
    }
    return 0;
}
```

ASCII: observations

The most commonly used sequences of characters exist in ASCII as sequences:

- characters '0' to '9' (digits)
- characters 'A' to 'Z' (uppercase latin alphabet)
- characters 'a' to 'z' (lowercase latin alphabet)

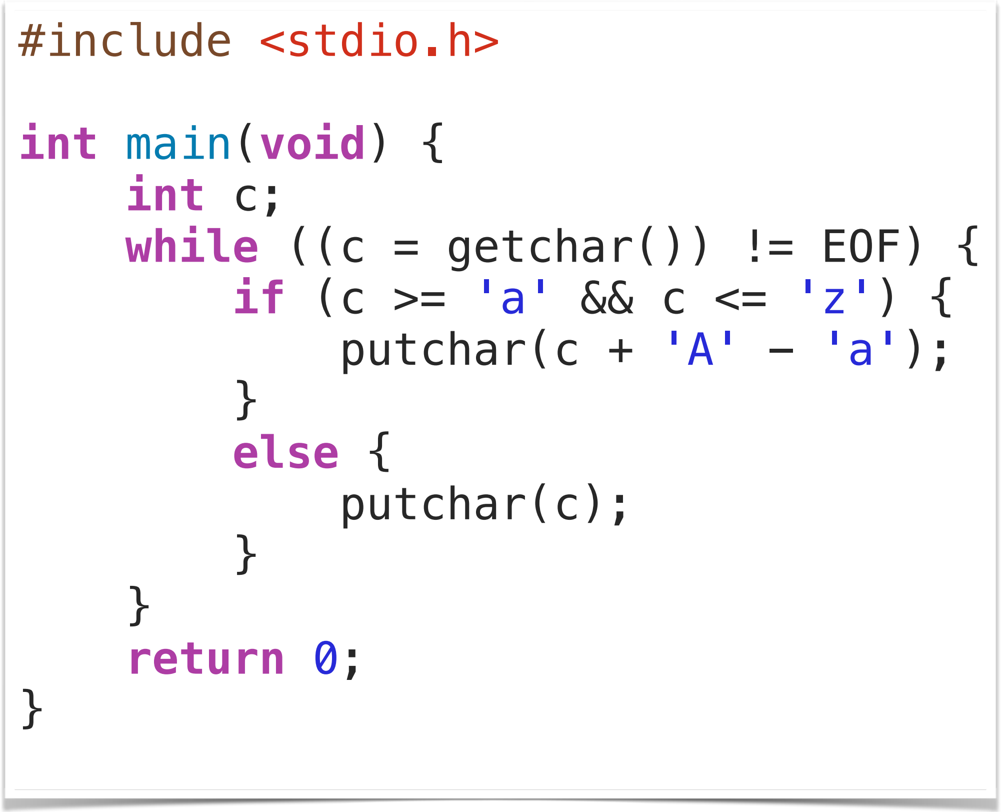


ASCII (1977/1986)

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

Checking whether some character is a digit, an uppercase letter or a lowercase letter, can be done with a condition that checks for an **interval**.

Example: Capitalizing Letters

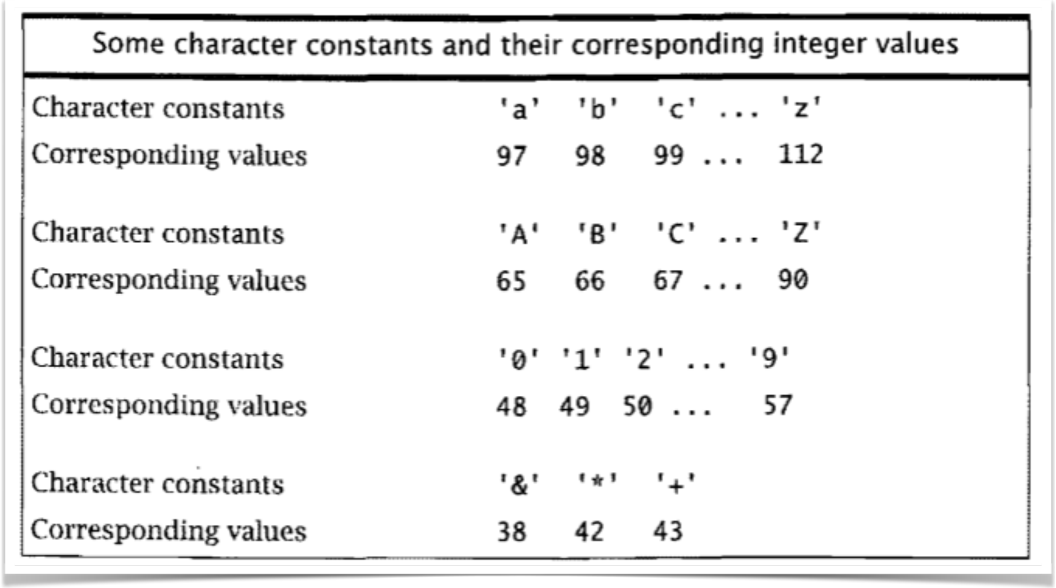


```c
#include <stdio.h>

int main(void) {
    int c;
    while ((c = getchar()) != EOF) {
        if (c >= 'a' && c <= 'z') {
            putchar(c + 'A' - 'a');
        }
        else {
            putchar(c);
        }
    }
    return 0;
}
```

No need to learn ASCII codes

It is not necessary to memorize the integer ASCII codes that correspond to characters.

It’s generally enough to remember that character codes are integers, and groups of related commonly used symbols are organized in sorted intervals.



Some character constants and their corresponding integer values

Character constants	'a'	'b'	'c'	...	'z'
Corresponding values	97	98	99	...	112
Character constants	'A'	'B'	'C'	...	'Z'
Corresponding values	65	66	67	...	90
Character constants	'0'	'1'	'2'	...	'9'
Corresponding values	48	49	50	...	57
Character constants	'&'	'*'	'+'		
Corresponding values	38	42	43		