Lecture 7: C Programming II

ME/AE 6705

Introduction to Mechatronics

Dr. Jonathan Rogers

Georgia Tech

Lesson Objectives

- Explain the origin and use of the ASCII standard
- Understand specific features of C programming language
  - Strings, Arrays, and Pointers
- Be able to articulate the memory structure and use of a string, array, and pointer
- Be able to use arrays to hold a vector of data
- Demonstrate concepts through multiple examples

ASCII Standard

- **American Standard Code for Information Interchange**
- Used to represent text on computers
- Uses a 7 bit integer assigned to each character
- 2^7 = 128 characters (52 english letters, upper and lower case, 10 numbers, punctuation, control codes
- Published in 1963, latest version 1986
- Originally designed for “teleprinters”
- Other text standards (UTF-8, UTF-16, etc)

Georgia Tech

ASCII Standard

American Standard Code for Information Interchange (ASCII)

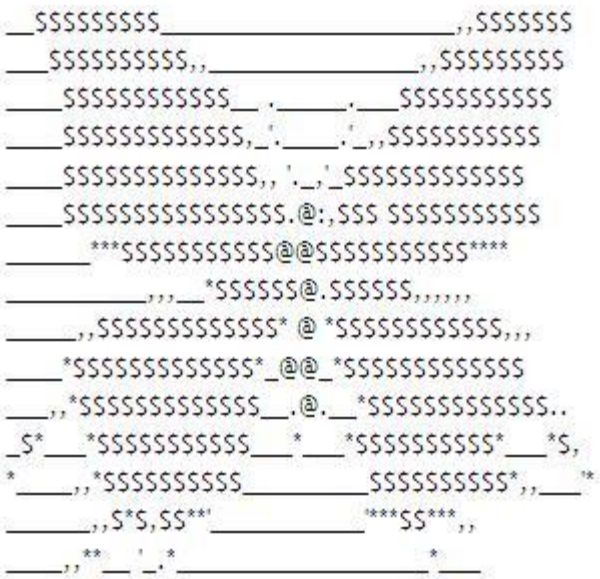
$B_4B_3B_2B_1$	$B_7B_6B_5$ = 000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SOH	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

'#' = 0100011 (35 dec)
'$' = 0100100 (36 dec)

'#' + '$' = ?

Georgia Tech

ASCII Art



Georgia Tech

Important ASCII “Chars”

- \0 NULL Null character
- \n LF Line feed
- \r CR Carriage return
- \t TAB Tab character

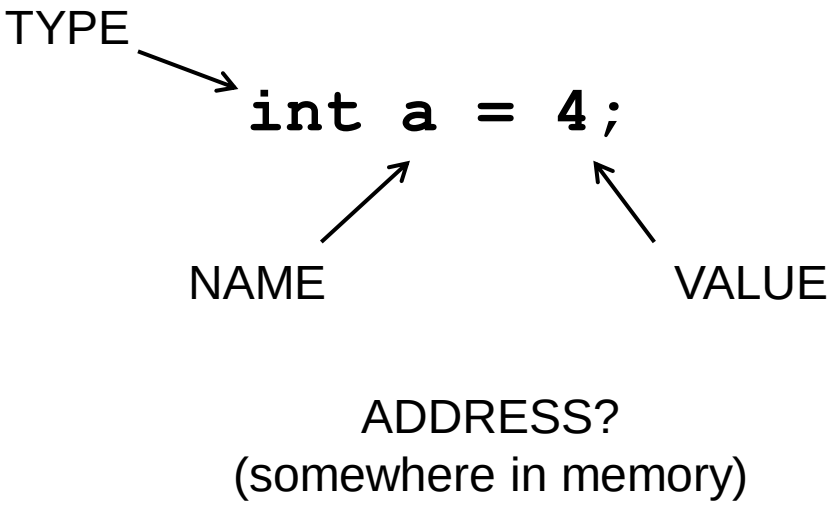
```c
#include <stdio.h>

int main()
{
        char   str1[]  = "Hello World!";          // compiler adds '\0'
        char   str2[4] = {'a', 'b', 'c', '\0'};
        char   str3[2] = {'a' + 'b' + 'c' ,'\0'};  // ???
        printf("%s\t"   ,str1);   // "Hello World!(tab)"
        printf("%s\t"   ,str2);   // "abc(tab)"
        printf("%s\r\n" ,str3);   // "&(carriage return + newline)"
        return 0;
}
```

Pointers

- Essential and powerful part of the C language
- Have a certain stigma with new programmers
- Every variable has a NAME, ADDRESS, and VALUE

NAME	a
ADDRESS	0x28feb8
VALUE	4



(Integer Address: 2686648)

Pointers

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE		…	…	…	…	

```c
#include <stdio.h>

int main()
{
        int a = 4; // Integer with value of 4
        // variable NAME is "a" has a VALUE of 4, and ADDRESS is 28feb8
        int *a_addr = 0; // Pointer to an "int"
        // variable NAME is "a_addr" has a VALUE of NULL, and ADDRESS is 28febc
        a_addr = &a; // Storing the address of "a" in "a_addr" (28feb8)
        printf("Address of \"a\":      %x\n" ,&a);       // 28feb8
        printf("Value of \"a_addr\":   %x\n" ,a_addr);   // 28feb8
        printf("Address of \"a_addr\": %x\n" ,&a_addr);  // 28febc
        printf("Value   of \"a\":      %i\n" ,*a_addr);  // 4
        return 0 ;
}
```

Georgia Tech

Pointers

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE		…	…	…	…	

```
#include <stdio.h>

int main()
{
        int a = 4; // Integer with value of 4
        // variable NAME is "a" has a VALUE of 4, and ADDRESS is 28feb8
        int *a_addr = 0; // Pointer to an "int"
        // variable NAME is "a_addr" has a VALUE of NULL, and ADDRESS is 28febc
        a_addr = &a; // Storing the address of "a" in "a_addr" (28feb8)
        printf("Address of \"a\":      %x\n" ,&a);       // 28feb8
        printf("Value of \"a_addr\":   %x\n" ,a_addr);   // 28feb8
        printf("Address of \"a_addr\": %x\n" ,&a_addr);  // 28febc
        printf("Value   of \"a\":      %i\n" ,*a_addr);  // 4
        return 0 ;
}
```

Georgia Tech

Pointers

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE	4	…	…	…	…	0

```
#include <stdio.h>

int main()
{
        int a = 4; // Integer with value of 4
        // variable NAME is "a" has a VALUE of 4, and ADDRESS is 28feb8
        int *a_addr = 0; // Pointer to an "int"
        // variable NAME is "a_addr" has a VALUE of NULL, and ADDRESS is 28febc
        a_addr = &a; // Storing the address of "a" in "a_addr" (28feb8)
        printf("Address of \"a\":      %x\n" ,&a);       // 28feb8
        printf("Value of \"a_addr\":   %x\n" ,a_addr);   // 28feb8
        printf("Address of \"a_addr\": %x\n" ,&a_addr);  // 28febc
        printf("Value   of \"a\":      %i\n" ,*a_addr);  // 4
        return 0 ;
}
```

Georgia Tech

Pointers

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE	4	…	…	…	…	28feb8

```c
#include <stdio.h>

int main()
{
        int a = 4; // Integer with value of 4
        // variable NAME is "a" has a VALUE of 4, and ADDRESS is 28feb8
        int *a_addr = 0; // Pointer to an "int"
        // variable NAME is "a_addr" has a VALUE of NULL, and ADDRESS is 28febc
        a_addr = &a; // Storing the address of "a" in "a_addr" (28feb8)
        printf("Address of \"a\":      %x\n" ,&a);        // 28feb8
        printf("Value of \"a_addr\":   %x\n" ,a_addr);    // 28feb8
        printf("Address of \"a_addr\": %x\n" ,&a_addr);   // 28febc
        printf("Value   of \"a\":      %i\n" ,*a_addr);   // 4
        return 0 ;
}
```

Pointers

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE	4	…	…	…	…	28feb8

```c
#include <stdio.h>

int main()
{
        int a = 4; // Integer with value of 4
        // variable NAME is "a" has a VALUE of 4, and ADDRESS is 28feb8
        int *a_addr = 0; // Pointer to an "int"
        // variable NAME is "a_addr" has a VALUE of NULL, and ADDRESS is 28febc
        a_addr = &a; // Storing the address of "a" in "a_addr" (28feb8)
        printf("Address of \"a\":      %x\n" ,&a);       // 28feb8
        printf("Value of \"a_addr\":   %x\n" ,a_addr);   // 28feb8
        printf("Address of \"a_addr\": %x\n" ,&a_addr);  // 28febc
        printf("Value   of \"a\":      %i\n" ,*a_addr);  // 4
        return 0 ;
}
```

Georgia Tech

Pointers

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE	4	…	…	…	…	28feb8

```c
#include <stdio.h>

int main()
{
        int a = 4; // Integer with value of 4
        // variable NAME is "a" has a VALUE of 4, and ADDRESS is 28feb8
        int *a_addr = 0; // Pointer to an "int"
        // variable NAME is "a_addr" has a VALUE of NULL, and ADDRESS is 28febc
        a_addr = &a; // Storing the address of "a" in "a_addr" (28feb8)
        printf("Address of \"a\":      %x\n" ,&a);       // 28feb8
        printf("Value of \"a_addr\":   %x\n" ,a_addr);   // 28feb8
        printf("Address of \"a_addr\": %x\n" ,&a_addr);  // 28febc
        printf("Value   of \"a\":      %i\n" ,*a_addr);  // 4
        return 0 ;
}
```

Pointers

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE	4	…	…	…	…	28feb8

```c
#include <stdio.h>

int main()
{
        int a = 4; // Integer with value of 4
        // variable NAME is "a" has a VALUE of 4, and ADDRESS is 28feb8
        int *a_addr = 0; // Pointer to an "int"
        // variable NAME is "a_addr" has a VALUE of NULL, and ADDRESS is 28febc
        a_addr = &a; // Storing the address of "a" in "a_addr" (28feb8)
        printf("Address of \"a\":      %x\n" ,&a);        // 28feb8
        printf("Value of \"a_addr\":   %x\n" ,a_addr);    // 28feb8
        printf("Address of \"a_addr\": %x\n" ,&a_addr);   // 28febc
        printf("Value   of \"a\":      %i\n" ,*a_addr);   // 4
        return 0 ;
}
```

Pointers

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE	4	…	…	…	…	28feb8

```c
#include <stdio.h>

int main()
{
        int a = 4; // Integer with value of 4
        // variable NAME is "a" has a VALUE of 4, and ADDRESS is 28feb8
        int *a_addr = 0; // Pointer to an "int"
        // variable NAME is "a_addr" has a VALUE of NULL, and ADDRESS is 28febc
        a_addr = &a; // Storing the address of "a" in "a_addr" (28feb8)
        printf("Address of \"a\":      %x\n" ,&a);        // 28feb8
        printf("Value of \"a_addr\":   %x\n" ,a_addr);    // 28feb8
        printf("Address of \"a_addr\": %x\n" ,&a_addr);   // 28febc
        printf("Value   of \"a\":      %i\n" ,*a_addr);   // 4
        return 0 ;
}
```

Georgia Tech

Asterisk

*

- Used to tell the compiler, “this is a pointer to a “int”, not an int”

```
int *a_addr; // Pointer to an "int"  (not a variable)
```

- Also used to tell the compiler, “get the value of the variable stored at the address stored in this pointer”

```
int b = *a_addr; // Get the value of the variable stored at "a_addr" (4)
// OR
*a_addr = 10;    // Change the value of the variable stored at "a_addr"
```

Georgia Tech

Ampersand

&

- Used to tell the compiler, “get the address of this variable”

```
a_addr = &a; // (28feb8) Storing the address of "a" in "a_addr"
```


Ampersand

&

- Also used in a “pass by reference” case

```
void func_add(int b, int &c) // compiler makes a copy of “b” in memory
and makes copy of the address of “c”, not a full copy of the data in “c”
{
        c = b + c;
}

int main()
{
        int a = 4;
        func_add(10,a);   // a is now 14
        return 0;
}
```

- Often used to “return” multiple values

Georgia Tech

Pointer Vocabulary

- **Referencing a pointer** – using the address stored in the pointer
- **Dereferencing a pointer** - getting the value of the variable at the stored address (*)
- A pointer "**points**" to a variable when the pointer stores the address of that variable.

Significance of Pointers

- **List sort:** It takes less computation to rearrange a list of variable addresses, than to rearrange the data in the variables directly.
- **Arrays:** Pointers enable low level, high speed sequential and random memory access
- **Everybody uses them (often behind the scenes)**
  - In Java, non-primitive function arguments are passed-by-reference (kind of) and all variables are really pointers although programmers can't modify the pointers directly.
  - In Matlab, when function arguments are "handles"

Georgia Tech

Drawbacks of Pointers

- Confusing, hard to read

```
int data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9}; // Define an array of "ints"
printf("%i\t%i\t%i\r\n", data[4], *(data + 4), *(&data[4]));
// prints 5(tab)5(tab)5(tab)
```

- "With great power, comes great responsibility"

```
char buffer[8];
*(buffer + 8) = 'a'; // "Memory stomp" or "buffer overrun"

// OR

int array[10], i;          // "i" is located in memory just after array
for (i = 0; i < 11 ; i++) // causes an infinite loop
        array[i] = 0;     // array[10] is actually modifying "i"
```

- More difficult to debug and maintain

Georgia Tech

Pointers - Conclusions

- Avoid pointer arithmetic
- Manipulate pointers only when necessary
- Add copious comments to avoid confusion

- Pointers save computational resources (important for microcontrollers)
  - The program can pass the address of the data to many functions and they all can work on the same copy of the data (not simultaneously) rather than making a separate copy for each function call.

Georgia Tech

Fun Fact

Ever wonder why a 32bit computer can only have a maximum of 4GB of memory?

Because the size of a pointer in memory on a 32 bit machine is only 32 bits! So there are only 2^{32} = 4294967296 possible unique addresses!

On a 64 bit machine there are a maximum of 18 EB (18 trillion bytes, or exabytes, 9 orders of magnitude more)

Georgia Tech

Arrays - Initialization

```
int data[6];        // Memory is allocated but is uninitialized JUNK!

int data[6] = { }; // Memory is allocated and initialized to zero

int data[] = { 7, 42, 6, 12, 99, -155}; // Allocated and initialized
(size is implicit)

char str[7] = "abc";  // Initialized to "abc\0\0\0\0" (padded with NULL)

int data[][2]  = {{22, 8}, {1, -16}, {47, 109}}; // Define a 2D array
int data[4][2] = {{22, 8}, {1, -16}, {47, 109}}; // Define a 2D array
(zero padded)

// RULES
// 1. Size MUST be specified when declared, size can't change afterward.
// 2. In general, just define an array that will always be big enough to
hold your data (size*2 etc).  Easy, but considered bad practice.
// 3. Dynamically sized arrays are available if memory is tight. See:
http://www.codingunit.com/c-tutorial-the-functions-malloc-and-free
```

Georgia Tech

Arrays – Accessing Values

```
// 1D array
int data[] = { 7, 42, 6, 12, 99, -155};

printf("%i", data[1]); // 42

data[1] = 10;

printf("%i", data[1]); // 10, { 7, 10, 6, 12, 99, -155};

// 2D array
int data2D[][2] = {{22, 8}, {1, -16}, {47, 109}};  // Define a 2D array
of "ints" (3x2 array)

printf("%i", data2D[1][1]); // -16

data2D[2][1] = 10; // 109 changed to a 10
```

- Arrays are always "passed by reference" into a function

Arrays

- Memory architecture

```
int data[] = { 7, 42, 6, 12, 99, -155}; // Define an array of "ints"
```

NAME	data[0]	data[1]	data[2]	data[3]	data[4]	data[5]
ADDRESS	28fea0	28fea4	28fea8	28feac	28feb0	28feb4
VALUE	7	42	6	12	99	-155

`sizeof(int) // 4 bytes`

- `data` is a pointer to `data[0]`
- `*data` is equivalent to `data[0]`
- `*(data+1)` is equivalent to `data[1]`

Georgia Tech

Arrays

```
int    a_i[] = {7, 42, 6, 12, 99, -155}; // Define an array of ints
double a_d[] = {7.0, 42.0, 6.0, 12.0, 99.0, -155.0};//Array of doubles
char str[]   = { 'H', 'e', 'l', 'l', 'o', '\0'}; // Define an array of chars
```

NAME	a_i[0]	a_i[1]	a_i[2]	a_i[3]	a_i[4]	a_i[5]
ADDRESS	28fea0	28fea4	28fea8	28feac	28feb0	28feb4
VALUE	7	42	6	12	99	-155
Bytes	1 2 3 4 …					

NAME	a_d[0]	a_d[1]	a_d[2]	…
ADDRESS	28fec0	28fec8	28fed0	…
VALUE	7.0	42.0	6.0	…
Bytes	1 2 3 4 5 6 7 8 …			

NAME	str[0]	str[1]	str[2]	str[3]	str[4]	str[5]
ADDRESS	…	…	…	…	…	…
VALUE	H	e	l	l	o	\0
Bytes	1	2	3	4	5	6

“int”, “float”, “double” sizes are platform dependent!
MPS432: int = 4 bytes, float = 4 bytes, double = N/A
Atmel328: int = 2 bytes, float = 4 bytes, double = 4

Quiz

- How are pointers and arrays related?
- Pointer arithmetic

```
int data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9}; // Define an array of “ints”


*data

*(data + 4)

*(&data[4])

*(data + (*(data[5] + data + 2)-4)
```

Georgia Tech

Multidimensional Arrays

```
// 2D array initialization
int data2D[10][2]; // Allocate memory for a 10x2 array (uninitialized)
int data2D[10][2] = {}; // Allocate memory, values initialized to zero
int data2D[][2]  = {{22, 8}, {1, -16}, {47, 109}}; // Allocate a 2D
array (3x2)
int data2D[4][2] = {{22, 8}, {1, -16}, {47, 109}}; // Allocate a 2D
array (4,2 zero padded)

data2D[1][0] = 33; // Changing the value 1 to 33
```

- The syntax suggests a 2 dimensional matrix format, which is very useful, but the memory is actually stored linearly.
- Modifying values uses the same method as for 1D arrays.

Georgia Tech

Multidimensional Arrays

```
int data2D[][2] = {{22, 8}, {1, -16}, {47, 109}};  // Define a 2D array
```

NAME	**data2D																					
NAME	*data2D[0]							*data2D[1]							*data2D[2]							
NAME	data2D[0][0]				data2D[0][1]				data2D[1][0]				data2D[1][1]				data2D[2][0]				data2D[2][1]			
ADDRESS	28fea0				28fea4				28fea8				28feac				28feb0				28feb4			
VALUE	22				8				1				-16				47				109			
Byte	1	2	3	4	…																			

```
data2D[0][1]; // value = 8
data2D[2][0]; // value = 47

data2D[1];    // value = 28fea8  (address of data2D[1][0])
*data2D[1];   // value = 1
```

Georgia Tech

Arrays as Function Arguments

- Functions treat array arguments like pointers

```
void func(int array[], int n);

// Is equivalent to

void func(int *array, int n)
{
        array[n] = 2; // You can use the array normally.  However, it
                      will modify the ORIGINAL memory of the
                      variable passed into the function.  It will
                      NOT make a copy of the array.
}
```

Copy Array

```
int array1[100];
int array2[100];

array2 = array1;   // This won't work

// This is what you have to do
void array_copy(int array1[], int array2[], int n)
{
        for(int i=0; i<n; i++)
        {
                array2[i] = array1[i]; // Assigning the value
        }
}
```


Other Function Arguments

- Passing by reference is slightly different than passing a pointer

```
// Although it is passing in a pointer, the variable "val" can be
treated like a variable directly.  It kind of hides the fact.
int func1(int &val)
{
        val = 2;
}
// This function works the same way but "val" must be treated like a
pointer
int func2(int *val)
{
        *val = 2;
}
int main()
{
        int a = 0;
        func1(a);
        func2(&a);
}
```

Arrays – Conclusions

- Arrays store data in memory sequentially
- Array indices (`[0][1][5]`) can be used like matrix indices
- Array indexing is one of the most common bugs. Make sure not to exceed the size of the array.

Power of arrays

- Matrix multiplication
- Gaussian elimination
- LU decomposition
- Eigenvalues
- Regression
- Control systems
- Image processing
- Data acquisition (12-bit ADC on MSPM0)

Georgia Tech

Strings

- Special type of char array
- Last byte should contain the NULL character `'\0'` to demarcate the end of the string
- The programmer is responsible to allocate space for the `'\0'`

Strings

```
// Initialization examples
char str1[3];              // Memory allocated but not initialized
char str2[3]  = "";        // Memory allocated and filled with '\0'
char str3[4]  = "abc";     // Memory allocated and initialized, compiler adds '\0'
char str4[4]  = {'a','b','c','\0'}; // Memory allocated and initialized
char str5[10] = "thing"; // Memory allocated and initialized, pads with '\0'
char *str6    = "string literal!";  // Don't do this.  This can't be modified.

printf("%s",str4);  // Prints chars until it reaches a '\0'

char str7[][6] = {"fox", "in", "socks"};   // str5 is 3x6, 2D array of chars
char *str8[3]  = {"fox", "in", "socks"};   // Don't do this.
str6 is an array of 3 char pointers to 3 STRING LITERALS of length 4, 3, and 6.

char *str9[] = {str2,str3,str4};  // This is fine.  3x1 Array of pointers to
those strings.  Values and pointers are modifiable.  Each string only has the
amount of memory allocated to it when it was initialized.  NOT a 3x4 char array.
```

Georgia Tech

Strings

```
char str5[][6] = {"fox", "in", "socks"};   // str5 is 3x6, 2D array of chars
```

f	o	x	\0	\0	\0
i	n	\0	\0	\0	\0
s	o	c	k	s	\0

3x6 array

In memory:

f	o	x	\0	\0	\0	i	n	\0	\0	\0	\0	s	o	c	k	s	\0

`str5[0][0]` through `str5[2][5]` are valid memory addresses and modifiable

Strings – Functions

- `strcmp(str1,str2)`
- `strcpy(str1,str2)`
- `strncpy(str1,str2,n)`
- `strcat(str1,str2)`
- `strlen(str)`

(functions above found in string.h)

Georgia Tech

Strings – Conclusions

- A string is a special array of chars
- A string is '\0' (null character) terminated
- There are standard library functions for string manipulation (most in string.h)
- Using strings makes code human readable but can cause unforeseen bugs if not careful

Review Questions

- What is the ASCII standard/ASCII table?

Review Questions

- What is the ASCII standard/ASCII table?

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

Review Questions

- **Pointers** – What is the difference between an instance of a type and a pointer to that type? Draw the name, address, & value table.

Review Questions

- **Pointers** – What is the difference between an instance of a type and a pointer to that type? Draw the name, address, & value table.

- A variable stores a value (int, char, double, etc). A pointer holds the address to a variable.

NAME	a	…	…	…	…	a_addr
ADDRESS	28feb8	…	…	…	…	28febc
VALUE	4	…	…	…	…	0

Georgia Tech

Review Questions

- **Arrays** – What is the difference between an array and a pointer and how do you use an array in assignments and function arguments?

- An array is a contiguous chunk of memory where multiple variables of the same type are stored serially. The name of the array (without indexing) returns the address of the first variable in the array
  - Square brackets [ ] to index
  - Send function arguments with * symbol

Georgia Tech