ASCII

- The American Standard Code for Information Interchange **(ASCII)** character set, has 128 characters designed to encode the Roman alphabet used in English and other Western European languages.
- C was designed to work with ASCII and we will only use the ASCII character set in this course. The **`char`** data type is used to store ASCII characters in C.
- ASCII can represent 128 characters and is encoded in one eight bit byte with a leading 0. Seven bits can encode numbers 0 to 127. Since integers in the range of 0 to 127 can be stored in 1 byte of space, the **sizeof(char)** is 1.
- The characters 0 through 31 represent **control characters** (e.g., line feed, back space), 32-126 are **printable characters**, and 127 is **delete** .

char type

- C supports the char data type for storing a single character.
- **`char`** uses one byte of memory.
- **`char`** constants are enclosed in single quotes.

```
char myGrade = 'A';
char yourGrade = '?';
```

ASCII Character Chart

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

Special Characters

- The backslash character, \, is used to indicate that the char that follows has special meaning. E.g. for unprintable characters and special characters.
- For example:
  - \n is the newline character
  - \t is the tab character
  - \" is the double quote (necessary since double quotes are used to enclose strings
  - \' is the single quote (necessary since single quotes are used to enclose chars
  - \\ is the backslash (necessary since \ now has special meaning
  - \a is beep which is unprintable

Special Char Example Code

- What is the output from these statements?
  - `printf("\t\tMove over\n\nWorld, here I come\n");`

```
>              Move over
>
> World, here I come
```

  - `printf("I've written \"Hello World\"\n\t many times\n\a");`

```
> I've written "Hello World"
>       many times <beep>
```

Character Library

- There are many functions to handle characters. To use these functions in your code,

```
#include <ctype.h>
```

- You will see on the following slide that the function parameters are of type **`int`**, not **`char`**. Why is this ok?

- Note that the return type for some functions is **`int`** since ANSI C does not support the **`bool`** data type. Recall that zero is "false", non-zero is "true".

- A few of the commonly used functions are listed on the next slide. For a full list of **`ctype.h`** functions, type man **`ctype.h`** at the unix prompt.

ctype.h

```
int isdigit (int c);
```

- Determine if c is a decimal digit (‘0’ - ‘9’)

```
int isxdigit(int c);
```

- Determines if c is a hexadecimal digit (‘0’ - ’9’, ‘a’ - f’, or ‘A’ - ‘F’)

```
int isalpha (int c);
```

- Determines if c is an alphabetic character (‘a’ - ‘z’ or ‘A- ‘Z’)

```
int isspace (int c);
```

- Determines if c is a whitespace character (space, tab, etc)

```
int isprint (int c);
```

- Determines if c is a printable character

```
int tolower (int c);
int toupper (int c);
```

- Returns c changed to lower- or upper-case respectively, if possible

Character Input/Output

- Use **`%c`** in **`printf( )`** and **`fprintf( )`** to output a single character.

```
char yourGrade = 'A';
printf( "Your grade is %c\n", yourGrade);
```

- Input char(s) using **`%c`** with **`scanf( )`** or **`fscanf( )`**

```
char grade, scores[3];
```

- **`%c`** inputs the next character, which may be whitespace

```
scanf("%c", &grade);
```

Array of char

- An array of chars may be (partially) initialized. This declaration reserves 20 char (bytes) of memory, but only the first 5 are initialized

```
char name2 [ 20 ] = { 'B', 'o', 'b', 'b', 'y' };
```

- You can let the compiler count the chars for you. This declaration allocates and initializes exactly 5 chars (bytes) of memory

```
char name3 [ ] = { 'B', 'o', 'b', 'b', 'y' };
```

- An array of chars is NOT a string (can you explain why not?)

Strings in C

- In C, a string is an array of characters terminated with the "null" character ('\0', value = 0, see ASCII chart).

- A string may be defined as a char array by initializing the last char to '\0'

```
char name4[ 20 ] = {'B', 'o', 'b', 'b', 'y', '\0' };
```

- Char arrays are permitted a special initialization using a string constant. Note that the size of the array must account for the '\0' character.

```
char name5[6] = "Bobby";  // this is NOT assignment
```

- Or let the compiler count the chars and allocate the appropriate array size

```
char name6[ ] = "Bobby";
```

- All string constants are enclosed in double quotes and include the terminating '\0 character

String Output

- Use **`%s`** in **`printf()`** or **`fprintf()`** to print a string. All chars will be output until the '**`\0`**' character is seen.

```
char name[ ] = "Bobby Smith";
printf( "My name is %s\n", name);
```

- As with all conversion specifications, a minimum field width and justification may be specified

```
char book1[ ] = "Flatland";
char book2[ ] = "Brave New World";

printf ("My favorite books are %12s and %12s\n",
book1, book2);
printf ("My favorite books are %-12s and %-12s\n",
book1, book2);
```

Dangerous String Input

- The most common and most dangerous method to get string input from the user is to use **`%s`** with **`scanf( )`** or **`fscanf( )`**

- This method interprets the next set of consecutive non-whitespace characters as a string, stores it in the specified char array, and appends a terminating '**`\0`**' character.
  - `char name[22];`
  - `printf(" Enter your name: ");`
  - `scanf( "%s", name);`

- Why is this dangerous?
- See scan fString.c and fscanfStrings.c

Safer String Input

- A safer method of string input is to use **`%ns`** with **`scanf( )`** or **`fscanf( )`** where **`n`** is an integer

- This will interpret the next set of consecutive non-whitespace characters up to a maximum of n characters as a string, store it in the specified char array, and append a terminating '\0' character.

```
char name[ 22 ];
printf( "Enter your name: ");
scanf("%21s", name);            // note 21, not 22
```

C String Library

- C provides a library of string functions.
- To use the string functions, include `<string.h>`.
- Some of the more common functions are listed here and on the next slide.
- To see all the string functions, type `man string.h` at the unix prompt.
- Commonly used string functions: These functions look for the '`\0`' character to determine the end and size of the string.

```
strlen( const char string[ ]  )
```
  - Returns the number of characters in the string, not including the "null" character

```
strcpy( char s1[ ], const char s2[ ] )
```
  - Copies s2 on top of s1.
  - The order of the parameters mimics the assignment operator

```
strcmp ( const char s1[ ] , const char s2[ ] )
```
  - Returns < 0, 0, > 0 if s1 < s2, s1 == s2 or s1 > s2 lexigraphically

```
strcat( char s1[ ] , const char s2[ ])
```
  - Appends (concatenates) s2 to s1

C String Library (cont)

- Some function in the C String library have an additional size parameter:
  - **`strncpy( char s1[ ], const char s2[ ], int n );`**
    - Copies at most n characters of s2 on top of s1.
    - The order of the parameters mimics the assignment operator
  - **`strncmp ( const char s1[ ] , const char s2[ ], int n );`**
    - Compares up to n characters of s1 with s2
    - Returns < 0, 0, > 0 if s1 < s2, s1 == s2 or s1 > s2 lexigraphically
  - **`strncat( char s1[ ], const char s2[ ] , int n);`**
    - Appends at most n characters of s2 to s1

String Code – example 1

```
char first[10] = "bobby";
char last[15] = "smith";
char name[30];
char you[ ] = "bobo";

strcpy( name, first );
strcat( name, last );
printf( "%d, %s\n", strlen(name), name );

strncpy( name, last, 2 );
printf( "%d, %s\n", strlen(name), name );

int result = strcmp( you, first );
result = strncmp( you, first, 3 );
strcat( first, last );
```

Simple Encryption– example 2

```
char c, msg[] = "this is a secret message";
int i = 0;
char code[26] =      /* Initialize our encryption code */
 {'t','f','h','x','q','j','e','m','u','p','i','d','c',
'k','v','b','a','o','l','r','z','w','g','n','s','y'} ;

/* Print the original phrase */
printf ("Original phrase: %s\n", msg);

/* Encrypt */
while( msg[i] != '\0' ){
  if( isalpha( msg[ i ] ) ) {
    c = tolower( msg[ i ] ) ;
    msg[ i ] = code[ c - 'a' ] ;
  }
  ++i;
}

printf("Encrypted: %s\n", msg ) ;
```

Arrays of Strings

- Since strings are arrays themselves, using an array of strings can be a little tricky

- An initialized array of string constants:

```
char months[ ][ 10 ] = {
  "Jan", "Feb", "March", "April", "May", "June",
  "July", "Aug", "Sept", "Oct", "Nov", "Dec"
};
int m;
for ( m = 0; m < 12; m++ )
  printf( "%s\n", months[ m ] );
```

- An array of string 12 variables, each 20 chars long:

```
char names[ 12 ] [ 21 ];
int n;
for( n = 0; n < 12; ++n )
{
  printf( "Please enter your name: " );
  scanf( "%20s", names[ n ] );
}
```

gets( ) to read a line

- The **`gets( )`** function is used to read a **line of input** (including the whitespace) from **`stdin`** until the **`\n`** character is encountered. The \n character is replaced with the terminating **`\0`** character.

```
#include <stdio.h>
char myString[ 101 ];
gets( myString );
```

- Why is this dangerous?
- See gets.c

fgets( ) to read a line

- The **`fgets( )`** function is used to read a **line of input** (including the whitespace) from the specified **FILE** until the **`\n`** character is encountered or until the specified number of chars is read.

- See fgets.c

fgets( ) example

```
#include <stdio.h>
#include <stdlib.h>          /* exit */

int main ( )
{
  double x ;
  FILE *ifp ;
  char myLine[42 ];          /*  for terminating \0 */

  ifp = fopen("test_data.dat", "r");
  if (ifp == NULL) {
    printf ("Error opening test_data.dat\n");
    exit (-1);
  }

  fgets(myLine, 42, ifp ); /* read up to 41 chars*/
  fclose(ifp);             /* close the file when
                              finished */

  /* check to see what you read */

  printf("myLine = %s\n", myLine);
  return 0;
}
```

Detecting EOF with fgets( )

- **`fgets( )`** returns the memory address in which the line was stored (the char array provided). However, when **`fgets( )`** encounters **`EOF`**, the special value **`NULL`** is returned.

```
FILE *inFile;
inFile = fopen( "myfile", "r" );

/* check that the file was opened */

char string[120];
while ( fgets(string, 120, inFile ) != NULL )
  printf( "%s\n", string );

fclose( inFile );
```

Using fgets( ) instead of gets( )

- Since **`fgets( )`** can read any file, it can be used in place of **`gets( )`** to get input from the user:

```
#include <stdio.h>
char myString[ 101 ];
```

  - Instead of:

```
gets( myString );
```

  - Use

```
fgets( mystring, 100, stdin );
```

"Big Enough"

- The "owner" of a string is responsible for allocating array space which is "big enough" to store the string (including the null character).
  - `scanf( )`, `fscanf( )`, and `gets( )` assume the char array argument is "big enough"
- String functions that do not provide a parameter for the length rely on the '`\0`' character to determine the end of the string.
- Most string library functions do not check the size of the string memory. E.g. `strcpy`
- See strings.c

What can happen?

```
int main( )
{
  char first[10] = "bobby";
  char last[15] = "smith";

  printf("first contains %d chars: %s\n", strlen(first), first);
  printf("last contains %d chars: %s\n", strlen(last), last);

  strcpy(first, "1234567890123");   /* too big */

  printf("first contains %d chars: %s\n", strlen(first), first);
  printf("last contains %d chars: %s\n", strlen(last), last);

  return 0;
}

/* output */
first contains 5 chars: bobby
last contains 5 chars: smith
first contains 13 chars: 1234567890123
last contains 5 chars: smith
Segmentation fault
```

The Lesson

- Avoid `scanf( "%s", buffer);`
- Use `scanf("%100s", buffer);` instead

- Avoid `gets( );`
- Use `fgets(..., ..., stdin);` instead

sprintf( )

- Sometimes it's necessary to format a string in an array of chars. Something akin to **`toString( )`** in Java.
- **`sprintf( )`** works just like **`printf( )`** or **`fprintf( )`**, but puts its "output" into the specified character array.
- As always, the character array must be big enough.
- See sprintf.c

```
char message[ 100 ];
int myAge = 4;
sprintf( message, "I am %d years old\n", age);
printf( "%s\n", message);
```

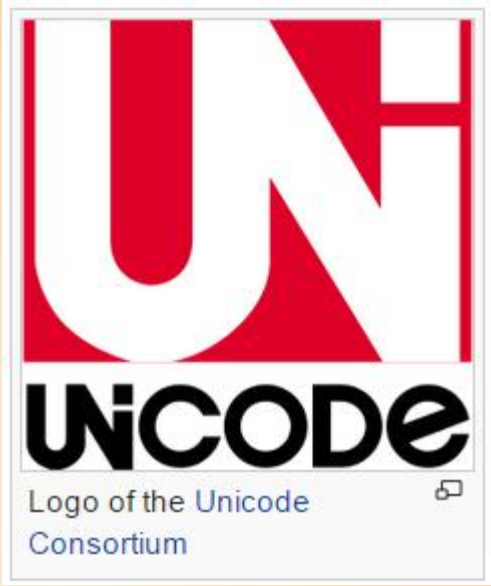
Appendix: extended ASCII Character Chart

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
80	€ 20AC		‚ 201A		„ 201E	… 2026	† 2020	‡ 2021		‰ 2030	Š 0160	‹ 2039	Ś 015A	Ť 0164	Ž 017D	Ź 0179
90		‘ 2018	’ 2019	“ 201C	” 201D	• 2022	– 2013	— 2014		™ 2122	š 0161	› 203A	ś 015B	ť 0165	ž 017E	ź 017A
A0	NBSP 00A0	ˇ 02C7	˘ 02D8	Ł 0141	¤ 00A4	Ą 0104	¦ 00A6	§ 00A7	¨ 00A8	© 00A9	Ş 015E	« 00AB	¬ 00AC	- 00AD	® 00AE	Ż 017B
B0	° 00B0	± 00B1	˛ 02DB	ł 0142	´ 00B4	µ 00B5	¶ 00B6	· 00B7	¸ 00B8	ą 0105	ş 015F	» 00BB	Ľ 013D	˝ 02DD	ľ 013E	ż 017C
C0	Ŕ 0154	Á 00C1	Â 00C2	Ă 0102	Ä 00C4	Ĺ 0139	Ć 0106	Ç 00C7	Č 010C	É 00C9	Ę 0118	Ë 00CB	Ě 011A	Í 00CD	Î 00CE	Ď 010E
D0	Đ 0110	Ń 0143	Ň 0147	Ó 00D3	Ô 00D4	Ő 0150	Ö 00D6	× 00D7	Ř 0158	Ů 016E	Ú 00DA	Ű 0170	Ü 00DC	Ý 00DD	Ţ 0162	ß 00DF
E0	ŕ 0155	á 00E1	â 00E2	ă 0103	ä 00E4	ĺ 013A	ć 0107	ç 00E7	č 010D	é 00E9	ę 0119	ë 00EB	ě 011B	í 00ED	î 00EE	ď 010F
F0	đ 0111	ń 0144	ň 0148	ó 00F3	ô 00F4	ő 0151	ö 00F6	÷ 00F7	ř 0159	ů 016F	ú 00FA	ű 0171	ü 00FC	ý 00FD	ţ 0163	˙ 02D9

Appendix: Unicode

From Wikipedia, the free encyclopedia

Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. Developed in conjunction with the Universal Character Set standard and published as The Unicode Standard, the latest version of Unicode contains a repertoire of more than 120,000 characters covering 129 modern and historic scripts, as well as multiple symbol sets. The standard consists of a set of code charts for visual reference, an encoding method and set of standard character encodings, a set of reference data files, and a number of related items, such as character properties, rules for normalization, decomposition, collation, rendering, and bidirectional order (for the correct display of text containing both right-to-left scripts, such as Arabic and Hebrew, and left-to-right scripts). As of June 2015, the most recent version is Unicode 8.0. The standard is maintained by the Unicode Consortium.



Logo of the Unicode Consortium

The origins of Unicode date to 1987, when Joe Becker from Xerox and Lee Collins and Mark Davis from Apple started investigating the practicalities of creating a universal character set. In it's original form, entitled Unicode 88, Becker outlined a 16-bit character model. In 1996, a surrogate character mechanism was implemented in Unicode 2.0, so that Unicode was no longer restricted to 16 bits. This increased the Unicode codespace to over a million code points, which allowed for the encoding of many historic scripts (e.g., Egyptian Hieroglyphs) and thousands of rarely used or obsolete characters that had not been anticipated as needing encoding.