CS 137

Programming Principles

Chapter 10

Characters and Strings

Victoria Sakhnini

Table of Contents

ASCII & Characters 2
Strings 4
Functions for Strings 6
gets vs scanf 8
printf for Strings 8
More Functions for Strings 9
Unicode 11
Additional Examples 13
Extra Practice Problems 16

ASCII & Characters

We have already seen this briefly earlier in the term.

Syntax: `char c;`

A `char` is an 8-bit integer. The integer can be a code representing printable and unprintable characters, or it can store a single letter, e.g., `char c = 'a';`.

Example 1

```c
#include <stdio.h>

int main(void)
{
    char c1 = 'a';
    char c2 = 97;
    printf("%c\n", c1);  // output: a
    printf("%d\n", c1);  // output: 97
    printf("%d\n", c2);  // output: 97
    printf("%c\n", c2);  // output: a
    return 0;
}
```

Example 2

```c
#include <stdio.h>

int main(void)
{
    char c1 = 'A';
    char c2 = 65;
    if (c1 == c2)  // Output: wow
        printf("wow\n");
    else
        printf("hmmmm\n");
    return 0;
}
```

📌 ASCII (American Standard Code for Information Interchange) codes represent text in computers and other devices. ASCII uses 7 bits (range 0000000–1111111), with the 8th bit used for a parity check or extended ASCII. Characters 0–31: control characters Characters 48–57: digits '0' to '9' Characters 65–90: uppercase letters 'A' to 'Z' Characters 97–122: lowercase letters 'a' to 'z' Note: 'A' and 'a' differ by exactly 32.

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

Example 3 — int vs char

```
#include <stdio.h>

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

Example 4 — Basic Calculator Using char

```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;
}
```

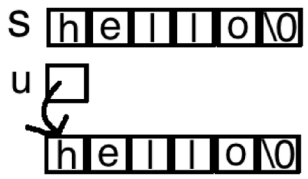
For the English language, ASCII is enough for most applications. However, many languages have far more complex letter systems. Unicode was developed to handle them — discussed later in this chapter.

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

    // Equivalent to the above:
    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;
}
```

The last form, `char *u = "Hello";`, is slightly different — it is a **pointer to a string literal**. This matters more than it might seem:




```c
#include <stdio.h>

int main(void)
{
    char s[] = "Hello";
    s[1] = 'a';                     // OK: s is an array — mutable
    printf("s=%s\n", s);            // Output: s=Hallo
    printf("sizeof s is %zu\n", sizeof(s));  // 6 (6-char array, 1 byte each)

    char *u = "Hello";
    // u[1] = 'a';  // ERROR! String literals are read-only.
    printf("u=%s\n", u);            // Output: u=Hello
    printf("sizeof u is %zu\n", sizeof(u));  // 8 (size of a pointer)

    char *hi = "Hello" " world!";   // Adjacent string literals are joined
    printf("hi=%s\n", hi);          // Output: hi=Hello world!
    printf("sizeof hi is %zu\n", sizeof(hi));  // 8 (pointer)
    return 0;
}
```

⚠ In char *u = "Hello", the string "Hello" is a string literal stored in read-only memory. Attempting to modify it (e.g., u[1] = 'a') causes undefined behaviour. Also note: sizeof(s) differs from sizeof(u) because s is an array and u is a pointer.

💡 Adjacent string literals are automatically concatenated by the compiler: char *hi = "Hello" " world!"; is the same as char *hi = "Hello world!";

Processing a String Character by Character

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

Functions for Strings

String manipulations in C are tedious and cumbersome. There is a library, `<string.h>`, that helps with the basics. Before discussing it, we need a brief digression into the `const` type qualifier.

const Type Qualifier

- The keyword `const` indicates that something is **not modifiable** — it is read-only.
- Assignment to a `const` piece of data causes a compiler error.
- It is helpful for communicating the intent of a variable to other programmers.

```
const int i = 10;  // i is a constant initialized to 10
i = 5;             // ERROR: cannot assign to a constant
```

Even though a value is `const`, through poor programming you could still change it via a pointer — but doing so is **undefined behaviour**:

```
#include <stdio.h>

int main(void) {
    const int i = 10;
    printf("%d\n", i);
    int *a = &i;  // Warning! Incompatible types: 'int *' vs 'const int *'
    *a = 3;       // Undefined behaviour
    printf("%d\n", i);
    return 0;
}
```

const with Pointers

Important: `const int *p` and `int *const q` are very different:

- `const int *p` — p is a pointer to a **constant int**. You cannot modify the integer through p. So `p = &i` is fine (reassigning the pointer), but `*p = 5` is an error.
- `int *const q` — q is a **constant pointer** to an int. You cannot reassign q itself (`q = p` is an error), but you can modify what q points at.

⚠ If you have int *r and you write r = p (where p is const int *), you get a warning. r = (int *)p will suppress the warning but is dubious — *r = 5 will bypass the intended const restriction.

const vs #define

Feature	const	#define
Type checking	Yes (compiler-enforced)	No (text substitution)
Scope	Follows normal scoping rules	Global (no scoping)
Debugger visibility	Yes	No
Can take address of	Yes	No
Can be used for arrays	Yes	Yes

String Equality Pitfall

⚠ Comparing strings with == compares their MEMORY ADDRESSES, not their contents! The following program prints "Sad." because str1 and str2 are stored at different addresses: char str1[10] = "abc", str2[10] = "abc"; if (str1 == str2) // compares addresses, not content! printf("Happy !\n"); else printf("Sad.\n"); // this is printed Use strcmp() for content comparison.

```
#include <stdio.h>

int main(void) {
    char str1[10] = "abc", str2[10] = "abc";
             if (str1 == str2)
                    printf("Happy !\n");
             else
                    printf("Sad.\n");
       return 0;
}
```

String Functions — strlen, strcpy, strncpy, strcat, strncat, strcmp

- `size_t strlen(const char *s)` — returns the length of s, not counting the null character. The `const` means strlen only reads the string and does not mutate it.

- `char *strcpy(char *s0, const char *s1)` — copies string s1 into s0 (up to and including the null character) and returns s0. s0 must have enough room; this is **not** checked internally. Overwriting beyond the end is undefined behaviour.

- `char *strncpy(char *s0, const char *s1, size_t n)` — copies at most n characters from s1 to s0. Null-padded if `strlen(s1) < n`. If s1 is longer than n, **no null character is appended** to s0.

- `char *strcat(char *s0, const char *s1)` — concatenates s1 onto the end of s0 and returns s0. Does not check for sufficient room. The two strings must not overlap in memory.

- `char *strncat(char *s0, const char *s1, size_t n)` — concatenates at most n characters from s1 onto s0. Always appends a null character after the concatenation.

- `int strcmp(const char *s0, const char *s1)` — compares the two strings character by character using ASCII values. Let i be the index of the first mismatch:

  Returns < 0 if s0[i] < s1[i], or if all matched characters are equal but s1 is longer.
  Returns > 0 if s0[i] > s1[i], or if all matched characters are equal but s0 is longer.
  Returns 0 if s0 and s1 are identical.

String Functions Example

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

int main(void) {
    char s[100] = "apples";
    char t[]    = " to monkeys";
    char u[100];
    strcpy(u, s);           // u = "apples"
    strncat(s, t, 4);       // s = "apples to"  (appends first 4 chars of t)
    strcat(s, u);           // s = "apples toapples"
    printf("%s\n", s);

    int comp = strcmp("abc", "azenew");
    if (comp < 0)
        printf("value is %d\n", comp);

    comp = strcmp("ZZZ ", "a");  // Hint: check the ASCII table
    if (comp < 0)
        printf("value is %d\n", comp);
}
```

Do a manual trace of the above program and verify your results before running it.

gets vs scanf

When reading a string with `scanf`, it stops at any whitespace character. This is not always desired. The function `gets` reads until a newline instead. Both functions are risky because they do not check whether the destination array is large enough. In practice, C programmers often write their own safe input functions.

printf for Strings

On certain compilers (e.g., `gcc -std=c11`), the code:

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

gives a warning that `s` is not a string literal and has no format arguments. This is a potential security issue if the string contains formatting characters (e.g., if the string was user-supplied). Avoid this by using:

```
printf("%s", s);  // safe form
```

More Functions for Strings

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

memcpy

- `void *memcpy(void *restrict s1, const void *restrict s2, size_t n)` — copies n bytes from s2 to s1. s1 and s2 must **not overlap**. The keyword `restrict` tells the compiler that only this pointer will access that memory region, enabling optimisations.

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

int main()
{
    char s[5] = {'s', 'a', '\0', 'c', 'h'};
    char s1[5];
    char s2[5];

    strcpy(s1, s);     // copies until '\0' — s1 gets "sa"
    memcpy(s2, s, 5);  // copies all 5 bytes, including '\0' mid-array

    for (int i = 0; i < 5; ++i)
        printf("%c ", s1[i]);
    printf("\n");

    for (int i = 0; i < 5; ++i)
        printf("%c ", s2[i]);
    printf("\n");
    return 0;
}
```

📌 Key difference between strcpy and memcpy: strcpy — stops at '\0'; acts on value. memcpy — copies exactly n bytes regardless of '\0'; acts on raw memory.

memmove

- `void *memmove(void *s1, const void *s2, size_t n)` — similar to memcpy, but s1 and s2 **may overlap**. Use memmove whenever the source and destination regions might overlap.

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

int main()
{
    char str[] = "memmove can be very useful.....";
    printf("%s\n", str);
    memmove(str + 20, str + 15, 11);
    printf("%s\n", str);
    return 0;
}
```

If you replace memmove with memcpy in the above example, you will not get the same results. Try it!

memcmp

- `int memcmp(const void *s1, const void *s2, size_t n)` — similar to strcmp, but compares n raw bytes of memory rather than null-terminated strings.

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

int main(void)
{
    char s[10] = "abc";
    char t[10] = "abd";
    int val = memcmp(s, t, 2);  // compare first 2 bytes: 'a','b' vs 'a','b'
    if (val == 0)
        printf("Amazing !\n");
    return 0;
}
```

memset

- `void *memset(void *s, int c, size_t n)` — fills the first n bytes of the memory area pointed to by s with byte value c. Note: the parameter is `int`, but the function uses an `unsigned char` conversion internally.

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

int main(void)
{
    char a[50];
    memset(a, '$', 5);
    a[5] = '\0';
```

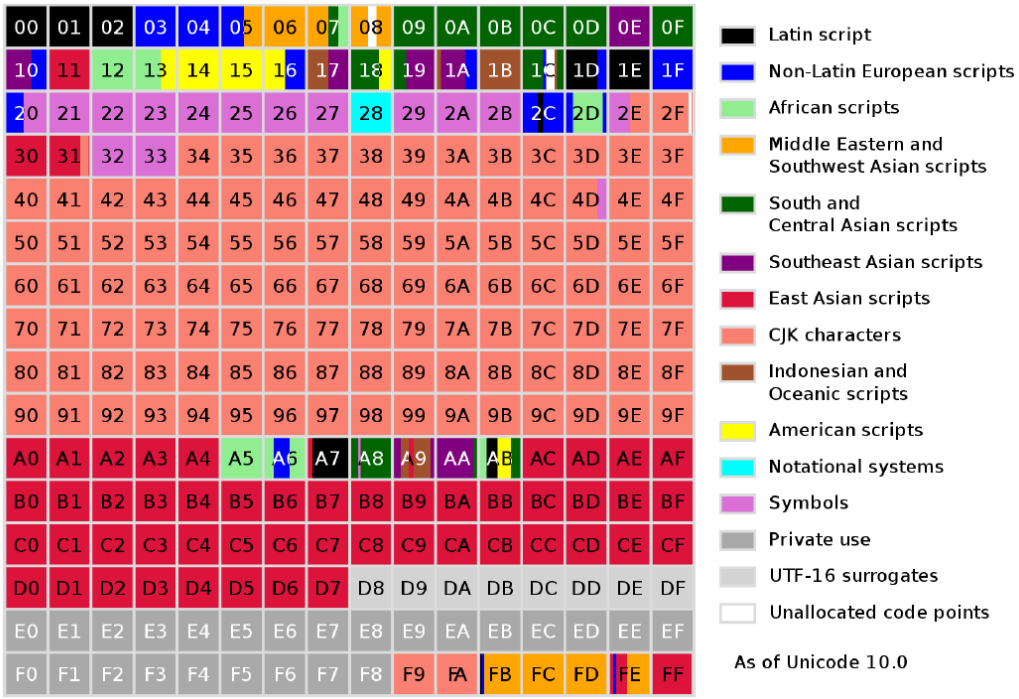
```
    printf("%s\n", a);  // $$$$$

    char str[50] = "This is sooooooooooo fun!!!!!!.";
    printf("\nBefore memset(): %s\n", str);
    memset(str + 10, '.', 9 * sizeof(char));  // fill 9 chars from str[10]
with '.'
    printf("After memset():  %s\n", str);
    return 0;
}
```

Unicode

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

- ASCII is far from sufficient to represent all characters across all languages and alphabets.
- Unicode spans more than 100,000 characters across real and constructed languages.
- A Unicode character spans **21 bits**, with a range of 0 to 1,114,112 (about 3 bytes per character). This comes from 17 planes × 2^{16} code points.
- Plane 0 is the **BMP (Basic Multilingual Plane)**.
- Unicode letters share the same values as ASCII for backward compatibility with the Western world.

Unicode Planes



Plane	Range
Plane 0 (BMP)	U+0000 to U+FFFF
Plane 1	U+10000 to U+1FFFF
...	...
Plane 15	U+F0000 to U+FFFFF
Plane 16	U+100000 to U+10FFFF

Unicode Encoding

- The Unicode specification defines a character code for each letter.
- There are multiple ways to encode Unicode: **UTF-8**, UTF-16, UTF-32, UCS-2.
- UTF-8 is one of the best-supported encodings and is the most popular on the web.

Byte Usage in UTF-8

Range	Bytes used	Bit pattern
U+0000 to U+007F (ASCII)	1 byte	0xxxxxxx
U+0080 to U+07FF	2 bytes	110xxxxx 10xxxxxx
U+0800 to U+FFFF	3 bytes	1110xxxx 10xxxxxx 10xxxxxx
U+10000 to U+10FFFF	4 bytes	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

📌 UTF-8 range summary: 1-byte characters: ASCII (0 to 0x7F = 127) 2-byte characters: up to 11 bits, range 128 to 2047 3-byte characters: up to 16 bits, range 2048 to 65535 4-byte characters: up to 21 bits, range 65536 to 2097151 In C, the standard library <wchar.h> provides Unicode support. The ICU (International Components for Unicode) library is popular among major companies (Adobe, Amazon, Apple, Google, IBM, etc.). See: http://site.icu-project.org/

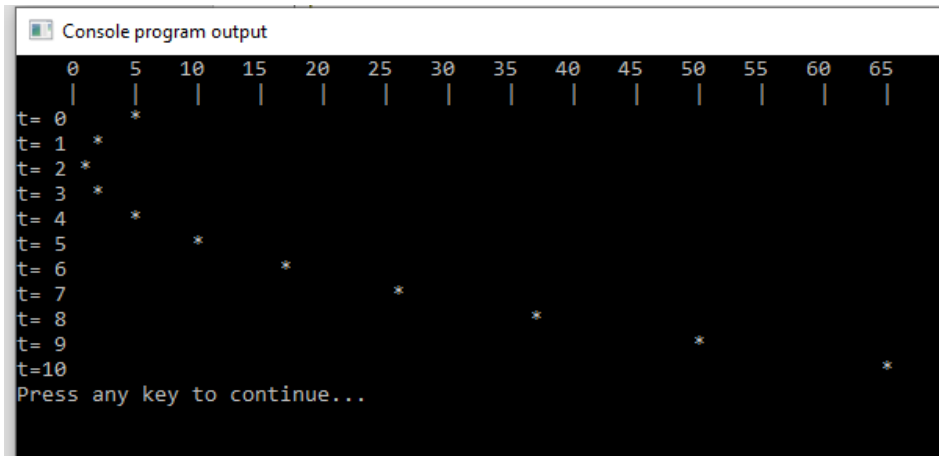
Example Using <wchar.h>

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

int main(void)
{
    // L prefix means wchar_t literal (vs normal char)
    wchar_t wc = L'\x3b1';          // Unicode code point U+03B1 (Greek alpha:
α)
    setlocale(LC_ALL, "en_US.UTF-8");
    // %lc or %C is used for wide characters
    wprintf(L"%lc\n", wc);
    // Once wprintf is used, use it consistently
    // (mixing with printf is undefined behaviour)
    wprintf(L"%zu\n", sizeof(wchar_t));
    return 0;
}
```

Additional Examples

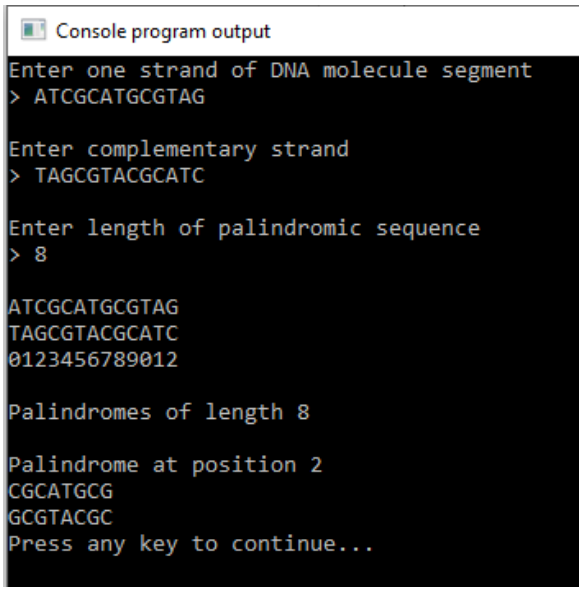
Plotting f(t) = t² – 4t + 5

```
#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;

    /* Display 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");

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

    /* Compute and plot f(t) for each 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);
}
```



DNA Palindrome Finder

```
#include <stdio.h>
#include <string.h>
#define STRANDSIZ 100

int main(void)
{
    char strand1[STRANDSIZ], strand2[STRANDSIZ];
    int palin_len;
    int i, j, match;

    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);

    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 program to determine whether a string is a valid password.
A valid password must contain:
- at least 8 characters
- at least one uppercase letter (e.g., 'P')
- at least one lowercase letter (e.g., 's')
- at least one digit (e.g., '6')
- at least one special character (a non-whitespace printable character that is none of the above, e.g., '$')

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

bool is_upper(char c)   { return; }
bool is_lower(char c)   { return; }
bool is_numeric(char c) { return; }

bool contains_something(bool (*f)(char), const char *s)
{
    int n = strlen(s);
    for (int i = 0; i < n; ++i)
        if (f(s[i])) return true;
    return false;
}

bool contains_upper_case_char(const char *s) { return; }
bool contains_lower_case_char(const char *s) { return; }
bool contains_numeric(const char *s)         { return; }

bool contains_special_char(const char *s)
{
    int n = strlen(s);
    // complete this
}

bool is_valid_password(const char *s)
{
    // complete this
}

int main(void)
{
    assert( is_valid_password("Pass$No1"));
    assert( is_valid_password("PassNo1*"));
    assert(!is_valid_password("Pas$tNo"));
    assert(!is_valid_password("Pas$Nooooooo"));
    assert( is_valid_password("Pas$No1!!"));
    assert(!is_valid_password("PasswordNo1"));
    assert(!is_valid_password("pass$no1"));
    assert(!is_valid_password("PAS$WORDNO1"));
}
```

2.

a) Write void remove_char(char *s, char c) that mutates s by removing all occurrences of character c. Be careful not to lose the null terminator.

b) Write void remove_chars(char *s, const char chars[], int len) that removes all characters in the array chars from string s.

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

void remove_char(char *s, char c) { }

void remove_chars(char *s, const char chars[], int len) { }

int main(void)
{
    char ans1[] = "Ni ce Wor k !";
    remove_char(ans1, ' ');
    assert(strcmp(ans1, "NiceWork!") == 0);
    char ans2[] = "C*S**1*37**";
    remove_char(ans2, '*');
    assert(strcmp(ans2, "CS137") == 0);
    char ans3[]   = "aPwer fe ctx!";
    char remove[] = "!wa x";
    remove_chars(ans3, remove, 5);
    assert(strcmp(ans3, "Perfect") == 0);

    return 0;
}
```

3.

Write bool allWordsSameLetters(const char *str) that returns true if all words in str consist of the same set of letters.

Notes: words are separated by at least one space; ignore repeated letters in the same word; letters are not case-sensitive.
A solution is provided — try to solve it yourself first.

```
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#define len ('z' - 'a' + 1)

int main(void)
{
    assert( allWordsSameLetters(" "));
    assert( allWordsSameLetters("abc bBca acb"));
    assert( allWordsSameLetters("Abc bca Acb"));
    assert(!allWordsSameLetters("abcd bBca acb"));
    assert(!allWordsSameLetters("ab bBca acb"));
    return 0;
}
```

4.

Write bool isAnagram(const char *s, const char *t) that returns true if t is an anagram of s.
The function should ignore any character that is not a letter. Assume all letters are lowercase.

```
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#define len ('z' - 'a' + 1)

int main(void)
{
    assert( isAnagram("a gentleman",      "elegant man"));
    assert(!isAnagram("a gentleman",      "elegant men"));
    assert( isAnagram("election results", "lies - let's recount"));
    assert(!isAnagram("election results", "lies - let recount"));
    assert( isAnagram("the hilton",       "hint: hotel"));
    assert(!isAnagram("the hilton",       "hint: a hotel"));
    return 0;
}
```

5.

Consider a function fun_3

```
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 mirror and unmirror.

mirror(str) — appends a reversed copy of str to itself.
Example: mirror("Test!") => "Test!!tseT"

unmirror(str) — removes the mirrored portion from a previously mirrored string.
Example: unmirror("Test!!tseT") => "Test!"

For mirror, assume the array is large enough. For unmirror, assume the string is validly mirrored.

7.

Implement a search algorithm to find words in a square matrix of letters.
Print 1 if the string is found, 0 otherwise.
Words can only be formed horizontally (along a row) or vertically (along a column) — not diagonally or backward.

Example 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.

Create a program that takes n foods (each with a name, calorie count, and category: 'healthy' or 'junk') and arranges them so that junk food comes first in descending calorie order, then healthy food in ascending calorie order.

Sample input (9 foods): 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

Expected 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.

Build a dynamic library management system. Each book has: title (string), author (string), year (int), copies (int).

Implement:
Book *addBook(Book *books, int *size) — dynamically resize and add a book
void searchBook(Book *books, int size, const char *title) — case-insensitive search
void sortBooks(Book *books, int size) — sort by publication year (ascending)
void updateCopies(Book *books, int size, const char *title)
void displayBooks(Book *books, int size) — already implemented below

Free all allocated memory before program termination.

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

typedef struct {
    char title[100];
    char author[100];
    int  year;
    int  copies;
} Book;

Book *addBook(Book *books, int *size)             { }
void  searchBook(Book *books, int size, const char *title) { }
void  sortBooks(Book *books, int size)            { }
void  updateCopies(Book *books, int size, const char *title) { }

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);
}
```

10.

Write void findReindeer(char *word) that searches for the word "REINDEER" in a given string (case-insensitive). Print all positions where it is found; if not found, print an appropriate message.

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

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

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

11.

Write char *updateOldDate(char *oldDate) that converts a date string from "YYYYMMDD" to "Month Day, Year" (e.g., "20231221" => "December 21, 2023").
Return a heap-allocated string. If the input is invalid or wrongly formatted, return "Invalid Date".

```
int main(void) {
    char validDate[] = "20231221";
    char *r1 = updateOldDate(validDate);
    assert(strcmp(r1, "December 21, 2023") == 0);
    free(r1);

    char invalidDate[] = "20231321";
    char *r2 = updateOldDate(invalidDate);
    assert(strcmp(r2, "Invalid Date") == 0);
    free(r2);

    char notDateNumbers[] = "19201010101911";
    char *r3 = updateOldDate(notDateNumbers);
    assert(strcmp(r3, "Invalid Date") == 0);
    free(r3);

    char notDateString[] = "kaejkaj;aea";
    char *r4 = updateOldDate(notDateString);
    assert(strcmp(r4, "Invalid Date") == 0);
    free(r4);
    return 0;
}
```

12.

A gift tag starts with a letter A-Z, followed by alternating digits and letters. Each letter after the first is shifted forward by the preceding digit's value (wraps around the alphabet; case-sensitive).

Example: "A3B4b" => 'A' unshifted; 'B' shifted by 3 => 'E'; 'b' shifted by 4 => 'f'. Output: "AEf".

Implement a program gift_tag.c that reads an encoded tag (at most 185 characters) and prints the decoded version.

Sample 1: Input: A3B4b => Decoded: AEf
Sample 2: Input: R1c => Decoded: Rd

13.

Write a function that takes a heap-allocated string s1 and returns a new heap-allocated string containing the reverse of s1.

14.

Write a function fname(char *s1, int n) that takes a heap-allocated string s1 and an integer n, and returns a new heap-allocated string containing s1 repeated n times.
Example: fname("abc", 3) => "abcabcabc"

15.

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

16.

Implement the function:
int longestpalindrome(char *st, int start, int end);
that returns the length of the longest palindrome in st between index start and index end (inclusive).

```
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;
}
```