Documentation

glib-functions

General guidelines for users:

- *All functions which are hyperlinked are glib functions, and link to their respective documentation online.*
- *All other functions which are not hyperlinked are custom made functions, and may be wrapper functions around standard glib functions, or different functions altogether.*
- *All output printed to* `stdout`.
- *Functions are described in the following format:*

Title of function

return_type `function_name(`*`arg_type arg_name, arg_type arg_name…`*`)`

One sentence function description.
Arguments: *Describe the arguments if not trivial.*
Returns*: One sentence description of what the function returns.*

1 Utility: General I/O

1.1 g_printf()

```
gint g_printf (gchar const *format, ...);
```
A wrapper around the standard printf() function.

Arguments: *C string that contains the text to be written to* `stdout`.
It can optionally contain embedded format specifiers that are replaced by the values specified in subsequent additional arguments and formatted as requested.

Returns: *On success, the total number of characters written is returned.*
If a writing error occurs, the error indicator (ferror) is set and a negative number is returned.

2 Utility: File I/O

2.1 g_file_get_contents()

```
gboolean g_file_get_contents (const gchar *filename, gchar **contents, gsize *length, GError **error);
```
Reads the contents of a file into a string.

Arguments: *filename : name of a file to read contents from.*
contents : location to store an allocated string, use g_free() to free the returned string.
length : location to store length in bytes of the contents, or NULL.

Returns: TRUE on success, FALSE if an error occurred

3 Utility: Comparators

3.1 gint int_comparator()

```
gint (*GCompareFunc) (gconstpointer a, gconstpointer b);
```
An integer comparator function. Used to sort GSList, and GArray.

Arguments: *a: a value*
b: a value to compare with

Returns: *negative value if a < b ; zero if a = b ; positive value if a > b*

4 Utility: Random number generators

4.1 g_random_double()

```
gdouble g_random_double (void);
```

Returns a random gdouble equally distributed over the range [0..1).

Arguments: *none*

Returns: *random gdouble equally distributed over the range [0..1).*

4.2 g_random_int_range()

```
gint32 g_random_int_range (gint32 begin, gint32 end);
```

Returns a random int equally distributed over the range [`begin` ..`end` :1).

Arguments: *begin: lower closed bound of the interval*

end: upper open bound of the interval

Returns: *random int equally distributed over the range [`begin` ..`end` :1)*

5 Utility: Timer

5.1 g_timer_new()

```
GTimer* g_timer_new (void);
```

Creates a new GTimer, and starts timing.

Arguments: *none*

Returns: *a new GTimer*

5.2 g_timer_stop()

```
void g_timer_stop (GTimer *timer);
```

Marks an end time.

Arguments: *timer: a GTimer*

Returns: *none*

5.3 g_timer_elapsed()

`gdouble g_timer_elapsed (GTimer *timer, gulong *microseconds);`
If GTimer has been stopped, obtains the elapsed time between the time it was started and the time it was stopped.

Arguments: *timer: a Gtimer*
microseconds: return location for the fractional part of seconds elapsed, in microseconds (that is, the total number of microseconds elapsed, modulo 1000000), or NULL

Returns: *seconds elapsed as a floating point value, including any fractional part.*

5.4 g_timer_reset()

`void g_timer_reset (GTimer *timer);`
Resets timer .

Arguments: *timer: a Gtimer*

Returns: *none*

5.5 g_timer_destroy()

`void g_timer_destroy (GTimer *timer);`
Destroys a timer, freeing associated resources.

Arguments: *timer: a GTimer*

Returns: *none*

6 Data structure: Arrays

6.1 g_array_new()

`GArray* g_array_new (gboolean zero_terminated, gboolean clear_, guint element_size);`
Creates a new GArray.

Arguments: *zero_terminated : TRUE if the array should have an extra element at the end which is set to 0*
clear_ : TRUE if GArray elements should be automatically cleared to 0 when they are allocated
*element_size **:** the size of each element in bytes*

Returns: *the new GArray*

6.2 g_array_append_val()

GArray* g_array_append_val (GArray * array, gconstpointer data);

Adds the data to the end of the dynamic array.

Arguments: *array: a GArray*
data: the value to append to the GArray

Returns: *the GArray*

6.3 g_array_sort()

void g_array_sort (GArray *array,GCompareFunc compare_func);
Sorts the given GArray according to the given comparison function.

Arguments: *array: pointer to GArray to be sorted*
compare_func: comparison function

Returns: *none*

6.4 find_element()

void find_element (int x, GArray *array);
Searches for element x in a GArray and prints if it is found or not.

Arguments: *x: element to be searched for*
array: GArray to be searched

Returns: *none*

6.5 print_array()

void print_array (GArray *array);
Prints the elements of the given GArray.

Arguments: *array: GArray to be printed*

Returns: *none*

6.6 g_array_index()

***void** ** g_array_index(GArray * array, guint element_size, guint index);*
Returns the element of GArray at an index, casted to the given type, whose size is mentioned as `element_size`.

Arguments: *array : GArray whose element is required*
element_size : size of element (the type of the element)
index : the index of the element to return

***Returns**:* *the element of the GArray at the index given by i, casted to the given element type.*

6.7 array_print_2d()

***void** array_print_2d (GArray *array, int row, int col);*
Prints the two dimensional matrix stored in a one dimensional array.

Arguments: *array : Pointer to input GArray which is to be printed*
row : number of rows of the two dimensinal matrix
col : no of columsn of the two dimensional matrix

***Returns**:* *none*

6.8 array_trace_2d()

***int** array_print_2d (GArray *array, int no_of_rows);*
Calculates the trace of a two dimensional square matrix stored in a one dimensional array.

Arguments: *array: Pointer to input Garray*
row: number of rows in the input Garray

***Returns**:* *Integer representing the trace of the array.*

7 Data structure: Balanced BST (AVL Tree)

7.1 balanced_bst_new ()

***Bst** balanced_bst_new ();*
Creates a new balanced bst and returns a pointer to it

Arguments: *none*

Returns: *pointer to a newly created balanced Bst*

7.2 balanced_bst_insert_file()

gboolean `balanced_bst_insert_file (Bst * tree, const char * in_file);`
Inserts elements from a file to the tree.If file contains integers, they are inserted into the tree. Otherwise an empty BST is loaded.

Arguments: *tree : Pointer to a Bst*
in_file : name of file which contains the elements to be inserted
(File format: Integers, one in a line, in a text file)

Returns*:* *TRUE if successfully inserted, FALSE otherwise*

7.3 balanced_bst_insert()

gboolean `balanced_bst_insert (Bst * tree, int element);`
Inserts an element into the tree and balances the tree

Arguments: *tree: a balanced Bst*
element: integer to be inserted

Returns: *TRUE if inserted successfully, FALSE otherwise*

7.4 balanced_bst_search ()

gboolean `balanced_bst_search (Bst * tree, int element);`
Searches for an element in the tree

Arguments: *tree: balanced Bst to be searched*
element: integer to be searched for

Returns: *TRUE if element found, FALSE otherwise*

7.5 balanced_bst_delete()

gboolean `balanced_bst_delete (Bst * tree, int element);`
Deletes the element and balances the tree

Arguments: *tree: pointer to a balanced Bst*
element: integer to be deleted

Returns: *TRUE if element deleted successfully , FALSE otherwise*

7.6 balanced_bst_inorder()

***GArray** ** balanced_bst_inorder (*Bst * tree, GArray * traversal*);*
Stores the inorder traversal of the tree in the given GArray.

Arguments: *tree : Pointer to the input Bst*
traversal : Pointer to output GArray

***Returns**:* *Pointer to GArray with required traversal*

7.7 balanced_bst_preoder ()

***GArray** ** balanced_bst_preorder (*Bst * tree, GArray * traversal*);*
Stores the preorder traversal of the tree in the given GArray.
Arguments: *tree : Pointer to the input balanced Bst*
traversal : Pointer to output GArray

***Returns**:* *Pointer to GArray with required traversal*

7.8 balanced_bst_postorder()

***GArray** ** balanced_bst_postorder (*Bst * tree, GArray * traversal*);*
Stores the postorder traversal of the tree in the given GArray.
Arguments: *tree : Pointer to the input Bst*
traversal : Pointer to output GArray

***Returns**:* *Pointer to GArray with required traversal*

8 *Data structure: BST*

8.1 bst_new ()

***Bst** ** bst_new ();*
Creates a new unbalanced Bst and returns a pointer to it

Arguments: *none*

Returns: *pointer to a newly created Bst*

8.2 bst_insert_file ()

gboolean `bst_insert_file (`*`Bst * tree, const char * in_file`*`);`
Inserts elements from a file to the tree

Arguments: *tree : pointer to Bst to which elements are to be inserted*
in_file : name of input file

Returns*:* *TRUE if inserted successfully, FALSE otherwise*

8.3 bst_insert()

gboolean `bst_insert (`*`Bst * tree, int element`*`);`
Inserts an element into the tree

Arguments: *tree: a Bst*
element: integer to be inserted to the Bst

Returns: *TRUE if inserted successfully, FALSE otherwise*

8.4 bst_search()

gboolean `bst_search (`*`Bst * tree, int element`*`);`
Searches for an element in the tree

Arguments: *tree: Bst to be searched*
element: integer to be searched for

Returns: *TRUE if element found , FALSE otherwise*

8.5 bst_delete ()

gboolean `bst_delete (`*`Bst * tree, int element`*`);`
Deletes the element and balances the tree

Arguments: *tree: Bst to be deleted from*
element: integer to be deleted

Returns: *TRUE if deleted successfully, FALSE otherwise*

8.6 bst_inorder ()

***GArray** ** `bst_inorder (`*`Bst * tree, GArray * traversal`*`);`
Stores the inorder traversal of the tree in the given GArray.

Arguments: *tree : Pointer to the input Bst*
traversal : Pointer to output GArray

Returns*:* *Pointer to GArray with required traversal*

8.7 bst_preoder ()

***GArray** ** `bst_preorder (`*`Bst * tree, GArray * traversal`*`);`
Stores the preorder traversal of the tree in the given GArray.

Arguments: *tree : Pointer to the input Bst*
traversal : Pointer to output GArray

Returns*:* *Pointer to GArray with required traversal*

8.8 bst_postorder()

***GArray** ** *`bst_postorder (Bst * tree, GArray * traversal);`*
Stores the postorder traversal of the tree in the given GArray.

Arguments: *tree : Pointer to the input Bst*
traversal : Pointer to output GArray

Returns*:* *Pointer to GArray with required traversal*

9 *Data structure: Complex Numbers*

9.1 complex_add()

COMPLEX `complex_add  (`*`COMPLEX c1, COMPLEX c2`*`);`
The real parts and the imaginary parts are added respectively.

Arguments: *c1: a COMPLEX number*
c2: another COMPLEX number

Returns: *sum of c1 and c2*

9.2 complex_subtract()

```
COMPLEX complex_subtract  (COMPLEX c1, COMPLEX c2);
```
The real parts and the imaginary parts are subtracted respectively.

Arguments: *c1: a COMPLEX number*
c2: another COMPLEX number

Returns: *c2 subtracted from c1*

9.3 complex_multiply()

```
COMPLEX complex_multiply  (COMPLEX c1, COMPLEX c2);
```
Multiplies two complex numbers.

Arguments: *c1: a COMPLEX number*
c2: another COMPLEX number

Returns: *the resultant COMPLEX number of c1 multiplied by c2*

9.4 complex_magnitude()

```
double complex_magnitude  (COMPLEX c1);
```
Returns a float value which is absolute value of the complex number.

Arguments: *c1:a COMPLEX number*

Returns: *float value which is absolute value of the COMPLEX number*

9.5 complex_argument()

```
double complex_argument  (COMPLEX c1);
```
Returns a float value which is argument of the COMPLEX number.

Arguments: *c1: a COMPLEX number*

Returns: *float value which is argument of the COMPLEX number*

9.6 complex_init()

```
void complex_init (COMPLEX *Ptr);
```
Initializes the COMPLEX number which Ptr is pointing to.

Arguments: *Ptr: pointer to the COMPLEX number to be initialized*
Returns*:* *none*

9.7 complex_assign()

void `complex_assign (`*`COMPLEX *fPtr, double real, double img`*`);`
Assigns the corresponding members of the COMPLEX with the given values.

Arguments: *fptr: pointer to the COMPLEX number to be initialized with real and img*

Returns*:* *none*

9.8 complex_print()

void `complex_print (`*`COMPLEX c`*`);`
The COMPLEX c is printed in the format real + i imaginary.

Arguments: *c: COMPLEX number to be printed*

Returns*:* *none*

10 Data structure: Graphs

10.1 graph_min_distance()

int `graph_min_distance (`*`int dist[], bool visited[], int v`*`);`
Find the vertex with minimum distance value, from the set of vertices not yet included in shortest path tree

Arguments: *dist : the distance array*
visited : the array which contains the set of visited vertices
v : number of vertices in the graph

Returns*: Returns the integer representation of vertex with minimum distance value.*

10.2 graph_dijkstra()

Path * `graph_dijkstra (`*`Graph * g, int src`*`);`
Function that implements Dijkstra's single source shortest path algorithm for a graph represented using adjacency matrix representation

Arguments: *g : pointer to input Graph*
src : integer corresponding to the source vertex

Returns*:* *pointer to the Path calculated.*

10.3 graph_new()

***Graph** ** graph_new (*int V, int E, int d*);*
Creates a graph with V vertices, E edges and allocates space for edge list & adjacency matrix

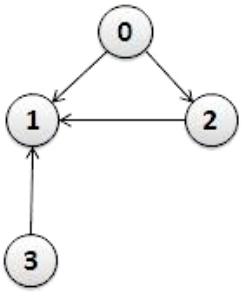
Arguments: *V : number of vertices for which space is to be allocated*
E : number of edges for which space is to be allocated
d : 1 if graph is directed, 0 if undirected

***Returns**:* *Pointer the new Graph*

10.4 graph_file_insert()

***int** graph_file_insert (Graph * g, const char * filename);*
Inserts data from file(adjacency matrix representation) to graph

Arguments: *g : Pointer to Graph to which data is to be loaded*
filename : name of the file which contains the adjacency matrix corresponding to g



For the given graph, the contents of the input file would be:

0 1 1 0
0 0 0 0
0 1 0 0
0 1 0 0

***Returns**:* *1 if insert is successful, 0 otherwise.*

10.5 graph_find()

```
int graph_find (subset s[], int i);
```
Function to find set of an element i (uses path compression technique)

Arguments: *s : the subset array storing heads of every vertex*
i : integer corresponding to vertex whose head is to be found

Returns*:* *integer corresponding to the head of the subset*

10.6 graph_union()

```
void graph_union (subset  s[], int x, int y);
```
Function that does union of two sets of x and y (uses union by rank)

Arguments: *s : the subset array storing heads of every vertex*
x : integer corresponding to the head of one set
y : integer corresponding to the head of other set

Returns*:* *void*

10.7 edge_comparator()

```
int edge_comparator (const void* a, const void* b);
```
Compare two edges according to their weights. Used in qsort() for sorting an array of edges.

Arguments: *a: pointer to the former edge in sequence*
b: pointer to the later edge in sequence

Returns*:* *negative value if a < b ; zero if a = b ; positive value if a > b*

10.8 graph_mst_kruskal()

```
Mst * graph_mst_kruskal (Graph * graph);
```
Function to construct MST using Kruskal's algorithm

Arguments: *graph : pointer to input Graph*

Returns*:* *Pointer to an Mst of the provided Graph.*

11 Data structure: Hashtables

11.1 g_hash_table_new()

```
GHashTable* g_hash_table_new (GHashFunc hash_func, GEqualFunc key_equal_func);
```
Creates a new GHashTable with the given hash function and comparison function.

Arguments: *hash_func : a function to create a hash value from a key*
key_equal_func : a function to check two keys for equality

Returns: *Pointer to* new GHashTable

11.2 g_str_hash()

```
guint g_str_hash (gconstpointer v);
Converts the string v to a hash value.
```

Arguments: *v: the string key to be hashed*

Returns: *a hash value corresponding to the key*

11.3 g_hash_table_insert()

```
gboolean g_hash_table_insert (GHashTable *hash_table, gpointer key, gpointer
value);
```
Inserts a key and value into the given GHashTable.

Arguments: *hash_table: a GHashtable*
key: key to be inserted
value: value to be associated with the key

Returns: *TRUE if key does not exist yet*

11.4 g_hash_table_remove()

```
gboolean g_hash_table_remove (GHashTable *hash_table, gconstpointer key);
```
Removes the key-value pair associated with the key k from the GHashTable hash.

Arguments: *hash_table: a GHashtable*
key: key to be removed

Returns: *TRUE if key was found and removed*

11.5 g_hash_table_contains()

```
gboolean g_hash_table_contains (GHashTable *hash_table, gconstpointer key);
```

Checks if key k is in GHashTable hash.

Arguments: *hash_table: a GHashtable*
key: key to be removed

Returns: *TRUE if key is in the hashtable, FALSE otherwise*

11.6 g_hash_table_lookup()

```
gpointer g_hash_table_lookup (GHashTable *hash_table, gconstpointer key);
```

Returns a pointer to the key k in the GHashTable hash.

Arguments: *hash_table: a GHashtable*
key: key to look up

Returns: *the associated value, or NULL if the key is not found*

11.7 g_hash_table_size()

```
guint g_hash_table_size (GHashTable *hash_table);
```

Returns the size of the GHashTable hash.

Arguments: *hash_table: a GHashtable*

Returns*:* *the number of key-value pairs in the GHashTable*

11.8 g_hash_table_foreach()

```
void g_hash_table_foreach (GHashTable *hash_table, GHFunc func, gpointer
user_data);
```

Calls the given function for each of the key/value pairs in the GHashTable

Arguments: *hashtable : pointer to input GHashTable*
func : the function to call for each key/value pair
user_data : user data to pass to the function

Returns*:* *none*

11.9 hash_print()

void `hash_print (`*`gpointer key, gpointer val, gpointer user_data`*`);`
Utility function to print a key-value pair in the format given by user_data.

Arguments: *key: the key to be printed*
value: value to be printed
user_data: C string with the format in which the key-value pair is to be printed.

Returns: *none*

11.10 hash_print_file()

void `hash_print_file (`***gpointer*** *key,* ***gpointer*** *val,* ***gpointer*** *user_data*`);`
Utility function to print a key-value pairs to file represented by user_data.

Arguments: *key: the key to be printed*
value: value to be printed
user_data: C string containing name of the file.

Returns: *none*

11.11 g_hash_table_destroy()

void `g_hash_table_destroy (`*`GHashTable *hash_table`*`);`
Destroys all keys and values in the GHashTable.

Arguments: *hash_table: pointer to the GHashTable to be destroyed*

Returns*:* *none*

12 *Data structure: Queues*

12.1 queue_new()

queue `queue_new ();`
Allocates space for a new queue

Arguments: *none*

Returns: *A newly created queue*

12.2 queue_enqueue()

```
void queue_enqueue (queue  q, char * s);
```
The element is inserted into queue

Arguments: *q: the queue to be inserted into*
s: the string to be inserted in the queue

Returns: *none*

12.3 queue_dequeue()

```
char * queue_dequeue (queue q);
```
Dequeues an element and returns it

Arguments: *q: the queue to be dequeued from*

Returns: *pointer to the string that is dequeued*

12.4 queue_show_head()

```
char * queue_show_head (queue q);
```
Returns head of queue

Arguments: *q: a queue*

Returns: *pointer to the head of the queue*

12.5 queue_show_tail()

```
char * queue_show_tail (queue q);
```
Returns tail of the queue

Arguments: *q: a queue*

Returns: *pointer to the tail of the queue*

12.6 queue_length()

```
int queue_length (queue q);
```
Returns length of queue

Arguments: *q: a queue*

Returns: *length of queue*

13 Data structure: Singly Linked Lists

13.1 g_slist_append()

```
GSList* g_slist_append (GSList *list, gpointer data);
```
Adds an element into the end of a GSList.

Arguments: *list: pointer to a GSList*
data: pointer to the data to be inserted

Returns: *pointer to a GSList formed after appending data to list*

13.2 g_slist_length()

```
guint g_slist_length (GSList *list);
```
Returns the length of the GSList represented by list.

Arguments: *list: pointer to GSList*

Returns: *length of the GSList*

13.3 g_slist_sort()

```
GSList* g_slist_sort (GSList *list, GCompareFunc compare_func);
```
Sorts the given string according to the given comparison function.

Arguments: *list : pointer to the input Slist*
compare_func : the comparison function used to sort the GSList.
This function is passed the data from two elements of the GSList and should return 0 if they are equal, a negative value if the first element comes before the second, or a positive value if the first element comes after the second.

Returns*:* *pointer to the sorted GSList*

14 Data Structure: Sparse Matrix

14.1 sm_create_matrix()

```
Sparsemat * sm_create_matrix (int maxRow, int maxCol);
```
Creates a new sparse matrix with given number of rows and columns

Arguments: *maxRow: number of rows in the sparse matrix*
maxCol: number of columns in the sparse matrix

Returns: *pointer to a newly created sparse matrix*

14.2 sm_get_element()

gboolean `sm_get_element (`*`Sparsemat * smat, int row, int col, int * value`*`);`
Get value of the element at given row and column. Returns FALSE if element not present in matrix.

Arguments: *smat : Pointer to input Sparsemat*
row : row of the element to be found
col : column of element to be found
value : pointer to location where value of element is to be stored

Returns*:* *TRUE if element exists in sparsematrix, FALSE otherwise.*

14.3 sm_set_element()

gboolean `sm_set_element (`*`Sparsemat * smat, int row, int col, int el`*`);`
Sets the element at given row and column to the given value.

Arguments: *smat: pointer to a Sparsemat*

Returns: *TRUE if element set successfully, FALSE otherwise.*

14.4 sm_search_element()

gboolean `sm_search_element (`*`Sparsemat * smat, int el, element * e`*`);`
Searches for an element and stores the resultant row and column in the given element.

Arguments*:* *smat: Pointer to input Sparsemat*
el: Integer representing search key
e: Pointer to location where result is stored. The result is stored in the data structure `element`*(which consists of integers* `row`*,* `column` *and* `value`*).*

Returns*:* *TRUE if element found, FALSE otherwise*

14.5 sm_print_matrix()

void `sm_print_matrix (`*`Sparsemat * smat`*`);`
Prints the entire sparse matrix.

Arguments: *smat: pointer to a Sparsemat*

Returns: *none*

14.6 sm_add_matrix()

***Sparsemat** ** `sm_add_matrix (`*`Sparsemat * sm1, Sparsemat * sm2`*`);`
Adds two compatible sparse matrices and returns the resultant matrix.

Arguments: *sm1: pointer to a Sparsemat*
sm2: pointer to another Sparsemat

Returns: *pointer to resultant Sparsemat of sm1+sm2*

14.7 sm_transpose()

***Sparsemat** ** `sm_transpose (`*`Sparsemat * smat`*`);`
Returns the transpose of the given sparse matrix.

Arguments: *smat: pointer to a Sparsemat*

Returns: *pointer to transpose of Sparsemat*

14.8 sm_trace_matrix()

gboolean `sm_trace_matrix (S`*`parsemat * smat, int * sum`*`);`
Stores the trace of the sparse matrix in the variable represented by `sum`*. Returns FALSE if sparse matrix is not a square matrix.*

Arguments: *smat: pointer to a Sparsemat*
sum: pointer to location where trace of Sparsemat is stored

Returns: *TRUE if smat is a square matrix, FALSE otherwise*

14.9 sm_scale_matrix()

***Sparsemat** ** `sm_scale_matrix (`*`Sparsemat * smat, int scale_factor`*`);`
Returns a matrix scaled to the given scale factor.

Arguments: *smat: pointer to a Sparsemat*
scale_factor: factor by which the Sparsemat is to be scaled

Returns: *pointer to the resultant Sparsemat*

15 Data Structure: Strings

15.1 g_string_new()

`Gstring* g_string_new (const gchar *init);`
Creates a new GString, initialized with the given string.

Arguments: *the initial text to copy into the string, or NULL to start with an empty string.*

Returns: *the new Gstring*

15.2 g_string_append()

`GString* g_string_append (GString *string, const gchar *val);`
Adds a string onto the end of a GString, expanding it if necessary.

Arguments: *string: a GString*
val: the string to append onto the end of string

Returns: *the new string*

15.3 g_strstr_len()

`gchar* g_strstr_len (const gchar *haystack, gssize haystack_len, const gchar *needle);`
Searches the string `haystack` *for the first occurrence of the string* `needle` *, limiting the length of the search to* `haystack_len`*.*

Arguments: *haystack : input string*
haystack_len : the maximum length of haystack . Note that :1 is a valid length, if haystack is null:terminated, meaning it will search through the whole string.
needle : the string to search for

Returns*: a pointer to the found occurrence, or NULL if not found.*

15.4 g_strdup()

`gchar* g_strdup (const gchar *str);`
Duplicates the given string.

Arguments: *str: the string to be duplicated*

Returns: *a newly-allocated copy of str*

15.5 g_strtod()

```
gdouble g_strtod (const gchar *nptr, gchar **endptr);
```
Converts a string to a gdouble value.

Arguments: *nptr: the string to convert to a duplicate value*
endptr: if non-NULL, it returns the character after the last character used in the conversion.

Returns: *the gdouble value*

15.6 g_str_equal()

```
gboolean g_str_equal (gconstpointer v1, gconstpointer v2);
```
Compares two strings for byte-by-byte equality and returns TRUE if they are equal.

Arguments: *v1: a key*
v2: a key to be compared with v1

Returns: *TRUE if equal, FALSE otherwise*

15.7 g_strsplit()

```
gchar** g_strsplit (const gchar *string, const gchar *delimiter, gint max_tokens);
```
Splits string into an array of strings.

Arguments: *string : A string to split*
delimiter : a string which specifies the places at which to split the string. The delimiter is not included in any of the resulting strings, unless max_tokens is reached.
max_tokens : the maximum number of pieces to split string into. If this is less than 1, the string is split completely.

Returns*:* a newly allocated NULL-terminated array of strings.

15.8 g_ascii_strcasecmp()

```
gint g_ascii_strcasecmp (const gchar *s1, const gchar *s2);
```
Compares the given two strings.

Arguments: *s1: string to compare with s2*
s2: string to compare with s1

Returns: *0 if the strings match, a negative value if s1 < s2 , or a positive value if s1 > s2*