<string.h>

`size_t strlen(const char *s);`

Return the length of the string pointed to by `s`, not including the null character.

`int strcmp(const char *s1, const char *s2);`

Return a negative/zero/positive integer, depending on whether the string pointed to by `s1` is less than/equal to/greater than the string pointed to by `s2`.

`int strncmp(const char *s1, const char *s2, size_t n);`

Same as `strcmp` except compare no more than `n` characters.

`char *strcpy(char *s1, const char *s2);`

Copy the string pointed to by `s2` into the array pointed to by `s1`. Return `s1`.

`char *strncpy(char *s1, const char *s2, size_t n);`

Same as `strcpy` but copy no more than `n` characters.

`char *strcat(char *s1, const char *s2);`

Append characters from the string pointed to by `s2` to the string pointed to by `s1`. Return `s1`.

`char *strncat(char *s1, const char *s2, size_t n);`

Same as `strcat` but copy no more than `n` characters.

`char *strtok(char *str, const char *delim);`

Breaks string `str` into a series of tokens using the delimiter `delim`. Returns a pointer to the first token found in the string. A null pointer is returned if there are no tokens left to retrieve.

`void *memcpy(void *dest, const void *src, size_t n);`

Copies `n` bytes from memory area `src` to memory area `dest`. Returns the pointer to the destination.

`void *memmove(void *str1, const void *str2, size_t n);`

Copies `n` bytes from `str2` to `str1`, but for overlapping memory blocks, `memmove()` is a safer approach than `memcpy()`. Returns the pointer to the destination.

<stdlib.h>

`void exit(int status);`

Terminate the entire program with the given status.

`void *malloc(size_t size);`

Allocate a block of memory with `size` bytes. The block is NOT cleared. Return a pointer to the beginning of the block, or `NULL` if memory could not be allocated.

`void *realloc(void *ptr, size_t size);`

Attempt to resize the memory block pointed to by `ptr` to `size` bytes. Returns a pointer to the newly allocated memory, or `NULL` on failure.

`void free(void *ptr);`

Release the memory block pointed to by `ptr`. The block must have been allocated by a call to `malloc` (or `calloc` or `realloc`), except if `ptr == NULL` (in which case the call to `free` has no effect).

`int atoi(const char *str);`

Converts the string argument `str` to an integer (`int`).

`double atof(const char *str);`

Converts the string argument `str` to a floating-point number (`double`).

`void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));`

Sort the array pointed to by `base` with `nmemb` elements, each `size` bytes long, using the comparison function `compar`. The comparison function must return a negative/zero/positive integer if the first argument is less than/equal to/greater than the second.

Sample comparator for `int *`:

```
int cmp_int(const void *a, const void *b) {
    int x = *(const int *)a;
    int y = *(const int *)b;
    if (x < y) return -1;
    if (x > y) return 1;
    return 0;
}
/* Usage: */
int arr[] = {5, 2, 8, 1, 9};
qsort(arr, 5, sizeof(int), cmp_int);
```

<stdio.h>

`FILE *fopen(const char *filename, const char *mode);`

Open the file whose name is pointed to by `filename` and associate it with a stream — mode specifies the mode in which the file is to be opened (`"r"` or `"rb"` for reading, `"w"` or `"wb"` for writing, where `"b"` is for binary files). Return a file pointer to be used for subsequent operations on the file, or `NULL` if the file could not be opened.

`int fclose(FILE *stream);`

Close the stream pointed to by `stream`. Return 0 if successful, `EOF` if any error is detected.

`int feof(FILE *stream);`

Return a nonzero value if the end-of-file indicator is set for the stream pointed to by `stream`; otherwise return 0.

`int ferror(FILE *stream);`

Return a nonzero value if the error indicator is set for the stream pointed to by `stream`; otherwise return 0.

`long ftell(FILE *stream);`

Return the current file position indicator for the stream pointed to by `stream`. If the call fails, return `-1L`.

`int fseek(FILE *stream, long offset, int whence);`

Change the file position indicator for the stream pointed to by `stream`. If `whence` is `SEEK_SET`, the new position is the beginning of the file plus `offset` bytes; if `whence` is `SEEK_CUR`, the new position is the current position plus `offset` bytes; if `whence` is `SEEK_END`, the new position is the end of the file plus `offset` bytes. The value of `offset` may be negative. Return 0 if successful, nonzero otherwise.

`int putchar(int c);`

Write the character `c` to the `stdout` stream. Return `c`, or `EOF` in the case of error.

`int putc(int c, FILE *stream);`

Same as `putchar` except write to the given stream.

`int getchar(void);`

Read a character from the `stdin` stream. Return the character

read, or EOF in the case of any error.

`int getc(FILE *stream);`

Same as `getchar` except read from the given stream.

`int fputs(const char *s, FILE *stream);`

Write the string pointed to by `s` to the stream pointed to by `stream`. Return a nonnegative value if successful, or `EOF` in the case of an error.

`char *fgets(char *s, int n, FILE *stream);`

Read characters from the stream pointed to by `stream` and store them in the array pointed to by `s`. Reading stops at the first new-line character (which is stored in the string), when `n-1` characters have been read, or at end-of-file. Always append a null character to the string. Return `s`, or `NULL` if any read errors happen or if end-of-file is encountered before reading any character.

`size_t fread(void *ptr, size_t size, size_t n, FILE *stream);`

Read `n` elements, each `size` bytes long, from the stream pointed to by `stream`, and store them in the array pointed to by `ptr`. Return the number of elements actually read.

`size_t fwrite(void *ptr, size_t size, size_t n, FILE *stream);`

Write `n` elements, each `size` bytes long, from the array pointed to by `ptr` to the stream pointed to by `stream`. Return the number of elements actually written.

`int printf(const char *format, ...);`

Write output to the `stdout` stream. The string pointed to by `format` specifies how subsequent arguments are displayed. Return the number of characters written, or a negative value in case of error.

`int fprintf(FILE *stream, const char *format, ...);`

Same as `printf` except write to the given stream.

`int scanf(const char *format, ...);`

Read input values from the `stdin` stream. The string pointed to by `format` specifies the format of the items to be read. The arguments that follow `format` point to memory locations where to store the values. Return the number of values successfully read and stored, or `EOF` in case of input failure prior to any conversion.

`int fscanf(FILE *stream, const char *format, ...);`

Same as `scanf` except read from the given stream.

printf conversion specifiers (basic)

`%d`	`int`, printed in decimal
`%hd`	`short int`, printed in decimal
`%ld`	`long int`, printed in decimal
`%u`	`unsigned int`, printed in decimal
`%hu`	`unsigned short int`, printed in decimal
`%lu`	`unsigned long int`, printed in decimal
`%f`	`double` (or `float`), fixed-point
`%e`	`double` (or `float`), scientific notation
`%g`	`double` (or `float`), fixed or scientific
`%c`	`char`, printed as a single character
`%s`	`char *`, printed as a string
`%p`	`void *` (any pointer), memory address

An optional **field width**: a positive integer between the `%` and the conversion character; indicates a minimum number of characters to use (padded on the left with spaces). E.g., `"%6d"` prints an integer using at least 6 characters.

An optional **precision**: a positive integer preceded by `.`, between the `%` and the conversion character; indicates digits after the decimal point (for `e`, `f`), total significant digits (for `g`), or maximum bytes to print (for `s`). E.g., `"%6.2f"` prints a float using 6 characters with 2 decimal places.

scanf conversion specifiers (basic)

`%d`	`int`, read in decimal
`%hd`	`short int`, read in decimal
`%ld`	`long int`, read in decimal
`%u`	`unsigned int`, read in decimal
`%hu`	`unsigned short int`, read in decimal
`%lu`	`unsigned long int`, read in decimal
`%f/%e/%g`	`double` (or `float`)
`%c`	`char`, read as a single character
`%s`	`char *`, read as a string

Operator Precedence

Operators with the highest precedence appear at the top; lowest at the bottom. Within an expression, higher precedence operators are evaluated first.

Category	Operator	Assoc.
Postfix	`() [] -> . ++ -`	L to R
Unary	`+ - ! ~ ++ - (type) * & sizeof`	R to L
Multiplicative	`* / %`	L to R
Additive	`+ -`	L to R
Shift	`<< >>`	L to R
Relational	`< <= > >=`	L to R
Equality	`== !=`	L to R
Bitwise AND	`&`	L to R
Bitwise XOR	`^`	L to R
Bitwise OR	`|`	L to R
Logical AND	`&&`	L to R
Logical OR	`||`	L to R
Conditional	`?:`	R to L
Assignment	`= += -= *= /= %=` etc.	R to L
Comma	`,`	L to R