Characters & Strings Lesson 1 Outline

1. Characters & Strings Lesson 1 Outline
2. Numeric Encoding of Non-numeric Data #1
3. Numeric Encoding of Non-numeric Data #2
4. Representing Characters
5. How Characters Are Represented #1
6. How Characters Are Represented #2
7. Representing Digits
8. Representing Punctuation
9. ASCII
10. ASCII Table #1
11. ASCII Table #2
12. ASCII Table #3
13. ASCII Table #4
14. ASCII Confirmation Program #1
15. ASCII Confirmation Program #2
16. ASCII Confirmation Program #3
17. ASCII Confirmation Program #4
18. A `char` is an `int` #1
19. A `char` is an `int` #2
20. Declaring `char` Scalar Variables #1
21. Declaring `char` Scalar Variables #2
22. `char` Like `int` Example
23. `char` Scalar Literal Constants
24. `char` Scalar Literal Constant Example
25. Using `char` Scalar Variables
26. Using `char` Scalar Variables Example
27. `char` Arrays #1
28. `char` Arrays #2
29. Character Array Example #1
30. Character Array Example #2
31. Character Strings #1
32. String Terminator
33. Character String Assignment Example #1
34. Character String Assignment Example #2

Numeric Encoding of Non-numeric Data #1

In Programming Project #4, we ***encoded*** (represented) the cloud types using **integer** values:

1. cirrus
2. cumulus
3. stratus
4. alto-stratus
5. alto-cumulus
6. cirro-stratus
7. cirro-cumulus
8. cumulo-nimbus
9. strato-cumulus

If we wanted to add other items, for example:

10. nimbus
11. strato-nimbus

Numeric Encoding of Non-numeric Data #2

1. cirrus
2. cumulus
3. stratus

…

The numbers in these cases have no standard meaning with respect to the items that they encode; they've been chosen essentially at random.

So, we see that we can encode ***qualitative*** (non-numeric) values with ***quantitative*** (numeric) values, using **arbitrary** but **distinct** numeric values to encode a set of qualities.

Representing Characters

What’s the most important set of non-numeric values in computing?

It’s the one that allows the computer to communicate with us in a way that makes sense to actual real live human beings: **natural language**.

The most efficient way for computers to communicate in a natural language is by **writing**.

Writing is based on **characters**.

Characters are **non-numeric**.

So, we want a way to **encode** characters numerically.

How Characters Are Represented #1

Here's a code you might have used to play secret code games when you were a kid:

'A' = 1, 'B' = 2, 'C' = 3, 'D' = 4, . . ., 'Z' = 26

Now that you've grown up and taken CS1313, you realize that the numbers that you choose can be <u>**arbitrary**</u>, as long as they're <u>**fixed**</u> and <u>**distinct**</u>.

So you could just as easily choose:
'A' = 65, 'B' = 66, 'C' = 67, 'D' = 68, . . ., 'Z' = 90

This is a perfectly reasonable encoding, if the only characters that you care about are upper case letters.

What about lower case?

How Characters Are Represented #2

'A' = 65, 'B' = 66, 'C' = 67, 'D' = 68, . . ., 'Z' = 90

What about lower case?

Well, you could add, for example:

'a' = 97, 'b' = 98, 'c' = 99, 'd' = 100, . . ., 'z' = 122

Are these the only characters that you need?

Representing Digits

Another kind of very important character is a digit.

Here’s a possible encoding of the decimal digits:

'0' = 48, '1' = 49, '2' = 50, '3' = 51, . . ., '9' = 57

Notice that there’s an important distinction between the character to be represented, which happens to be a digit, and the numeric encoding, whose value doesn’t have to have anything to do with the value of the digit being encoded.

Representing Punctuation

In addition to the upper case letters, the lower case letters and the digits, we also need to encode special characters such as punctuation.

This is starting to get pretty complicated, so maybe it'd help to have a standardized system.

ASCII

The ***American Standard Code for Information Interchange*** (ASCII)[*] is a standardized system for encoding characters numerically.

It has several categories of characters:

- letters:
  - upper case ('A' = 65 through 'Z' = 90);
  - lower case ('a' = 97 through 'z' = 122);
- digits ('0' = 48 through '9' = 57);
- punctuation
  - space = 32 through slash = 47;
  - colon = 58 through at sign = 64;
  - open square bracket = 91 through backquote = 96;
  - open curly brace = 123 through tilde = 126;
- ***control characters***, encoded as 0 through 31; also **DEL** (encoded as 127).

[*] http://www.asciitable.com/

ASCII Table #1

Code	Char	Kbd	Name	Code	Char	Kbd	Name
0	NUL		Null	16	DLE	Ctrl-P	Data Line Escape
1	SOH	Ctrl-A	Start of Heading	17	DC1	Ctrl-Q	Device Control 1
2	STX	Ctrl-B	Start of Text	18	DC2	Ctrl-R	Device Control 2
3	ETX	Ctrl-C	End of Text	19	DC3	Ctrl-S	Device Control 3
4	EOT	Ctrl-D	End of Transmission	20	DC4	Ctrl-T	Device Control 4
5	ENQ	Ctrl-E	Enquiry	21	NAK	Ctrl-U	Negative Acknowledge
6	ACK	Ctrl-F	Acknowledge	22	SYN	Ctrl-V	Synchronous File
7	BEL	Ctrl-G	Ring Bell	23	ETB	Ctrl-W	End Transmission Block
8	BS	Ctrl-H	Backspace	24	CAN	Ctrl-X	Cancel
9	HT	Ctrl-I	Horizontal Tab	25	EM	Ctrl-Y	End of Medium
10	LF	Ctrl-J	Line Feed	26	SUB	Ctrl-Z	Substitute
11	VT	Ctrl-K	Vertical Tab	27	ESC	Ctrl-Shift-K	Escape
12	FF	Ctrl-L	Form Feed	28	FS	Ctrl-Shift-L	File Separator
13	CR	Ctrl-M	Carriage Return	29	GS	Ctrl-Shift-M	Group Separator
14	SO	Ctrl-N	Shift Out	30	RS	Ctrl-Shift-N	Record Separator
15	SI	Ctrl-O	Shift In	31	US	Ctrl-Shift-O	Unit Separator

ASCII Table #2

Code	Char	Name	Code	Char	Name
32		Blank space	48	0	
33	!	Exclamation point	49	1	
34	"	Double quote	50	2	
35	#	Pound	51	3	
36	$	Dollar sign	52	4	
37	%	Percent	53	5	
38	&	Ampersand	54	6	
39	'	Single quote	55	7	
40	(	Open parenthesis	56	8	
41	)	Close parenthesis	57	9	
42	*	Asterisk	58	:	Colon
43	+	Plus	59	;	Semicolon
44	,	Comma	60	<	Less than
45	-	Hyphen	61	=	Equals
46	.	Period	62	>	Greater than
47	/	Slash	63	?	Question mark

ASCII Table #3

Code	Char	Name	Code	Char	Name
64	@	At	80	P	
65	A		81	Q	
66	B		82	R	
67	C		83	S	
68	D		84	T	
69	E		85	U	
70	F		86	V	
71	G		87	W	
72	H		88	X	
73	I		89	Y	
74	J		90	Z	
75	K		91	[	Open square bracket
76	L		92	\	Backslash
77	M		93	]	Close square bracket
78	N		94	^	Caret
79	O		95	_	Underscore

ASCII Table #4

Code	Char	Name	Code	Char	Name
96	`	Accent grave	112	p	
97	a		113	q	
98	b		114	r	
99	c		115	s	
100	d		116	t	
101	e		117	u	
102	f		118	v	
103	g		119	w	
104	h		120	x	
105	i		121	y	
106	j		122	z	
107	k		123	{	Open curly brace
108	l		124	|	Vertical bar
109	m		125	}	Close curly brace
110	n		126	~	Tilde
111	o		127	DEL	Delete

ASCII Confirmation Program #1

```
#include <stdio.h>

int main ()
{ /* main */
    const int first_printable_character_code =  32;
    const int last_printable_character_code  = 126;
    const int program_success_code           =   0;
    int index;

    for (index = first_printable_character_code;
         index <= last_printable_character_code;
         index++) {
        printf("ASCII Code #%3d is: %c\n",
            index, index);
    } /* for index */
    return program_success_code;
} /* main */
```

ASCII Confirmation Program #2

```
% gcc -o asciitest asciitest.c
% asciitest
ASCII Code # 32 is:
ASCII Code # 33 is: !
ASCII Code # 34 is: "
ASCII Code # 35 is: #
ASCII Code # 36 is: $
ASCII Code # 37 is: %
ASCII Code # 38 is: &
ASCII Code # 39 is: '
ASCII Code # 40 is: (
ASCII Code # 41 is: )
ASCII Code # 42 is: *
ASCII Code # 43 is: +
ASCII Code # 44 is: ,
ASCII Code # 45 is: -
ASCII Code # 46 is: .
ASCII Code # 47 is: /
```

```
ASCII Code # 48 is: 0
ASCII Code # 49 is: 1
ASCII Code # 50 is: 2
ASCII Code # 51 is: 3
ASCII Code # 52 is: 4
ASCII Code # 53 is: 5
ASCII Code # 54 is: 6
ASCII Code # 55 is: 7
ASCII Code # 56 is: 8
ASCII Code # 57 is: 9
ASCII Code # 58 is: :
ASCII Code # 59 is: ;
ASCII Code # 60 is: <
ASCII Code # 61 is: =
ASCII Code # 62 is: >
ASCII Code # 63 is: ?
```

ASCII Confirmation Program #3

```
ASCII Code # 64 is: @
ASCII Code # 65 is: A
ASCII Code # 66 is: B
ASCII Code # 67 is: C
ASCII Code # 68 is: D
ASCII Code # 69 is: E
ASCII Code # 70 is: F
ASCII Code # 71 is: G
ASCII Code # 72 is: H
ASCII Code # 73 is: I
ASCII Code # 74 is: J
ASCII Code # 75 is: K
ASCII Code # 76 is: L
ASCII Code # 77 is: M
ASCII Code # 78 is: N
ASCII Code # 79 is: O
```

```
ASCII Code # 80 is: P
ASCII Code # 81 is: Q
ASCII Code # 82 is: R
ASCII Code # 83 is: S
ASCII Code # 84 is: T
ASCII Code # 85 is: U
ASCII Code # 86 is: V
ASCII Code # 87 is: W
ASCII Code # 88 is: X
ASCII Code # 89 is: Y
ASCII Code # 90 is: Z
ASCII Code # 91 is: [
ASCII Code # 92 is: \
ASCII Code # 93 is: ]
ASCII Code # 94 is: ^
ASCII Code # 95 is: _
```

ASCII Confirmation Program #4

```
ASCII Code # 96 is: ‘
ASCII Code # 97 is: a
ASCII Code # 98 is: b
ASCII Code # 99 is: c
ASCII Code #100 is: d
ASCII Code #101 is: e
ASCII Code #102 is: f
ASCII Code #103 is: g
ASCII Code #104 is: h
ASCII Code #105 is: i
ASCII Code #106 is: j
ASCII Code #107 is: k
ASCII Code #108 is: l
ASCII Code #109 is: m
ASCII Code #110 is: n
ASCII Code #111 is: o
```

```
ASCII Code #112 is: p
ASCII Code #113 is: q
ASCII Code #114 is: r
ASCII Code #115 is: s
ASCII Code #116 is: t
ASCII Code #117 is: u
ASCII Code #118 is: v
ASCII Code #119 is: w
ASCII Code #120 is: x
ASCII Code #121 is: y
ASCII Code #122 is: z
ASCII Code #123 is: {
ASCII Code #124 is: |
ASCII Code #125 is: }
ASCII Code #126 is: ~
```

A char is an int #1

```
#include <stdio.h>

int main ()
{ /* main */
    const int  first_printable_character_code =  32;
    const int  last_printable_character_code  = 126;
    const int  program_success_code           =   0;
    int  index;

    for (index = first_printable_character_code;
         index <= last_printable_character_code;
         index++) {
        printf("ASCII Code #%3d is: %c\n",
            index, index);
    } /* for index */
    return program_success_code;
} /* main */
```

Notice that the variable named `index` is declared as an `int`, but in the `printf` statement, `index` can be used not only as an `int` but also as a `char`. The reverse is also true.

A `char` is an `int` #2

```
#include <stdio.h>

int main ()
{ /* main */
    const int  program_success_code           =   0;
    const char first_printable_character_code =  32;
    const char last_printable_character_code  = 126;
    char index;

    for (index = first_printable_character_code;
         index <= last_printable_character_code;
         index++) {
        printf("ASCII Code #%3d is: %c\n",
            index, index);
    } /* for index */
    return program_success_code;
} /* main */
```

Notice that the variable named `index` is declared as a `char`, but in the `printf` statement, `index` can be used not only as a `char` but also as an `int`. The reverse is also true.

Declaring `char` Scalar Variables #1

Here's a declaration of a `char` scalar variable:

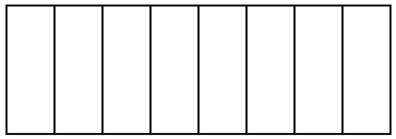
```
char first_initial;
```

This declaration tells the compiler to grab a group of bytes, name them `first_initial`, and think of them as storing a `char`.

How many bytes in a `char` scalar?

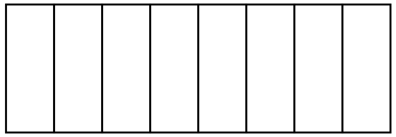
Each `char` scalar takes one byte:

`first_initial` :



Declaring `char` Scalar Variables #2

```
char first_initial;
```

`first_initial` :


REMEMBER: A `char` is just like an `int`, except that it uses fewer bytes:
typically, a `char` is 1 byte and an `int` is 4 bytes.

So, we can use `char` variables and constants in exactly the same ways that we use `int` variables and constants.

char Like int Example

```
% cat charadd.c
#include <stdio.h>

int main ()
{ /* main */
    const int program_success_code = 0;
    int addend, augend;
    char sum;

    printf("What are the addend and augend?\n");
    scanf("%d %d", &addend, &augend);
    sum = addend + augend;
    printf("The sum is %d.\n", sum);
    return program_success_code;
} /* main */
% gcc -o charadd charadd.c
% charadd
What are the addend and augend?
1 4
The sum is 5.
```

`char` Scalar Literal Constants

A ***character scalar literal constant*** is a single `char` enclosed in single quotes:

```
'H'
```

Note that

```
'''
```

is illegal.

However, you can also represent an individual `char` literal using the ***octal*** (base 8) code that represents it.

For example, the apostrophe character corresponds to ASCII code 39 decimal, which converts to 47 octal. So we can represent the apostrophe character like so:

```
'\047'
```

char Scalar Literal Constant Example

```
% cat apostrophe.c
#include <stdio.h>

int main ()
{ /* main */
    const int program_success_code = 0;
    printf("Apostrophe: %c\n", '\047');
    return program_success_code;
} /* main */
% gcc -o apostrophe apostrophe.c
% apostrophe
Apostrophe: '
```

Using `char` Scalar Variables

In C, we can use `char` scalar variables in many of the same ways that we use `int` scalar variables. As we saw, for example, we can declare them:

```
char first_initial;
```

We can also assign `char` scalar values to `char` scalar variables, by enclosing them in single quotes:

```
first_initial = 'H';
```

We can output `char` scalar values from `char` scalar variables, like so:

```
printf("My first initial is %c.\n",
    first_initial);
```

Using char Scalar Variables Example

```
% cat charscalar.c
#include <stdio.h>

int main ()
{ /* main */
    const char computers_favorite_character = 'q';
    const int  program_success_code         = 0;
    char users_favorite_character;

    printf("What is your favorite character?\n");
    scanf("%c", &users_favorite_character);
    printf("Your favorite character is '%c'.\n",
        users_favorite_character);
    printf("My favorite character is '%c'.\n",
        computers_favorite_character);
    return program_success_code;
} /* main */
% gcc -o charscalar charscalar.c
% charscalar
What is your favorite character?
Z
Your favorite character is 'Z'.
My favorite character is 'q'.
```

`char` Arrays #1

In C, you can have an array of type `char`, just as you can have arrays of numeric types:

```
char my_name[12];
```

We can fill this `char` array with characters and be able to print them out.

`char` Arrays #2

```
my_name[ 0] = 'H';
my_name[ 1] = 'e';
my_name[ 2] = 'n';
my_name[ 3] = 'r';
my_name[ 4] = 'y';
my_name[ 5] = ' ';
my_name[ 6] = 'N';
my_name[ 7] = 'e';
my_name[ 8] = 'e';
my_name[ 9] = 'm';
my_name[10] = 'a';
my_name[11] = 'n';
```

Is this a good solution?

Character Array Example #1

```
#include <stdio.h>
int main ()
{ /* main */
    const int my_name_length = 12;
    char my_name[my_name_length];
    int  index;
    my_name[ 0] = 'H';
    my_name[ 1] = 'e';
    my_name[ 2] = 'n';
    my_name[ 3] = 'r';
    my_name[ 4] = 'y';
    my_name[ 5] = ' ';
    my_name[ 6] = 'N';
    my_name[ 7] = 'e';
    my_name[ 8] = 'e';
    my_name[ 9] = 'm';
    my_name[10] = 'a';
    my_name[11] = 'n';
    printf("My name is ");
    for (index = 0; index < my_name_length; index++) {
        printf("%c", my_name[index]);
    } /* for index */
    printf(".\n");
    return 0;
} /* main */
```

Character Array Example #2

```
% gcc -o chararray chararray.c
% chararray
My name is Henry Neeman.
```

This is an improvement, but it’s still not an efficient way to assign a sequence of characters to a variable.

What we want is a kind of `char` variable whose use will be convenient for inputting, outputting and using sequences of characters.

Character Strings #1

A ***character string*** is a sequence of characters with the following properties:

- it is **stored** like a `char` array;
- it is **used** like a `char` scalar.

In C, we declare a character string like so:

```
char my_name[my_name_length+1];
```

Notice that a character string is declared **exactly** like a `char` array; in fact, a character string **is** a `char` array.

String Terminator

The only **difference** between a `char` **array** and a character **string** is that the **length** of the `char` **string** is **one greater** than the number of characters to be stored, and that the last character in any C character string is the ***null character***, called `NUL`, which corresponds to integer value 0:

```
'\0'
```

A **null character** (integer 0) used to indicate the end of a string is known as a ***character string terminator***.

In general, a numeric value that is used to indicate that a particular state has been reached – for example, the end of a list – is called a ***sentinel*** value.

So, the character string terminator `NUL` is a sentinel that indicates the end of the string in question.

Character String Assignment Example #1

```
% cat charstrassnbad.c
#include <stdio.h>

int main ()
{ /* main */
    const int my_name_length       = 12;
    const int program_success_code =  0;
    char my_name[my_name_length + 1];

    my_name = "Henry Neeman"; /* <-- DOESN'T WORK! */
    printf("My name is %s.\n", my_name);
    return program_success_code;
} /* main */
% gcc -o charstrassnbad charstrassnbad.c
charstrassnbad.c: In function ‘main’:
charstrassnbad.c:8: incompatible types
  in assignment
```

The version above seems like it should work, but it doesn't!

Character String Assignment Example #2

```
% cat charstrassn.c
#include <stdio.h>
#include <string.h>

int main ()
{ /* main */
    const int my_name_length         = 12;
    const int program_success_code =  0;
    char my_name[my_name_length + 1];

    strcpy(my_name, "Henry Neeman"); /* <-- WORKS! */
    printf("My name is %s.\n", my_name);
    return program_success_code;
} /* main */
% gcc -o charstrassn charstrassn.c
% charstrassn
My name is Henry Neeman.
```

This version works!