Snake

Généré par Doxygen 1.8.11

Table des matières

Chapitre 1

Index des classes

1.1 Liste des classes

Liste des classes, structures, unions et interfaces avec une brève description :

Chapitre 2

Index des fichiers

2.1 Liste des fichiers

Liste de tous les fichiers avec une brève description :

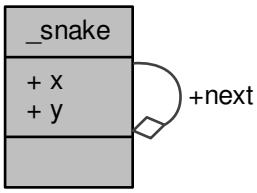
Chapitre 3

Documentation des classes

3.1 Référence de la structure _snake

`#include <snakelib.h>`

Graphe de collaboration de _snake :



Attributs publics

- int x
- int y
- struct _snake * next

3.1.1 Documentation des données membres

3.1.1.1 struct _snake* _snake : :next

3.1.1.2 int _snake : :x

3.1.1.3 int _snake : :y

La documentation de cette structure a été générée à partir du fichier suivant :

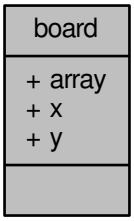
- INCLUDE/snakelib.h

3.2 Référence de la structure board

int **array and int size tableau 2d et sa taille

`#include <snakelib.h>`

Graphe de collaboration de board :



Attributs publics

- char ** array
- int x
- int y

3.2.1 Description détaillée

int **array and int size tableau 2d et sa taille

3.2.2 Documentation des données membres

3.2.2.1 char** board : :array

3.2.2.2 int board : :x

3.2.2.3 int board : :y

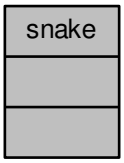
La documentation de cette structure a été générée à partir du fichier suivant :

- INCLUDE/snakelib.h

3.3 Référence de la structure snake

linked list avec deux entiers x et y représente chaque élément du snake

Graphe de collaboration de snake :



3.3.1 Description détaillée

linked list avec deux entiers x et y représente chaque élément du snake

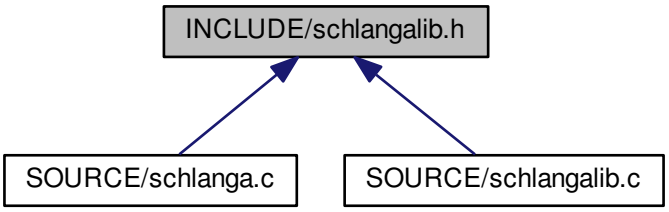
La documentation de cette structure a été générée à partir du fichier suivant :
- INCLUDE/snakelib.h

Chapitre 4

Documentation des fichiers

4.1 Référence du fichier INCLUDE/schlangalib.h

Ce graphe montre quels fichiers incluent directement ou indirectement ce fichier :



Fonctions

— void schlanga_ia (board ∗b, const snake ∗s, snake ∗c, int len)

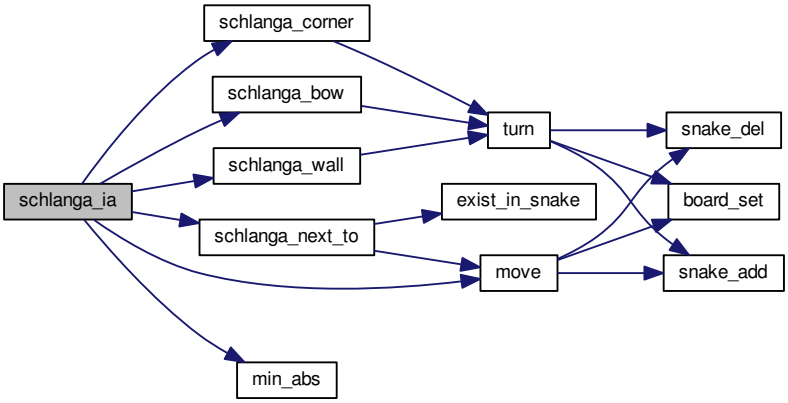
4.1.1 Documentation des fonctions

4.1.1.1 void schlanga_ia (board ∗ *b,* const snake ∗ *s,* snake ∗ *c,* int *len*)

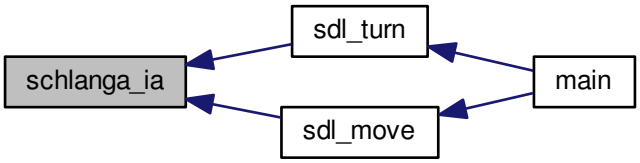
```
437 {
438     static int code = CODE_NULL;
439     static int count = 0;
440     static int corner = 0;
441     int x_board = 1;
442     int y_board = 1;
443 
444     int dx_board = b->x - 2;
445     int dy_board = b->y - 2;
446 
447     int schlanga_x = (*c)->x;
448     int schlanga_y = (*c)->y;
449 
```

```
450     int schlanga_dir_x = (*c)->x - (*c)->next->x;
451     int schlanga_dir_y = (*c)->y - (*c)->next->y;
452 
453     if(schlanga_next_to(b,s,c))
454     {
455         code = CODE_NEXT_TO;
456         return;
457     }
458     if(schlanga_corner(b,c))
459     {
460         code = CODE_CORNER;
461         corner = len/2-2;
462         return;
463     }
464     if(corner)
465     {
466         if(schlanga_bow(b,s,c))
467         {
468             corner = 0;
469             code = CODE_BOW;
470         }
471         else
472         {
473             move(b,c,'&');
474             --corner;
475         }
476         return;
477     }
478     if(code != CODE_WALL)
479     {
480         if(schlanga_wall(b,c))
481         {
482             code = CODE_WALL;
483             unsigned m = 0;
484             if(schlanga_dir_x == 0)
485             {
486                 m = min_abs(schlanga_x - x_board, schlanga_x - dx_board);
487             }
488             if(schlanga_dir_y == 0)
489             {
490                 m = min_abs(schlanga_y - y_board, schlanga_y - dy_board);
491             }
492             if(m != 0)
493             {
494                 count = rand() % m;
495             }
496             return;
497         }
498     }
499     else
500     {
501         if(count == 0)
502         {
503             code = CODE_NULL;
504         }
505         else
506         {
507             if(schlanga_bow(b,s,c))
508             {
509                 code = CODE_BOW;
510                 return;
511             }
512             else
513             {
514                 move(b,c,'&');
515                 --count;
516                 return;
517             }
518         }
519     }
520     if(schlanga_bow(b,s,c))
521     {
522         code = CODE_BOW;
523         return;
524     }
525     move(b,c,'&');
526 }
```

Voici le graphe d'appel pour cette fonction :



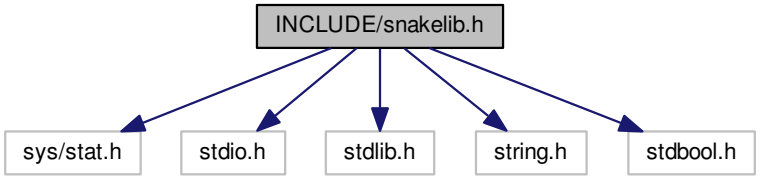
Voici le graphe des appelants de cette fonction :



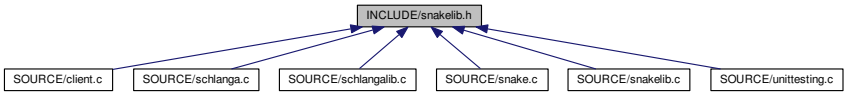
4.2 Référence du fichier INCLUDE/snakelib.h

```
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
```

Graphe des dépendances par inclusion de snakelib.h :



Ce graphe montre quels fichiers incluent directement ou indirectement ce fichier :



Classes

- struct board
  *int **array and int size tableau 2d et sa taille*
- struct _snake

Définitions de type

- typedef struct board board
- typedef struct _snake _snake
- typedef _snake * snake

Fonctions

- int board_size_x (board const *b)
  taille x du plateau
- int board_size_y (board const *b)
- board board_init (int x, int y)
- void board_set (board *brdin, int x, int y, char value)
- char board_get (board const *brdin, int x, int y)
  return la valeur
- void board_free (board *brdin)
  liberer la memoire occupé par un plateau
- void board_print (const board *brdin)
  pour afficher un plateau
- void board_pxmap (const board *brdin, char *foldername, int filename, int zoom)
  créer une image ppm du tableau pour mieux voir et interpreter nos fonctions de teste
- void snake_add (snake *snkin, int x, int y)
  un snake est équivalent à une file d'attente (FIFO « first in, first out »), cette fonction permet d'ajouter à l'entête de la liste
- void snake_addl (snake *snkin, int x, int y)
- void snake_getl (snake *snkin, int *x, int *y)
- void snake_del (snake *snkin, int *x, int *y)

un snake est équivalent à une file d'attente (FIFO « first in, first out »), cette fonction permet de supprimer au queue de la liste

- void snake_free (snake *snkin)
  libérer la mémoire
- void snake_print (const snake *snkin)
- void move (board *brdin, snake *snkin, char id)
- void turn (board *brdin, snake *snkin, int drctn, char id)
- void snake_init (board *brdin, snake *snkin, int len, char id)
  init a snake
- bool choc_snake (snake const *s1, snake const *s2)
  test choc entre deux snake
- bool choc_sc (snake const *s1, snake const *s2)
- bool choc_wall (board const *b, snake const *s)
  test choc snake contre le mur
- void board_apple (board *b, int *x, int *y)
- bool snake_apple (snake const *s, int *x, int *y)

4.2.1 Documentation des définitions de type

4.2.1.1 typedef struct _snake _snake

4.2.1.2 typedef struct board board

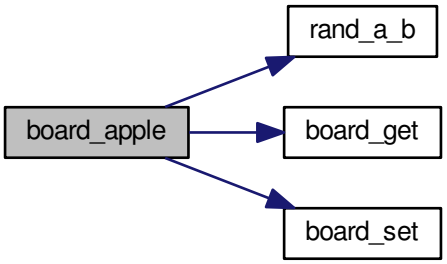
4.2.1.3 typedef _snake* snake

4.2.2 Documentation des fonctions

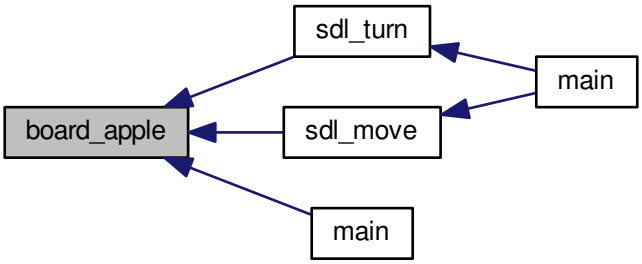
4.2.2.1 void board_apple (board * b, int * x, int * y)

```
512 {
513     bool t = true;
514     while(t)
515     {
516             *x = rand_a_b(1, b->x-2);
517             *y = rand_a_b(1, b->y-2);
518             if(board_get(b,*x,*y) == ' ')
519             {
520                 board_set(b,*x,*y,'$');
521                 t = false;
522             }
523     }
524 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :



4.2.2.2 void board_free (board * *brdin*)

liberer la memoire occupé par un plateau

Paramètres

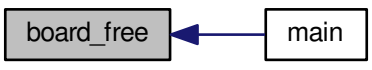
adresse	du plateau à libérer

Renvoie

une fonction de type void

```
91 {
92     int i;
93 
94     for(i = 0; i < brdin->y ; i++)
95     {
96         free(brdin->array[i]);
97         brdin->array[i] = NULL;
98     }
99 
100     free(brdin->array);
101     brdin->array = NULL;
102 }
```

Voici le graphe des appelants de cette fonction :



4.2.2.3 char board_get (board const * *brdin*, int *x*, int *y*)

return la valeur

Paramètres

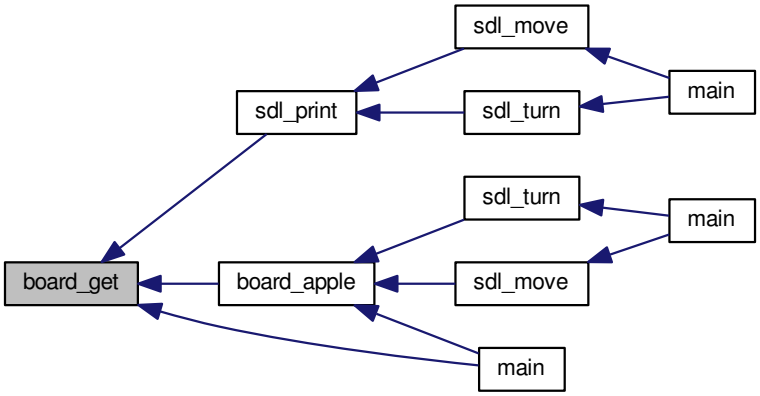
adresse	d'un plateau pour accéder et modifier simplement ses variables
x	et y la case à visiter

Renvoie

char la valueur du point (x,y)

```
70 {
71     return brdin->array[y][x];
72 }
```

Voici le graphe des appelants de cette fonction :



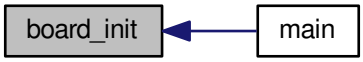
4.2.2.4 board board_init (int *x*, int *y*)

borders values is '#'.

```
22 {
23     board brdout;
24 
25     brdout.x = x;
26     brdout.y = y;
27 
28     int i, j;
29 
30     brdout.array=malloc(y*sizeof(char*));
31     for(i = 0; i < y; i++)
32         brdout.array[i] = malloc(x*sizeof(int));
33 
34 
36     for(i = 0; i < x; i++)
37     {
38         brdout.array[0][i] = '#';
39         brdout.array[y-1][i] = '#';
40     }
41     for(i = 0;i < y; i++)
42     {
43         brdout.array[i][0] = '#';
44         brdout.array[i][x-1] = '#';
```

```
45      }
46
47      for(i = 1; i < y-1; i++)
48          for(j = 1; j < x-1; j++)
49              brdout.array[i][j] = ' ';
50
51      return brdout;
52  }
```

Voici le graphe des appelants de cette fonction :



4.2.2.5 void board_print (const board * *brdin*)

pour afficher un plateau

Paramètres

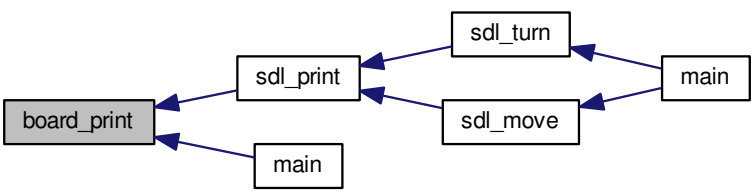
une	adresse d'un plateau concéder constant (droit que pour la lecture) pour éviter de copier le plateau

Renvoie

une fonction de type void

```
113 {
114     int i,j;
115     /* traversing 2d-array */
116     for(i=0 ; i < brdin->y; i++)
117     {
118         for(j=0 ;j < brdin->x; j++)
119         {
120             printf("%c", brdin->array[i][j]);
121         }
122         printf("\n");
123     }
124 }
```

Voici le graphe des appelants de cette fonction :



4.2.2.6 void board_pxmap (const board * *brdin*, char * *foldername*, int *filename*, int *zoom*)

créer une image ppm du tableau pour mieux voir et interpreter nos fonctions de teste

Paramètres

une	adresse d'un plateau concéder constant (droit que pour la lecture) pour éviter de copier le plateau
*char**	foldername : donner un nom au dossier créé
int	filename : numero de la map
int	zoom : pour definir le zoom appliqué a la map (taille d'une case en px = zoom * 1px)

Renvoie

une fonction de type void

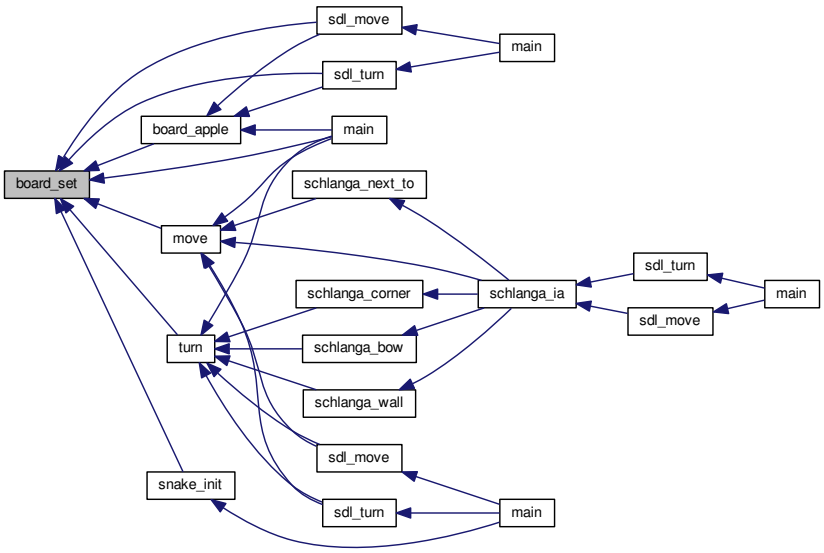
le dossier n'existe pas

```
141 {
142     struct stat dir_stat;
143     if(stat(foldername, &dir_stat) < 0)
145         mkdir(foldername, S_IRWXU);
146 
147 
148     char ffilename[strlen(foldername) + 7];
149     sprintf(ffilename, "%s/%d.ppm", foldername, filename);
150     FILE* fdout = fopen( ffilename, "w");
151     fprintf(fdout, "P3\n%d %d\n255\n", brdin->y * zoom,\
152             brdin->x * zoom);
153 
154 
155     int i,j,l,k;
156     /* traversing 2d-array */
157     for(i = 0; i < brdin->y; i++)
158     {
159      for(l = 0; l < zoom; l++)
160      {
161       for(j = 0; j < brdin->x; j++)
162       {
163        for(k = 0; k < zoom; k++)
164        {
165          if(brdin->array[i][j] == ' ')
166           fprintf(fdout,"175 175 175 ");
167          else if(brdin->array[i][j] == '#')
168           fprintf(fdout,"48 48 48 ");
169          else if(brdin->array[i][j] == '@')
170           fprintf(fdout,"110 11 20 ");
171        }
172       }
173       fprintf(fdout, "\n");
174      }
175      fprintf(fdout, "\n");
176     }
177     fclose(fdout);
178 
179 
180 }
```

4.2.2.7 void board_set (board * *brdin*, int *x*, int *y*, char *value*)

```
65 {
66     brdin->array[y][x] = value;
67 }
```

Voici le graphe des appelants de cette fonction :



4.2.2.8 int board_size_x (board const * *b*)

taille x du plateau

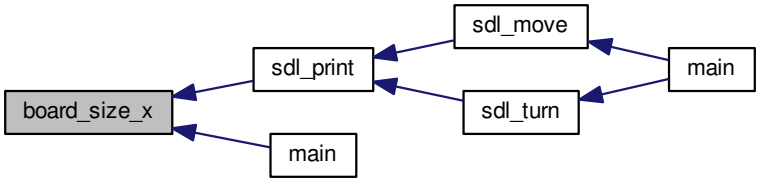
taille y du plateau

Paramètres

adresse	du plateau

```
75 {
76     return b->x;
77 }
```

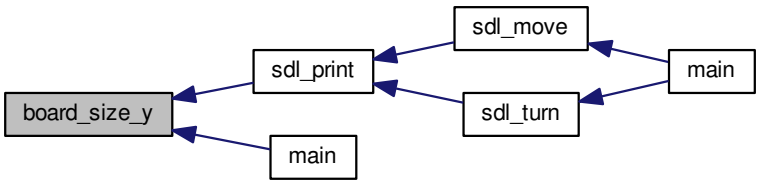
Voici le graphe des appelants de cette fonction :



4.2.2.9 int board_size_y (board const * b)

```
80 {
81     return b->y;
82 }
```

Voici le graphe des appelants de cette fonction :



4.2.2.10 bool choc_sc (snake const * s1, snake const * s2)

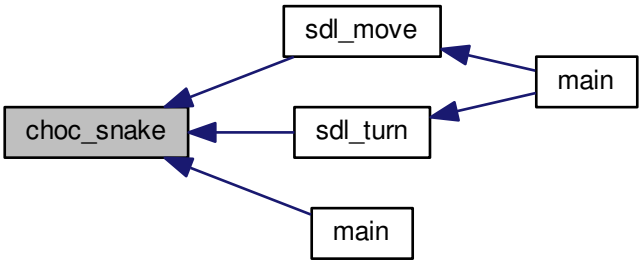
```
490 {
491     if ((*s1)->x == (*s2)->x && (*s1)->y == (*s2)->y)
492         return true;
493     return false;
494 }
```

4.2.2.11 bool choc_snake (snake const * s1, snake const * s2)

test choc entre deux snake

```
478 {
479     snake tmp = (*s2)->next;
480     while(tmp)
481     {
482         if((*s1)->x == tmp->x && (*s1)->y == tmp->y)
483             return true;
484         tmp = tmp->next;
485     }
486     return false;
487 }
```

Voici le graphe des appelants de cette fonction :

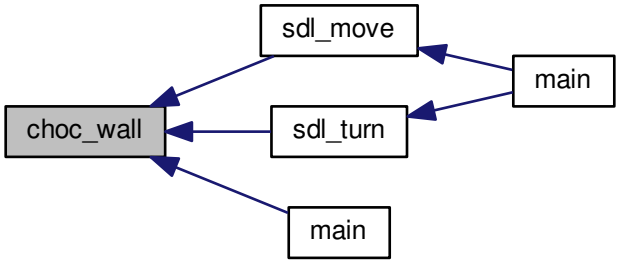


4.2.2.12 bool choc_wall (board const * *b*, snake const * *s*)

test choc snake contre le mur

```
497 {
498     if((*s)->x == 0) return true;
499     if((*s)->x == b->x-1) return true;
500     if((*s)->y == 0) return true;
501     if((*s)->y == b->y-1) return true;
502 
503     return false;
504 }
```

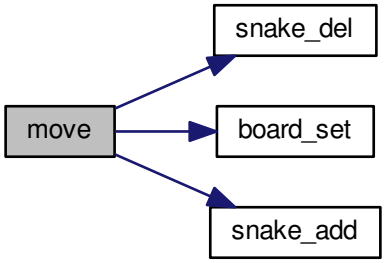
Voici le graphe des appelants de cette fonction :



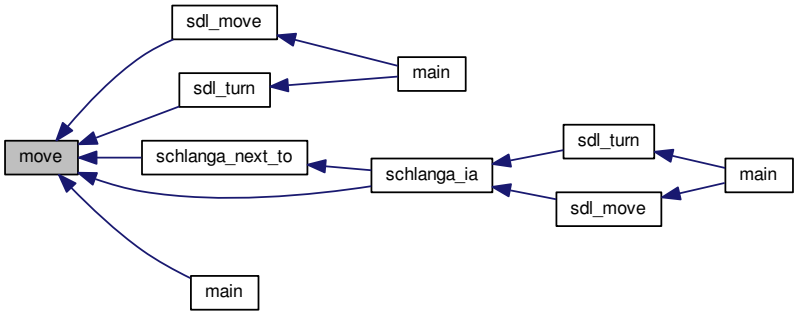
4.2.2.13 void move (board * *brdin*, snake * *snkin*, char *id*)

```
371 {
372     int x,y,dx,dy;
373     snake_del(snkin, &x, &y);
374     board_set(brdin, x, y, ' ');
375     dx = ((*snkin)->x) - ((*snkin)->next->x);
376     dy = ((*snkin)->y) - ((*snkin)->next->y);
377     snake_add(snkin, (*snkin)->x+dx , (*snkin)->y+dy);
378     board_set(brdin, (*snkin)->x ,(*snkin)->y , id);
379 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :



4.2.2.14 void snake_add (snake * *snkin,* int *x,* int *y*)

un snake est équivalent à une file d'attente (FIFO « first in, first out »), cette fonction permet d'ajouter à l'entête de la liste

Paramètres

snake	*snkin : adresse d'un snake
int	x, int y : coordonnée du nouvel élément

Renvoie

fonction de type void

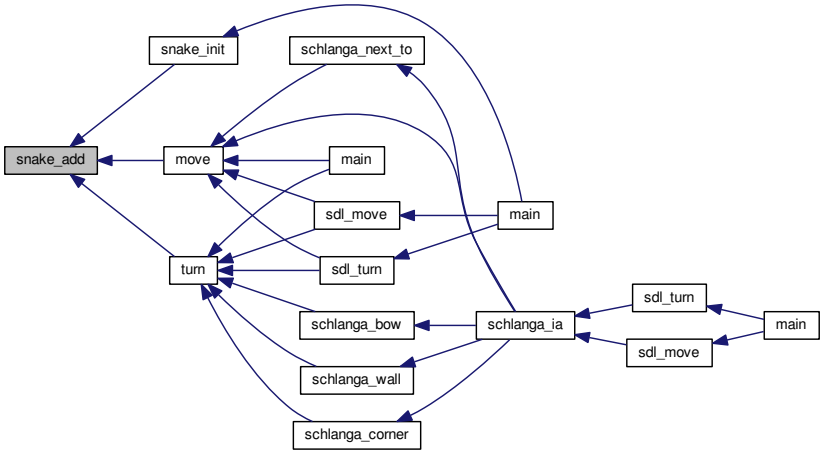
define list output with coord x,y

the next element is list input

edit list input(*input=output)

```
192 {
194     snake snkout=malloc(sizeof(_snake));
195     snkout->x = x;
196     snkout->y = y;
198     snkout->next = *snkin;
200     *snkin = snkout;
201 }
```

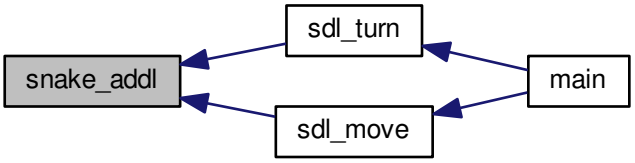
Voici le graphe des appelants de cette fonction :



4.2.2.15 void snake_addl (snake * *snkin*, int *x*, int *y*)

```
262 {
263     snake new = malloc(sizeof(_snake));
264     *new = (_snake)
265     {
266         .x = x,
267         .y = y,
268         .next = NULL,
269     };
270     snake tmp = *snkin;
271     while(tmp->next != NULL)
272     {
273         tmp = tmp->next;
274     }
275     tmp->next = new;
276 }
```

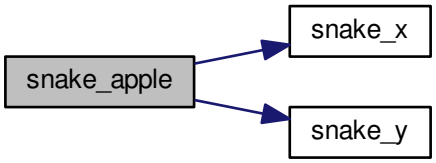
Voici le graphe des appelants de cette fonction :



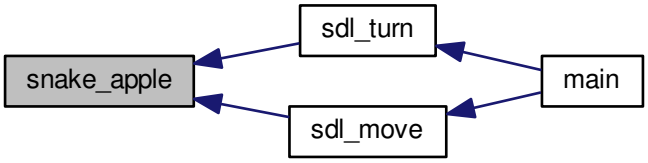
4.2.2.16 bool snake_apple (snake const * s, int * x, int * y)

```
527 {
528     if(snake_x(s) == *x && snake_y(s) == *y) return true;
529     return false;
530 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :



4.2.2.17 void snake_del (snake * snkin, int * x, int * y)

un snake est équivalent à une file d'attente (FIFO « first in, first out »), cette fonction permet de supprimer au queue de la liste

Paramètres

snake	*snkin : adresse d'un snake
snake	int *x, int *y : adresses de deux entiers pour pouvoir récupérer les coordonnées d'élément supprimer

Renvoie

fonction de type void

if the list is empty

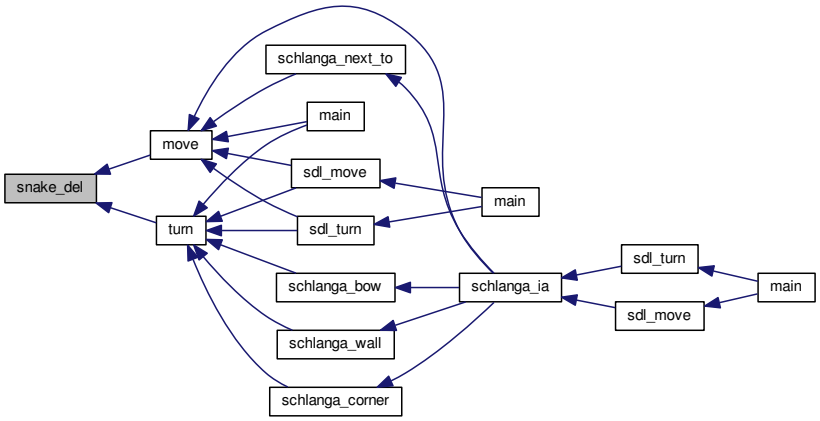
default value x=-1 y=-1

if the list contains only one item

we move along the linked list keeping the last two consecutive element

```
215 {
216     if(*snkin == NULL)
217     {
219         if((x != NULL) && (y != NULL))
220         {
222             *x = -1;
223             *y = -1;
224         }
225     }
226     else if((*snkin)->next == NULL)
227     {
228
230         if((x != NULL) && (y != NULL))
231         {
232             *x = (*snkin)->x;
233             *y = (*snkin)->y;
234         }
235         free((*snkin));
236         *snkin = NULL;
237     }
238     else
239     {
244         snake tmp1 = *snkin;
245         snake tmp2 = *snkin;
246         while(tmp1->next != NULL)
247         {
248             tmp2 = tmp1;
249             tmp1 = tmp1->next;
250         }
251         if((x != NULL) && (y != NULL))
252         {
253             *x = tmp1->x;
254             *y = tmp1->y;
255         }
256         tmp2->next = NULL;
257         free(tmp1);
258     }
259 }
```

Voici le graphe des appelants de cette fonction :



4.2.2.18 void snake_free (snake * *snkin*)

libérer la mémoire

Paramètres

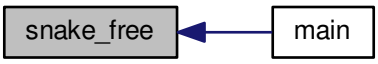
adresse	du snake

Renvoie

fonction de type void

```
309 {
310     snake tmp = *snkin;
311     snake tmpnext;
312 
313     while(tmp != NULL)
314     {
315         tmpnext = tmp->next;
316         free(tmp);
317         tmp = tmpnext;
318     }
319 
320     *snkin = NULL;
321 }
```

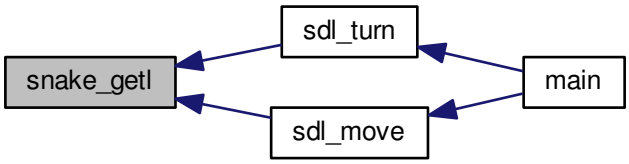
Voici le graphe des appelants de cette fonction :



4.2.2.19 void snake_getl (snake * *snkin*, int * *x*, int * *y*)

```
279 {
280     if(*snkin == NULL)
281     {
282         *x = -1;
283         *y = -1;
284     }
285     else if((*snkin)->next == NULL)
286     {
287         *x = (*snkin)->x;
288         *y = (*snkin)->y;
289     }
290     else
291     {
292         snake tmp = *snkin;
293         while(tmp->next != NULL)
294         {
295             tmp = tmp->next;
296         }
297         *x = tmp->x;
298         *y = tmp->y;
299     }
300 }
```

Voici le graphe des appelants de cette fonction :

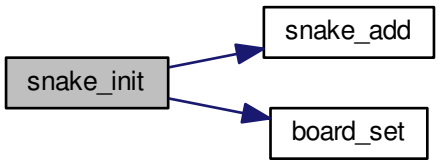


4.2.2.20 void snake_init (board * *brdin*, snake * *snkin*, int *len*, char *id*)

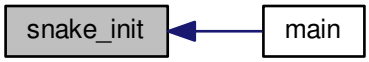
init a snake

```
429 {
430     if(id == '@')
431     {
432 
433         snake_add( snkin, brdin->x / 2, brdin->y - 2 );
434         board_set( brdin, (*snkin)->x, (*snkin)->y, id );
435 
436         int i;
437         for(i = 0; i < len - 1; i++)
438         {
439             snake_add( snkin, (*snkin)->x, (*snkin)->y -1 );
440             board_set( brdin, (*snkin)->x, (*snkin)->y, id );
441         }
442 
443     }
444 
445     if(id == '&')
446     {
447 
448         snake_add( snkin, brdin->x / 2, 1 );
449         board_set( brdin, (*snkin)->x, (*snkin)->y, id );
450 
451         int i;
452         for(i = 0; i < len - 1; i++)
453         {
454             snake_add( snkin, (*snkin)->x, (*snkin)->y +1 );
455             board_set( brdin, (*snkin)->x, (*snkin)->y, id );
456         }
457 
458     }
459 
460     if(id == '+')
461     {
462 
463         snake_add( snkin, 1, brdin->y / 2 );
464         board_set( brdin, (*snkin)->x, (*snkin)->y, id );
465 
466         int i;
467         for(i = 0; i < len - 1; i++)
468         {
469             snake_add( snkin, (*snkin)->x + 1, (*snkin)->y);
470             board_set( brdin, (*snkin)->x, (*snkin)->y, 1 );
471         }
472 
473     }
474 
475 }
```

Voici le graphe d'appel pour cette fonction :



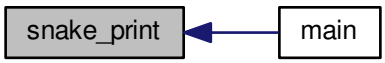
Voici le graphe des appelants de cette fonction :



4.2.2.21 void snake_print (const snake * *snkin*)

```
332 {
333     snake tmp = *snkin;
334     while(tmp != NULL)
335     {
336         printf("[%d:%d]", tmp->x, tmp->y);
337         tmp = tmp->next;
338     }
339     printf("\n");
340 }
```

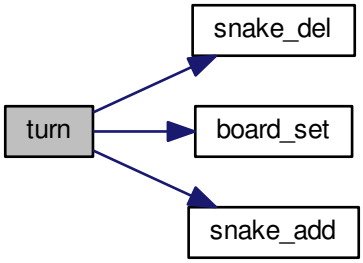
Voici le graphe des appelants de cette fonction :



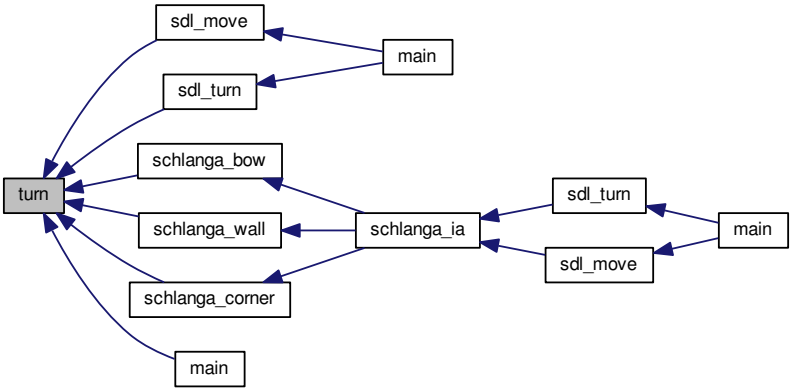
4.2.2.22 void turn (board * *brdin*, snake * *snkin*, int *drctn*, char *id*)

```
394 {
395     int x,y,dx,dy;
396 
397     snake_del(snkin,&x,&y);
398     board_set(brdin,x,y, ' ');
399 
400     dx=((*snkin)->x) - ((*snkin)->next->x);
401     dy=((*snkin)->y) - ((*snkin)->next->y);
402 
403     if(dx == 1)
404     {
405         snake_add(snkin, (*snkin)->x, (*snkin)->y + drctn);
406         board_set(brdin, (*snkin)->x, (*snkin)->y, id);
407     }
408 
409     else if(dx == -1)
410     {
411         snake_add(snkin, (*snkin)->x, (*snkin)->y - drctn);
412         board_set(brdin, (*snkin)->x, (*snkin)->y, id);
413     }
414 
415     else if(dy == 1)
416     {
417         snake_add(snkin,(*snkin)->x-drctn,(*snkin)->y);
418         board_set(brdin,(*snkin)->x,(*snkin)->y,id);
419     }
420 
421     else if(dy == -1)
422     {
423         snake_add(snkin,(*snkin)->x+drctn,(*snkin)->y);
424         board_set(brdin,(*snkin)->x,(*snkin)->y,id);
425     }
426 }
```

Voici le graphe d'appel pour cette fonction :



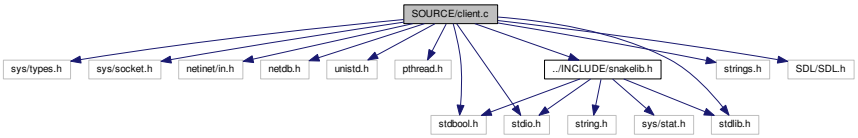
Voici le graphe des appelants de cette fonction :



4.3 Référence du fichier SOURCE/client.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "../INCLUDE/snakelib.h"
#include <SDL/SDL.h>
```

Graphe des dépendances par inclusion de client.c :



Macros

- #define err(msg) {fprintf(stderr, "%s\n", msg) ;exit(1) ;}
- #define PORT 1234
- #define ADDR "localhost"
- #define tabX 51
- #define tabY 31
- #define slen 3

Fonctions

- static int client_socket ()
- void sdl_print ()
- void * sdl_move ()
- void * sdl_turn ()
- int main ()

Variables

- pthread_t threadpt
- pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
- SDL_Surface * E
- SDL_Surface * A
- SDL_Surface * B
- SDL_Surface * C
- SDL_Surface * D
- board b
- snake s
- snake c
- bool X
- char t
- char o
- char r
- char l
- char m
- int n

4.3.1 Documentation des macros

4.3.1.1 #define ADDR "localhost"

4.3.1.2 #define err(*msg*) {fprintf(stderr, "%s\n", msg);exit(1);}

4.3.1.3 #define PORT 1234

4.3.1.4 #define slen 3

4.3.1.5 #define tabX 51

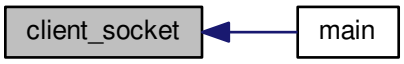
4.3.1.6 #define tabY 31

4.3.2 Documentation des fonctions

4.3.2.1 static int client_socket ( ) [static]

```
38 {
39     int n = socket(AF_INET, SOCK_STREAM, 0);
40     if(0 > n) err("socket()");
41 
42     struct hostent* server_host = gethostbyname(ADDR);
43     if(NULL == server_host) err("gethostbyname()");
44 
45     struct sockaddr_in server_addr;
46     bzero(&server_addr,sizeof(server_addr));
47     server_addr.sin_family = AF_INET;
48     server_addr.sin_port = htons(PORT);
49 
50     bcopy((char *)server_host->h_addr_list[0], (char *)&server_addr.sin_addr.s_addr, server_host->h_length)
    ;
51     if(0 > connect(n, (struct sockaddr*)&server_addr, sizeof(server_addr))) err("connect()");
52     return n;
53 }
```

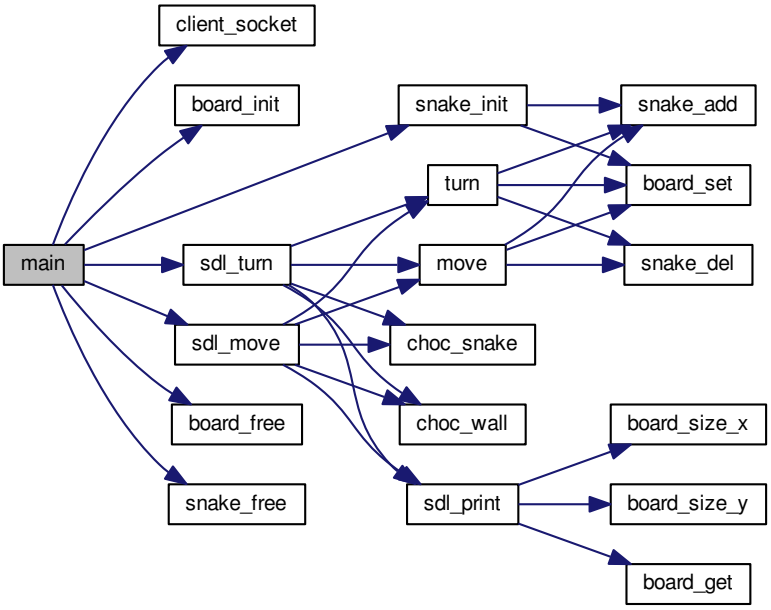
Voici le graphe des appelants de cette fonction :



4.3.2.2 int main ( )

```
204 {
205     n = client_socket();
206     read(n,&t,sizeof(char*));
207     o = (t == '@')? '&' : '@';
208     m = 'm';
209     r = 'r';
210     l = 'l';
211     /* la variable globale du jeu */
212     X = true;
213     /* rand init */
214     srand(time(0));
215     /* init board + snake */
216     b = board_init(tabX,tabY);
217     snake_init(&b,&s,slen,t);
218     snake_init(&b,&c,slen,o);
219     /* init sdl en mode video */
220     SDL_Init(SDL_INIT_VIDEO);
221     /* init surface E avec une taille 20*tabX et 20*tabY */
222     E = SDL_SetVideoMode(20*tabX, 20*tabY, 32, SDL_HWSURFACE);
223     /* creer le reste des surfaces avec une taille de 20*20 px */
224     A = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
225     B = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
226     C = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
227     D = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
228     /* init avec couleur rgb */
229     SDL_FillRect(A, 0, SDL_MapRGB(E->format, 95, 0, 0));
230     SDL_FillRect(B, 0, SDL_MapRGB(E->format, 137, 137, 137));
231     SDL_FillRect(C, 0, SDL_MapRGB(E->format, 47, 47, 47));
232     SDL_FillRect(D, 0, SDL_MapRGB(E->format, 157, 62, 12));
233     /* init barre de fenetre */
234     SDL_WM_SetCaption("Snack", 0);
235     /* nos variabe de thread */
236     pthread_t thread_turn;
237     pthread_t thread_move;
238     /* creer les thread */
239     pthread_create (&thread_turn, 0, &sdl_turn, 0);
240     pthread_create (&thread_move, 0, &sdl_move,0);
241     /* attendre leurs fin */
242     pthread_join(thread_turn, 0);
243     pthread_join(thread_move, 0);
244     /* libere surface */
245     SDL_FreeSurface(E);
246     SDL_FreeSurface(A);
247     SDL_FreeSurface(B);
248     SDL_FreeSurface(C);
249     /* libere board + snake */
250     board_free(&b);
251     snake_free(&s);
252     snake_free(&c);
253     SDL_Quit();
254     return 0;
255 }
```

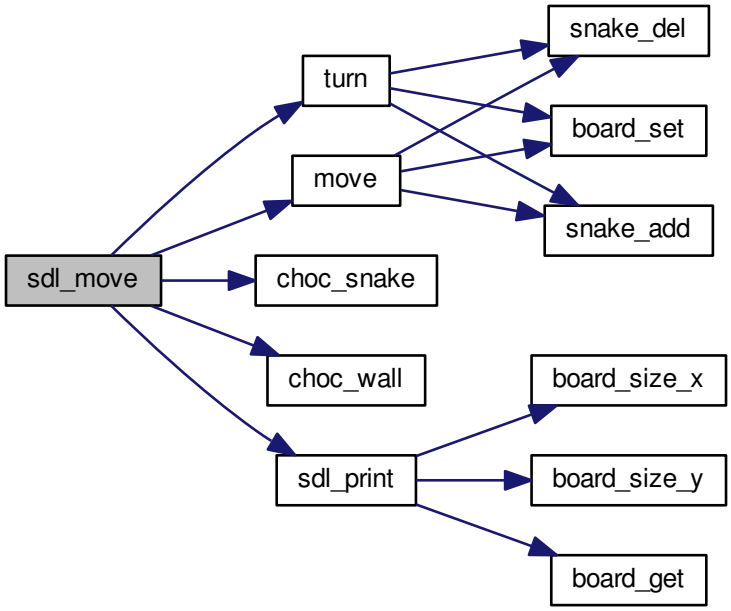
Voici le graphe d'appel pour cette fonction :



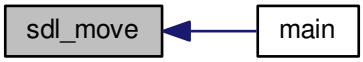
4.3.2.3 void ∗ sdl_move ( )

```
90 {
91     char buff;
92     while(X)
93     {
94         /* on ferme la variable mutex */
95         pthread_mutex_lock(&mutex);
96         /* move snake */
97         move(&b,&s,t);
98         write(n,&m,1);
99         read(n,&buff,sizeof(char*));
100         switch(buff)
101         {
102             case 'm':
103                 move(&b,&c,o);
104                 break;
105             case 'l':
106                 turn(&b,&c,o,-1);
107                 break;
108             case 'r':
109                 turn(&b,&c,o,1);
110                 break;
111         }
112         /* affiche SDL */
113         if ((X = !(choc_snake(&s,&s)) && !(choc_wall(&b,&
    s))))
114             sdl_print();
115         /* libere le mutex pour les autres thread */
116         pthread_mutex_unlock(&mutex);
117         /*u*/sleep (1/*00000*/);
118     }
119     pthread_exit(0);
120 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

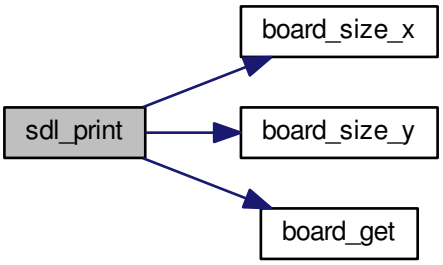


4.3.2.4 void sdl_print ( )

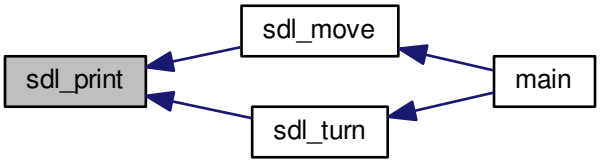
```
56                  {
57     int x,y, dx = board_size_x (&b), dy = board_size_y (&
       b);
58     /* def une position */
59     SDL_Rect p;
60     for (x = 0; x < dx; x++)
61     {
62         for (y = 0; y < dy; y++)
63         {
64             p.x = x * 20 ; p.y = y * 20 ;
65             /* lire dans le plateau et dessiner la surface */
66             switch (board_get (&b,x,y))
67             {
68                 case '@':
69                     SDL_BlitSurface(A, 0, E, &p);
70                     break;
```

```
71                  case ' ':
72                      SDL_BlitSurface(B, 0, E, &p);
73                      break;
74                  case '#':
75                      SDL_BlitSurface(C, 0, E, &p);
76                      break;
77                  case '&':
78                      SDL_BlitSurface(D, 0, E, &p);
79                      break;
80              }
81          }
82      }
83      /* reafficher un nouvelle ecran */
84      SDL_Flip(E);
85  }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

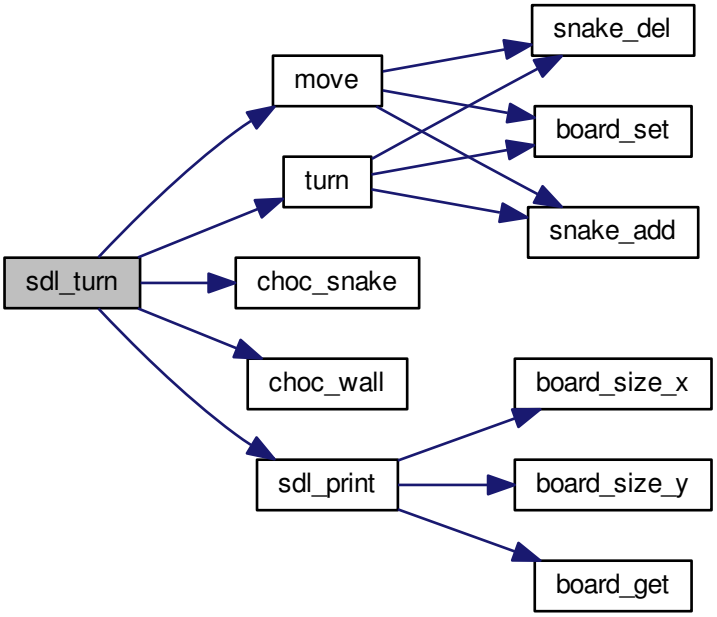


4.3.2.5 void * sdl_turn ( )

```
123 {
124     SDL_Event event;
125     char buff;
126     while (X)
127     {
128     SDL_WaitEvent(&event);
129         switch (event.type)
130         {
131             case SDL_QUIT:
132                 X = false;
```

```
133                 break;
134             case SDL_KEYDOWN:
135                 switch (event.key.keysym.sym)
136                 {
137                     case SDLK_ESCAPE:
138                         X = false;
139                         break;
140                     case SDLK_RIGHT:/* key right */
141                     case SDLK_KP3:/* keypad 3 */
142                     case SDLK_KP9:/* keypad 3 */
143                     case 'e':/* code ascii */
144                         /* on ferme la variable mutex */
145                         pthread_mutex_lock(&mutex);
146                         /* turn snake */
147                         turn(&b,&s,1,t);
148                         write(n,&r,1);
149                         read(n,&buff,sizeof(char*));
150                         switch(buff)
151                         {
152                             case 'm':
153                                 move(&b,&c,o);
154                                 break;
155                             case 'l':
156                                 turn(&b,&c,o,-1);
157                                 break;
158                             case 'r':
159                                 turn(&b,&c,o,1);
160                                 break;
161                         }
162                         /* affiche SDL */
163                         if ((X = !(choc_snake(&s,&s)) && !(
    choc_wall(&b,&s))))
164                             sdl_print();
165                         pthread_mutex_unlock(&mutex);
166                         break;
167                     case SDLK_LEFT:/* key left */
168                     case SDLK_KP1:/* keypad 1*/
169                     case SDLK_KP7:/* keypad 7*/
170                     case 'a':/* code ascii */
171                         /* on ferme la variable mutex */
172                         pthread_mutex_lock(&mutex);
173                         /* turn snake */
174                         turn(&b,&s,-1,t);
175                         write(n,&l,1);
176                         read(n,&buff,sizeof(char*));
177                         switch(buff)
178                         {
179                             case 'm':
180                                 move(&b,&c,o);
181                                 break;
182                             case 'l':
183                                 turn(&b,&c,o,-1);
184                                 break;
185                             case 'r':
186                                 turn(&b,&c,o,1);
187                                 break;
188                         }
189                         /* affiche SDL */
190                         if ((X = !(choc_snake(&s,&s)) && !(
    choc_wall(&b,&s))))
191                             sdl_print();
192                         /* libere le mutex pour les autres thread */
193                         pthread_mutex_unlock(&mutex);
194                         break;
195                     default:/* ne fait rien */break;
196                 }
197             default:/* ne fait rien */break;
198         }
199     }
200     pthread_exit(0);
201 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

4.3.3 Documentation des variables

4.3.3.1 SDL_Surface * A

4.3.3.2 SDL_Surface * B

4.3.3.3 board b

4.3.3.4 SDL_Surface * C

4.3.3.5 snake c

4.3.3.6 SDL_Surface * D

4.3.3.7 SDL_Surface* E

4.3.3.8 char l

4.3.3.9 char m

4.3.3.10 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER

4.3.3.11 int n

4.3.3.12 char o

4.3.3.13 char r

4.3.3.14 snake s

4.3.3.15 char t

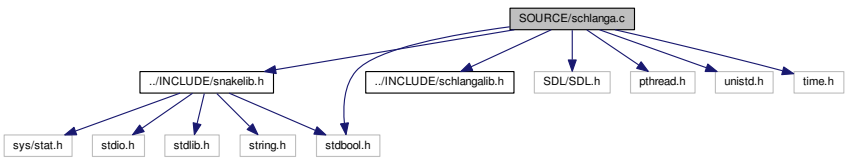
4.3.3.16 pthread_t threadpt

4.3.3.17 bool X

4.4 Référence du fichier SOURCE/schlanga.c

```
#include "../INCLUDE/snakelib.h"
#include "../INCLUDE/schlangalib.h"
#include <SDL/SDL.h>
#include <pthread.h>
#include <stdbool.h>
#include <unistd.h>
#include <time.h>
```

Graphe des dépendances par inclusion de schlanga.c :



Macros

- #define BOARD_SIZE_X 68
- #define BOARD_SIZE_Y 33
- #define SNAKE_LEN 12
- #define SLEEP 105000

Fonctions

— void * sdl_turn ()
— void * sdl_move ()
— void sdl_print ()
— int main ()

Variables

— pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
— bool X
— SDL_Surface * E
— SDL_Surface * A
— SDL_Surface * B
— SDL_Surface * C
— SDL_Surface * D
— board b
— snake s
— snake c

4.4.1 Documentation des macros

4.4.1.1 #define BOARD_SIZE_X 68

4.4.1.2 #define BOARD_SIZE_Y 33

4.4.1.3 #define SLEEP 105000

4.4.1.4 #define SNAKE_LEN 12

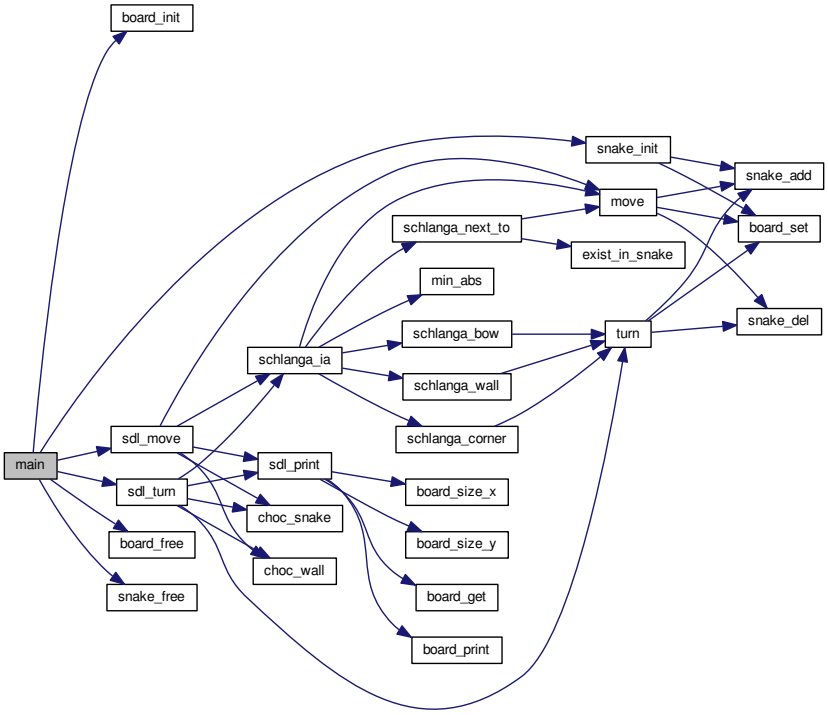
4.4.2 Documentation des fonctions

4.4.2.1 int main ( )

```
26 {
27     /* rand init */
28     srand(time(0));
29     X = true;
30     /* init board + snake */
31     b = board_init(BOARD_SIZE_X,BOARD_SIZE_Y);
32     snake_init(&b,&s,SNAKE_LEN,'@');
33     snake_init(&b,&c,SNAKE_LEN,'&');
34     /* init sdl en mode video */
35     SDL_Init(SDL_INIT_VIDEO);
36     /* init surface E avec une taille 20*BOARD_SIZE_X et 20*BOARD_SIZE_Y */
37     E = SDL_SetVideoMode(20*BOARD_SIZE_X, 20*BOARD_SIZE_Y, 32, SDL_HWSURFACE);
38     /* creer le reste des surfaces avec une taille de 20*20 px */
39     A = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
40     B = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
41     C = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
42     D = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
43     /* init avec couleur rgb */
44     SDL_FillRect(A, 0, SDL_MapRGB(E->format, 95, 0, 0));
45     SDL_FillRect(B, 0, SDL_MapRGB(E->format, 137, 137, 137));
46     SDL_FillRect(C, 0, SDL_MapRGB(E->format, 47, 47, 47));
47     SDL_FillRect(D, 0, SDL_MapRGB(E->format, 157, 62, 12));
48     /* init barre de fenetre */
49     SDL_WM_SetCaption("Snack", 0);
50     /* nos variabe de thread */
51     pthread_t thread_turn;
52     pthread_t thread_move;
53     /* creer les thread */
54     pthread_create(&thread_turn, 0, &sdl_turn, 0);
55     pthread_create(&thread_move, 0, &sdl_move,0);
56     /* attendre leurs fin */
```

```
57      pthread_join(thread_turn, 0);
58      pthread_join(thread_move, 0);
59      /* libere surface */
60      SDL_FreeSurface(E);
61      SDL_FreeSurface(A);
62      SDL_FreeSurface(B);
63      SDL_FreeSurface(C);
64      /* libere board + snake */
65      board_free(&b);
66      snake_free(&s);
67      snake_free(&c);
68      SDL_Quit();
69      return 0;
70  }
```

Voici le graphe d'appel pour cette fonction :

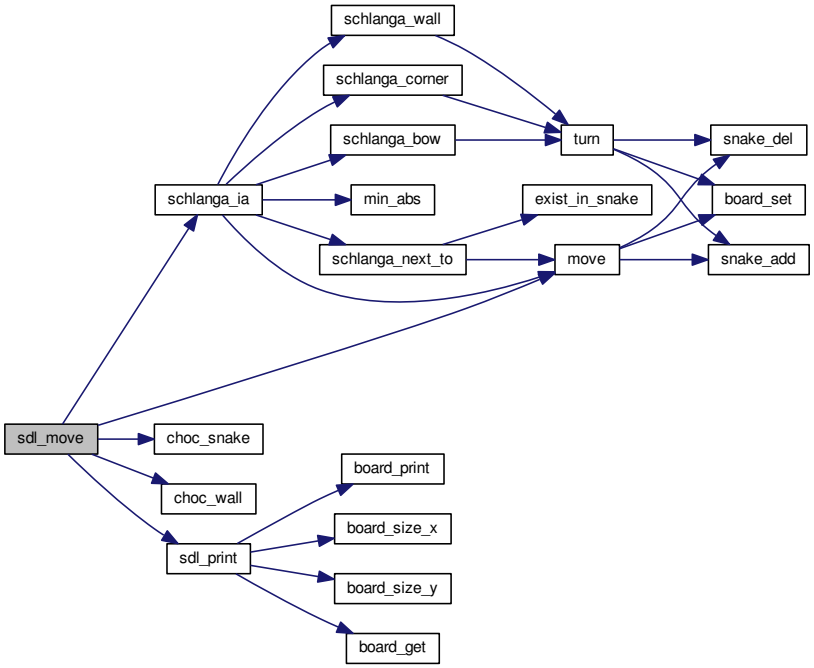


4.4.2.2 void* sdl_move ( )

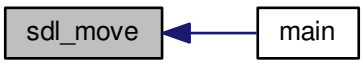
```
145 {
146     while(X)
147     {
148             /* on ferme la variable mutex */
149             pthread_mutex_lock(&mutex);
150             /* move snake */
151             move(&b,&s,'@');
152             schlanga_ia(&b,&s,&c,SNAKE_LEN);
153             if ((X = !(choc_snake(&s,&s))\
154                 && !(choc_wall(&b,&s))\
155                 && !(choc_snake(&c,&c))\
156                 && !(choc_wall(&b,&c))\
157                 && !(choc_snake(&c,&s))\
158                 && !(choc_snake(&s,&c))))
159
160             {
161                 sdl_print();
162             }
163             /* libere le mutex pour les autres thread */
```

```
164             pthread_mutex_unlock(&mutex);
165             usleep(SLEEP);
166         }
167         pthread_exit(0);
168 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

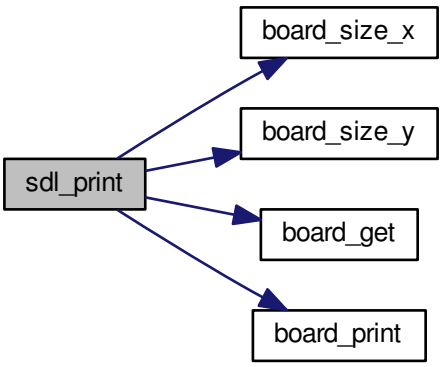


4.4.2.3 void sdl_print ( )

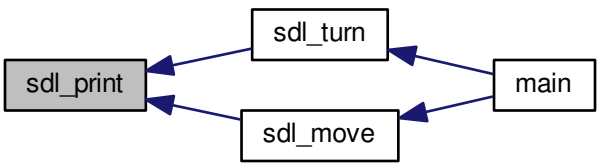
```
173 {
174     int x,y, dx = board_size_x (&b), dy = board_size_y (&
      b);
175     /* def une position */
176     SDL_Rect p;
177     for (x = 0; x < dx; x++)
178     {
179         for (y = 0; y < dy; y++)
180         {
```

```
181             p.x = x * 20 ; p.y = y * 20 ;
182             /* lire dans le plateau et dessiner la surface */
183             switch (board_get (&b,x,y))
184             {
185                 case '@':
186                     SDL_BlitSurface(A, 0, E, &p);
187                     break;
188                 case ' ':
189                     SDL_BlitSurface(B, 0, E, &p);
190                     break;
191                 case '#':
192                     SDL_BlitSurface(C, 0, E, &p);
193                     break;
194                 case '&':
195                     SDL_BlitSurface(D, 0, E, &p);
196                     break;
197                 }
198         }
199     }
200     board_print(&b);
201     /* reafficher un nouvelle ecran */
202     SDL_Flip(E);
203 }
```

Voici le graphe d'appel pour cette fonction :



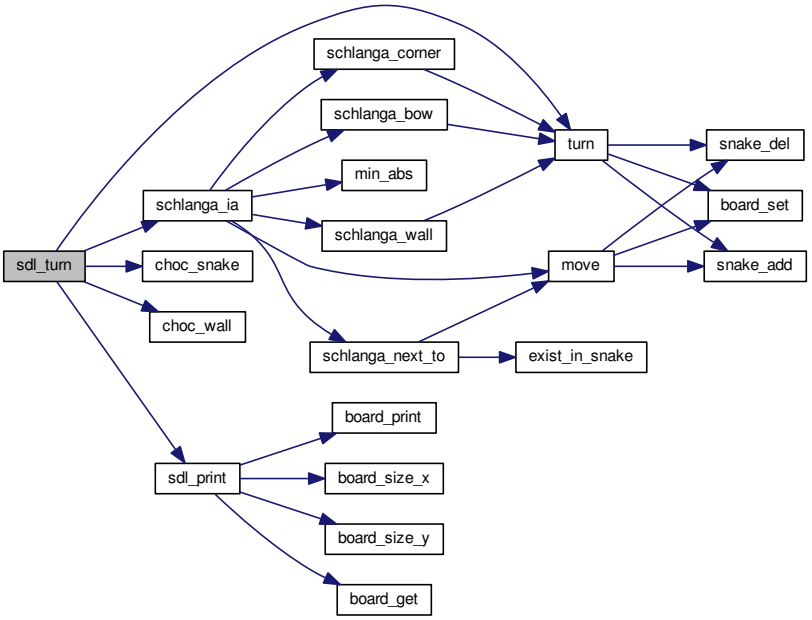
Voici le graphe des appelants de cette fonction :



4.4.2.4 void* sdl_turn ( )

```
80 {
81     SDL_Event event;
82     while (X)
83     {
84     SDL_WaitEvent(&event);
85         switch (event.type)
86         {
87             case SDL_QUIT:
88                 X = false;
89                 break;
90             case SDL_KEYDOWN:
91                 switch (event.key.keysym.sym)
92                 {
93                     case SDLK_ESCAPE:
94                         X = false;
95                         break;
96                     case SDLK_RIGHT:/* key right */
97                     case SDLK_KP3:/* keypad 3 */
98                     case SDLK_KP9:/* keypad 3 */
99                     case 'e':/* code ascii */
100                         /* on ferme la variable mutex */
101                         pthread_mutex_lock(&mutex);
102                         /* turn snake */
103                         turn(&b,&s,1,'@');
104                         schlanga_ia(&b,&s,&c,SNAKE_LEN);
105                         if ((X = !(choc_snake(&s,&s))\
106                             && !(choc_wall(&b,&s))\
107                             && !(choc_snake(&c,&c))\
108                             && !(choc_wall(&b,&c))\
109                             && !(choc_snake(&c,&s))\
110                             && !(choc_snake(&s,&c))))
111 
112                         {
113                             sdl_print();
114                         }
115                         pthread_mutex_unlock(&mutex);
116                         break;
117                     case SDLK_LEFT:/* key left */
118                     case SDLK_KP1:/* keypad 1*/
119                     case SDLK_KP7:/* keypad 7*/
120                     case 'a':/* code ascii */
121                         /* on ferme la variable mutex */
122                         pthread_mutex_lock(&mutex);
123                         /* turn snake */
124                         turn(&b,&s,-1,'@');
125                         schlanga_ia(&b,&s,&c,SNAKE_LEN);
126                         if ((X = !(choc_snake(&s,&s))\
127                             && !(choc_wall(&b,&s))\
128                             && !(choc_snake(&c,&c))\
129                             && !(choc_wall(&b,&c))\
130                             && !(choc_snake(&c,&s))\
131                             && !(choc_snake(&s,&c))))
132                         {
133                             sdl_print();
134                         }
135                         pthread_mutex_unlock(&mutex);
136                         break;
137                     default:/* ne fait rien */break;
138                 }
139             default:/* ne fait rien */break;
140         }
141     }
142     pthread_exit(0);
143 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

4.4.3 Documentation des variables

4.4.3.1 SDL_Surface * A

4.4.3.2 SDL_Surface * B

4.4.3.3 board b

4.4.3.4 SDL_Surface * C

4.4.3.5 snake c

4.4.3.6 SDL_Surface * D

4.4.3.7 SDL_Surface* E

4.4.3.8 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER

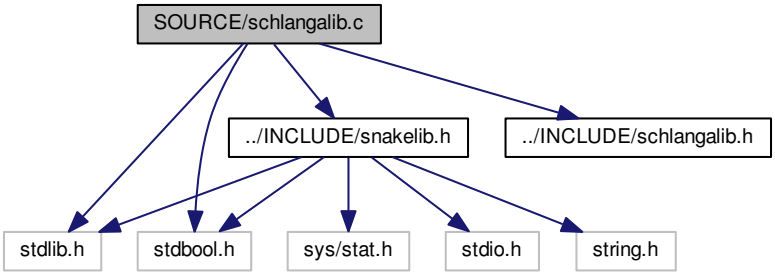
4.4.3.9 snake s

4.4.3.10 bool X

4.5 Référence du fichier SOURCE/schlangalib.c

```
#include <stdlib.h>
#include <stdbool.h>
#include "../INCLUDE/snakelib.h"
#include "../INCLUDE/schlangalib.h"
```

Graphe des dépendances par inclusion de schlangalib.c :



Macros

- #define CODE_NULL 0
- #define CODE_BOW 1
- #define CODE_WALL 2
- #define CODE_CORNER 3
- #define CODE_NEXT_TO 4

Fonctions

- bool exist_in_snake (const snake *s, int x, int y)
- bool schlanga_bow (board *b, const snake *s, snake *c)
- bool schlanga_wall (board *b, snake *c)
- bool schlanga_corner (board *b, snake *c)
- bool schlanga_next_to (board *b, const snake *s, snake *c)
- int min_abs (int a, int b)
- void schlanga_ia (board *b, const snake *s, snake *c, int len)

4.5.1 Documentation des macros

4.5.1.1 #define CODE_BOW 1

4.5.1.2 #define CODE_CORNER 3

4.5.1.3 #define CODE_NEXT_TO 4

4.5.1.4 #define CODE_NULL 0

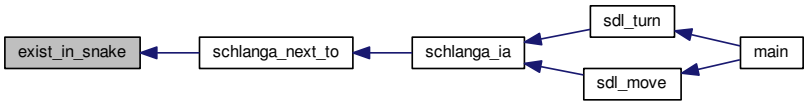
4.5.1.5 #define CODE_WALL 2

4.5.2 Documentation des fonctions

4.5.2.1 bool exist_in_snake (const snake * *s*, int *x*, int *y*)

```
21 {
22     snake curr = *s;
23     while(curr != 0)
24     {
25         if(curr->x == x && curr->y == y)
26         {
27             return true;
28         }
29         curr = curr->next;
30     }
31     return false;
32 }
```

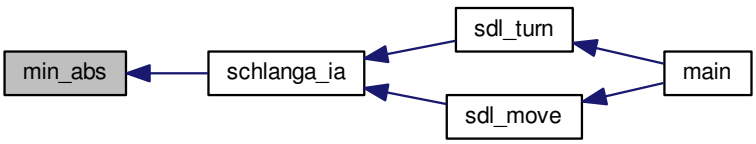
Voici le graphe des appelants de cette fonction :



4.5.2.2 int min_abs (int *a*, int *b*)

```
421 {
422     if(a < 0)
423         a = -a;
424     if(b < 0)
425         b = -b;
426     return (a < b)? a : b;
427 }
```

Voici le graphe des appelants de cette fonction :

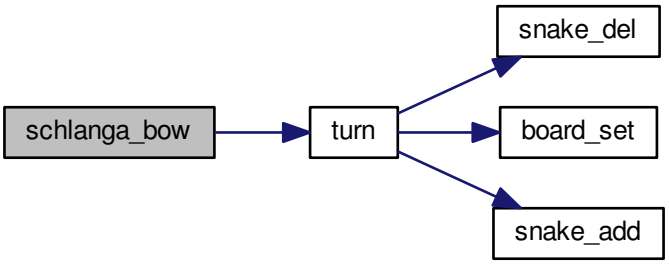


4.5.2.3 bool schlanga_bow (board * b, const snake * s, snake * c)

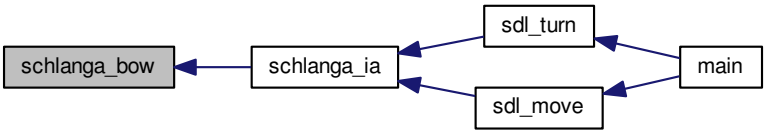
```
43 {
44     snake curr = *s;
45     snake head = *s;
46     bool danger = false;
47     int schla_dir_x = (*c)->x - (*c)->next->x;
48     int schla_dir_y = (*c)->y - (*c)->next->y;
49     while(curr != 0)
50     {
51         if(\
52             ((*c)->x + schla_dir_x == curr->x) && ((*c)->y == curr->y)\
53         )
54         {
55             danger = true;
56             break;
57         }
58         if(\
59             ((*c)->x + 2*schla_dir_x == curr->x) && ((*c)->y == curr->y)\
60         )
61         {
62             danger = true;
63             break;
64         }
65         if(\
66             ((*c)->x + 3*schla_dir_x == curr->x) && ((*c)->y == curr->y)\
67         )
68         {
69             danger = true;
70             break;
71         }
72
73
74         if(\
75             ((*c)->y + schla_dir_y == curr->y) && ((*c)->x == curr->x)\
76         )
77         {
78             danger = true;
79             break;
80         }
81         if(\
82             ((*c)->y + 2*schla_dir_y == curr->y) && ((*c)->x == curr->x)\
83         )
84         {
85             danger = true;
86             break;
87         }
88         if(\
89             ((*c)->y + 3*schla_dir_y == curr->y) && ((*c)->x == curr->x)\
90         )
91         {
92             danger = true;
93             break;
94         }
95         curr = curr->next;
96     }
97     if(danger)
98     {
99         int snake_dir_x = head->x - head->next->x;
100         int snake_dir_y = head->y - head->next->y;
101         int snake_bow_x = 0, snake_bow_y = 0;
102         int dx = 0, dy = 0;
103         curr = *s;
104         while(snake_bow_x == 0\
105             && snake_bow_y == 0\
106             && curr->next->next != 0)
107         {
108             dx = curr->x - curr->next->next->x;
109             dy = curr->y - curr->next->next->y;
110             if(dx != 0 && dy != 0)
111             {
112                 snake_bow_x = dx;
113                 snake_bow_y = dy;
114                 break;
115             }
116             curr = curr->next;
117         }
118         if(snake_bow_x == 1\
119             && snake_bow_y == 1)
120         {
121             if(snake_dir_x == 1)
122             {
123                 turn(b,c,-1,'&');
124                 return true;
125             }
126             if(snake_dir_y == 1)
```

```
127             {
128                 turn(b,c,1,'&');
129                 return true;
130             }
131         }
132         if(snake_bow_x == 1\
133             && snake_bow_y == -1)
134         {
135             if(snake_dir_x == 1)
136             {
137                 turn(b,c,1,'&');
138                 return true;
139             }
140             if(snake_dir_y == -1)
141             {
142                 turn(b,c,-1,'&');
143                 return true;
144             }
145         }
146         if(snake_bow_x == -1\
147             && snake_bow_y == 1)
148         {
149             if(snake_dir_x == -1)
150             {
151                 turn(b,c,1,'&');
152                 return true;
153             }
154             if(snake_dir_y == 1)
155             {
156                 turn(b,c,-1,'&');
157                 return true;
158             }
159         }
160         if(snake_bow_x == -1\
161             && snake_bow_y == -1)
162         {
163             if(snake_dir_x == -1)
164             {
165                 turn(b,c,-1,'&');
166                 return true;
167             }
168             if(snake_dir_y == -1)
169             {
170                 turn(b,c,1,'&');
171                 return true;
172             }
173         }
174         turn(b,c,(rand() % 2) * 2 - 1,'&');
175         return true;
176     }
177     return false;
178 }
```

Voici le graphe d'appel pour cette fonction :



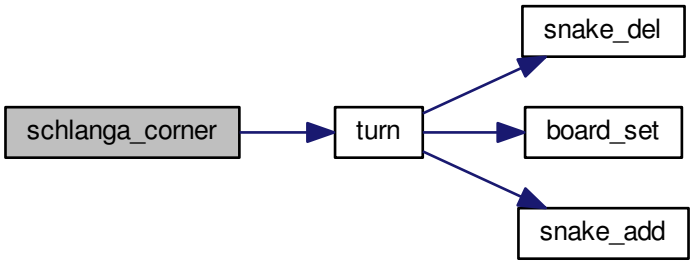
Voici le graphe des appelants de cette fonction :



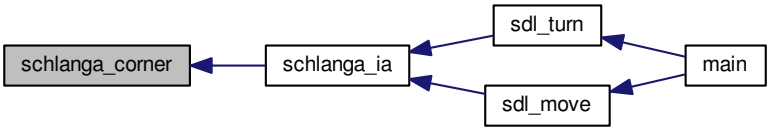
4.5.2.4 bool schlanga_corner (board * b, snake * c)

```
288 {
289     int x_board = 1;
290     int y_board = 1;
291 
292     int dx_board = b->x - 2;
293     int dy_board = b->y - 2;
294 
295     snake head = (*c);
296 
297     int schlanga_dir_x = head->x - head->next->x;
298     int schlanga_dir_y = head->y - head->next->y;
299 
300     if(x_board == head->x && y_board == head->y)
301     {
302         if(schlanga_dir_x)
303         {
304             turn(b,c,-1,'&');
305         }
306         if(schlanga_dir_y)
307         {
308             turn(b,c,1,'&');
309         }
310         return true;
311     }
312     if(x_board == head->x && dy_board == head->y)
313     {
314         if(schlanga_dir_x)
315         {
316             turn(b,c,1,'&');
317         }
318         if(schlanga_dir_y)
319         {
320             turn(b,c,-1,'&');
321         }
322         return true;
323     }
324     if(dx_board == head->x && y_board == head->y)
325     {
326         if(schlanga_dir_x)
327         {
328             turn(b,c,1,'&');
329         }
330         if(schlanga_dir_y)
331         {
332             turn(b,c,-1,'&');
333         }
334         return true;
335     }
336     if(dx_board == head->x && dy_board == head->y)
337     {
338         if(schlanga_dir_x)
339         {
340             turn(b,c,-1,'&');
341         }
342         if(schlanga_dir_y)
343         {
344             turn(b,c,1,'&');
345         }
346         return true;
347     }
348     return false;
349 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

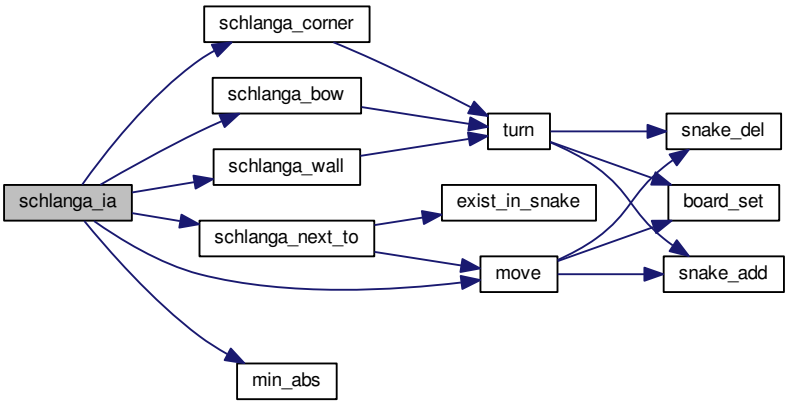


4.5.2.5 void schlanga_ia (board * *b*, const snake * *s*, snake * *c*, int *len*)

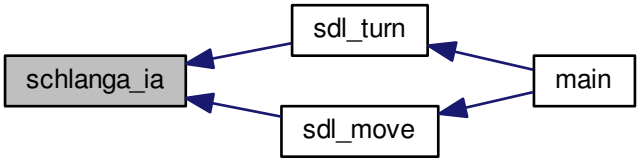
```
437 {
438     static int code = CODE_NULL;
439     static int count = 0;
440     static int corner = 0;
441     int x_board = 1;
442     int y_board = 1;
443 
444     int dx_board = b->x - 2;
445     int dy_board = b->y - 2;
446 
447     int schlanga_x = (*c)->x;
448     int schlanga_y = (*c)->y;
449 
450     int schlanga_dir_x = (*c)->x - (*c)->next->x;
451     int schlanga_dir_y = (*c)->y - (*c)->next->y;
452 
453     if(schlanga_next_to(b,s,c))
454     {
455         code = CODE_NEXT_TO;
456         return;
457     }
458     if(schlanga_corner(b,c))
459     {
460         code = CODE_CORNER;
461         corner = len/2-2;
462         return;
463     }
464     if(corner)
465     {
466         if(schlanga_bow(b,s,c))
467         {
```

```
468             corner = 0;
469             code = CODE_BOW;
470         }
471         else
472         {
473             move(b,c,'&');
474             --corner;
475         }
476         return;
477     }
478     if(code != CODE_WALL)
479     {
480         if(schlanga_wall(b,c))
481         {
482             code = CODE_WALL;
483             unsigned m = 0;
484             if(schlanga_dir_x == 0)
485             {
486                 m = min_abs(schlanga_x - x_board, schlanga_x - dx_board);
487             }
488             if(schlanga_dir_y == 0)
489             {
490                 m = min_abs(schlanga_y - y_board, schlanga_y - dy_board);
491             }
492             if(m != 0)
493             {
494                 count = rand() % m;
495             }
496             return;
497         }
498     }
499     else
500     {
501         if(count == 0)
502         {
503             code = CODE_NULL;
504         }
505         else
506         {
507             if(schlanga_bow(b,s,c))
508             {
509                 code = CODE_BOW;
510                 return;
511             }
512             else
513             {
514                 move(b,c,'&');
515                 --count;
516                 return;
517             }
518         }
519     }
520     if(schlanga_bow(b,s,c))
521     {
522         code = CODE_BOW;
523         return;
524     }
525     move(b,c,'&');
526 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

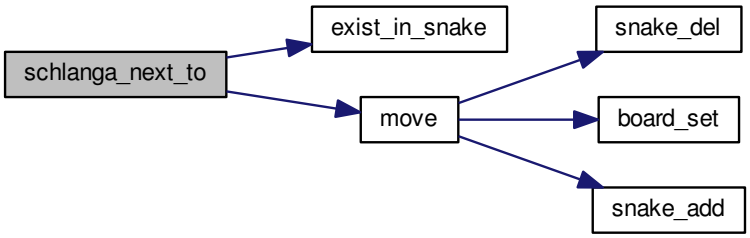


4.5.2.6 bool schlanga_next_to (board * *b*, const snake * *s*, snake * *c*)

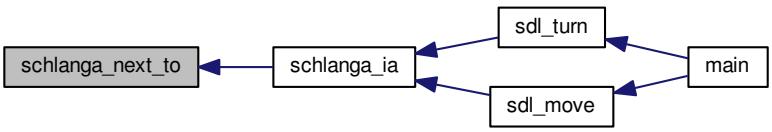
```
360 {
361
362     int x_board = 1;
363     int y_board = 1;
364
365     int dx_board = b->x - 2;
366     int dy_board = b->y - 2;
367
368     int schlanga_x = (*c)->x;
369     int schlanga_y = (*c)->y;
370
371     int schlanga_dir_x = (*c)->x - (*c)->next->x;
372     int schlanga_dir_y = (*c)->y - (*c)->next->y;
373
374     if(schlanga_x == x_board)
375     {
376         if(schlanga_dir_x == 0)
377         {
378             if(exist_in_snake(s,schlanga_x+1,schlanga_y))
379             {
380                 move(b,c,'&');
381                 return true;
382             }
383         }
```

```
384     }
385     if(schlanga_x == dx_board)
386     {
387         if(schlanga_dir_x == 0)
388         {
389             if(exist_in_snake(s,schlanga_x-1,schlanga_y))
390             {
391                 move(b,c,'&');
392                 return true;
393             }
394         }
395     }
396     if(schlanga_y == y_board)
397     {
398         if(schlanga_dir_y == 0)
399         {
400             if(exist_in_snake(s,schlanga_x,schlanga_y+1))
401             {
402                 move(b,c,'&');
403                 return true;
404             }
405         }
406     }
407     if(schlanga_y == dy_board)
408     {
409         if(schlanga_dir_y == 0)
410         {
411             if(exist_in_snake(s,schlanga_x,schlanga_y-1))
412             {
413                 move(b,c,'&');
414                 return true;
415             }
416         }
417     }
418     return false;
419 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

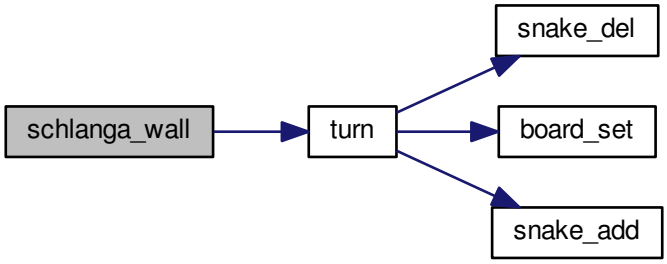


4.5.2.7 bool schlanga_wall (board * b, snake * c)

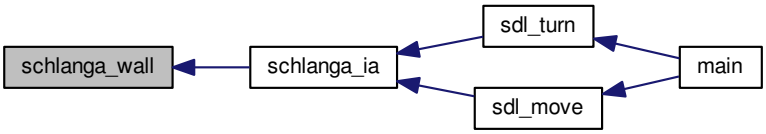
```
188 {
189     int x_board = 1;
190     int y_board = 1;
191     int dx_board = b->x - 2;
192     int dy_board = b->y - 2;
193     int x_schla = (*c)->x;
194     int y_schla = (*c)->y;
195     int dx_schla = (*c)->x - (*c)->next->x;
196     int dy_schla = (*c)->y - (*c)->next->y;
197     if(y_schla == y_board)
198     {
199         if(dx_schla == 0)
200         {
201             if(x_schla == x_board)
202             {
203                 turn(b,c,1,'&');
204                 return true;
205             }
206             if(x_schla == dx_board)
207             {
208                 turn(b,c,-1,'&');
209                 return true;
210             }
211             turn(b,c,(rand() % 2) * 2 - 1,'&');
212             return true;
213         }
214         turn(b,c,dx_schla,'&');
215         return true;
216     }
217     if(y_schla == dy_board)
218     {
219         if(dx_schla == 0)
220         {
221             if(x_schla == x_board)
222             {
223                 turn(b,c,-1,'&');
224                 return true;
225             }
226             if(x_schla == dx_board)
227             {
228                 turn(b,c,1,'&');
229                 return true;
230             }
231             turn(b,c,(rand() % 2) * 2 - 1,'&');
232             return true;
233         }
234         turn(b,c,-dx_schla,'&');
235         return true;
236     }
237     if(x_schla == dx_board)
238     {
239         if(dy_schla == 0)
240         {
241             if(y_schla == y_board)
242             {
243                 turn(b,c,1,'&');
244                 return true;
245             }
246             if(y_schla == dy_board)
247             {
248                 turn(b,c,-1,'&');
249                 return true;
250             }
251             turn(b,c,(rand() % 2) * 2 - 1,'&');
252             return true;
253         }
254         turn(b,c,dy_schla,'&');
255         return true;
256     }
257     if(x_schla == x_board)
258     {
259         if(dy_schla == 0)
260         {
261             if(y_schla == y_board)
262             {
263                 turn(b,c,-1,'&');
264                 return true;
265             }
266             if(y_schla == dy_board)
267             {
268                 turn(b,c,1,'&');
269                 return true;
270             }
271             turn(b,c,(rand() % 2) * 2 - 1,'&');
```

```
272             return true;
273         }
274         turn(b,c,-dy_schla,'&');
275         return true;
276     }
277     return false;
278 }
```

Voici le graphe d'appel pour cette fonction :



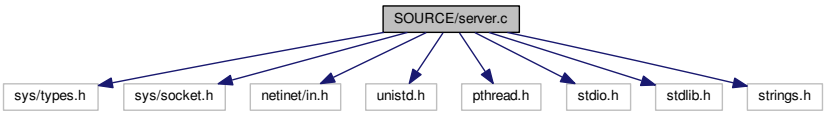
Voici le graphe des appelants de cette fonction :



4.6 Référence du fichier SOURCE/server.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
```

Graphe des dépendances par inclusion de server.c :



Macros

- #define err(msg) {fprintf(stderr, "%s\n", msg) ;exit(1) ;}
- #define PORT 1234
- #define ADDR INADDR_ANY
- #define BACKLOG 10

Fonctions

- static int server_socket ()
- static int server_accept (int p)
- static void * server_handler (void *arg)
- int main ()

Variables

- static int clients [2]
- static pthread_t threadpt [2]

4.6.1 Documentation des macros

4.6.1.1 #define ADDR INADDR_ANY

4.6.1.2 #define BACKLOG 10

4.6.1.3 #define err(*msg*) {fprintf(stderr, "%s\n", msg) ;exit(1) ;}

4.6.1.4 #define PORT 1234

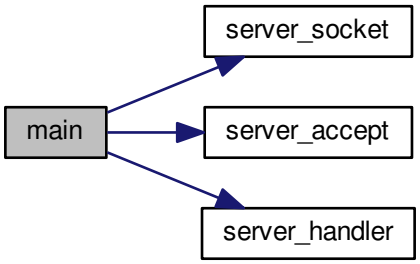
4.6.2 Documentation des fonctions

4.6.2.1 int main ( )

```
58 {
59     short count = 0;
60     int n = server_socket();
61     while(2 != count)
62     {
63         clients[count] = server_accept(n);
64         ++count;
65     }
66     for(count = 0; count != 2; count++)
67     {
```

```
68          int pth = pthread_create(&threadpt[count], NULL, &server_handler, &
        clients[count]);
69          if(0 > pth) err("pthread_create()");
70      }
71      char buf[2] = {'@','&'};
72      for(count = 0; count != 2; count++)
73      {
74          write(clients[count], &buf[count], 1);
75      }
76      for(count = 0; count != 2; count++)
77      {
78          pthread_join(threadpt[count],NULL);
79      }
80      return 0;
81  }
```

Voici le graphe d'appel pour cette fonction :



4.6.2.2 static int server_accept (int *p*) [static]

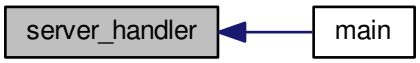
```
36 {
37      struct sockaddr_in client_addr;
38      unsigned len = sizeof(client_addr);
39
40      int n = accept(p, (struct sockaddr*)&client_addr, &len);
41      if(0 > n) err("accept()");
42      return n;
43  }
```

Voici le graphe des appelants de cette fonction :

4.6.2.3 static void* server_handler (void * *arg*) [static]

```
45 {
46     int n = *((int*)arg);
47     int i = (n == clients[0])? 0 : 1;
48     int j = (i+1) % 2;
49     char dir = '\0';
50     while(read(clients[i], &dir, sizeof(char*)))
51     {
52         write(clients[j], &dir, 1);
53     }
54     close(clients[i]);
55     pthread_exit(0);
56 }
```

Voici le graphe des appelants de cette fonction :



4.6.2.4 static int server_socket ( ) [static]

```
20 {
21     int n = socket(AF_INET, SOCK_STREAM, 0);
22     if(0 > n) err("socket()");
23
24     struct sockaddr_in server_addr;
25     bzero(&server_addr,sizeof(server_addr));
26     server_addr.sin_family = AF_INET;
27     server_addr.sin_port = htons(PORT);
28     server_addr.sin_addr.s_addr = htonl(ADDR);
29
30     if(0 > bind(n, (struct sockaddr*)&server_addr, sizeof(server_addr))) err("bind()");
31     if(0 > listen(n, BACKLOG)) err("listen()");
32
33     return n;
34 }
```

Voici le graphe des appelants de cette fonction :

4.6.3 Documentation des variables

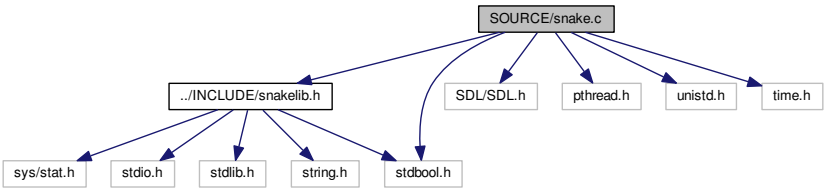
4.6.3.1 int clients[2] [static]

4.6.3.2 pthread_t threadpt[2] [static]

4.7 Référence du fichier SOURCE/snake.c

```
#include "../INCLUDE/snakelib.h"
#include <SDL/SDL.h>
#include <pthread.h>
#include <stdbool.h>
#include <unistd.h>
#include <time.h>
```

Graphe des dépendances par inclusion de snake.c :



Macros

- #define BOARD_SIZE_X 51
- #define BOARD_SIZE_Y 31
- #define SNAKE_LEN 6
- #define SLEEP 100000

Fonctions

- void * sdl_turn ()
- void * sdl_move ()
- void sdl_print ()
- int main ()

Variables

- pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
- SDL_Surface * E
- SDL_Surface * A
- SDL_Surface * B
- SDL_Surface * C
- SDL_Surface * D
- board b
- snake s
- int apple_x
- int apple_y
- int last_x
- int last_y
- bool X

4.7.1 Documentation des macros

4.7.1.1 #define BOARD_SIZE_X 51

4.7.1.2 #define BOARD_SIZE_Y 31

4.7.1.3 #define SLEEP 100000

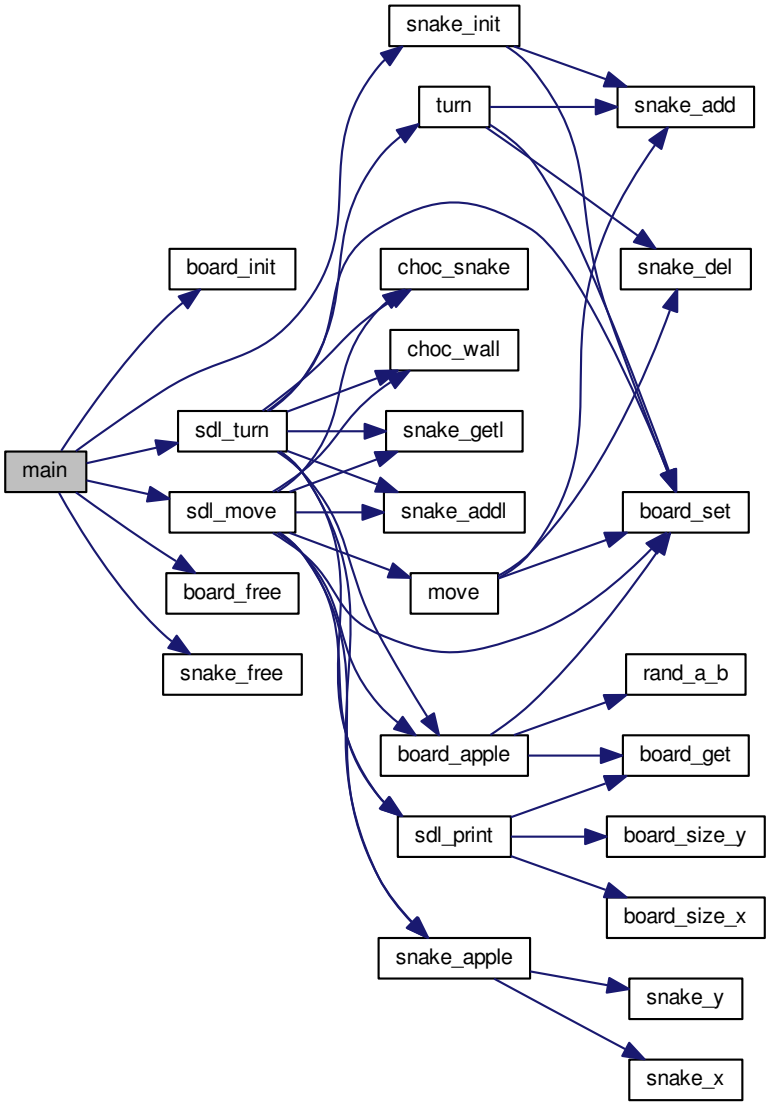
4.7.1.4 #define SNAKE_LEN 6

4.7.2 Documentation des fonctions

4.7.2.1 int main ( )

```
26 {
27     /* la variable globale du jeu */
28     X = true;
29     /* rand init */
30     srand(time(0));
31     /* init board + snake */
32     b = board_init(BOARD_SIZE_X,BOARD_SIZE_Y);
33     snake_init(&b,&s,SNAKE_LEN,'@');
34     /* init sdl en mode video */
35     SDL_Init(SDL_INIT_VIDEO);
36     /* init surface E avec une taille 20*BOARD_SIZE_X et 20*BOARD_SIZE_Y */
37     E = SDL_SetVideoMode(20*BOARD_SIZE_X, 20*BOARD_SIZE_Y, 32, SDL_HWSURFACE);
38     /* creer le reste des surfaces avec une taille de 20*20 px */
39     A = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
40     B = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
41     C = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
42     D = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 20, 32, 0, 0, 0, 0);
43     /* init avec couleur rgb */
44     SDL_FillRect(A, 0, SDL_MapRGB(E->format, 95, 0, 0));
45     SDL_FillRect(B, 0, SDL_MapRGB(E->format, 137, 137, 137));
46     SDL_FillRect(C, 0, SDL_MapRGB(E->format, 47, 47, 47));
47     SDL_FillRect(D, 0, SDL_MapRGB(E->format, 157, 62, 12));
48     /* init barre de fenetre */
49     SDL_WM_SetCaption("Snack", 0);
50     /* nos variabe de thread */
51     pthread_t thread_turn;
52     pthread_t thread_move;
53     /* creer les thread */
54     pthread_create (&thread_turn, 0, &sdl_turn, 0);
55     pthread_create (&thread_move, 0, &sdl_move,0);
56     /* attendre leurs fin */
57     pthread_join(thread_turn, 0);
58     pthread_join(thread_move, 0);
59     /* libere surface */
60     SDL_FreeSurface(E);
61     SDL_FreeSurface(A);
62     SDL_FreeSurface(B);
63     SDL_FreeSurface(C);
64     /* libere board + snake */
65     board_free(&b);
66     snake_free(&s);
67     SDL_Quit();
68     return 0;
69 }
```

Voici le graphe d'appel pour cette fonction :

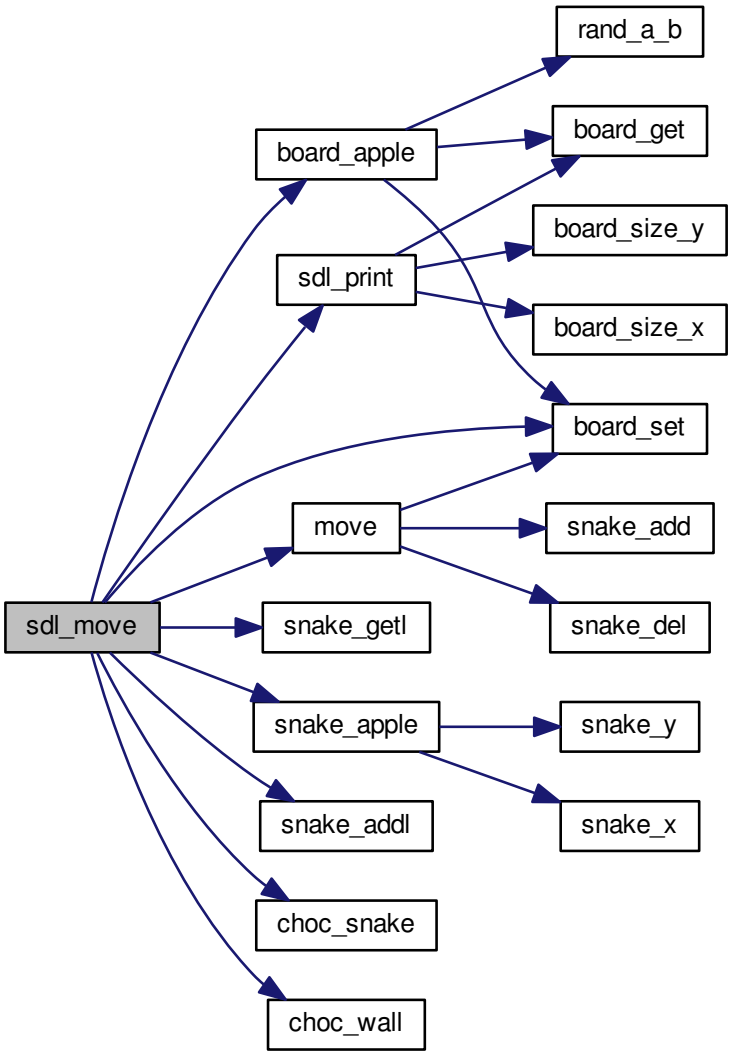


4.7.2.2 void* sdl_move ( )

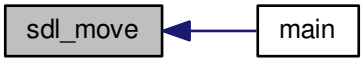
```
148 {
149     board_apple(&b, &apple_x, &apple_y);
150     while(X)
151     {
152             /* on ferme la variable mutex */
153             pthread_mutex_lock(&mutex);
154             /* on cherche le dernier de notre liste */
155             snake_getl(&s,&last_x,&last_y);
156             /* move snake */
157             move(&b,&s,'@');
158             /* teste snake/pomme */
159             if(snake_apple(&s,&apple_x,&apple_y))
160             {
161                 /* si oui on ajoute le dernier avant de tourner */
162                 snake_addl(&s, last_x, last_y);
```

```
163                 /* on l'ajoute aussi pour le plateau */
164                 board_set(&b,last_x,last_y,'@');
165                 /* genere nouvelle pomme */
166                 board_apple(&b, &apple_x, &apple_y);
167             }
168             /* affiche SDL */
169             if ((X = !(choc_snake(&s,&s)) && !(choc_wall(&
    b,&s))))
170                 sdl_print();
171             /* libere le mutex pour les autres thread */
172             pthread_mutex_unlock(&mutex);
173             usleep (SLEEP);
174         }
175         pthread_exit(0);
176 }
```

Voici le graphe d'appel pour cette fonction :



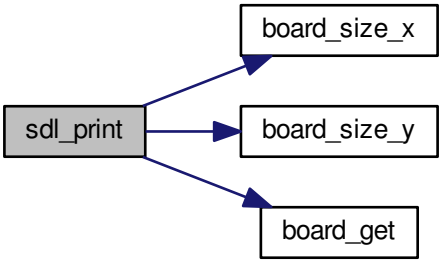
Voici le graphe des appelants de cette fonction :



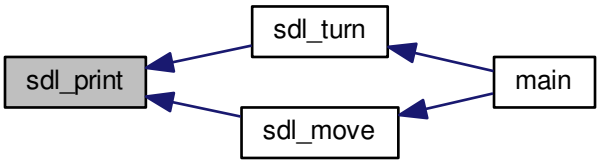
4.7.2.3 void sdl_print ( )

```
181 {
182     int x,y, dx = board_size_x(&b), dy = board_size_y(&
      b);
183     /* def une position */
184     SDL_Rect p;
185     for (x = 0; x < dx; x++)
186     {
187         for (y = 0; y < dy; y++)
188         {
189             p.x = x * 20 ; p.y = y * 20 ;
190             /* lire dans le plateau et dessiner la surface */
191             switch (board_get (&b,x,y))
192             {
193                 case '@':
194                     SDL_BlitSurface(A, 0, E, &p);
195                     break;
196                 case ' ':
197                     SDL_BlitSurface(B, 0, E, &p);
198                     break;
199                 case '#':
200                     SDL_BlitSurface(C, 0, E, &p);
201                     break;
202                 case '$':
203                     SDL_BlitSurface(D, 0, E, &p);
204                     break;
205                 }
206         }
207     }
208     /* reafficher un nouvelle ecran */
209     SDL_Flip(E);
210 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

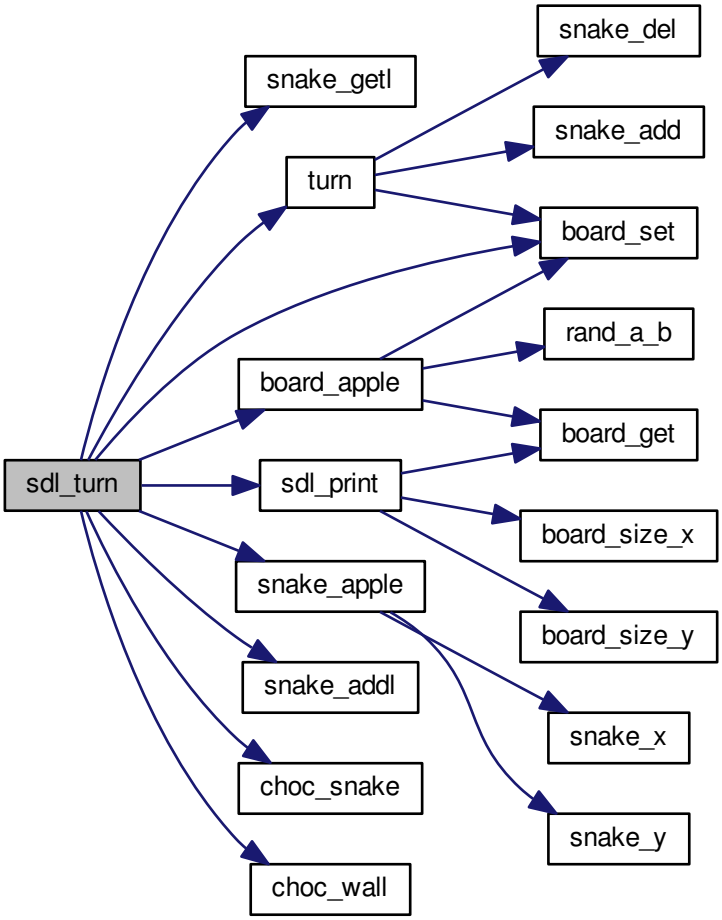


4.7.2.4 void* sdl_turn ( )

```
74 {
75     SDL_Event event;
76     while (X)
77     {
78     SDL_WaitEvent(&event);
79         switch (event.type)
80         {
81             case SDL_QUIT:
82                 X = false;
83                 break;
84             case SDL_KEYDOWN:
85                 switch (event.key.keysym.sym)
86                 {
87                     case SDLK_ESCAPE:
88                         X = false;
89                         break;
90                     case SDLK_RIGHT:/* key right */
91                     case SDLK_KP3:/* keypad 3 */
92                     case SDLK_KP9:/* keypad 3 */
93                     case 'e':/* code ascii */
94                         /* on ferme la variable mutex */
95                         pthread_mutex_lock(&mutex);
96                         /* on cherche le dernier de notre liste */
97                         snake_getl(&s,&last_x,&last_y);
98                         /* turn snake */
99                         turn(&b,&s,1,'@');
100                         /* teste snake/pomme */
101                         if(snake_apple(&s,&apple_x,&apple_y))
102                         {
103                             /* si oui on ajoute le dernier avant de tourner */
104                             snake_addl(&s, last_x, last_y);
105                             /* on l'ajoute aussi pour le plateau */
106                             board_set(&b,last_x,last_y,'@');
107                             /* genere nouvelle pomme */
108                             board_apple(&b, &apple_x, &
    apple_y);
109                         }
110                         if ((X = !(choc_snake(&s,&s)) && !(
    choc_wall(&b,&s))))
111                             sdl_print();
112                         pthread_mutex_unlock(&mutex);
113                         break;
114                     case SDLK_LEFT:/* key left */
115                     case SDLK_KP1:/* keypad 1*/
116                     case SDLK_KP7:/* keypad 7*/
117                     case 'a':/* code ascii */
118                         /* on ferme la variable mutex */
119                         pthread_mutex_lock(&mutex);
120                         /* on cherche le dernier de notre liste */
121                         snake_getl(&s,&last_x,&last_y);
122                         /* turn snake */
123                         turn(&b,&s,-1,'@');
124                         /* teste snake/pomme */
125                         if(snake_apple(&s,&apple_x,&apple_y))
126                         {
127                             /* si oui on ajoute le dernier avant de tourner */
128                             snake_addl(&s, last_x, last_y);
```

```
129                             /* on l'ajoute aussi pour le plateau */
130                             board_set(&b,last_x,last_y,'@');
131                             /* genere nouvelle pomme */
132                             board_apple(&b, &apple_x, &
    apple_y);
133                         }
134                         /* affiche SDL */
135                         if ((X = !(choc_snake(&s,&s)) && !(
    choc_wall(&b,&s))))
136                         sdl_print();
137                         /* libere le mutex pour les autres thread */
138                         pthread_mutex_unlock(&mutex);
139                         break;
140                     default:/* ne fait rien */break;
141                 }
142             default:/* ne fait rien */break;
143         }
144     }
145     pthread_exit(0);
146 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :

4.7.3 Documentation des variables

4.7.3.1 SDL_Surface * A

4.7.3.2 int apple_x

4.7.3.3 int apple_y

4.7.3.4 SDL_Surface * B

4.7.3.5 board b

4.7.3.6 SDL_Surface * C

4.7.3.7 SDL_Surface * D

4.7.3.8 SDL_Surface* E

4.7.3.9 int last_x

4.7.3.10 int last_y

4.7.3.11 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER

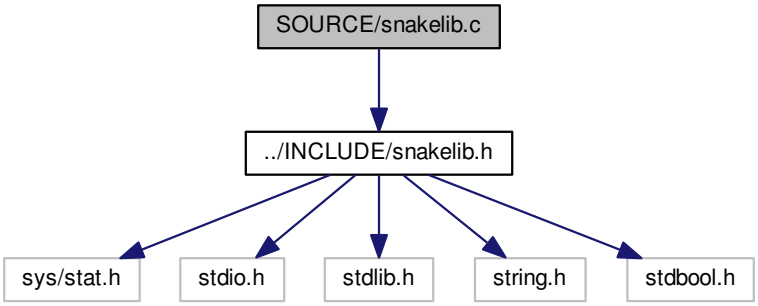
4.7.3.12 snake s

4.7.3.13 bool X

4.8 Référence du fichier SOURCE/snakelib.c

```
#include "../INCLUDE/snakelib.h"
```

Graphe des dépendances par inclusion de snakelib.c :



Fonctions

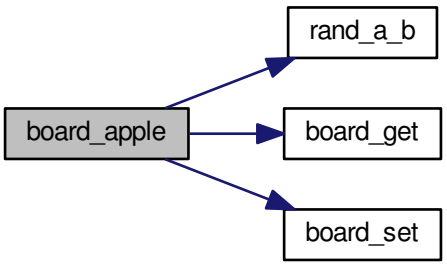
- board board_init (int x, int y)
- void board_set (board *brdin, int x, int y, char value)
- char board_get (board const *brdin, int x, int y)
  return la valeur
- int board_size_x (board const *b)
  taille x du plateau
- int board_size_y (board const *b)
- void board_free (board *brdin)
  liberer la memoire occupé par un plateau
- void board_print (const board *brdin)
  pour afficher un plateau
- void board_pxmap (const board *brdin, char *foldername, int filename, int zoom)
  créer une image ppm du tableau pour mieux voir et interpreter nos fonctions de teste
- void snake_add (snake *snkin, int x, int y)
  un snake est équivalent à une file d'attente (FIFO « first in, first out »), cette fonction permet d'ajouter à l'entête de la liste
- void snake_del (snake *snkin, int *x, int *y)
  un snake est équivalent à une file d'attente (FIFO « first in, first out »), cette fonction permet de supprimer au queue de la liste
- void snake_addl (snake *snkin, int x, int y)
- void snake_getl (snake *snkin, int *x, int *y)
- void snake_free (snake *snkin)
  libérer la mémoire
- void snake_print (const snake *snkin)
- int snake_x (snake const *s)
- int snake_y (snake const *s)
- snake * snake_next (snake const *s)
- void move (board *brdin, snake *snkin, char id)
- void turn (board *brdin, snake *snkin, int drctn, char id)
- void snake_init (board *brdin, snake *snkin, int len, char id)
  init a snake
- bool choc_snake (snake const *s1, snake const *s2)
  test choc entre deux snake
- bool choc_sc (snake const *s1, snake const *s2)
- bool choc_wall (board const *b, snake const *s)
  test choc snake contre le mur
- int rand_a_b (int a, int b)
- void board_apple (board *b, int *x, int *y)
- bool snake_apple (snake const *s, int *x, int *y)

4.8.1 Documentation des fonctions

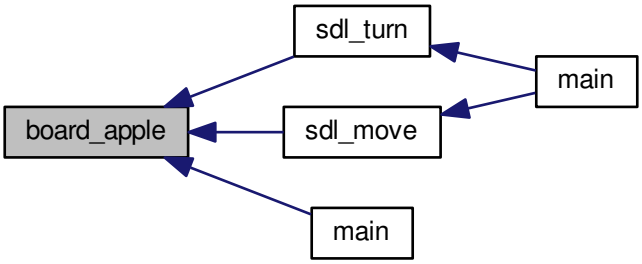
4.8.1.1 void board_apple (board * b, int * x, int * y)

```
512 {
513     bool t = true;
514     while(t)
515     {
516             *x = rand_a_b(1, b->x-2);
517             *y = rand_a_b(1, b->y-2);
518             if(board_get(b,*x,*y) == ' ')
519             {
520                 board_set(b,*x,*y,'$');
521                 t = false;
522             }
523     }
524 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :



4.8.1.2 void board_free (board * brdin)

liberer la memoire occupé par un plateau

Paramètres

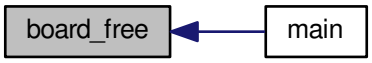
adresse	du plateau à libérer

Renvoie

une fonction de type void

```
91 {
92     int i;
93 
94     for(i = 0; i < brdin->y ; i++)
95     {
96         free(brdin->array[i]);
97         brdin->array[i] = NULL;
98     }
99 
100     free(brdin->array);
101     brdin->array = NULL;
102 }
```

Voici le graphe des appelants de cette fonction :



4.8.1.3 char board_get (board const * *brdin*, int *x*, int *y*)

return la valeur

Paramètres

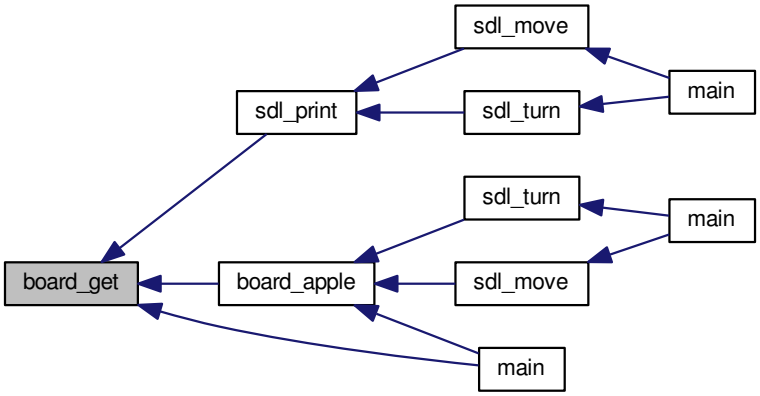
adresse	d'un plateau pour accéder et modifier simplement ses variables
x	et y la case à visiter

Renvoie

char la valueur du point (x,y)

```
70 {
71     return brdin->array[y][x];
72 }
```

Voici le graphe des appelants de cette fonction :

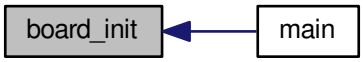


4.8.1.4 board board_init (int *x*, int *y*)

borders values is '#'.

```
22 {
23     board brdout;
24 
25     brdout.x = x;
26     brdout.y = y;
27 
28     int i, j;
29 
30     brdout.array=malloc(y*sizeof(char*));
31     for(i = 0; i < y; i++)
32         brdout.array[i] = malloc(x*sizeof(int));
33 
34 
36     for(i = 0; i < x; i++)
37     {
38         brdout.array[0][i] = '#';
39         brdout.array[y-1][i] = '#';
40     }
41     for(i = 0;i < y; i++)
42     {
43         brdout.array[i][0] = '#';
44         brdout.array[i][x-1] = '#';
45     }
46 
47     for(i = 1; i < y-1; i++)
48         for(j = 1; j < x-1; j++)
49             brdout.array[i][j] = ' ';
50 
51     return brdout;
52 }
```

Voici le graphe des appelants de cette fonction :



4.8.1.5 void board_print (const board * *brdin*)

pour afficher un plateau

Paramètres

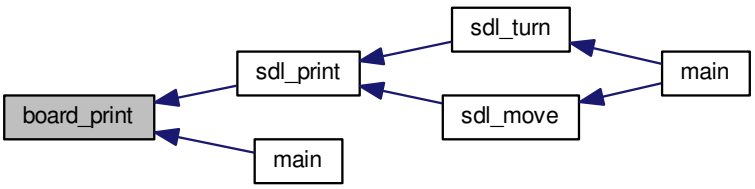
une	adresse d'un plateau concéder constant (droit que pour la lecture) pour éviter de copier le plateau

Renvoie

une fonction de type void

```
113 {
114     int i,j;
115     /* traversing 2d-array */
116     for(i=0 ; i < brdin->y; i++)
117     {
118         for(j=0 ;j < brdin->x; j++)
119         {
120             printf("%c", brdin->array[i][j]);
121         }
122         printf("\n");
123     }
124 }
```

Voici le graphe des appelants de cette fonction :



4.8.1.6 void board_pxmap (const board * *brdin*, char * *foldername*, int *filename*, int *zoom*)

créer une image ppm du tableau pour mieux voir et interpreter nos fonctions de teste

Paramètres

une	adresse d'un plateau concéder constant (droit que pour la lecture) pour éviter de copier le plateau
char∗	foldername : donner un nom au dossier créé
int	filename : numero de la map
int	zoom : pour definir le zoom appliqué a la map (taille d'une case en px = zoom ∗ 1px)

Renvoie

une fonction de type void

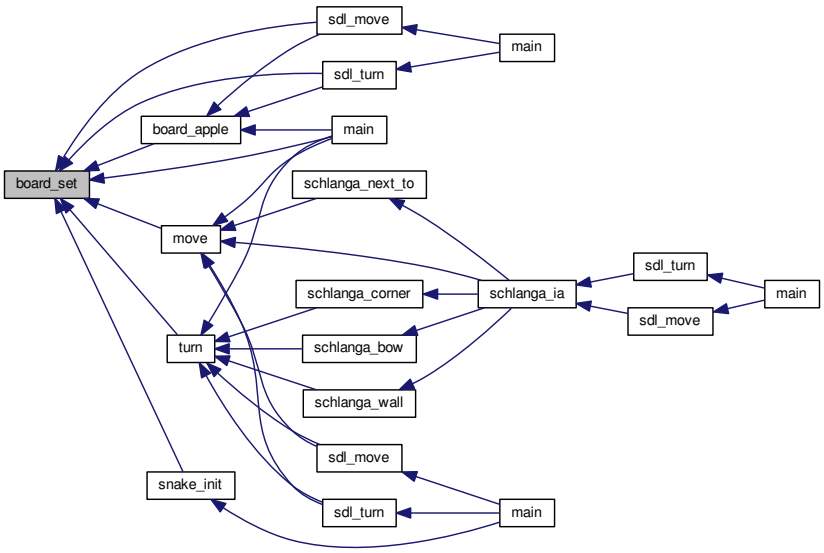
le dossier n'existe pas

```
141 {
142     struct stat dir_stat;
143     if(stat(foldername, &dir_stat) < 0)
145         mkdir(foldername, S_IRWXU);
146 
147 
148     char ffilename[strlen(foldername) + 7];
149     sprintf(ffilename, "%s/%d.ppm", foldername, filename);
150     FILE* fdout = fopen( ffilename, "w");
151     fprintf(fdout, "P3\n%d %d\n255\n", brdin->y * zoom,\
152             brdin->x * zoom);
153 
154 
155     int i,j,l,k;
156     /* traversing 2d-array */
157     for(i = 0; i < brdin->y; i++)
158     {
159      for(l = 0; l < zoom; l++)
160      {
161       for(j = 0; j < brdin->x; j++)
162       {
163        for(k = 0; k < zoom; k++)
164        {
165          if(brdin->array[i][j] == ' ')
166           fprintf(fdout,"175 175 175 ");
167          else if(brdin->array[i][j] == '#')
168           fprintf(fdout,"48 48 48 ");
169          else if(brdin->array[i][j] == '@')
170           fprintf(fdout,"110 11 20 ");
171        }
172       }
173       fprintf(fdout, "\n");
174      }
175      fprintf(fdout, "\n");
176     }
177     fclose(fdout);
178 
179 
180 }
```

4.8.1.7 void board_set (board ∗ *brdin*, int *x*, int *y*, char *value*)

```
65 {
66     brdin->array[y][x] = value;
67 }
```

Voici le graphe des appelants de cette fonction :



4.8.1.8 int board_size_x (board const * *b*)

taille x du plateau

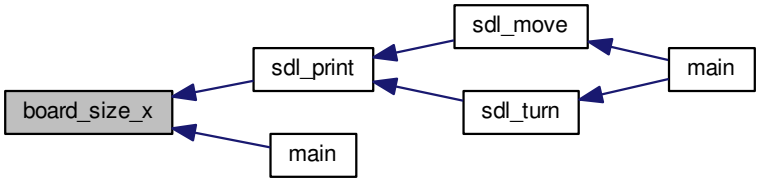
taille y du plateau

Paramètres

adresse	du plateau

```
75 {
76     return b->x;
77 }
```

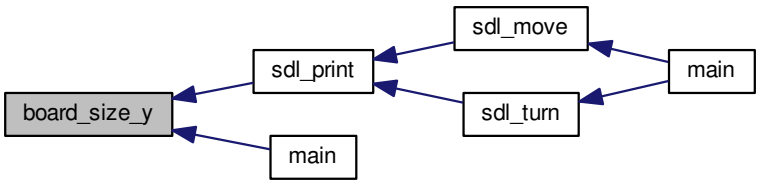
Voici le graphe des appelants de cette fonction :



4.8.1.9 int board_size_y (board const * *b*)

```
80 {
81     return b->y;
82 }
```

Voici le graphe des appelants de cette fonction :



4.8.1.10 bool choc_sc (snake const * *s1*, snake const * *s2*)

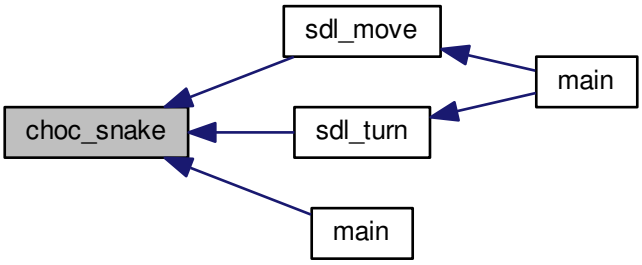
```
490 {
491     if ((*s1)->x == (*s2)->x && (*s1)->y == (*s2)->y)
492         return true;
493     return false;
494 }
```

4.8.1.11 bool choc_snake (snake const * *s1*, snake const * *s2*)

test choc entre deux snake

```
478 {
479     snake tmp = (*s2)->next;
480     while(tmp)
481     {
482         if((*s1)->x == tmp->x && (*s1)->y == tmp->y)
483             return true;
484         tmp = tmp->next;
485     }
486     return false;
487 }
```

Voici le graphe des appelants de cette fonction :

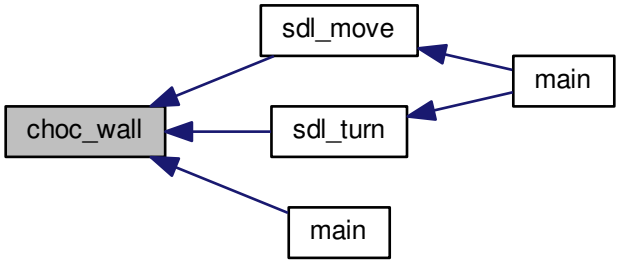


4.8.1.12 bool choc_wall (board const * *b*, snake const * *s*)

test choc snake contre le mur

```
497 {
498     if((*s)->x == 0) return true;
499     if((*s)->x == b->x-1) return true;
500     if((*s)->y == 0) return true;
501     if((*s)->y == b->y-1) return true;
502 
503     return false;
504 }
```

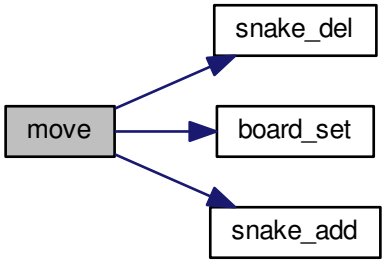
Voici le graphe des appelants de cette fonction :



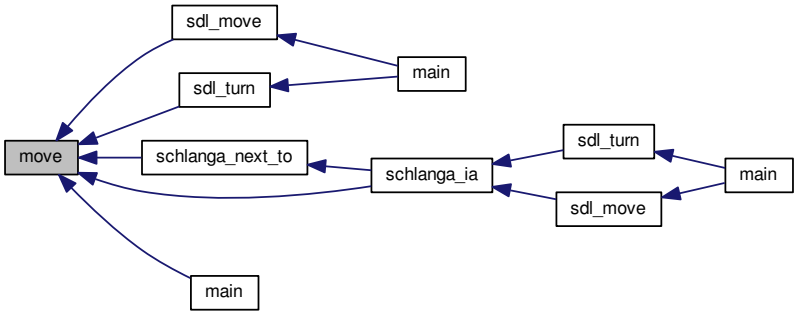
4.8.1.13 void move (board * *brdin*, snake * *snkin*, char *id*)

```
371 {
372     int x,y,dx,dy;
373     snake_del(snkin, &x, &y);
374     board_set(brdin, x, y, ' ');
375     dx = ((*snkin)->x) - ((*snkin)->next->x);
376     dy = ((*snkin)->y) - ((*snkin)->next->y);
377     snake_add(snkin, (*snkin)->x+dx , (*snkin)->y+dy);
378     board_set(brdin, (*snkin)->x ,(*snkin)->y , id);
379 }
```

Voici le graphe d'appel pour cette fonction :



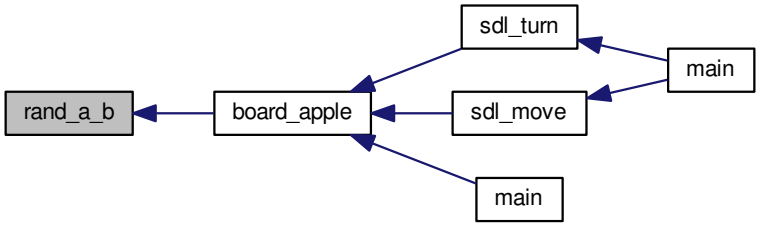
Voici le graphe des appelants de cette fonction :



4.8.1.14 int rand_a_b (int *a*, int *b*)

```
507 {
508     return (rand()%(b-a) + a);
509 }
```

Voici le graphe des appelants de cette fonction :



4.8.1.15 void snake_add (snake * *snkin*, int *x*, int *y*)

un snake est équivalent à une file d'attente (FIFO « first in, first out »), cette fonction permet d'ajouter à l'entête de la liste

Paramètres

snake	*snkin : adresse d'un snake
int	x, int y : coordonnée du nouvel élément

Renvoie

fonction de type void

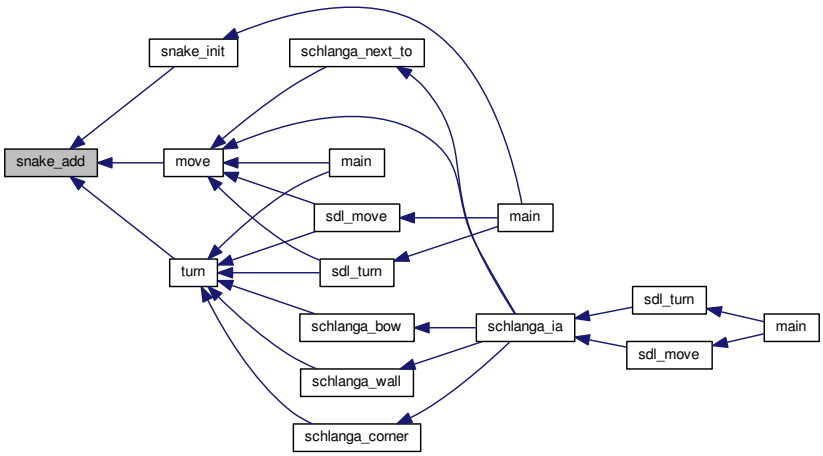
define list output with coord x,y

the next element is list input

edit list input(*input=output)

```
192 {
194     snake snkout=malloc(sizeof(_snake));
195     snkout->x = x;
196     snkout->y = y;
198     snkout->next = *snkin;
200     *snkin = snkout;
201 }
```

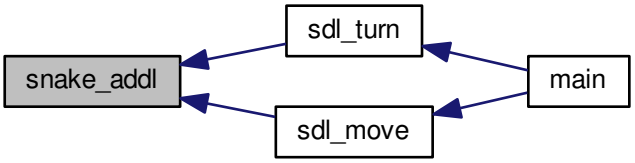
Voici le graphe des appelants de cette fonction :



4.8.1.16 void snake_addl (snake * *snkin*, int *x*, int *y*)

```
262 {
263     snake new = malloc(sizeof(_snake));
264     *new = (_snake)
265     {
266         .x = x,
267         .y = y,
268         .next = NULL,
269     };
270     snake tmp = *snkin;
271     while(tmp->next != NULL)
272     {
273         tmp = tmp->next;
274     }
275     tmp->next = new;
276 }
```

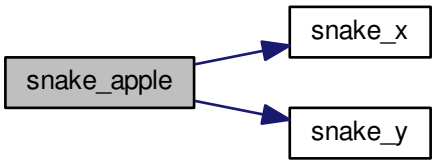
Voici le graphe des appelants de cette fonction :



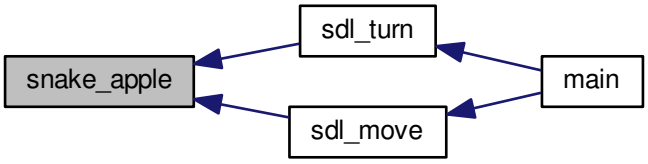
4.8.1.17 bool snake_apple (snake const * s, int * x, int * y)

```
527 {
528     if(snake_x(s) == *x && snake_y(s) == *y) return true;
529     return false;
530 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :



4.8.1.18 void snake_del (snake * *snkin,* int * *x,* int * *y*)

un snake est équivalent à une file d'attente (FIFO « first in, first out »), cette fonction permet de supprimer au queue de la liste

Paramètres

snake	∗snkin : adresse d'un snake
snake	int ∗x, int ∗y : adresses de deux entiers pour pouvoir récupérer les coordonnées d'élément supprimer

Renvoie

fonction de type void

if the list is empty

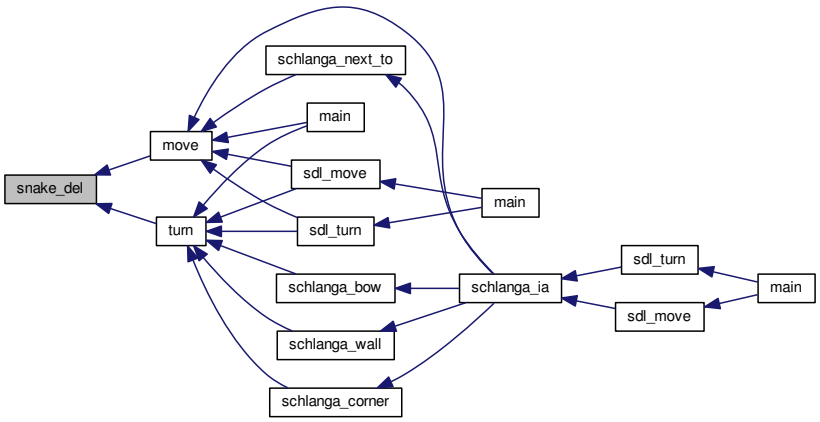
default value x=-1 y=-1

if the list contains only one item

we move along the linked list keeping the last two consecutive element

```
215 {
216     if(*snkin == NULL)
217     {
219         if((x != NULL) && (y != NULL))
220         {
222             *x = -1;
223             *y = -1;
224         }
225     }
226     else if((*snkin)->next == NULL)
227     {
228
230         if((x != NULL) && (y != NULL))
231         {
232             *x = (*snkin)->x;
233             *y = (*snkin)->y;
234         }
235         free((*snkin));
236         *snkin = NULL;
237     }
238     else
239     {
244         snake tmp1 = *snkin;
245         snake tmp2 = *snkin;
246         while(tmp1->next != NULL)
247         {
248             tmp2 = tmp1;
249             tmp1 = tmp1->next;
250         }
251         if((x != NULL) && (y != NULL))
252         {
253             *x = tmp1->x;
254             *y = tmp1->y;
255         }
256         tmp2->next = NULL;
257         free(tmp1);
258     }
259 }
```

Voici le graphe des appelants de cette fonction :



4.8.1.19 void snake_free (snake * *snkin*)

libérer la mémoire

Paramètres

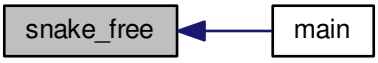
adresse	du snake

Renvoie

fonction de type void

```
309 {
310     snake tmp = *snkin;
311     snake tmpnext;
312 
313     while(tmp != NULL)
314     {
315         tmpnext = tmp->next;
316         free(tmp);
317         tmp = tmpnext;
318     }
319 
320     *snkin = NULL;
321 }
```

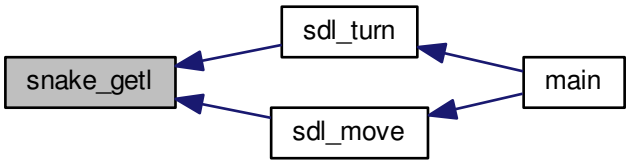
Voici le graphe des appelants de cette fonction :



4.8.1.20 void snake_getl (snake * *snkin*, int * *x*, int * *y*)

```
279 {
280     if(*snkin == NULL)
281     {
282         *x = -1;
283         *y = -1;
284     }
285     else if((*snkin)->next == NULL)
286     {
287         *x = (*snkin)->x;
288         *y = (*snkin)->y;
289     }
290     else
291     {
292         snake tmp = *snkin;
293         while(tmp->next != NULL)
294         {
295             tmp = tmp->next;
296         }
297         *x = tmp->x;
298         *y = tmp->y;
299     }
300 }
```

Voici le graphe des appelants de cette fonction :



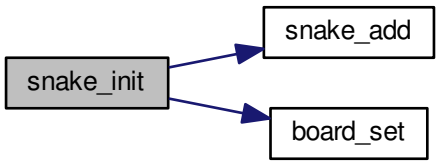
4.8.1.21 void snake_init (board * *brdin*, snake * *snkin*, int *len*, char *id*)

init a snake

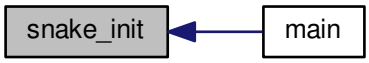
```
429 {
430     if(id == '@')
431     {
432 
433         snake_add( snkin, brdin->x / 2, brdin->y - 2 );
434         board_set( brdin, (*snkin)->x, (*snkin)->y, id );
435 
436         int i;
437         for(i = 0; i < len - 1; i++)
438         {
439             snake_add( snkin, (*snkin)->x, (*snkin)->y -1 );
440             board_set( brdin, (*snkin)->x, (*snkin)->y, id );
441         }
442 
443     }
444 
445     if(id == '&')
446     {
447 
448         snake_add( snkin, brdin->x / 2, 1 );
449         board_set( brdin, (*snkin)->x, (*snkin)->y, id );
450 
451         int i;
452         for(i = 0; i < len - 1; i++)
453         {
```

```
454             snake_add( snkin, (*snkin)->x, (*snkin)->y +1 );
455             board_set( brdin, (*snkin)->x, (*snkin)->y, id );
456         }
457 
458     }
459 
460     if(id == '+')
461     {
462 
463         snake_add( snkin, 1, brdin->y / 2 );
464         board_set( brdin, (*snkin)->x, (*snkin)->y, id );
465 
466         int i;
467         for(i = 0; i < len - 1; i++)
468         {
469             snake_add( snkin, (*snkin)->x + 1, (*snkin)->y);
470             board_set( brdin, (*snkin)->x, (*snkin)->y, 1 );
471         }
472 
473     }
474 
475 }
```

Voici le graphe d'appel pour cette fonction :



Voici le graphe des appelants de cette fonction :



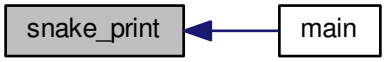
4.8.1.22 snake* snake_next (snake const * *s*)

```
355 {
356     return &((*s)->next);
357 }
```

4.8.1.23 void snake_print (const snake * *snkin*)

```
332 {
333     snake tmp = *snkin;
334     while(tmp != NULL)
335     {
336         printf("[%d:%d]", tmp->x, tmp->y);
337         tmp = tmp->next;
338     }
339     printf("\n");
340 }
```

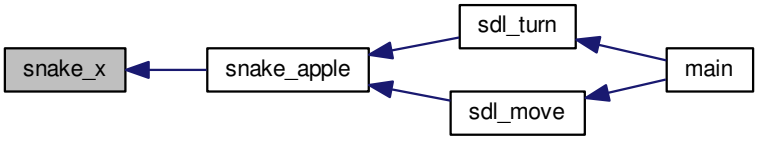
Voici le graphe des appelants de cette fonction :



4.8.1.24 int snake_x (snake const * *s*)

```
345 {
346     return (*s)->x;
347 }
```

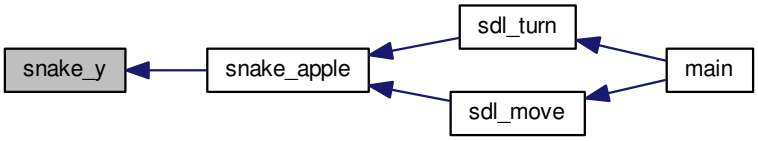
Voici le graphe des appelants de cette fonction :



4.8.1.25 int snake_y (snake const * *s*)

```
350 {
351     return (*s)->y;
352 }
```

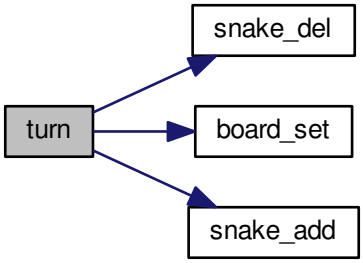
Voici le graphe des appelants de cette fonction :



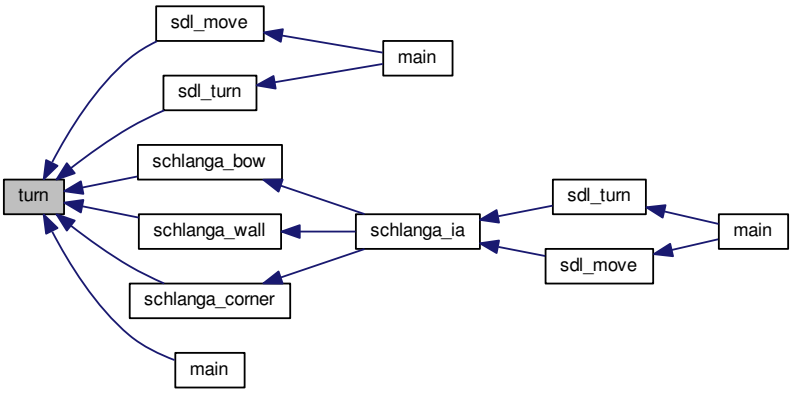
4.8.1.26 void turn (board * *brdin*, snake * *snkin*, int *drctn*, char *id*)

```
394 {
395     int x,y,dx,dy;
396 
397     snake_del(snkin,&x,&y);
398     board_set(brdin,x,y, ' ');
399 
400     dx=((*snkin)->x) - ((*snkin)->next->x);
401     dy=((*snkin)->y) - ((*snkin)->next->y);
402 
403     if(dx == 1)
404     {
405         snake_add(snkin, (*snkin)->x, (*snkin)->y + drctn);
406         board_set(brdin, (*snkin)->x, (*snkin)->y, id);
407     }
408 
409     else if(dx == -1)
410     {
411         snake_add(snkin, (*snkin)->x, (*snkin)->y - drctn);
412         board_set(brdin, (*snkin)->x, (*snkin)->y, id);
413     }
414 
415     else if(dy == 1)
416     {
417         snake_add(snkin,(*snkin)->x-drctn,(*snkin)->y);
418         board_set(brdin,(*snkin)->x,(*snkin)->y,id);
419     }
420 
421     else if(dy == -1)
422     {
423         snake_add(snkin,(*snkin)->x+drctn,(*snkin)->y);
424         board_set(brdin,(*snkin)->x,(*snkin)->y,id);
425     }
426 }
```

Voici le graphe d'appel pour cette fonction :



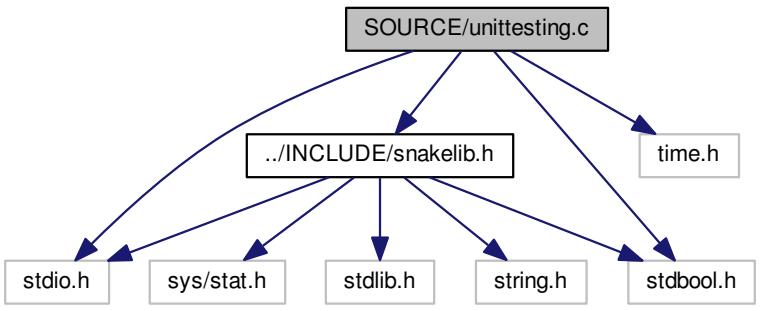
Voici le graphe des appelants de cette fonction :



4.9 Référence du fichier SOURCE/unittesting.c

```
#include "../INCLUDE/snakelib.h"
#include <stdio.h>
#include <time.h>
#include <stdbool.h>
```

Graphe des dépendances par inclusion de unittesting.c :



Fonctions

— int main ()

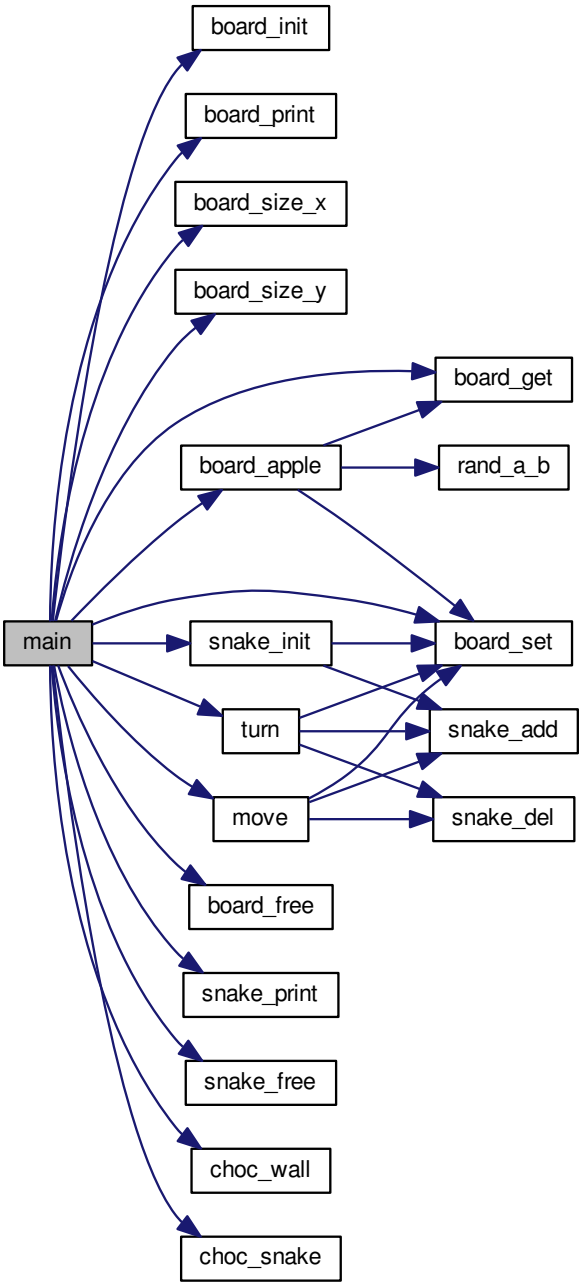
4.9.1 Documentation des fonctions

4.9.1.1 int main ( )

```
5           {
6     srand(time(0));
7     printf("\n\n\n<<< test pour les fonctions type board_# >>>\n\n\n");
8     board b = board_init(31,16);
9     printf("<<< test board_init | tabX=31 | tabY=16 >>>\n\n\n");
10     printf("<<< test board_print >>>\n");
11     board_print(&b);
12     printf("<<< test board_size_x & board_size_y >>>\n\n\n");
13     printf("[size_x=%d : size_y=%d]\n",board_size_x(&b),board_size_y(&b));
14     printf("<<< test board_set & board_get >>>\n\n\n");
15     board_set(&b,5,9,'@');
16     printf("<<< board_set(&b,5,9,'@') >>>\n\n\n");
17     board_print(&b);
18     printf("<<< board_get(&b,5,9) >>>\n\n\n");
19     printf("char=%c\n\n\n\n",board_get(&b,5,9));
20     int x,y;
21     printf("<<< test board_apple(&b, &x, &y) >>>\n\n\n");
22     board_apple(&b, &x, &y);
23     board_print(&b);
24     printf("apple coord [x=%d:y=%d]\n\n\n",x,y);
25     board_apple(&b, &x, &y);
26     board_print(&b);
27     printf("apple coord [x=%d:y=%d]\n\n\n",x,y);
28     board_free(&b);
29     printf("\n\n\n<<< test pour les fonctions type snake_# >>>\n\n\n");
30     printf("<<< test snake_init | len=5 >>>\n\n\n");
31     b=board_init(31,16);
32     snake s = 0;
33     snake_init(&b,&s,5,'@');
34     printf("<<< test snake_print >>>\n\n\n");
35     snake_print(&s);
36     printf("<<< snake print avec board_print >>>\n\n\n");
37     board_print(&b);
38     printf("<<< test move >>>\n\n\n");
39     move(&b,&s,'@');
40     printf("<<< snake_print >>>\n\n\n");
41     snake_print(&s);
42     printf("<<< board_print >>>\n\n\n");
43     board_print(&b);
44     move(&b,&s,'@');
45     printf("<<< snake_print >>>\n\n\n");
46     snake_print(&s);
47     printf("<<< board_print >>>\n\n\n");
48     board_print(&b);
49     move(&b,&s,'@');
50     printf("<<< snake_print >>>\n\n\n");
51     snake_print(&s);
52     printf("<<< board_print >>>\n\n\n");
53     board_print(&b);
54     printf("<<< test turn >>>\n\n\n");
55     turn(&b,&s,-1,'@');
56     printf("<<< snake_print >>>\n\n\n");
57     snake_print(&s);
58     printf("<<< board_print >>>\n\n\n");
59     board_print(&b);
60     turn(&b,&s,1,'@');
61     printf("<<< snake_print >>>\n\n\n");
62     snake_print(&s);
63     printf("<<< board_print >>>\n\n\n");
64     board_print(&b);
65     turn(&b,&s,-1,'@');
66     printf("<<< snake_print >>>\n\n\n");
67     snake_print(&s);
68     printf("<<< board_print >>>\n\n\n");
69     board_print(&b);
70     turn(&b,&s,1,'@');
71     printf("<<< snake_print >>>\n\n\n");
72     snake_print(&s);
73     printf("<<< board_print >>>\n\n\n");
74     board_print(&b);
75     board_free(&b);
76     snake_free(&s);
77     printf("\n\n\n<<< test choc_wall >>>\n\n\n");
78     b=board_init(31,16);
79     snake_init(&b,&s,5,'@');
80     printf("<<< snake_print >>>\n\n\n");
81     snake_print(&s);
82     printf("<<< board_print >>>\n\n\n");
83     board_print(&b);
84     bool test=false;
```

```
85      while(!test) {
86          move(&b,&s,'@');
87          test=choc_wall(&b,&s);
88          printf("<<< snake_print >>>\n\n\n");
89          snake_print(&s);
90          printf("<<< board_print >>>\n\n\n");
91          board_print(&b);
92          if(test)
93              printf("<<< choc_wall=>true >>>\n\n\n");
94          else
95              printf("<<< choc_wall=>false >>>\n\n\n");
96      }
97      board_free(&b);
98      snake_free(&s);
99      printf("\n\n\n<<< test choc_snake >>>\n\n\n");
100      b=board_init(31,16);
101      snake_init(&b,&s,5,'@');
102      printf("<<< snake_print >>>\n\n\n");
103      snake_print(&s);
104      printf("<<< board_print >>>\n\n\n");
105      board_print(&b);
106      test=false;
107      while(!test) {
108          turn(&b,&s,-1,'@');
109          test=choc_snake(&s,&s);
110          printf("<<< snake_print >>>\n\n\n");
111          snake_print(&s);
112          printf("<<< board_print >>>\n\n\n");
113          board_print(&b);
114          if(test)
115              printf("<<< choc_snake=>true >>>\n\n\n");
116          else
117              printf("<<< choc_snake=>false >>>\n\n\n");
118      }
119      board_free(&b);
120      snake_free(&s);
121      return 0;
122  }
```

Voici le graphe d'appel pour cette fonction :



Index