CS 137

Characters and Strings

Fall 2025

Victoria Sakhnini

Table of Contents

ASCII & Characters

We've already seen this briefly earlier in the term.

Syntax: `char c;`

It is an 8-bit integer. The integer can be code representing printable and unprintable characters. It can also store a single letter via, say, `char c = 'a';`

Example1:

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.          char c1 = 'a';
6.          char c2 = 97;
7.          printf("%c\n", c1);  // output: a
8.          printf("%d\n", c1);  // output: 97
9.          printf("%d\n", c2);  // output: 97
10.         printf("%c\n", c2);  // output: a
11.         return 0;
12.  }
```

Example2:

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.          char c1 = 'A';
6.          char c2 = 65;
7.          if (c1 == c2)       // Output: wow
8.                  printf("wow\n");
9.          else
10.                 printf("hmmmm\n");
11.         return 0;
12.  }
```

 How is text represented in computers? ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) codes represent text in computers and other devices. More modern schemes are based on ASCII to support many additional characters.

ASCII uses 7 bits, with the 8th bit either used for a parity check bit or extended ASCII, ranging from 0000000-1111111.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

The image is courtesy of http://www.hobbyprojects.com/ascii-table/ascii-table.html

Characters `0-31` are control characters
Characters `48-57` are the numbers `0` to `9`
Characters `65-90` are the letters `A` to `Z`
Characters `97-122` are the letters `a` to `z`
Note that `'A'` and `'a'` are `32` letters away

Example3: To demonstrate the similarity/differences between `int` and `char`

```c
#include <stdio.h>

int main(void){
        char c1 = 'A';
        char c2 = 65;
        int i = 'a';
        int j = 97;
        printf("%c\n", c1);  // Output: A
        printf("%d\n", c1);  // Output: 65
        printf("%c\n", c2);  // Output: A
        printf("%d\n", c2);  // Output: 65
        printf("%c\n", i);   // Output: a
        printf("%d\n", i);   // Output: 97
        printf("%c\n", j);   // Output: a
        printf("%d\n", j);   // Output: 97
        if (c1 == c2)        // Output: wow
                printf("wow\n");
        else
                printf("hmmmm\n");
        if (i == j)            // Output: nice
                printf("nice\n");
        else
                printf("what?????\n");
        return 0;
}
```

Example 4: A program of a basic calculator

```c
#include <stdio.h>

int main(void)
{
        char op;
        int a, b;
        printf("Enter an operation (+,-,*,/): ");
        scanf("%c", &op);
        printf("Enter two integers: ");
        scanf("%d", &a);
        scanf("%d", &b);
        printf("%d %c %d = ", a, op, b);
        switch (op)
        {
                case '+':
                        printf("%d\n", a + b);  break;
                case '-':
                        printf("%d\n", a - b);  break;
                case '*':
                        printf("%d\n", a * b);  break;
                case '/':
                        printf("%d\n", a / b);  break;
                default:
                        printf("wrong operation\n");
        }
        return 0;
}
```

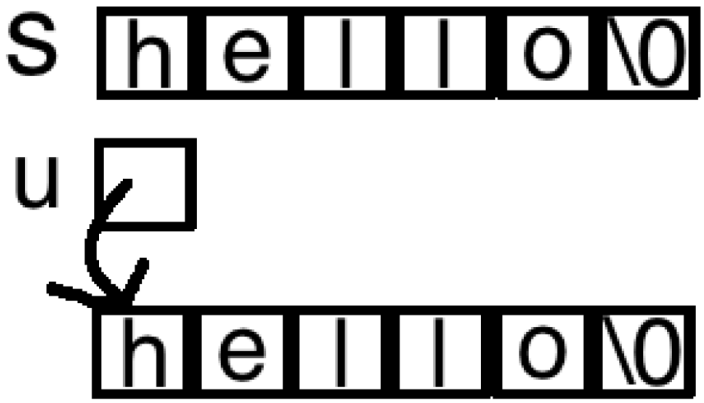
For the English language, ASCII is enough for most applications. However, many languages have far more complicated letter systems, and a new way to represent these would be required. To account for other languages, we now have Unicode, which we will discuss in a few lectures.

Strings

In C, a string is an array of characters **terminated by** a NULL character ('\0').

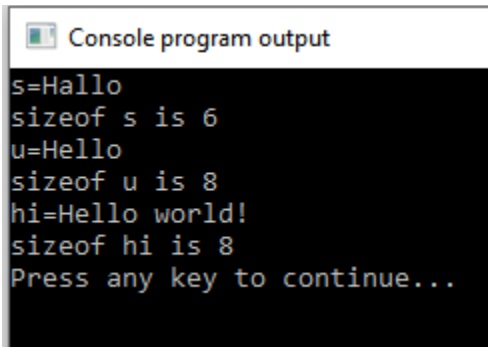
```c
#include <stdio.h>

int main(void){
        char s[] = "Hello";
        printf("%s\n", s); // Output: Hello

        // The next is the same as the previous.
        char t[] = { 'H', 'e', 'l', 'l', 'o', '\0 ' };
        printf("%s\n", t);  // Output: Hello

        // Slightly different
        char *u = "Hello";
        printf("%s\n", u);  // Output: Hello
        return 0;
}
```

Notice that the last one is slightly different than the previous two... The last one is a pointer to a string.



This doesn't seem like it but consider the following example:

```c
#include <stdio.h>

int main(void)
{
        char s[] = "Hello";
        s[1] = 'a';  // mutating the second letter
        printf("s=%s\n", s); // Output: s=Hallo
        printf("sizeof s is %zu\n", sizeof(s)); // output 6 (because s is an array of 6 characters and each char requires 1 byte.

        // Slightly different
        char *u = "Hello";
        // The next commented line causes an error! check the explanation below.
        //u [1] = 'a'    *********
        printf("u=%s\n", u);  // Output: u=Hello
        printf("sizeof u is %zu\n", sizeof(u));  // Output: 8 because the size of a pointer is 8 bytes

        char *hi = "Hello" " world!";
        printf("hi=%s\n", hi);  // Output: hi=Hello world!
        printf("sizeof hi is %zu\n", sizeof(hi));  // Output: 8 because the size of a pointer is 8 bytes
        return 0;
 }
```



In `char *u = "Hello";`, `"Hello"` is called a **string literal.**

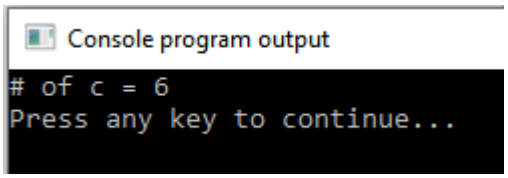
String literals cannot be changed, and attempting to change them causes undefined behaviour.

Reminder: Notice also that `sizeof(u)` is different from the `sizeof(s)` [**s is an array vs u is a pointer**]

Another note: `char *hi = "Hello"" world!";` will combine into one string literal.

One more example of how to process a string character by character if we don't know the length.:

```c
#include <stdio.h>

int main(void){
  char str[] = "count the number of times a character c occurs in a string";
  int i = 0;
  int cnt = 0;

  // str[i] is false if str[i] is the null character ('\0')
  // which is the end of the string
  while (str[i]) {
          if (str[i] == 'c')
                  cnt++;
          i++;
  }
  printf("# of c = %d\n", cnt);
  return 0;
  }
```



Functions for Strings

In C, string manipulations are very tedious and cumbersome. However, there is a library that can help with some of the basics. This being said other languages are far better at handling string manipulations than C. Before discussing these, we need a brief digression into `const` type qualifiers:

- The keyword `const` indicates that something is not modifiable, i.e., is read-only.
- Assignment to a `const` piece of data results in error.
- It is helpful to tell other programmers about the nature of a variable

Examples:

- `const int i = 10;` is a constant `i` whose value is initialized to be `10`.
- The command `i = 5;` will cause an error because you are trying to reassign a constant.
- Even though it is a constant - through bad programming, you could still change the value, but doing so is undefined behaviour as per the C standard:

```c
#include <stdio.h>

int main(void){
          const int i = 10;
          printf("%d\n", i);
          int *a = &i;    // This line will cause an error or a warning!!!!:
               // Operands of '=' have incompatible types 'int *' and 'const int *'.
          *a = 3;
         printf("%d\n", i);
          return 0;
  }
```

Let's summarize the differences between `const` and `#define`:

Constants

- `const` can be used to create read-only objects of any type we want, including arrays, structures, pointers etc.
- Constants are subject to the same scope rules as variables
- Constants have memory addresses.

Macros

- `#define` can only be used for numerical, character or string constants.
- Constants created with `#define` aren't subject to the same scoping rules as variables - they apply everywhere.
- Macros don't have addresses.

Important notes:

- The lines
  - `const int *p`
  - `int *const q`

  are very different. The declaration `const int *p` means `p` is a pointer to a constant `int`, so we cannot modify the integer that `p` points to (through `p`).
  For example, the line `p = &i` is okay, as this reassigns the pointer itself, not the `int` it points at. On the other hand, line `*p = 5` will cause an error.
- Continuing on this thought, if we have another pointer `int *r`, then `r = p` will give a warning, whereas `r = (int *)p` will give no warning but is dubious, and in fact, `*r = 5` will execute somewhat, bypassing the intended behaviour.
- The line `int *const q` means we cannot modify the pointer `q`. So, for example, the line `q = p` will cause an error.

Back to String:

As mentioned before, `C` has a library to handle strings, `<string.h>` but it contains relatively basic commands compared to a language like Python.

[IMPORTANT] Consider the following program:

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.   char str1[10] = "abc", str2[10] = "abc";
6.   if (str1 == str2)
7.           printf("Happy !\n");
8.   else
9.           printf("Sad.\n");
10.  return 0;
11.  }
12.
```

Comparing strings using == compares their memory addresses; it does not check if the two strings store the same characters in the same order! Thus, the program above prints **`Sad`**.

We probably don't want this behaviour. Thankfully, "string equality" is one of the functions inside the library.

Let's have a look at the first set of commands that we will investigate:

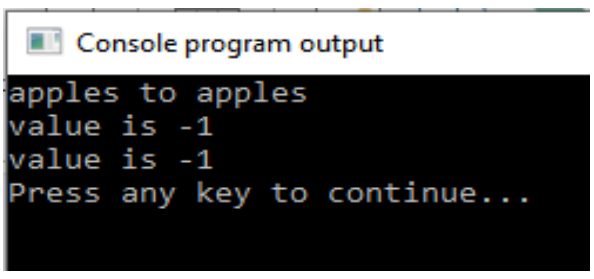
- `size_t strlen(const char *s);`
    - Returns the string length of `s`.
    - Does not include the null character.
    - Here, the keyword `const` means that `strlen` should only read the string and not mutate it.
- `char *strcpy(char *s0, const char *s1);`
    - Copies the string `s1` into `s0` (up to the first null character) and returns `s0`
    - `s0` must have enough room to store the contents of `s1`, but this check is not done inside this function.
    - If there is not enough room, `strcpy` will overwrite bits that follow `s0`, which is undefined behaviour - as in any access past the end of an array.
    - Why return a pointer? It makes it easier to nest the call if needed.
- `char *strncpy(char *s0, const char *s1,size_t n);`
    - Only copies the first `n` characters from `s1` to `s0`.
    - Null padded if `strlen(s1) < n`.
    - No null character is added to end of `s0` if it was not part of `s1`.

- `char *strcat(char *s0, const char *s1);`
  - Concatenates `s1` to `s0` and returns `s0`
  - Does not check if there is enough room in `s0` like `strcpy`.
  - Two strings should not overlap on the same memory! (Undefined behaviour otherwise).
- `char *strncat(char *s0, const char *s1, size_t n);`
  - Only concatenates the first `n` characters from `s1` to `s0`.
  - Adds null character after concatenation.
- `int strcmp(const char *s0, const char *s1);`
  - Compares the two strings, comparing ASCII values.
  - Let `i` be the index of the first character in the two strings that doesn't match, then:
    - `< 0` if `s0[i] < s1[i]` OR all characters up until `i` are equal, but `s1` is longer.
    - `> 0` if `s0[i] > s1[i]` OR all characters up until `i` are equal, but `s0` is longer.
    - `= 0` if `s0` equals `s1`

Example:

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main(void){
5.          char s[100] = "apples";
6.          char t[] = " to monkeys";
7.          char u[100];
8.          strcpy(u, s);
9.         strncat(s, t, 4);
10.         strcat(s, u);
11.         printf("%s\n", s);
12.         int comp = strcmp("abc", "azenew");
13.         if (comp < 0)
14.                 printf("value is %d\n", comp);
15.         comp = strcmp("ZZZ ", "a");    // Hint: Check the  ASCII table
16.         if (comp < 0)
17.                 printf("value is %d\n", comp);
18.  }
```

Do a manual trace and compare your results with the provided output! (on the next page)



gets vs scanf

When trying to read a string from the user using `scanf`, recall that it stops reading characters at any whitespace type character. This might not be the desired effect. You could use the gets function to change this, which stops reading input on a newline character. Both are risky functions as they don't check when the array storing the strings is full. Often, C programmers write their input functions to be safe.

printf for strings

On certain compilers, eg `gcc -std=c11`, the command

```
char *s = "abcj\n"; printf(s);
```

gives a warning that this is not a string literal and has no format arguments. This is a potential security issue if the string itself contains formatting arguments (for example, if it was user-created). You can avoid these errors by making the above string a constant or using `printf("%s",s);` type commands.

More Functions for Strings

[We don't really need to use them in CS137]

- `void *memcpy(void * restrict s1, const void *restrict s2, size_t n);`
  - Copies `n` bytes from `s2` to `s1` must not overlap on the same memory.
  - `restrict` indicates that only this pointer will access that memory area. This allows for compiler optimizations.

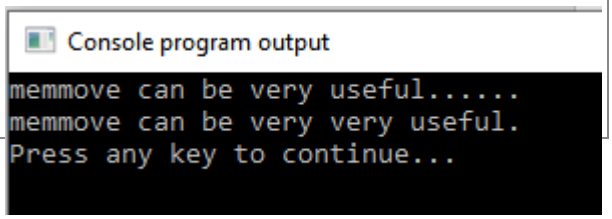
Consider the following program:

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4.
5. int main()
6. {
7.          char s[5]={'s','a','\0','c','h'};
8.
9.          char s1[5];
10.         char s2[5];
11.
12.
13.         strcpy(s1,s);
14.         memcpy(s2,s,5);
15.
16.         for(int i=0; i<5; ++i)
17.                 printf("%c ", s1[i]);
18.         printf("\n");
19.         for(int i=0; i<5; ++i)
20.                 printf("%c ", s2[i]);
21.         printf("\n");
22.         return 0;
23.  }
```

Note: `memcpy()` function is used to copy a specified number of bytes from one memory to another. Whereas the `strcpy()` function is used to copy the contents of one string into another. `memcpy()` function acts on memory rather than value. Whereas the `strcpy()` function acts on value rather than memory. `strcpy` stops when it encounters a `NUL ('\0')` character, `memcpy` does not.

- `void *memmove(void *s1, const void * s2, size_t n);`
    - Similar to `memcpy` but `s1` and `s2` can overlap on same memory.

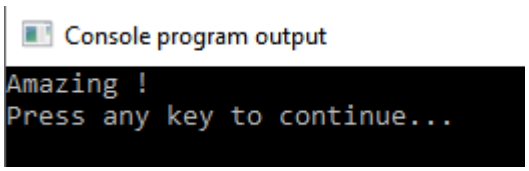
```
1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4.
5. int main ()
6. {
7.          char str[] = "memmove can be very useful......";
8.          printf("%s\n",str);
9.          memmove (str+20,str+15,11);
10.         printf("%s\n",str);
11.         return 0;
12.  }
```



Note: `memcpy()` function is used to copy a specified number of bytes from one memory to another. `memmove()` function is used to copy a specified number of bytes from one memory to another or to overlap on the same memory.
If you try to replace `memmove` with `memcpy` you will not get the same results. (Try it!!!)

- `int memcmp(const void *s1, const void *s2, size_t n);`
    - Similar to `strcmp`, except it compares the bytes of memory.

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main(void)
5. {
6.          char s[10] = "abc";
7.          char t[10] = "abd";
8.          int val = memcmp(s, t, 2);
9.          if (val == 0)
10.                 printf("Amazing !\n");
11.         return 0;
12.  }
```



- `void *memset(void *s, int c, size_t n);`
    - Fills the first `n` bytes of the area with byte `c`. (Note - parameter is `int`, but function will use an `unsigned char` conversion).

```
1. #include <stdio.h>
2. #include <string.h>
3. int main(void)
4. {
5.          char a[50];
6.          memset(a, '$', 5);
7.          a[5] = '\0';
8.          printf("%s\n", a);
9.
10.         char str[50] = "This is sooooooooo fun!!!!!!.";
11.         printf("\nBefore memset(): %s\n", str);
12.
13.         // Fill 9 characters starting from str[10] with '.'
14.         memset(str + 10, '.', 9 * sizeof(char));
15.
16.         printf("After memset():  %s\n", str);
17.
18.         return 0;
19.  }
```

```
Console program output
$$$$$

Before memset(): This is sooooooooo fun!!!!!!.
After memset():  This is so......... fun!!!!!!.
Press any key to continue...
```

Unicode

[*You will not be tested on this section, but it is useful to know*]

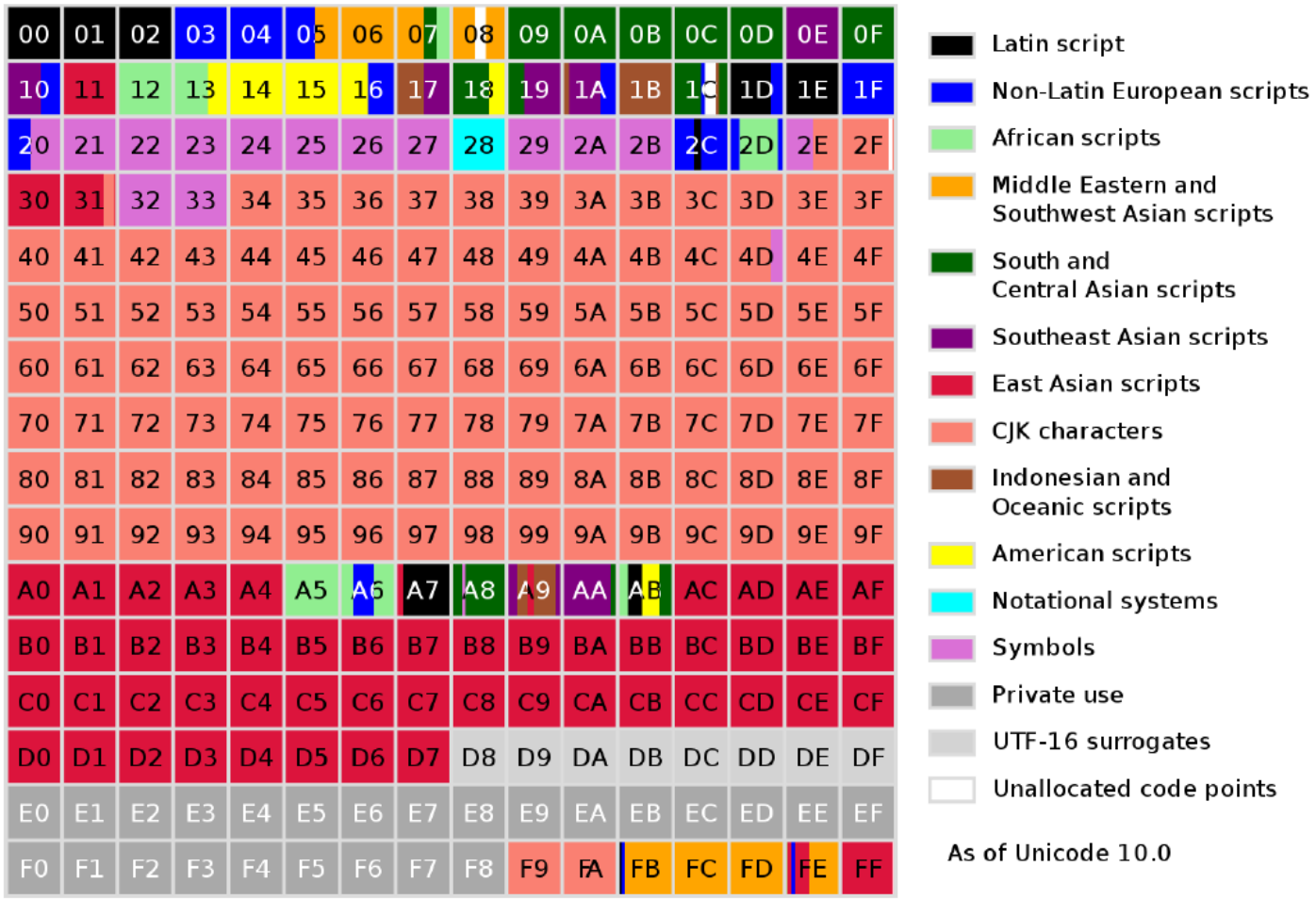
- As exciting as ASCII is, it is far from sufficient to handle all characters over all languages/alphabets.
- Unicode spans more than 100,000 characters over languages both real and fake!
- A Unicode character spans 21 bits and has a range of 0 to 1,114,112 or 3 bytes per character. This last number comes from the 17 planes, which Unicode is divided into and multiplied by the 2^{16} code points (contiguous block).
- Plane 0 is the BMP (Basic Multilingual Plane) - see next page.
- Unicode letters also share the same values as ASCII. This was necessary for adoption by the Western World, which had ASCII first.
- Examples:

UTF+13079

UTF+0061 $(6 \cdot 16 + 1 = 97)$

a

First Plane Basic Multilingual Plane:



- Plane 0 (BMP) consists of characters from U+0000 to U+FFFF
- Plane 1 consists of characters from U+10000 to U+1FFFF
- ... Plane 15 consists of characters from U+F0000 to U+FFFFF
- Plane 16 consists of characters from U+100000 to U+10FFFF

Unicode Encoding:

- The Unicode specification just defines a character code for each letter.
- There are different ways, however, to encode Unicode.
- Popular encodings include UTF-8, UTF-16, UTF-32, UCS-2.
- Different encodings have advantages and disadvantages
- We'll talk about UTF-8, one of the best-supported encodings.

Byte Usage in UTF-8

Code Point Range in Hex	UTF-8 Byte Sequence in Binary
000000-00007F	0xxxxxxx
000080-0007FF	110xxxxx 10xxxxxx
000800-00FFFF	1110xxxx 10xxxxxx 10xxxxxx
010000-10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

For example, let's look at the letter ä which has Unicode 0xE4 or 11100100 in binary. In UTF-8, this falls into range 2 above and so is encoded as 110**00011** 10**100100**. When you concatenate, the bolded text gives you the binary encoding of 0xE4.

The 1 byte characters (i.e. those in range 1) correspond to the ASCII characters (0 to 0x7F = 01111 1111 = 127)

The 2-byte characters are up to 11 bits long with a range 128 to 2^{11} - 1 (i.e. 2047)

The 3-byte characters are up to 16 bits long, with a range 2048 to 2^{16} - 1 (i.e. 65535)

The 4-byte characters are up to 21 bits long, with a range 65536 to 2^{21} - 1 (i.e. 2097151)

In C, a standard library called `<wchar.h>` has code for Unicode.

In fact, ICU (the International Components for Unicode) is more popular among companies such as Adobe, Amazon, Appache, Apple, Google, IBM, Oracle, etc.

For more details, visit http://site.icu-project.org/

We will mainly be dealing with ASCII. However, in an ever-international world, you will need to, at some point, understand Unicode encoding.

Example using <wchar.h> :

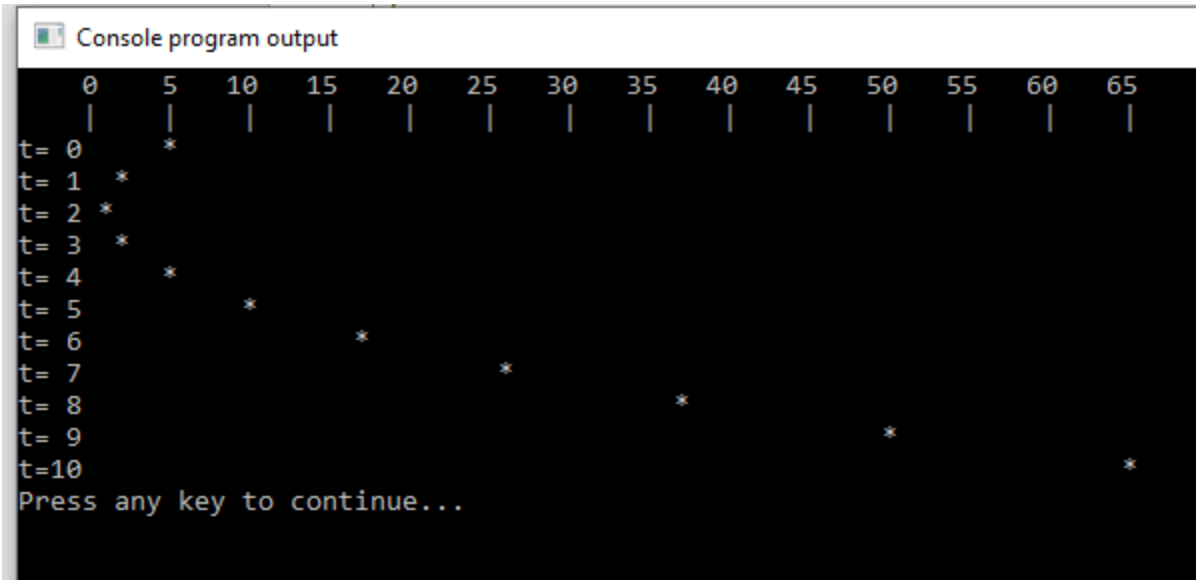
```c
#include <locale.h>
#include <wchar.h>

int main(void)
{
        //L means wchar_t literal vs a normal char.
        wchar_t wc = L'\x3b1';
        setlocale(LC_ALL, "en_US.UTF-8");

        //%lc or %C is wide character
        wprintf(L"%lc\n", wc);

        // Using wprintf once means you need
        // to use it all the time ( undefined
        // behaviour otherwise )
        wprintf(L"%zu\n", sizeof(wchar_t));
        return 0;
}
```

Output:

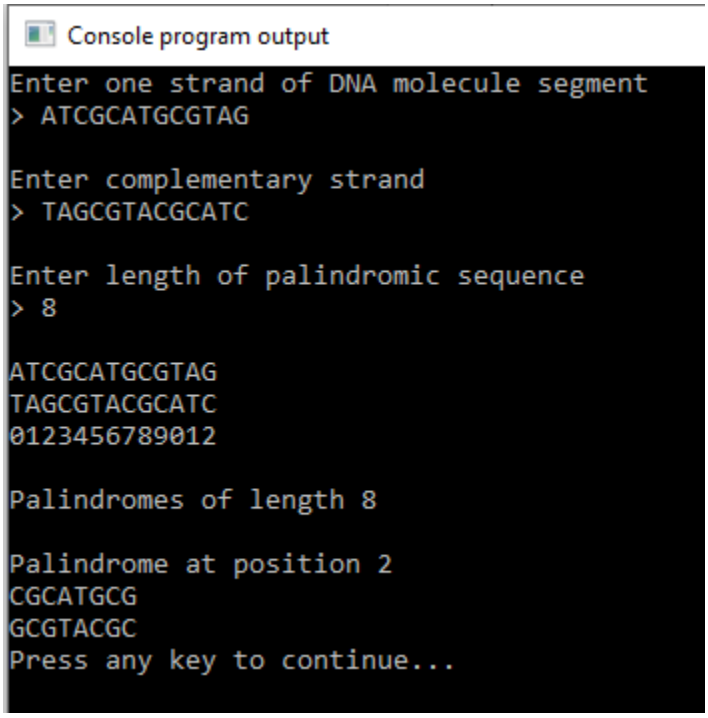
```
α
4
```

Additional Examples

```c
/*  Plots the function  f(t) = t^2  -  4t  +  5  for t between 0 and 10  */

#include <stdio.h>
#define MAX_VAL 65       /*  maximum function value */

int fn(int t);

int main(void)
{
        char plot[MAX_VAL + 2];  /* one line of plot */
        int i, t, funval;

        /* Displays heading lines */
        for (i = 0; i <= MAX_VAL; i += 5)
                printf("%5d", i);
        printf("\n");

        for (i = 0; i <= MAX_VAL; i += 5)
                printf("    |");
        printf("\n");

        /* Initializes plot to all blanks */
        for (i = 0; i <= MAX_VAL + 1; ++i)
                plot[i] = ' ';

        /* Computes and plots f(t) for each value of t from 0 through 10 */
        for (t = 0; t <= 10; ++t)
        {
                funval = fn(t);
                plot[funval] = '*';
                plot[funval + 1] = '\0';
                printf("t=%2d%s\n", t, plot);
                plot[funval] = ' ';
                plot[funval + 1] = ' ';
        }

        return (0);
}

/* f(t) = t^2  -  4t  +  5  */
int fn(int t)
{
        return (t * t - 4 * t + 5);
}
```


Console program output
0 5 10 15 20 25 30 35 40 45 50 55 60 65
t= 0 *
t= 1 *
t= 2 *
t= 3 *
t= 4 *
t= 5 *
t= 6 *
t= 7 *
t= 8 *
t= 9 *
t=10 *
Press any key to continue...

```c
/*
 *  Finds specified-length palindromic sequences of nucleotide pairs in
 *  a portion of a DNA molecule whose complementary strands are
 *  represented as strings
 */

#include <stdio.h>
#include <string.h>

#define STRANDSIZ  100   /*  maximum space for storing a strand  */

int main(void)
{
        char strand1[STRANDSIZ], /* complementary strands */
             strand2[STRANDSIZ]; /*    of DNA */
        int palin_len;   /* length of palindromic sequences
                                        of interest   */
        int i, j, match;

        /* Gets input data and displays reference lines   */
        printf("Enter one strand of DNA molecule segment\n> ");
        scanf("%s", strand1);
        printf("\nEnter complementary strand\n> ");
        scanf("%s", strand2);
        printf("\nEnter length of palindromic sequence\n> ");
        scanf("%d", &palin_len);
        printf("\n%s\n%s\n", strand1, strand2);
        for (i = 0; i < strlen(strand1); ++i)
                printf("%d", i % 10);

        printf("\n\nPalindromes of length %d\n\n", palin_len);

        /* Displays palindromes of interest   */
        for (i = 0; i <= strlen(strand1) - palin_len; ++i)
        {
                match = 1;
                for (j = 1; match && j <= palin_len; ++j)
                        if (strand1[i + j - 1] != strand2[i + palin_len - j])
                                match = 0;

                if (match)
                {
                        printf("Palindrome at position %d\n", i);
                        for (j = i; j < i + palin_len; ++j)
                                printf("%c", strand1[j]);
                        printf("\n");
                        for (j = i; j < i + palin_len; ++j)
                                printf("%c", strand2[j]);
                        printf("\n");
                }
        }
        return (0);
}
```


Console program output
Enter one strand of DNA molecule segment
> ATCGCATGCGTAG
Enter complementary strand
> TAGCGTACGCATC
Enter length of palindromic sequence
> 8
ATCGCATGCGTAG
TAGCGTACGCATC
0123456789012
Palindromes of length 8
Palindrome at position 2
CGCATGCG
GCGTACGC
Press any key to continue...

Extra Practice Problems

1) Complete the following program (That allows you to practice pointer to function) to determine whether a string is a valid password. A valid password must contain:

- at least 8 characters
- at least one upper case letter (e.g., `'P'`)
- at least one lower case letter (e.g., `'s'`)
- at least one digit (e.g., `'6'`)
- at least one special character, which is a non-whitespace printable character that is not mentioned above (e.g., `'$'`).

```
1. #include <assert.h>
2. #include <stdbool.h>
3. #include <stdio.h>
4. #include <string.h>
5.
6. bool is_upper(char c)
7. {
8.   return;
9. }
10.
11.  bool is_lower(char c)
12.  {
13.    return ;
14.  }
15.
16.  bool is_numeric(char c)
17.  {
18.    return ;
19.  }
20.
21.  bool contains_something(bool (*f)(char), const char *s)
22.  {
23.    int n = strlen(s);
24.    for(int i = 0; i < n; ++i)
25.    {
26.      if(f(s[i]))
27.      {
28.        return true;
29.      }
30.    }
31.    return false;
32.  }
33.
34.  bool contains_upper_case_char(const char *s)
35.  {
36.    return ;
37.  }
38.
39.  bool contains_lower_case_char(const char *s)
40.  {
41.    return ;
42.  }
43.
```

```
44.  bool contains_numeric(const char *s)
45.  {
46.    return ;
47.  }
48.
49.  bool contains_special_char(const char *s)
50.  {
51.    int n = strlen(s);
52.
53.  }
54.
55.  bool is_valid_password(const char *s)
56.  {
57.
58.  }
59.
60.  int main(void)
61.  {
62.    assert(is_valid_password("Pass$No1"));
63.    assert(is_valid_password("PassNo1*"));
64.    assert(!is_valid_password("Pas$tNo"));
65.    assert(!is_valid_password("Pas$Nooooooo"));
66.    assert(is_valid_password("Pas$No1!!"));
67.    assert(!is_valid_password("PasswordNo1"));
68.    assert(!is_valid_password("pass$no1"));
69.    assert(!is_valid_password("PAS$WORDNO1"));
70.  }
71.
```

2)

a) Write a function `void remove_white_spaces(char *s)`, which mutates the string `s` by removing all the white space characters. (be careful not to lose the null character at the end).

b) Write a function `void remove_chars(char *s, const char chars[], int len)`, which mutates the string `s` by removing all the characters in the array `chars` from the string `s`.

Here is a sample test

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <assert.h>
4.
5. void remove_char(char *s, char c)
6. {
7.
8. }
9.
10.  void remove_chars(char *s, const char chars[], int len)
11.  {
12.
13.  }
14.
15.  int main(void)
16.  {
17.    char ans1[] = "Ni ce Wor k !";
18.    remove_char(ans1, ' ');
19.    assert(strcmp(ans1, "NiceWork!") == 0);
20.
21.    char ans2[] = "C*S**1*37**";
22.    remove_char(ans2,'*');
23.    assert(strcmp(ans2, "CS137") == 0);
24.
25.    char ans3[] = "aPwer fe ctx!";
26.    char remove[] = "!wa x";
27.    remove_chars(ans3, remove, 5);
28.    assert(strcmp(ans3,"Perfect")==0);
29.
30.    return 0;
31.  }
32.
```

3) Write a function `bool allWordsSameLetters(const char *str)` that returns `true` if all the words in the string `str` consist of the same letters. (A solution[i] is provided; try to solve it first before looking)

Notes: two words are separated by at least one space. Ignore repetition of letters in the same word; letters are not case-sensitive.

Here is a sample test program:

```
1. #include <stdio.h>
2. #include <stdbool.h>
3. #include <assert.h>
4.
5. #define len  'z'-'a'+1
6.
7. int main(void)
8. {
9.          assert(allWordsSameLetters("     "));
10.         assert(allWordsSameLetters("abc bBca acb"));
11.         assert(allWordsSameLetters("Abc bca    Acb"));
12.         assert(!allWordsSameLetters("abcd bBca acb"));
13.         assert(!allWordsSameLetters("ab bBca acb"));
14.         return 0;
15.  }
16.
```

4) Write a function `bool isAnagram(const char*s, const char *t)` that returns `trues` if `t` is an anagram of `s`. The function should ignore any character that is not a letter. Assume that all letters are lowercase letters. (**solution[ii] is provided, try to solve it first before looking**)

Here is a sample test program:

```
1. #include <stdio.h>
2. #include <stdbool.h>
3. #include <assert.h>
4.
5. #define len  'z'-'a'+1
6.
7. int main(void)
8. {
9.          assert(isAnagram("a gentleman", "elegant man"));
10.         assert(!isAnagram("a gentleman", "elegant men"));
11.         assert(isAnagram("election results", "lies - let's recount"));
12.         assert(!isAnagram("election results", "lies - let recount"));
13.         assert(isAnagram("the hilton", "hint: hotel"));
14.         assert(!isAnagram("the hilton", "hint: a hotel"));
15.         return 0;
16.  }
17.
```

5) Consider the following function:

```
void fun_3(char *str, int n, char c)
{
    char *p;
    int i;

    for (p = str; *p != c && *p != '\0'; p++);
    if (*p == '\0')
        printf ("there is no %c in \"%s\"", c, str);
    else
        for (i = 0; *(p+i) != '\0' && i<n; i++)
            printf ("%c", *(p+i));
    printf ("\n");









}
```

a) What is the output for `fun_3("Welcome to the Jungle", 4, 'J')`?

b) What is the output for `fun_3("Patience", 2, 'u')`?

c) What is the purpose of function `fun_3`?

6) Implement the functions `mirror` and `unmirror`. The function `mirror` takes a string and appends a mirrored version of it. For example, `mirror("Test!")` would yield `"Test!!tseT"`. The function `unmirror` takes a previously mirrored string and removes the mirrored part from it. For example, `unmirror("Test!!tseT")` would yield `"Test!"`.

For mirror, you may assume that the array passed as parameter `char *str` will be large enough to hold the mirrored version of the input string.

For `unmirror`, you may assume that the parameter `string char *str` is appropriately mirrored.

7) Implement a search algorithm to find words in a square matrix of letters. Your program takes in a matrix, a string, and the dimensions of the matrix and prints out 1 if the string is found, 0 otherwise. Words can only be formed horizontally along the row of the matrix or vertically along the column of the matrix, not diagonally or backward.
E.g.:

```
char matrix[5][5]=[['h','e','l','l','o'],
                   ['o','z','b','w','i'],
                   ['w','h','i','y','l'],
                   ['d','f','g','h','j'],
                   ['y','v','n','m','k']];
```

Input: hello
Output: 1
Input: howdy
Output: 1
Input: sup
Output: 0

8) A food pyramid represents food in such a way that the food that needs to be consumed in more amounts is at the pyramid's base, and the least is at the pyramid's apex. Create a program that takes in the total number of foods and data on different foods from user input and arranges the food such that healthy food comes first in ascending order and junk food comes next in descending order.
E.g.:
Input:
9
soda 550 junk
alcohol 700 junk
chocolate 300 junk
candy 250 junk
fruits 100 healthy
cheese 250 healthy
veggies 25 healthy
water 0 healthy
nuts 250 healthy
Output:
Food: alcohol, Calories: 700, Category: junk
Food: soda, Calories: 550, Category: junk
Food: chocolate, Calories: 300, Category: junk
Food: candy, Calories: 250, Category: junk
Food: water, Calories: 0, Category: healthy
Food: veggies, Calories: 25, Category: healthy
Food: fruits, Calories: 100, Category: healthy
Food: cheese, Calories: 250, Category: healthy
Food: nuts, Calories: 250, Category: healthy

9) You are tasked with building a memory-efficient system to organize a dynamic collection of books in a library. A structure with the following fields represents each book:

- `title` (string): Title of the book.
- `author` (string): Author's name.
- `year` (integer): Year of publication.
- `copies` (integer): Number of copies available.

Your task is to implement a C program `library_manager.c` that:

Dynamically creates a structure array for n books, where the user enters n.

Allows the user to input book details (title, author, year, copies).

Implements the following functionalities using functions:

Search: Search for books by title (case-insensitive) and display their details.

Sort: Sort the books in ascending order according to the publication year.

Update: Update the number of copies for a specific book.

Ensures proper memory management by dynamically resizing the array when books are added or removed.

Frees all allocated memory before program termination.

Complete the following program to meet the above requirements.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define the structure for books
typedef struct {
    char title[100];
    char author[100];
    int year;
    int copies;
} Book;

Book* addBook(Book* books, int* size) {}

// Function to search for a book by title
void searchBook(Book* books, int size, const char* title) {}

// Function to sort books by publication year
void sortBooks(Book* books, int size) {}

// Function to update the number of copies for a book
void updateCopies(Book* books, int size, const char* title) {}

// Function to display all books in the collection
void displayBooks(Book* books, int size) {
    if (size == 0) {
        printf("No books in the library.\n");
        return;
```

```
    }

    printf("\nLibrary Collection:\n");
    for (int i = 0; i < size; i++) {
        printf("Book %d:\n", i + 1);
        printf("  Title: %s\n", books[i].title);
        printf("  Author: %s\n", books[i].author);
        printf("  Year: %d\n", books[i].year);
        printf("  Copies: %d\n", books[i].copies);
    }
}

int main(void){
    Book *lib = NULL;
    int n = 0;

    lib = addBook(lib, &n);
    lib = addBook(lib, &n);
    lib = addBook(lib, &n);
    searchBook(lib, n, "b2");
    displayBooks(lib, n);
    sortBooks(lib,n);
    displayBooks(lib, n);
    updateCopies(lib,n,"b3");
    displayBooks(lib, n);
}
```

Sample execution:

Enter the title of the book: b1
Enter the author of the book: a1
Enter the publication year: 2008
Enter the number of copies: 1
Book added successfully!
Enter the title of the book: b2
Enter the author of the book: a2
Enter the publication year: 2006
Enter the number of copies: 3
Book added successfully!
Enter the title of the book: b3
Enter the author of the book: a3
Enter the publication year: 2010
Enter the number of copies: 9
Book added successfully!
Book Found:
Title: b2
Author: a2
Year: 2006
Copies: 3

Library Collection:
Book 1:

Title: b1
Author: a1
Year: 2008
Copies: 1
Book 2:
Title: b2
Author: a2
Year: 2006
Copies: 3
Book 3:
Title: b3
Author: a3
Year: 2010
Copies: 9

Library Collection:
Book 1:
Title: b2
Author: a2
Year: 2006
Copies: 3
Book 2:
Title: b1
Author: a1
Year: 2008
Copies: 1
Book 3:
Title: b3
Author: a3
Year: 2010
Copies: 9
Enter the new number of copies: 10
Number of copies updated successfully!

Library Collection:
Book 1:
Title: b2
Author: a2
Year: 2006
Copies: 3
Book 2:
Title: b1
Author: a1
Year: 2008
Copies: 1
Book 3:
Title: b3
Author: a3
Year: 2010
Copies: 10

10) Write a function `void findReindeer(char* word)` that looks for the word "REINDEER" within a given string. If the word is found, the function should print its position in the string. If the word is not found, it should print an appropriate message. The function should use case-insensitive string comparison. If multiple occurrences of the word occur, the function should print all matching positions.

The sample code for testing is below.

```
int main(void) {
    char matching[] = "The REINdeer are ready for the holiday season!";
    findReindeer(matching);  // Expected output: Found REINDEER at position 4

    char notMatching[] = "It's snowing in this deer village!";
    findReindeer(notMatching);  // Expected output: REINDEER not found

    char multipleMatching[] = "Reindeers are a funny kind of reinDeer!";
    findReindeer(multipleMatching);  // Expected output: Found REINDEER at
position 0, 30

    return 0;
}
```

11) For the new year, a student was given a new planner to track their dates. However, they already wrote down all their events and plans in another date format.
Write a function `char* updateOldDate(char* oldDate)` that takes a string representing a date in the format "YYYYMMDD" and formats it into a more readable format "Month Day, Year". You cannot assume that the input is always valid, and you should print an error message if the date is invalid or in the wrong format. Make sure to handle the conversion of the month and day correctly.

The sample code for testing is below.

```
int main(void) {
    char validDate[] = "20231221";
    char *formattedValidDate = updateOldDate(validDate);
    assert(strcmp(formattedValidDate, "December 21, 2023") == 0);
    free(formattedValidDate);

    char invalidDate[] = "20231321";
    char *formattedInvalidDate = updateOldDate(invalidDate);
    assert(strcmp(formattedInvalidDate, "Invalid Date") == 0);
    free(formattedInvalidDate);

    char notDateNumbers[] = "19201010101911";
    char *formattedNotDateNumbers = updateOldDate(notDateNumbers);
    assert(strcmp(formattedNotDateNumbers, "Invalid Date") == 0);
    free(formattedNotDateNumbers);

    char notDateString[] = "kaejkaj;aea";
    char *formattedNotDateString = updateOldDate(notDateString);
    assert(strcmp(formattedNotDateString, "Invalid Date") == 0);
    free(formattedNotDateString);

    return 0;
}
```

12) People often ask the shopping center gift wrapping booth to wrap their presents. To remember which present should be returned to which customer, an encoded gift tag is added to each present, and the decoded version is given to the customer. The tag always contains a character from A-Z for the first letter, followed by a series of digits representing a shift pattern. Each letter in the tag (after the first) should then be shifted forward by the number of positions indicated by the digits. The shift will wrap around the alphabet, so 'Z' shifted by 1 would become 'A'. Assume all inputs are valid, and that case matters.

For example, if the gift tag is `"A3B4b"`, the encoding rule works as follows:

- The first character `'A'` is unshifted.
- The second character `'B'` is shifted forward by 3 positions to `'E'`.
- The third character `'b'` is shifted forward by 4 positions to `'f'`.
- As such, the decoded version would be `"AEf"`.

It can be time-consuming to decode each gift tag during busy times manually. As such, implement a C program, `gift_tag.c`, that reads the input tag and returns the decoded version. Assume that the input string has at most 185 characters.

Sample input 1:
Enter the encoded gift tag (max 185 characters): A3B4b
Expected output 1:
Decoded gift tag: AEf

Sample input 2:
Enter the encoded gift tag (max 185 characters): R1c
Expected output 2:
Decoded gift tag: Rd

13) Write a function that takes a string (s1) allocated on the heap and returns a new string similar to s1 but reversed.

14) Write a function `fname` that takes a string (`s1`) allocated on the heap and integer `n` and returns a new string similar to `s1` repeated `n` times `fname("abc", 3)` => `"abcabcabc"`

15) Write a function that takes two strings `s1` and `s2`, returns `true` if `s1` is a substring of `s2`, otherwise `false`.

16) Implement the following function:
// returns the length of the longest palindrome in st between index start and index end (inclusive for both)

```
int longestpalindrome(char* st, int start, int end);
int main(){
    assert(6==longestpalindrome("111111",0,5));
    assert(7==longestpalindrome("abbbcxyzazyxbba",0,14));
    assert(1==longestpalindrome("a",0,0));
    assert(6==longestpalindrome("aaa77777jbbbbbboh666aaa",0,23));
    assert(6==longestpalindrome("abccbacba",0,8));
    return 0;
}
```

Answers

---

i

```
char getSmallLetter(char ch)
{
    if (ch >= 'A' && ch <= 'Z')
        ch += 'a' - 'A';
    return ch;
}
bool allWordsSameLetters(const char *str)
{

    int first[len] = { 0 };
    int others[len];
    int i;
    //skip spaces
    while (*str != '\0' && *str == ' ')
        str++;
    if (*str == '\0')
        return true;
    //init array for first word
    while (*str != '\0' && *str != ' ')
    {
        first[getSmallLetter(*str) - 'a'] = 1;
        str++;
    }
    while (*str != '\0')
    {
        while (*str == ' ')
            str++;
        for (i = 0; i < len; i++)
            others[i] = 0;
        while (*str != '\0' && *str != ' ')
        {
            others[getSmallLetter(*str) - 'a'] = 1;
            str++;
        }
        for (i = 0; i < len && *str; i++)
        {
            if (first[i] != others[i])
                return false;
        }
    }
    return true;
}
```

ii

```
void countLetters(const char *s, int letters[])
{
    while (*s != '\0')
    {
        if (*s >= 'a' && *s <= 'z')
            letters[*s - 'a'] ++;
        s++;
    }
}

bool isAnagram(const char *s, const char *t)
{
    int s_letters[len], t_letters[len];
    int i;
    for (i = 0; i < len; i++)
        s_letters[i] = t_letters[i] = 0;
    countLetters(s, s_letters);
    countLetters(t, t_letters);
    for (i = 0; i < len; i++)
        if (s_letters[i] != t_letters[i])
            return false;
    return true;
}
```