Problem Solving Techniques 문제해결

Jinkyu Lee

Dept. of Computer Science and Engineering,
Sungkyunkwan University (SKKU)

Contents

- **Chapter 3 – String**
  - Program design example

Character code

- Character codes are mappings between numbers and the symbols which make up a particular alphabets.
- The *American Standard Code for Information Interchange* (ASCII) is a single-byte character code where 2^7=128 characters are specified. Bytes are eight-bit entities; so that means the highest-order bit is left as zero.

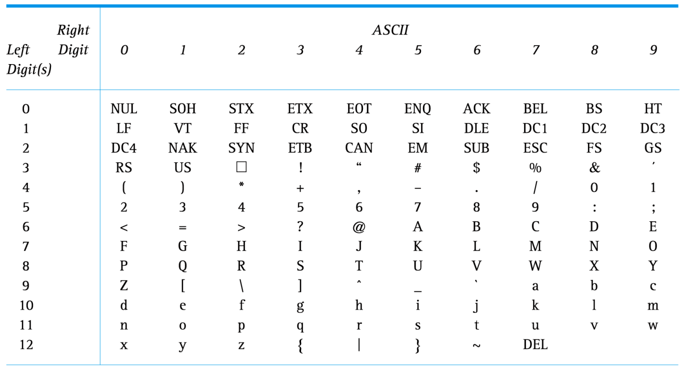
Character code – ASCII code

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

Properties of ASCII

- Several properties of the design make programming task easier

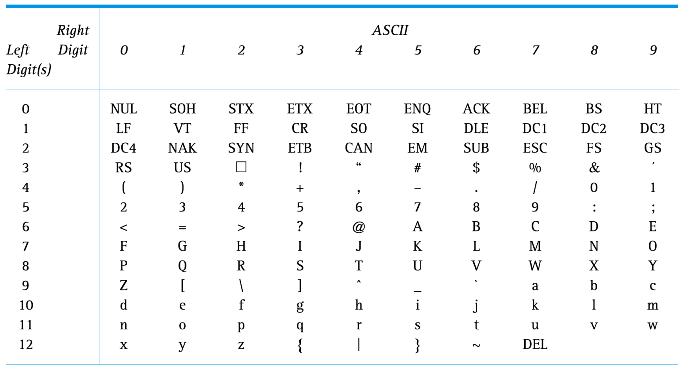
  - All non-printable characters have either the first three bits as zero or all seven lowest bits as one. This makes it very easy to eliminate them before displaying junk.



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

Properties of ASCII

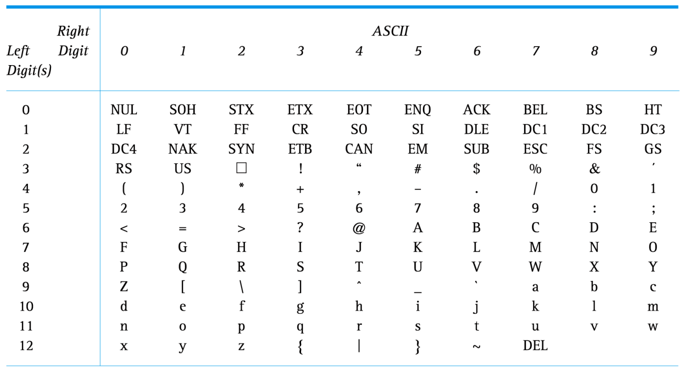
- Several properties of the design make programming task easier
  - All non-printable characters have either the first three bits as zero or all seven lowest bits as one. This makes it very easy to eliminate them before displaying junk.
  - Both the upper- and lowercase letters and the numerical digits appear sequentially. Thus we can iterate through all the letters/digits simply by looping from the value of the first symbol (say, ‘a’) to value of the last symbol (say, ‘z’).



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

Properties of ASCII

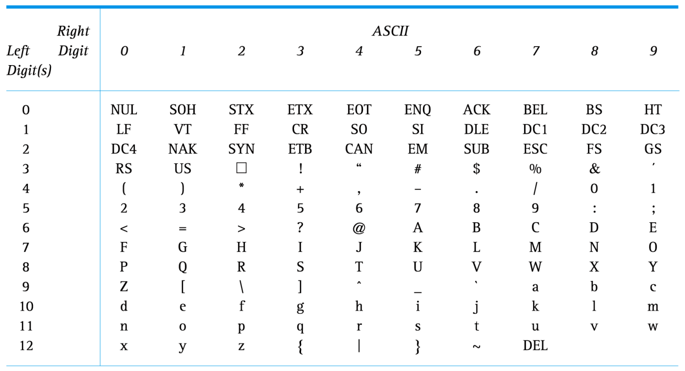
- Several properties of the design make programming task easier
  - All non-printable characters have either the first three bits as zero or all seven lowest bits as one. This makes it very easy to eliminate them before displaying junk.
  - Both the upper- and lowercase letters and the numerical digits appear sequentially. Thus we can iterate through all the letters/digits simply by looping from the value of the first symbol (say, ‘a’) to value of the last symbol (say, ‘z’).
  - We can convert a character (say, ‘I’) to its rank in the collating sequence (eighth, if ‘A’ is the zeroth character) simply by subtracting off the first symbol (‘A’).



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

Properties of ASCII

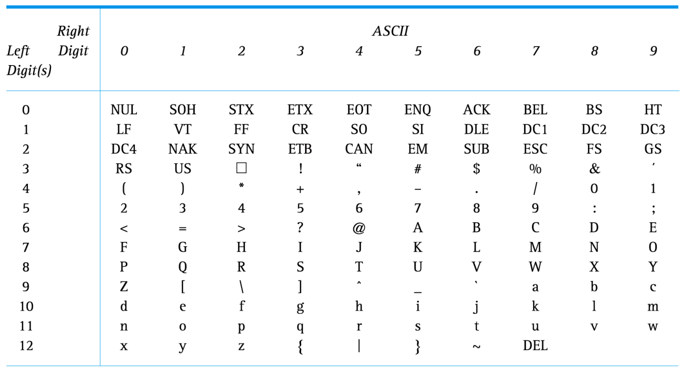
- We can convert (say ‘C’) from upper- to lowercase by adding the difference of the upper and lowercase starting character (‘C’-‘A”+‘a’). Similarly, a character x is uppercase if and only if it lies between ‘A’ and ‘Z’.



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

Properties of ASCII

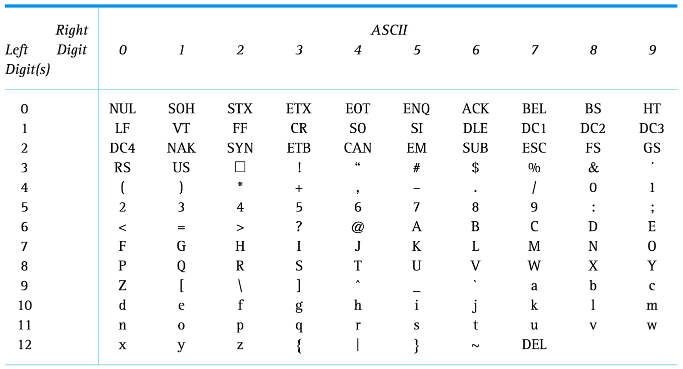
- We can convert (say ‘C’) from upper- to lowercase by adding the difference of the upper and lowercase starting character (‘C’-‘A”+‘a’). Similarly, a character x is uppercase if and only if it lies between ‘A’ and ‘Z’.
- The character code tells us what will happen when naively sorting text files. Which of ‘x’ or ‘3’ or ‘C’ appears first in alphabetical order? Sorting alphabetically means sorting by character code. Using a different collating sequence requires more complicated comparison functions.



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

Properties of ASCII

- We can convert (say ‘C’) from upper- to lowercase by adding the difference of the upper and lowercase starting character (‘C’-‘A”+‘a’). Similarly, a character x is uppercase if and only if it lies between ‘A’ and ‘Z’.
- The character code tells us what will happen when naively sorting text files. Which of ‘x’ or ‘3’ or ‘C’ appears first in alphabetical order? Sorting alphabetically means sorting by character code. Using a different collating sequence requires more complicated comparison functions.
- Non-printable character codes for new-line (10) and carriage return (13) are designed to delimit the end of text lines. Inconsistent use of these codes is one of the pains in moving text files between UNIX and Windows systems.



Right Digit	*ASCII*									
Left Digit(s)	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	“	#	$	%	&	´
4	(	)	*	+	,	–	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	ˆ	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Unicode

- More modern international character code designs such as *Unicode* use two or even three bytes per symbol, and can represent virtually any symbol in every language on earth. Older languages, like Pascal, C, and C++, view the char type as virtually synonymous with 8-bit entities. However, good old ASCII remains alive embedded in Unicode. Java, on the other hand, was designed to support Unicode, so characters are 16-bit entities. The upper byte is all zeros when working with ASCII/ISO Latin 1 text.

Representing strings

- Strings are sequences of characters, where order clearly matters. It is important to be aware of how your favorite programming language represents strings, because there are several different possibilities.

  - *Null-terminated Arrays* – C/C++ treats strings as arrays of characters. The string ends the instant it hits the null character "**\0**", i.e., zero ASCII. Failing to end your string explicitly with a null typically extends it by a bunch of unprintable characters.

Representing strings

- *Array Plus Length* – Another scheme uses the first array location to store the length of the string, thus avoiding the need for any terminating null character. Presumably this is what Java implementations do internally.

- *Linked Lists of Characters* – Text strings can be represented using linked lists, but this is typically avoided because of the high space-overhead associated with having a several-byte pointer for each single byte character.

Which string representation?

- The underlying string representation can have a big impact on which operations are easily or efficiently supported. Compare each of these three data structures with respect to the following properties
  - Which uses the least amount of space? On what sized strings?
  - Which constrains the content of the strings which can possibly be represented?
  - Which allow constant-time access to the ith character?
  - Which allow efficient checks that the ith character is in fact within the string, thus avoiding out-of-bounds errors?
  - Which allow efficient deletion or insertion of new characters at the ith position?
  - Which representation is used when users are limited to strings of length at most 255, e.g., file names in Windows?

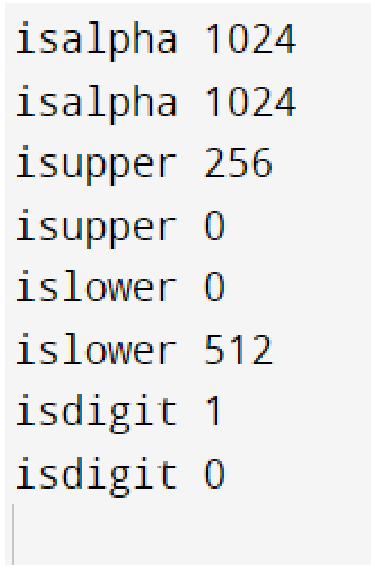
Null-terminated Arrays *Array Plus Length* *Linked Lists of Characters*

C string library functions

- The C language *character* library ctype.h contains several simple tests and manipulations on character codes. As with all C predicates, true is defined as any non-zero quantity, and false as zero.

```
#include <ctype.h>          /* include the character library */

int isalpha(int c);         /* true if c is either upper or lower case */
int isupper(int c);         /* true if c is upper case */
int islower(int c);         /* true if c is lower case */
int isdigit(int c);         /* true if c is a numerical digit (0-9) */
int ispunct(int c);         /* true if c is a punctuation symbol */
int isxdigit(int c);        /* true if c is a hexadecimal digit (0-9,A-F) */
int isprint(int c);         /* true if c is any printable character */

int toupper(int c);         /* convert c to upper case -- no error checking */
int tolower(int c);         /* convert c to lower case -- no error checking */
```

```
printf(" isalpha %d\n", isalpha('A'));
printf(" isalpha %d\n", isalpha('a'));
printf(" isupper %d\n", isupper('A'));
printf(" isupper %d\n", isupper('a'));
printf(" islower %d\n", islower('A'));
printf(" islower %d\n", islower('a'));
printf(" isdigit %d\n", isdigit('1'));
printf(" isdigit %d\n", isdigit('a'));
```



```
isalpha 1024
isalpha 1024
isupper 256
isupper 0
islower 0
islower 512
isdigit 1
isdigit 0
```

```
#include <ctype.h>       /* include the character library */

int isalpha(int c);      /* true if c is either upper or lower case */
int isupper(int c);      /* true if c is upper case */
int islower(int c);      /* true if c is lower case */
int isdigit(int c);      /* true if c is a numerical digit (0-9) */
int ispunct(int c);      /* true if c is a punctuation symbol */
int isxdigit(int c);     /* true if c is a hexadecimal digit (0-9,A-F) */
int isprint(int c);      /* true if c is any printable character */

int toupper(int c);      /* convert c to upper case -- no error checking */
int tolower(int c);      /* convert c to lower case -- no error checking */
```

C string library functions

- These appear in the C language *string* library string.h.

```
#include <string.h>      /* include the string library */

char *strcat(char *dst, const char *src);      /* concatenation */
int strcmp(const char *s1, const char *s2);    /* is s1 == s2? */
char *strcpy(char *dst, const char *src);      /* copy src to dist */
size_t strlen(const char *s);                  /* length of string */
char *strstr(const char *s1, const char *s2);  /* search for s2 in s1 */
char *strtok(char *s1, const char *s2);        /* iterate words in s1 */
```

Contents

- Chapter 3 – String
  - **Program design example**

From now on, slides are adapted from Prof. Chang Wook Ahn’ s slides.

Contents

1. Example: Corporate Renamings

- Your company, Digiscam, has put to work on a program that maintains a database of corporate name changes and does the appropriate substitutions to bring old documents up to date.
  - Take as input a file with a given number of corporate name changes, followed by a given number of lines of text to correct!
  - Only exact matches of the string should be replaced.
  - There are at most 100 corporate changes, each line is at most 1,000 characters

```
4   #corporate name changes
"Anderson Consulting" to "Accenture"
"Enron" to "Dynegy"
"DEC" to "Compaq"
"TWA" to "American"
5   #lines of text to correct
Anderson Accounting begat Accenture, which
offered advice to Dynergy before it CompaqLARED bankruptcy,
which made Anderson
Consulting quite happy it changed its name
in the first place!
```

Outcome!

2. String

- Since the string is a basic data structure, its gravity is growing in the string processing.
  - e.g., Internet search engine such as Google, The sequencing of the human genome
- How to represent characters and strings?
- How to discover a certain pattern in the string?

2. Character Codes

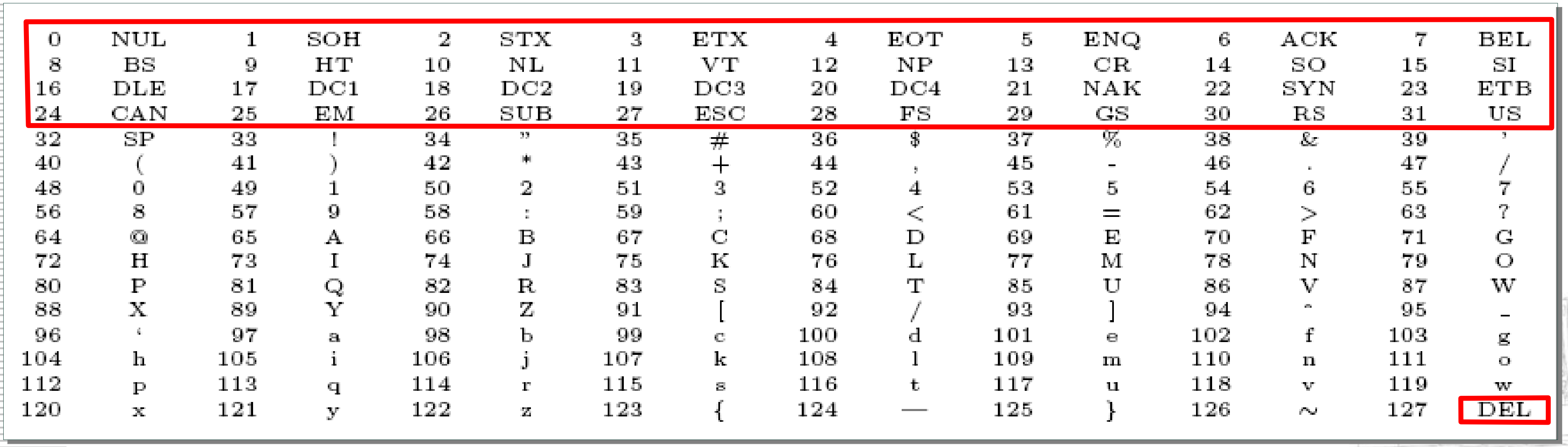
❖ Character Codes

- Mappings between numbers and the symbols making up a particular alphabet
- ASCII code – 1 byte (2^7 = 128) character code

❖ ASCII Code Properties

'F'-5='A'
'C'-'A'+'a'='c'

- **Non-printable characters:** the first three 3bits as '0', all seven lowest bits as '1'
- **Upper/lowercase letters and the numerical digits appear sequentially.**
  - We can convert a character from upper- to lowercase by adding difference of the starting character
- **Given the code, we can predict what will happen when naively sorting text files**
- **Non-printable character(10) and carriage return(13)** are to delimit the end of text lines



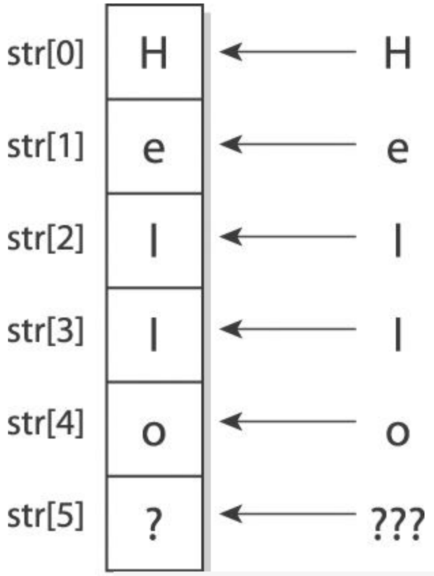
0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3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

3. Representing Strings <1>

❖ Null-terminated Arrays : C/C++

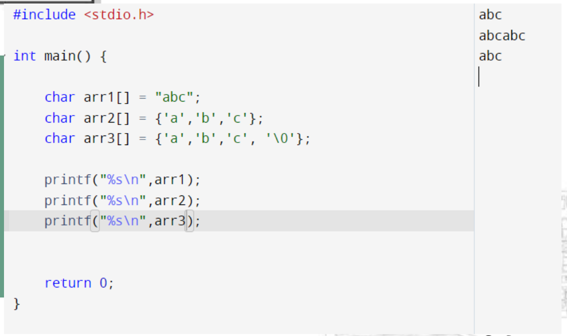
- C/C++ treats strings as arrays of characters
- The string ends the instant it hits the null character '\0', zero ASCII

```
int main(void)
{
    char str[6]="Hello";


    . . . . .
```



- "Hello0" and "Hello\0" are different
- String vs. Character array

```
char arr1[ ] = "abc";
char arr2[ ] = {'a', 'b', 'c'};
char arr3[ ] = {'a', 'b', 'c', '\0'};
```

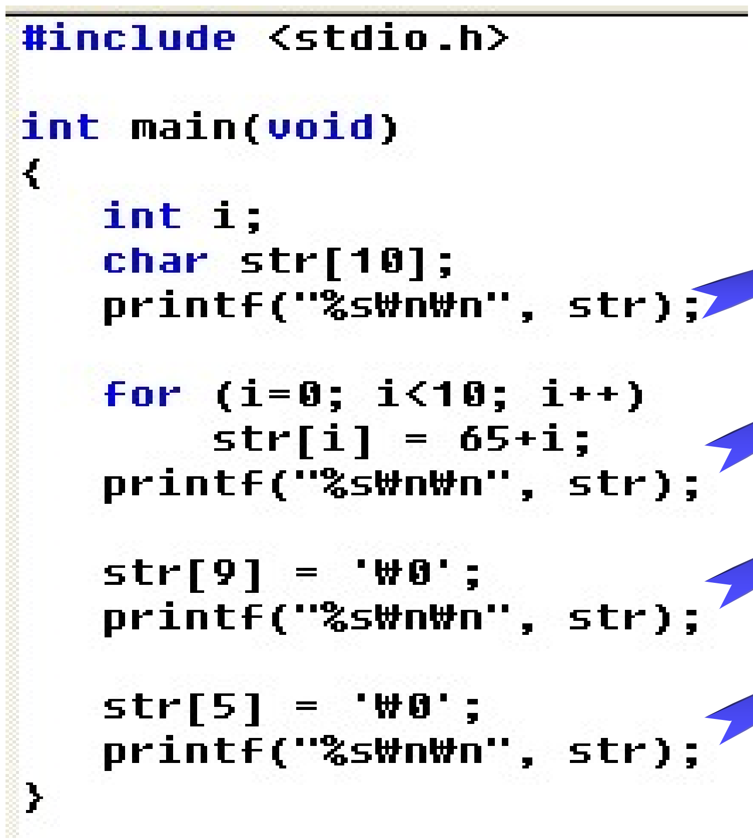


```
#include <stdio.h>

int main() {

    char arr1[] = "abc";
    char arr2[] = {'a','b','c'};
    char arr3[] = {'a','b','c', '\0'};

    printf("%s\n",arr1);
    printf("%s\n",arr2);
    printf("%s\n",arr3);


    return 0;
}
```

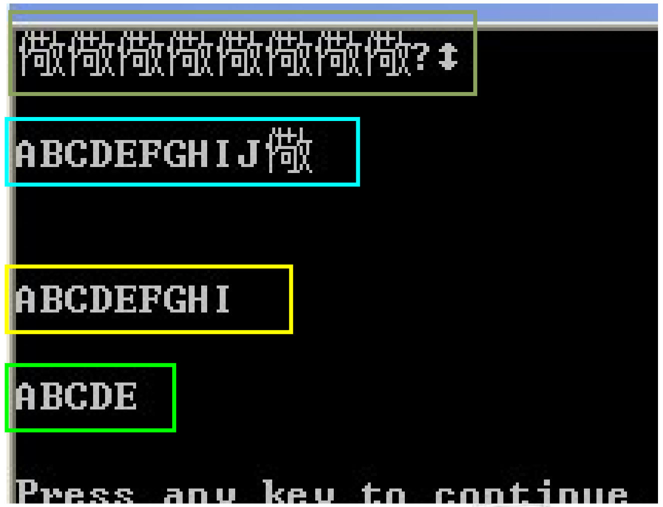
```
abc
abcabc
abc
```

3. Representing Strings <2>

- **Why is the null necessary?**
  - To represent the end of string
  - To discriminate between Garbage and String
  - To designate the range of print by printf()



```c
#include <stdio.h>

int main(void)
{
    int i;
    char str[10];
    printf("%s₩n₩n", str);

    for (i=0; i<10; i++)
        str[i] = 65+i;
    printf("%s₩n₩n", str);

    str[9] = '₩0';
    printf("%s₩n₩n", str);

    str[5] = '₩0';
    printf("%s₩n₩n", str);
}
```

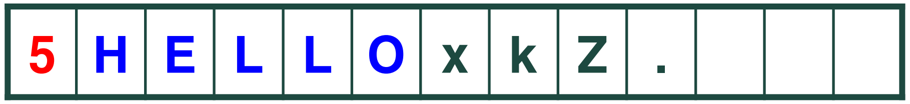


```
儆儆儆儆儆儆儆儆?↕

ABCDEFGHIJ儆

ABCDEFGHI

ABCDE

Press any key to continue
```

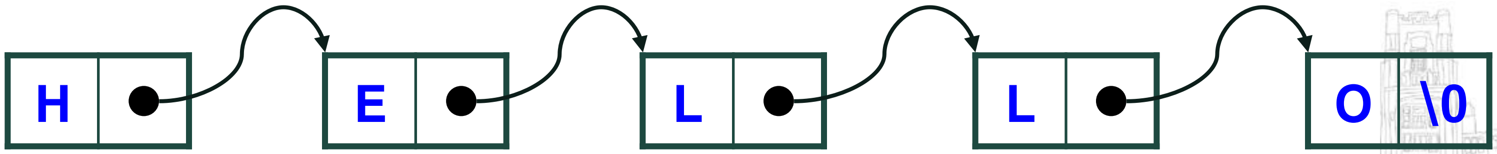
3. Representing Strings <3>

- **Array + Length**
  - **The length of string is stored at the first array location**

  ➔ **Not need to use ‘null’ character!**
  - **Presumably this is what Java implementations do internally.**

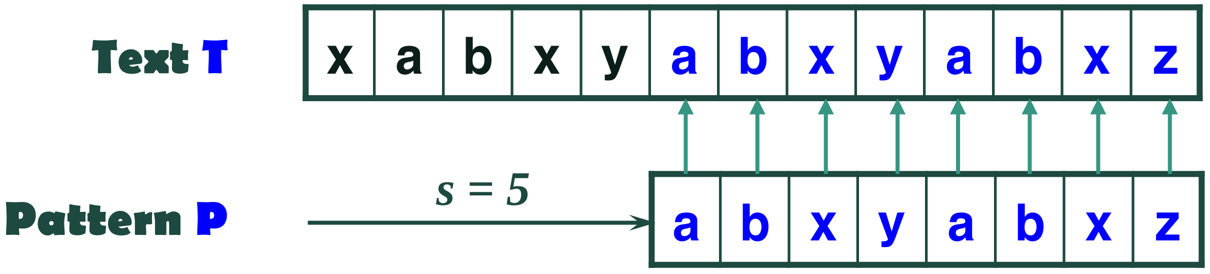


- **Linked List of Characters**
  - **Typically avoided due to the high space-overhead**
    - associated with having a several-byte pointer for each single byte character
  - **It might be useful when inserting/deleting substrings frequently.**



4. Searching for Patterns <1>

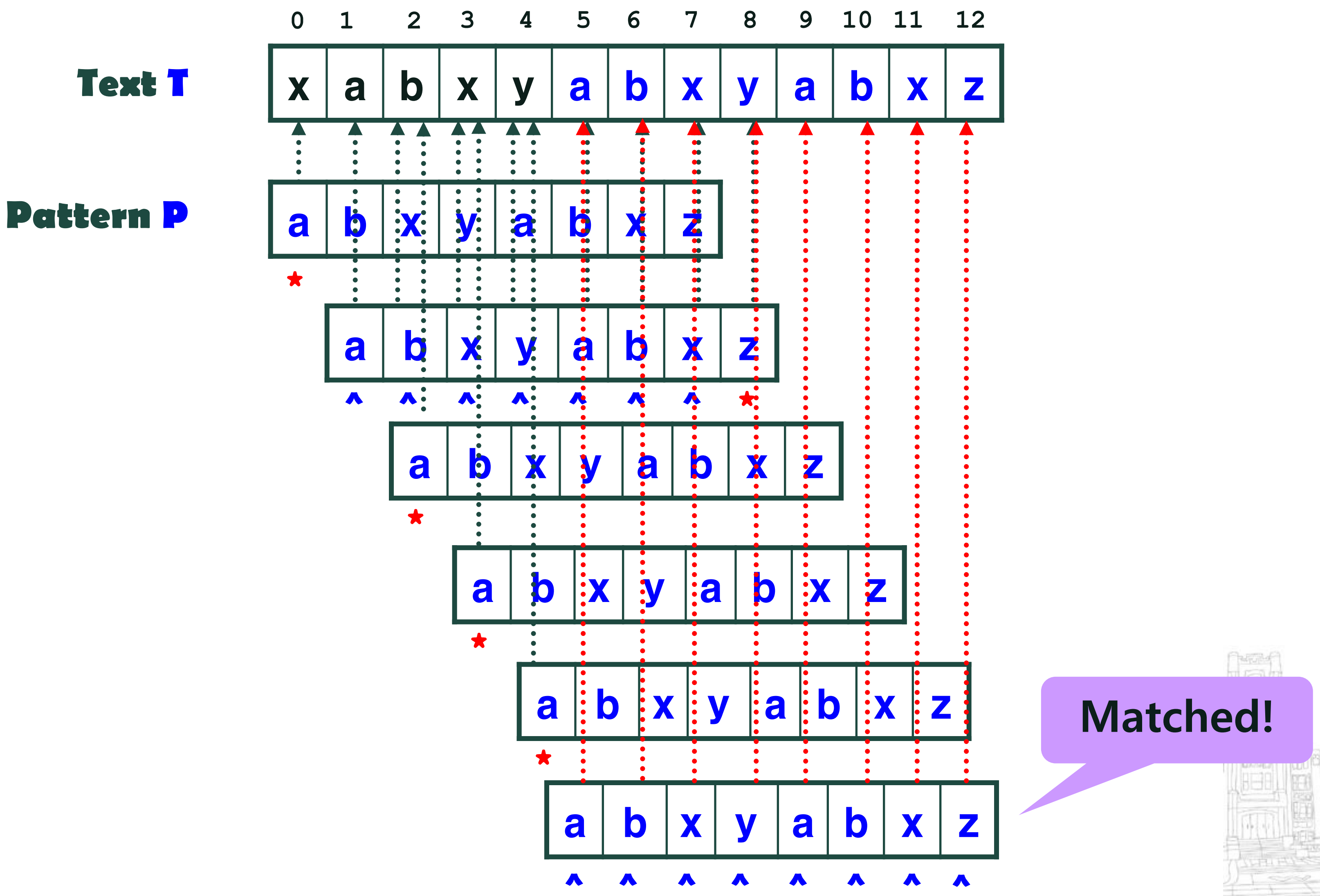
- Search for the pattern P in text T
  - T[1…n],   P[1…m], where m ≤ n
  - P *occurs with shift s* in text T if 0 ≤ *s* ≤ n-m
    - T[s+1…s+m] = P[1…m] , that is, (T[s+j] = P[j], for 1 ≤ j ≤m)



- Naïve method
  - It searches for exact pattern matches by comparing all the characters of P with those in text T.

4. Searching for Pattern ‹2›

❖ Naïve Method: Example



4. Searching for Pattern <3>

❖ Algorithm for Naïve method

```
Naïve-Match(P, T)
n ← length[T]
m ← length[P]
for s ← 0 to (n-m)
    do if P[1…m] = T[s+1…s+m]
        then return s
```

5. Example: Corporate Renamings <1>

❖ Your company, Digiscam, has put to work on a program that maintains a database of corporate name changes and does the appropriate substitutions to bring old documents up to date.

- Take as input a file with a given number of corporate name changes, followed by a given number of lines of text to correct!
- Only exact matches of the string should be replaced.
- There are at most 100 corporate changes, each line is at most 1,000 characters

```
4   #corporate name changes
"Anderson Consulting" to "Accenture"
"Enron" to "Dynegy"
"DEC" to "Compaq"
"TWA" to "American"
5   #lines of text to correct
Anderson Accounting begat Accenture, which
offered advice to Dynergy before it CompaqLARED bankruptcy,
which made Anderson
Consulting quite happy it changed its name
in the first place!
```

Outcome!

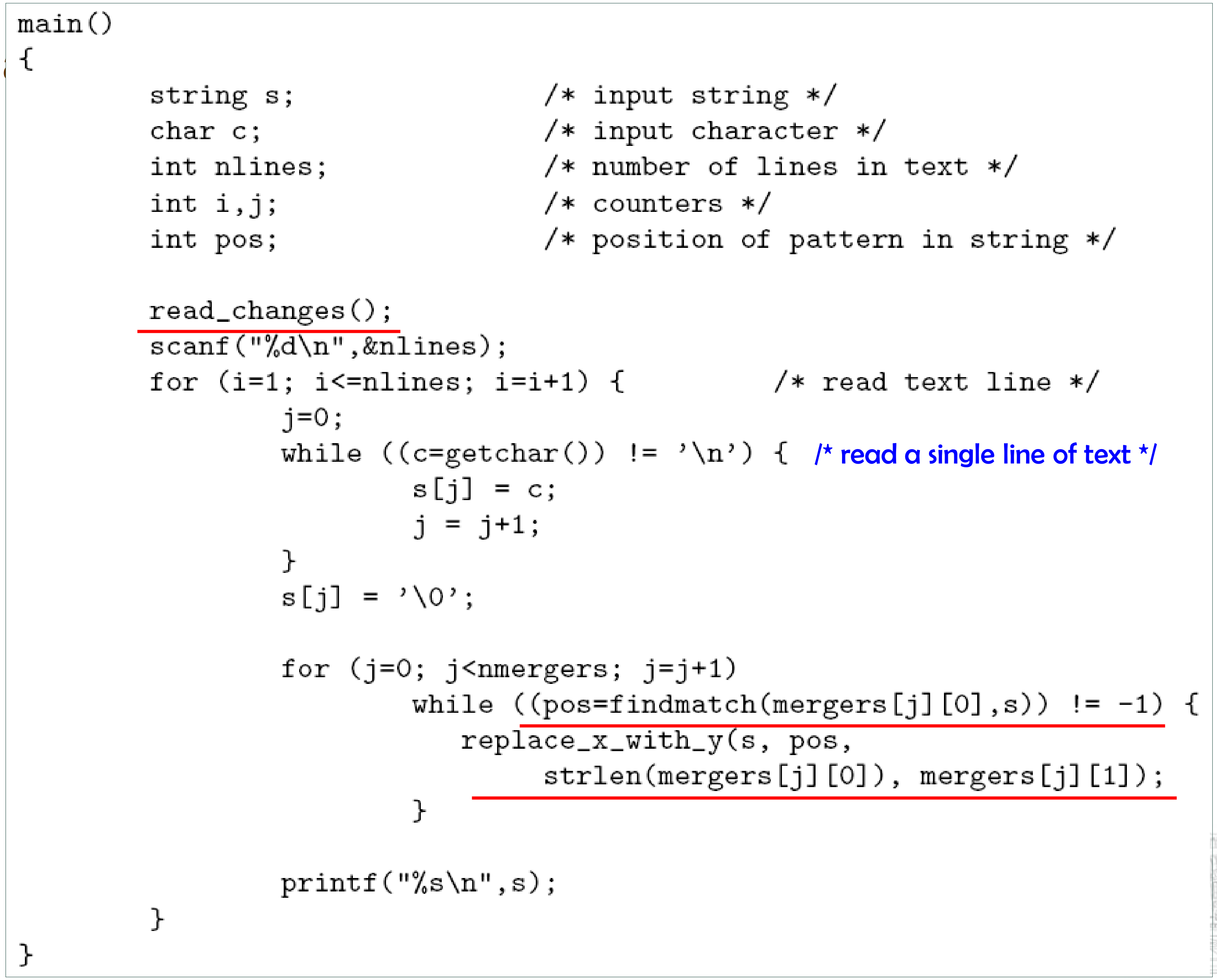
5. Example: Corporate Renamings <2>

- **Necessary Operators**
  - **Read and Store the strings**
  - **Searching for the Pattern, Substitution, and Printing out**
- **Input file consists of two parts**
  1. **Dictionary part: describe the corporate name changes**
  2. **Database part: a document stored in the database**
- **The names should be stored: Data structure**

```
#include <string.h>

#define MAXLEN          1001    /* longest possible string */
#define MAXCHANGES      101     /* maximum number of name changes */

typedef char string[MAXLEN];

string mergers[MAXCHANGES][2];  /* store before/after corporate names */
int nmergers;                   /* number of different name changes */
```

5. Example: Corporate Renamings <3>

❖ m



```
main()
{
        string s;                       /* input string */
        char c;                         /* input character */
        int nlines;                     /* number of lines in text */
        int i,j;                        /* counters */
        int pos;                        /* position of pattern in string */

        read_changes();
        scanf("%d\n",&nlines);
        for (i=1; i<=nlines; i=i+1) {             /* read text line */
                j=0;
                while ((c=getchar()) != '\n') {  /* read a single line of text */
                        s[j] = c;
                        j = j+1;
                }
                s[j] = '\0';

                for (j=0; j<nmergers; j=j+1)
                        while ((pos=findmatch(mergers[j][0],s)) != -1) {
                           replace_x_with_y(s, pos,
                                strlen(mergers[j][0]), mergers[j][1]);
                        }

                printf("%s\n",s);
        }
}
```

5. Example: Corporate Renamings <4>

❖ **Read and Store the corporate name changes**

```
read_changes()
{
        int i;                          /* counter */

        scanf("%d\n",&nmergers);
        for (i=0; i<nmergers; i++) {
                read_quoted_string(&(mergers[i][0]));
                read_quoted_string(&(mergers[i][1]));
        }
}
```

```
read_quoted_string(char *s)
{
        int i=0;                        /* counter */
        char c;                         /* latest character */

        while ((c=getchar()) != '\"') ;
        while ((c=getchar()) != '\"') {
                s[i] = c;
                i = i+1;
        }
        s[i] = '\0';
 }
```

Ex.,
"Enron" to "Dynegy"
"DEC" to "Compaq"

5. Example: Corporate Renamings <5>

❖ **Example of storing the name changes**

nmergers = 4, mergers[101][2]

	0	1
0	Anderson Consulting\0	Accenture\0
1	Enron\0	Dynegy\0
2	DEC\0	Compaq\0
3	TWA\0	American\0

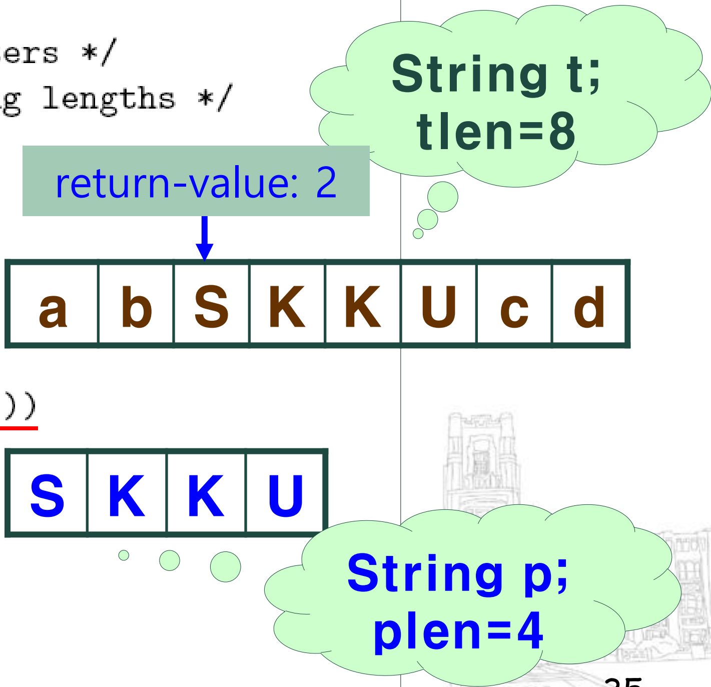
5. Example: Corporate Renamings ⟨6⟩

❖ Searching for Pattern

- **Discriminate Upper/lowercase letters**
- **Not possible to search for the pattern separated in two lines**

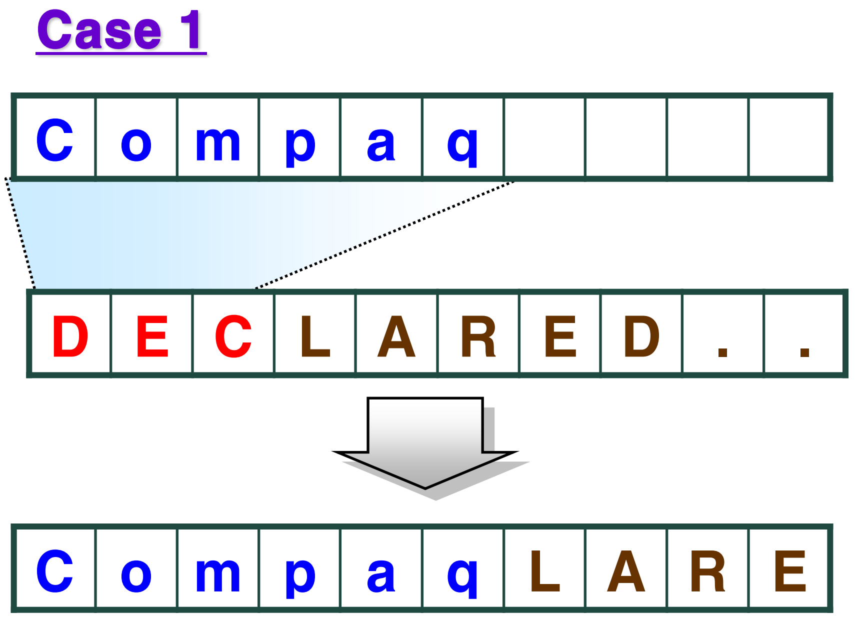
```
/*      Return the position of the first occurrence of the pattern p
        in the text t, and -1 if it does not occur.
*/

int findmatch(char *p, char *t)
{
        int i,j;                        /* counters */
        int plen, tlen;                 /* string lengths */

        plen = strlen(p);
        tlen = strlen(t);

        for (i=0; i<=(tlen-plen); i=i+1) {
                j=0;
                while ((j<plen) && (t[i+j]==p[j]))
                        j = j+1;
                if (j == plen) return(i);
        }

        return(-1);
}
```

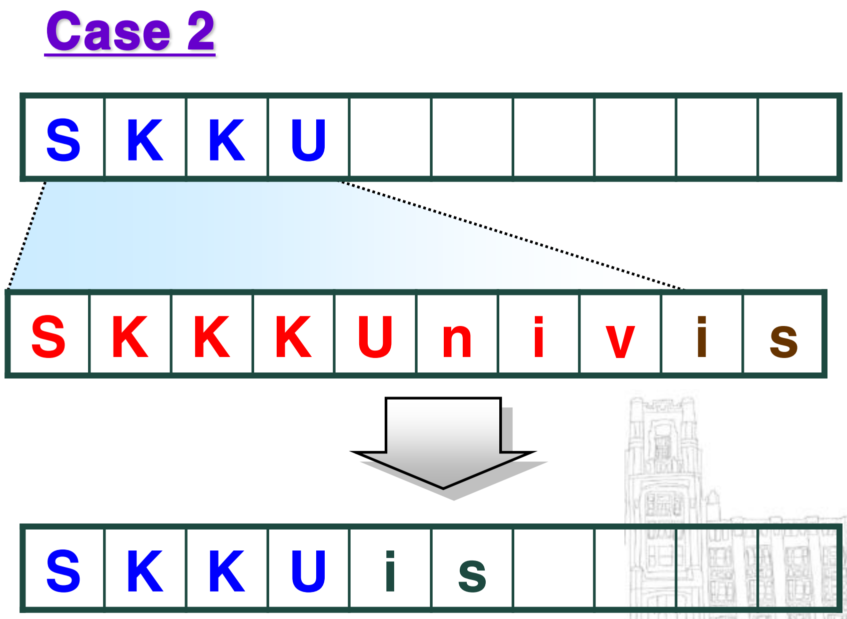
Cf)
Time Complexity
$O(|p||t|)$



5. Example: Corporate Renamings <7>

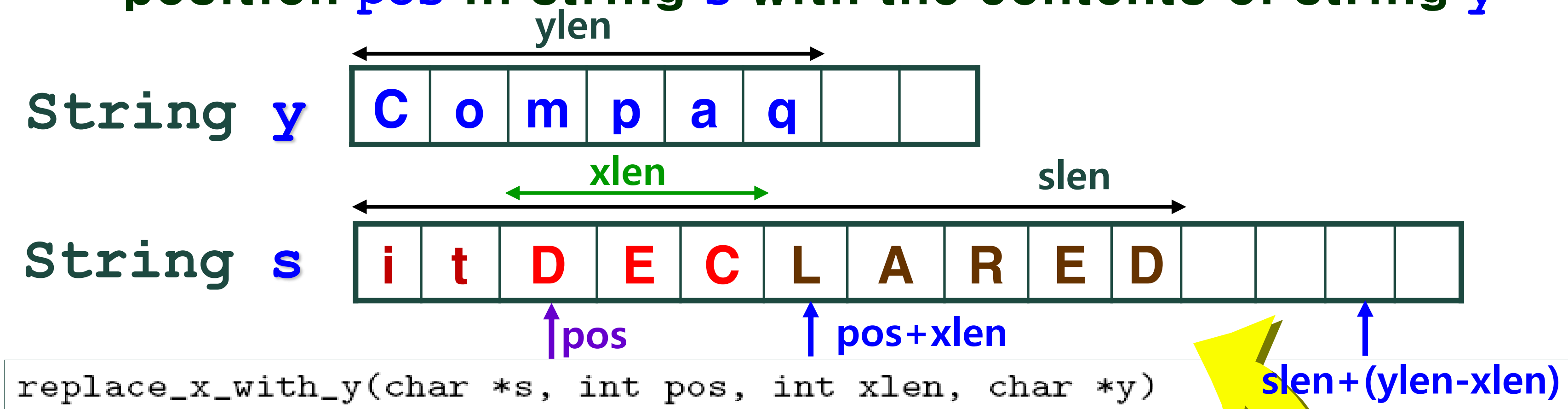
- String Manipulation – Computing length, Copying, Reversing
- In the Example, Replacing a substring is used
  - Case 1. If the replacement string is longer than the original
    → Move the suffix out of the way so it isn't stepped on
  - Case 2. If the replacement string is shorter
    → Move the suffix in to cover up the hole





5. Example: Corporate Renamings <8>

- **Replace the substring of length `xlen` starting at position `pos` in string `s` with the contents of string `y`**



```
replace_x_with_y(char *s, int pos, int xlen, char *y)
{
        int i;                              /* counter */
        int slen,ylen;                      /* lengths of relevant strings */

        slen = strlen(s);
        ylen = strlen(y);

        if (xlen >= ylen)
                for (i=(pos+xlen); i<=slen; i++) s[i+(ylen-xlen)] = s[i];
        else
                for (i=slen; i>=(pos+xlen); i--) s[i+(ylen-xlen)] = s[i];

        for (i=0; i<ylen; i++) s[pos+i] = y[i];
}
```