

## ESCAPE SEQUENCE

<code>\n</code>	New line
<code>\t</code>	Horizontal tab
<code>\r</code>	Carriage return (moves cursor to the beginning of current line without advancing to the next line)
<code>\\</code>	Backslash
<code>\"</code>	Double Quote
<code>\'</code>	Single Quote
<code>\b</code>	Backspace (Deletes the last character)
<code>\f</code>	Form feed (Advances paper feed in printers to the start of page)
<code>\v</code>	Vertical tab
<code>\a</code>	Audible bell (alert) → Beep sound
<code>\?</code>	Question mark
<code>\0</code>	Null character
<code>\x (\\xhh)</code>	For representing Hexadecimal values (specify any character using its hexadecimal value)
<code>\ddd</code>	Octal Escape sequence

Date

Notes

```
char ch = '\x 41';  
printf ("%c", ch);    Output → A
```

\x 20 → space

\x 0A → new line

\x 09 → tab

Only first 2 digits are considered part of the \x escape sequence '\x hh'.

#### ASCII Values

NUL	0
Space	32
0	48
1	49
A	65
a	97



# Notes

Date

## FORMAT SPECIFIERS

%d or %i	int	Signed integer
%u	unsigned int	unsigned integer
%f	float	floating point no (decimal)
%lf	double	double precision floating point
%c	char	single character
%s	char[]	string (array of characters)
%p	pointer	pointer address
%x or %X	unsigned int	unsigned hexadecimal (lower/upper)
%o	unsigned int	unsigned octal number
%ld	long int	signed long integer
%lu	unsigned long	unsigned long integer
%lld	long long int	signed long long integer
%llu	unsigned long long	unsigned long long integer
%e or %E	float or double	scientific notation (lower/upper)
%g or %G	float or double	shorter of %f & %e (lower/upper)
%h	short	signed short integer
%hu	unsigned short	unsigned short integer
%x		prints % character
%zu		prints value of type size_t

%g or %G is used to format floating point numbers in a way that dynamically chooses between using decimal notation (%f) and hexadecimal notation (%e)

If the exponent is less than -4 or greater than or equal to the precision, it switches to scientific notation.

Date    /    /   

Notes

- \* printf exhibits characteristics of 'function overloading' but it does not involve operator overloading, function overriding or operator overriding.

But it(c) actually does not support overloading.  
It supports 'Variadic functions' of 'printf'.

#### \* VARIADIC FUNCTIONS

are functions that can accept a variable number of arguments.

Defined using ellipsis (...)

- \* 'z' is a length modifier that specifies the size of the argument in 'size\_t'.



## Notes

Date

- \* "%+d" format specifier prints the integer with a sign (+ or -).

```
int num = -4;  
printf("%+d", num);
```

 Output: -4

```
int num2 = 10;  
printf("%+d", num2);
```

 Output: +4

```
int zero = 0;  
printf("%+d", zero);
```

 Output: +0

- \* "%+5d" with a specified width

```
int num = -25;  
printf("%+5d", num);
```

 Output: -25  
2 spaces

- \*  
int temp = -5;  
int pressure = 0;

```
printf("%+d°C", temp);  
printf("%+d hPa", pressure);
```

 Output: -5°C  
+0 hPa

- \* %5d: Right aligns the integer within a width of 5 characters, padding with spaces on the left.
- \* %+5d: Right aligns an integer, includes the sign & maintains a minimum width of 5 characters.
- \* %-5d: Left aligns the integer within a width of 5 characters, padding with spaces on the right.

Date      /      /     

Notes

\* int num = 10.4;

printf ("%.10.4f", num);

Output:

4.2000

4 decimal places

10 width

Total width of the output will be 10 characters

\* const char \*str = "abcdefgh";

printf ("%.10.4s", str);

Output:

abcd

Formatted string with width 10, max 4 characters.

\*

### INPUT - OUTPUT FUNCTIONS

printf, scanf, fgets, puts, scanf-s, getch, getche, fprintf / fscanf, getchar.

• fgets : safely read a string from input (includes spaces)

char str[100];

printf ("Enter a string");

fgets (str, sizeof (str), stdin);

printf ("You entered ", str);

• puts : Output a string followed by a new line

puts ("Hello World");



## Notes

Date / /

- `scanf_s` : Safer version of `scanf` (Microsoft specific).

It checks for buffer overflows by acquiring the size of buffer for certain types (strings).

```
char str[100];  
printf("Enter a string");  
scanf_s("%s", str, (unsigned)_countof(str));  
printf("%s", str);
```

reads a string with buffer size check.

- `getchar` : Reads a single character from standard input.

```
char ch;  
printf("Enter a character");  
ch = getchar();  
printf("You entered %c", ch);
```

`getchar`

- `getch` / `getche` : Reads a character from the keyboard without echoing it on the console (for `getch`) or echoing (for `getche`).

`putchar`

- `putch` : Outputs a single character to the console.

Date

Notes

- `fprintf` : used to output formatted data to a file.
- `fscanf` reads formatted data from a file.

```
FILE * file = fopen("example.txt", "w");  
fprintf (file, "Hello file"); // Writes to the file  
fclose (file);
```

```
char str[100];  
file = fopen("example.txt", "r");  
fscanf (file, "%s", str) // Reads from the file  
printf ("File content %s", str);  
fclose (file);
```



# Notes

Date

- \* **atoi** (ASCII to integer)  
converts a string to an integer

```
const char * str = "1234";  
int num = atoi(str);  
printf("%d", num); // Output: 1234
```

- \* **atof** (ASCII to float)

- \* ~~atoi~~ **itoa** (integer to ASCII)  
converts an integer to a string, returns a pointer to the resulting string

```
int num = 1234;  
char str[20];  
itoa(num, str, 10); // convert string → integer with base 10  
printf("%s", str); // Output: 1234
```

- \* **sprintf** Formats and stores a string in a buffer.  
Formats data into a string

```
char str[20];  
int num = 1234;  
sprintf(str, "%d", num);  
printf("%s", str); // Output: 1234
```

Date      /      /     

Notes

- \* **strtol** (String to Long)  
converts a string to a long integer with error checking.

```
const char * str = " 1234 abc ";
char * endptr;
long num = strtol ( str, &endptr, 10); // base
printf ("Long integer %ld", num); // long integer 1234
printf ("Remaining string", endptr); // Rem. string abc
```

- \* **strtod** (String to double)

- \* **Exit & Abort** are functions used ~~about~~ to terminate a program, but they do so in different ways, & have different purposes.

Exit	Abort
<ul style="list-style-type: none"> <li>• Returns a status code to the OS. (0 for success, non-zero for failure)</li> <li>• Performs cleanup.</li> <li>• Typically does not generate a core dump.</li> <li>• Use for normal program termination.</li> <li>• To terminate program gracefully.</li> </ul>	<ul style="list-style-type: none"> <li>• Does not return a status code.</li> <li>• Does not perform cleanup.</li> <li>• May generate a core dump.</li> <li>• Used to indicate a serious error / bug.</li> <li>• When you encounter a critical error from which you cannot recover &amp; want to terminate without cleanup.</li> </ul>



# Notes

Date      /      /     

- \* malloc : Allocates a specified no. of bytes
- free : Deallocates previously allocated memory
- calloc : Allocates memory for an array & initialises it to 0
- realloc : Resizes a previously allocated memory block.

\*) Switch case.

\*) ~~foreach~~ (range based loop)

a) goto in C/C++

a) Control flow statements

```
a) if (int b = 20; b > 40)
{
}
}
```

a) finally C++ CLI (has functionality of C#) <sup>try</sup>  
↳ not there in C++

a) catch (...) <sub>→ ellipse</sub>

a) C++ (98)

```
if (init statement; condition)
{
}
}
```

```
if (int b = 20; b > 40)
{
}
}
```

Date / /

Notes

- \* do while loop will execute atleast once.
- \* Never use expressions in switch.
- \* try catch is very costly (avoid it if possible)  
↳ it adds alot of binaries.

\* memcpy (Memory copy)

\* memset (Memory Set)

- \* Copies memory from one location to another.
- \* Sets memory to a specific value.
- \* Requires a source & destination.
- \* Only requires a memory block.
- \* Undefined behaviour if source & destination overlap.
- \* Not applicable.
- \* Used for copying binary data.
- \* Used for initialising or filling a memory.

memcpy is used when you need to copy raw memory blocks like copying contents of a struct, an array, or raw byte buffers. It's often used in performance critical code where you need to transfer chunks of memory without type checking.



## Notes

Date      /      /     

```
char src[] = "Hello, world!";  
char dest[50];
```

```
memcpy(dest, src, 13);  
→ Copy 13 characters from src → dest
```

\* **memset** fills a specific number of bytes in memory to a particular value.  
It is used to initialise memory, such as filling an array with zeroes or setting default values for newly allocated memory.

```
char buffer[50];
```

```
memset(buffer, '-', 10);  
→ set first 10 characters in buffer to '-'
```

```
buffer[10] = '\0';  
→ Null - terminates the string
```

Date      /      /     

Notes

## TIME FUNCTIONS

<ctime> → C++

<time.h> → C

- `time()` Returns the current time as `time_t` (since UNIX)
- `difftime()` Computes the difference in seconds b/w two `time_t` times
- `clock()` Returns the processor time consumed by the program
- `asctime()` Converts a `tm` structure to a human readable string, returning pointer to the string
- `ctime()` Similar to `asctime()` for `time_t`
- `gmtime()` Converts `time_t` to UTC in form of a `tm` structure
- `localtime()` Converts `time_t` to local time
- `mktime()` Converts a `tm` structure to `time_t`
- `strftime()` Converts & formats a `tm` structure into a string according to a specified format
- `clock_gettime()` Provides high resolution real time or monotonic time
- `chrono` in C++11 & later, `std::chrono` provides a comprehensive time API for high precision time management.

```
auto start = chrono::high_resolution_clock::now();
this_thread::sleep_for(chrono::seconds(2));
auto end = chrono::high_resolution_clock::now();
chrono::duration<double> duration = end - start;
cout << "Elapsed time" << duration.count();
```

→ `time_t` stores time in number of seconds

→ `tm` provides a broken down representation of time, allowing access to individual components like hour, minute, second



## ERROR FUNCTIONS

- errno** : A global variable set to a positive error code when a system or library call fails.
- perror** : prints the description of the last error that occurred, based on the current value of **errno**.
- strerror** : returns a pointer to the string describing the error, code passed as an argument.
- setjmp / longjmp** : provides non-local jumps (used to implement error handling or exception like behaviour).
- abort** : causes abnormal program termination & generates a core dump (if enabled).
- exit** : terminates the program normally, returning a status code to the calling process or shell.
- atexit** : registers a function to be called on exit.
- assert** : a macro that checks the condition, & if it evaluates to false, terminates the program.

## Common Errno values:

EPERM	Operation not permitted
ENOENT	No such file or directory
ESRCH	No such process
EINTR	Interrupted system call
ENOMEM	Out of memory
EACCESS	Permission denied
EBUSY	Resource busy
EXIST	File exists
EINVAL	Invalid argument
EAGAIN	Try again (resource temporarily unavailable)



Date      /      /     

Notes

## GOTO

goto statement is a control flow statement that allows for an unconditional jump to another point in the program, marked by a label.

```
int x = 0;
printf("Before goto");
```

```
if (x == 0)
```

```
{
    goto skip;
}
```

```
printf("After goto");
```

```
printf("Something");
```

```
printf("After goto");
```

```
goto;
```

```
printf("After goto");
```

Output: ~~Before goto~~

Output: Before goto  
After goto.

\* pow  $\rightarrow$  pow(x, y)  $\rightarrow x^y$

\* pow10  $\rightarrow$  pow10(x)  $\rightarrow 10^x$



# Notes

Date \_\_\_\_\_

## STRING METHODS

① `strlen` : Returns the length of a string (excluding null terminator)

```
char str[] = "Hello";  
printf("Length: %d", strlen(str)); // 5
```

`strlen()` returns the length of the string excluding the null terminator & its return type is `size_t`.

`sizeof()` gives the total size of array, including the null terminator & its return type is also `size_t`.

② `strcpy` : Copies a source string to a destination

```
strcpy(dest, src)
```

③ `strncpy` : Copies upto `n` characters from the source string to the destination

```
strncpy(dest, src, n-characters)  
↳ number
```

④ `strcat` : Appends source string to the destination string

```
strcat(str1, str2)
```



Date      /      /     

## Notes

- ⑤ **strncat** : appends upto  $n$  characters from the source string to the destination

**strncat** (dest, src, no. of characters)

- ⑥ **strcmp** : compares two strings lexicographically

**strcmp** (str1, str2)  $\rightarrow$   $== 0 \rightarrow$  equal  
else non equal

- ⑦ **strncmp** : compares upto  $n$  characters of two strings lexicographically

**strncmp** (str1, str2, no. of chars)

- ⑧ **strchr** : searches for the first occurrence of a character in the string

**strchr** (str, character)

```
char str[] = "Hello World";  
char *pos = strchr(str, 'W');  
printf(pos)       $\rightarrow$  World  
printf(str)       $\rightarrow$  Hello World
```

```
if (pos) {  
    printf("Found at %d", pos-str); // 6  
}
```



## Notes

Date / /

- ⑨ `strstr` : finds the first occurrence of a substring in a string

`strstr (Haystack, needle)`  
↳ string, ↓

- ⑩ `strtok` : splits a string into tokens based on delimiters

```
char str[] = "Hello, World, C, Programming";  
char * token = strtok (str, ",");
```

```
while (token != NULL) {  
    printf ("Token: %s", token);  
    token = strtok (NULL, ",");  
}
```

- ⑪ `sscanf` : reads formatted data from a string

```
char str[] = "123 234";  
int num1, num2;  
sscanf (str, "%d %d", &num1, &num2);  
printf ("Extracted nos. %d %d", num1, num2);  
↳ 123, 234
```

⑫ `strptime` :

⑫ `sprintf` : Writes formatted output to a string

```
char buffer[50];
int num = 123;
sprintf(buffer, "Number: %d", num);
printf("%s", buffer) // Number: 123
```

⑬ `snprintf` : Writes formatted output to a string with a size limit

```
char buffer[10];
int num = 12345;
snprintf(buffer, sizeof(buffer), "Number: %d", num);
printf("%s", buffer) // Number: 123
```

⑭ `memset` : Fills a block of memory with a particular value

```
char str[50] = "Hello World";
memset(str, '*', 5);
printf("%s", str) // ***** World
```

⑮ `memcpy` : Copies a block of memory from one block to another

```
char src[] = "Hello";
char dest[10];
memcpy(dest, src, strlen(src) + 1);
printf("%s", dest) // Hello
```

This is necessary to ensure that null terminator '\0' at the end of string is also copied.



Date      /      /     

Notes

- ⑫ `sprintf` : Writes formatted output to a string

```
char buffer[50];  
int num = 123;  
sprintf(buffer, "Number: %d", num);  
printf("%s", buffer) // Number: 123
```

- ⑬ `snprintf` : Writes formatted output to a string with a size limit

```
char buffer[10]  
int num = 12345;  
snprintf(buffer, sizeof(buffer), "Number: %d", num);  
printf("%s", buffer) // Number: 123
```

- ⑭ `memset` : Fills a block of memory with a particular value

```
char str[50] = "Hello World"  
memset(str, '*', 5)  
printf("%s", str) // ***** World
```

- ⑮ `memcpy` : Copies a block of memory from one block to another

```
char src[] = "Hello"  
char dest[10]  
memcpy(dest, src, strlen(src) + 1);  
printf("%s", dest) // Hello
```

This is necessary to ensure that null terminator '\0' at the end of string is also copied.

## Notes

Date

```
int linearSearch (int arr[], int n, int target) {  
    for (int i = 0; i < n; i++) {  
        if (arr[i] == target)  
            return i;  
    }  
    return -1;  
}
```

```
int binarySearch (int arr[], int left, int right, int target) {  
    while (left <= right) {  
        int mid = (left + right) / 2;  
        if (arr[mid] == target) return mid;  
        if (arr[mid] < target) left = mid + 1;  
        else right = mid - 1;  
    }  
    return -1;  
}
```

```
int hashSearch (unordered_map <int, int> &hashTable, int target) {  
    if (hashTable.find(target) != hashTable.end()) {  
        return hashTable[target];  
    }  
    return -1;  
}
```



Date      /      /     

Notes

```
int main() {  
    const int n = 100000;    // Array size  
    int arr[n];  
    for (int i = 0; i < n; i++) {  
        arr[i] = i + 1;    // Fill array with sorted  
                           // data for binary search  
    }
```

```
    int target = n - 1;    // element to search
```

```
    clock_t start, end;
```

```
    double time_taken;
```

```
    start = clock();
```

```
    int linear_result = linear_search(arr, n, target);
```

```
    end = clock();
```

```
    time_taken = ((double)(end - start)) / CLOCK_PER_SEC;
```

```
    printf("Linear search time: %f seconds, time taken);\n
```

Similar for BS & HT.

Output:

Linear search time : 0.004050 seconds

Binary search time : 0.000002 seconds

Hash table time : 0.000002 seconds

## Notes

Date

- \* `clock()` function returns number of clock ticks that have elapsed since the program started. A "clock tick" is a unit of time determined by the system & its not necessarily a second.
- \* `CLOCKS_PER_SEC` constant is defined in `<time.h>` library that represents the number of clock ticks per second on your system.
- \* `clock()` measures CPU time which is more precise & provides a finer resolution (micro/milli seconds depending on system) making it ideal for measuring the time spent executing small sections of code.
- \* `time()` measures real world calendar time in seconds which is coarse & not appropriate for short operations like search algorithms.



Date 23 / October 2024

Notes

✓ a) GetTrk Count ()

✓ a) Mathematical functions

✓ e) what happens when we pass null pointer

e) Debugger in C/C++

✓ a) How many bytes does each datatype use

e) Based on compiler, how ~~do~~ data much size do data types use

a) ~~that~~ which compiler follows what size

d) size\_t

e) auto Keyword

(↳ destroyed automatically when a f<sup>n</sup> is completed.  
→ different in C & C++

a) Storage classes in C → extern, static, extern, register, auto

a) user defined functions, variable scope, local global,  
Param passing ~~etc~~ cdecl (right to left)  
pascal (left to right)

# Notes

Date / /

## # Get Tick Count

is a window specific function that retrieves the number of milliseconds that have elapsed since the system was elapsed started. (system's uptime).

It is part of Windows API & is used to measure time intervals.

```
DWORD startTick = GetTickCount();  
for(int i=0; i<10000000; i++);  
DWORD endTick = GetTickCount();  
DWORD elapsedTime = endTick - startTick;  
printf("%lu", elapsedTime);    → 123    (in milliseconds)
```

# Mathematical functions : abs, fabs, pow, sqrt, pow10, sin, cos, tan, ceil, floor, log, log10, exp, fmod, round, hypot,

# Dereferencing a NULL pointer causes undefined behaviour (likely a crash or segmentation fault).

#



Date      /      /     

Notes

TYPE	STORAGE SIZE	in C
char	1 byte	
unsigned char	1 byte	
signed char	1 byte	
int	2 or 4 bytes	
unsigned int	2 or 4 bytes	
short	2 bytes	
unsigned short	2 bytes	
long	8 bytes	
unsigned long	8 bytes	