担当：長尾 智晴 (nagao@ynu.ac.jp)
濱上 知樹 (hamagami@ynu.ac.jp)

「アルゴリズムとデータ構造」講義日程

1. ~~基本的データ型~~
2. ~~基本的制御構造~~
3. ~~変数のスコープルール．関数~~
4. ~~配列を扱うアルゴリズムの基礎(1)．最大値，最小値~~
5. ~~配列を扱うアルゴリズムの基礎(2)．重複除去，集合演算，ポインタ~~
6. ~~ファイルの扱い~~
7. ~~整列(1)．単純挿入整列・単純選択整列・単純交換整列~~
8. ~~整列(2)．マージ整列・クイック整列~~
9. 再帰的アルゴリズムの基礎．再帰におけるスコープ．ハノイの塔など ← 本日の内容
10. バックトラックアルゴリズム．8 王妃問題など．
11. 線形リストを扱うアルゴリズム(1 回)
12. 木構造を扱うアルゴリズム(1) 基礎
13. 木構造を扱うアルゴリズム(2) 挿入，削除，バランスなど．
14. ハッシング
15. その他のアルゴリズム

第９回「再帰的アルゴリズムの基礎」

☆ 「再帰的」とは

「あるものが部分的にそれ自身で構成されていたり，それ自身によって定義されている」

例 1）自然数

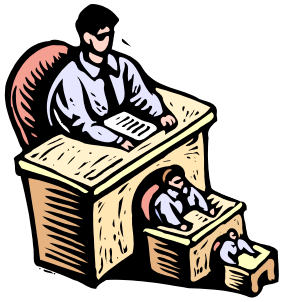
(i) 0 は自然数
(ii) 自然数の次の要素も自然数

例 2）階乗関数

(i) 0! → 1
(ii) n! → n*(n-1)!　(n>0)

例 3）木構造

(i) ϕ は木（空木（empty tree））
(ii) t1, t2 が木ならば，それらを 2 つの「子」としてもつ節点(node) ならびに t1, t2 から構成される構造も木



これも再帰的？！

☆ 再帰的アルゴリズムの図式

- 再帰はある意味での繰り返し → 「終了条件」が必要
- 図式

(a) P ⇔ if (B) { Comp[S,P] }
(b) P ⇔ Comp[S, if (B) { P }]

（ただし，S は P を含まない文の集合，Comp[S,P] は S と P の組み合わせ）

※ ある条件 B がいつかは偽にならないといけない．（そうでないと無限に自分自身を呼び続け，プログラムが終了しない．）

- 扱っている問題や扱うべきデータが再帰的に定義されているときに適する．

☆ 変数のスコープ（おさらい）

◎再帰呼び出しの場合はどうなるか?

「アルゴリズムとデータ構造」第 3 回「有効範囲」のプリント参照（ fact.c→本資料 p.5 ）

ポイント

- 関数の内部で宣言される変数（仮引数を含めて）は，**関数本体のブロック内でのみ有効**.

例）int func(int n1, int n2)

```
{
    int i, sum = 0;

    for (i=n1;i<=n2;i++)
        sum = sum + i;
    return sum;
}
```

この変数 i, sum はこの関数の外からは見えない（＝参照できない，代入できない）

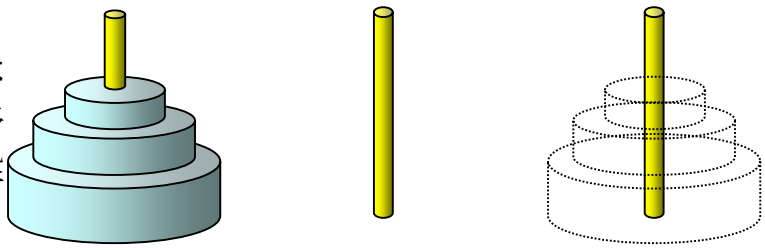
- **上記変数は関数呼び出しのたびに新しく別に作られる（例外：static 宣言）.**

※ 自分自身を再帰的に呼び出しても，値は保存され，混同することはない.

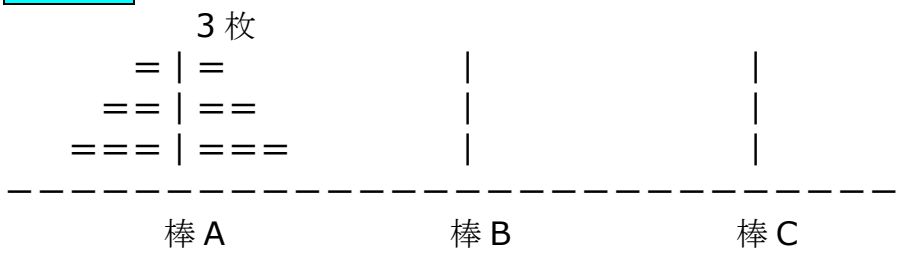
☆ 再帰プログラムの例その 1

◎ ハノイの塔（hanoi.c → p.6）

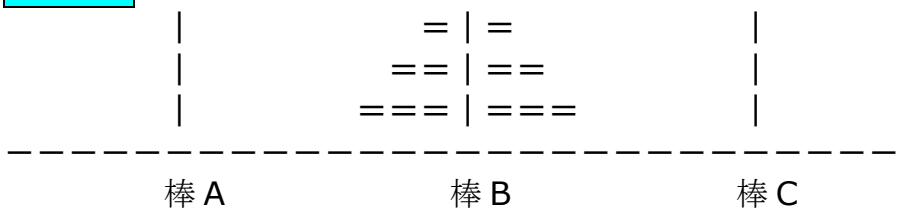
- 条件を満足しつつ，「**初期状態**」から「**目標状態**」まで移行するにはどのように円盤を移動させればよいか?



初期状態



目標状態



条件 1：一度に一枚の円盤だけが移動できる.
条件 2：どの円盤もその円盤より小さい円盤の上に置いてはいけない.
条件 3：棒 C を補助的な場所として使用してよい.

・再帰的な解法

棒 A から棒 B に n 枚移動する場合は,

1) まず，上から n-1 枚を棒 A から棒 C に移動. ← 「n-1 枚を動かす」再帰的
2) 棒 A にある円盤一つを棒 B に移動（最も下にあるベースとなる円盤の移動）
3) 棒 C の上から n-1 枚を棒 B に移動. ← 「n-1 枚を動かす」再帰的

※ 再帰を使うと良くない場合

- 問題が再帰的に定義できる ≠ 再帰的アルゴリズムが最良な方法
- ある場合には再帰的な定義を単純な繰り返しに直すことが可能 → こちらのほうが高速

※ 本質的には，非再帰的な機械を使って，再帰的手続きが実現されているのであるから，すべての再帰的なプログラムは繰り返しのプログラムに書き換え可能. しかし，「スタック」（先入れ後だしのデータ保存場所）が必要. 問題は，繰り返しアルゴリズムが明解であること.

◎ 典型的図式 --- 末尾再帰（tail recursion）

- 終り（もしくは始め）において，1 回だけ P の（再帰）呼び出しがある場合.

(a') P ⇔ if (B) { S; P }

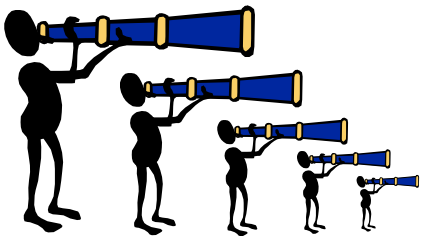
(b') P ⇔ S;　if (B) { P }　→ S の後に if(B){P}を実行

↓

P ⇔ f = f0;　while (B) { S }

※ f は一つ前の計算結果を保存するための変数.

例 --- 階乗計算（ fact.c → 本資料 p.5 ）



◎ 少し複雑な例 --- フィボナッチ数

・フィボナッチ数

fib(0) ⇔ 0,　fib(1) ⇔ 1,　fib(n) ⇔ fib(n-1) + fib(n-2)

・fib(i-1) と fib(i-2) を保存する変数を用意すればよい.

再帰版

```
int fib( int n )
{
  if (n == 0 )
    return 0;
  else if ( n == 1 )
    return 1;
  else
    return fib(n-1)+fib(n-2);
}
```

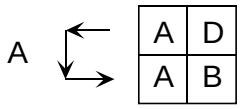
ループ版

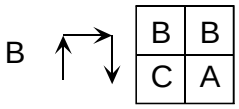
```
int fib( int n )
{
  int x=1, y=0, i=1, z;   /* x<->fib(i-1), y<->fib(i-2) */
  /* n==0, n==1 の場合は省略 */
  while ( i < n ){
    z = x;   x = x + y;   y = z;
    i++;
  }
  return x;
}
```

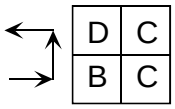
☆ 再帰プログラムの例その２

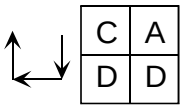
◎ ヒルベルト曲線（ hilbert.c → 本資料 p.8 ）

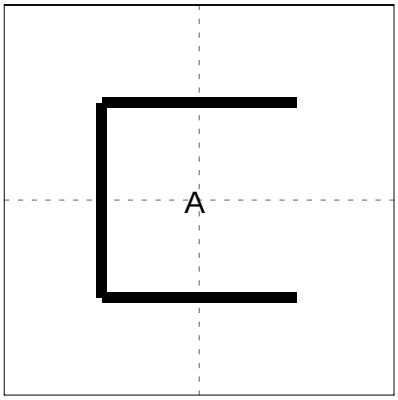
- 考案者 D. Hilbert
- 基本となる**"コの字"**形の線図形を，半分に縮小して回転したものを合成する操作を再帰的に行うことによって複雑な曲線を描くもの．
- このような曲線は一般に**空間充填曲線**と呼ばれ，ヒルベルト曲線以外にも，ペアノ曲線，ドラゴン曲線などがある．

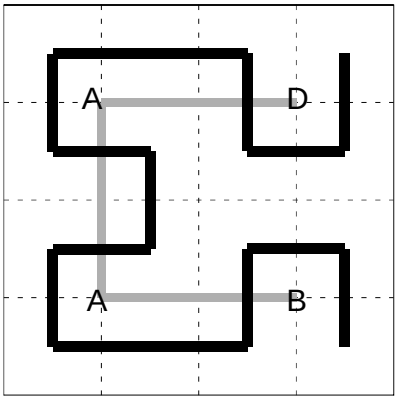


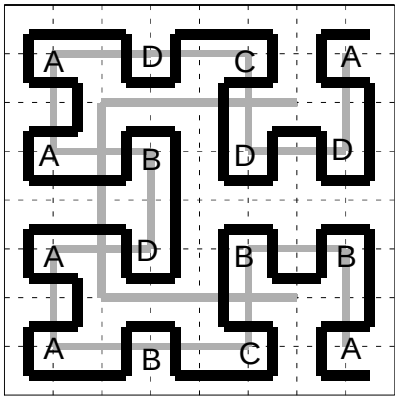












再帰は，最初は混乱するかも知れませんが，慣れれば非常に楽な方法であることがわかります．

ふんふん，
なるほど．．．

```
/*************************************************************
     「アルゴリズムとデータ構造」
     サンプルプログラム
     << 再帰プログラムの例: 階乗　fact.c　>>
     copyright (c)　T.Mori <mori@forest.dnj.ynu.ac.jp>
*************************************************************/
#include <stdio.h>
int fact_rec(int n);
int fact_nonrec(int m);

int main(void)
{
     int m;

     m = 10;
     printf("階乗(再帰)     = %d¥n", fact_rec(m));
     printf("階乗(非再帰) = %d¥n", fact_nonrec(m));
     return 0;
}

/* 再帰版 階乗計算 */
fact_rec( int n )
{
     if ( n <= 0 )  return 1;
     else   return n*fact_rec(n-1);      /* f(n) = n * f(n-1) */
}

/* 非再帰版 階乗計算 */
fact_nonrec( int m )
{
     int fn,n;

     fn = 1;                /* f(0) = 1 */
     n = 1;
     while (n<=m) {
          fn = n*fn;       /* f(n) = n*f(n-1) */
          n++;
     }
     return fn;
}
```

【実行結果】

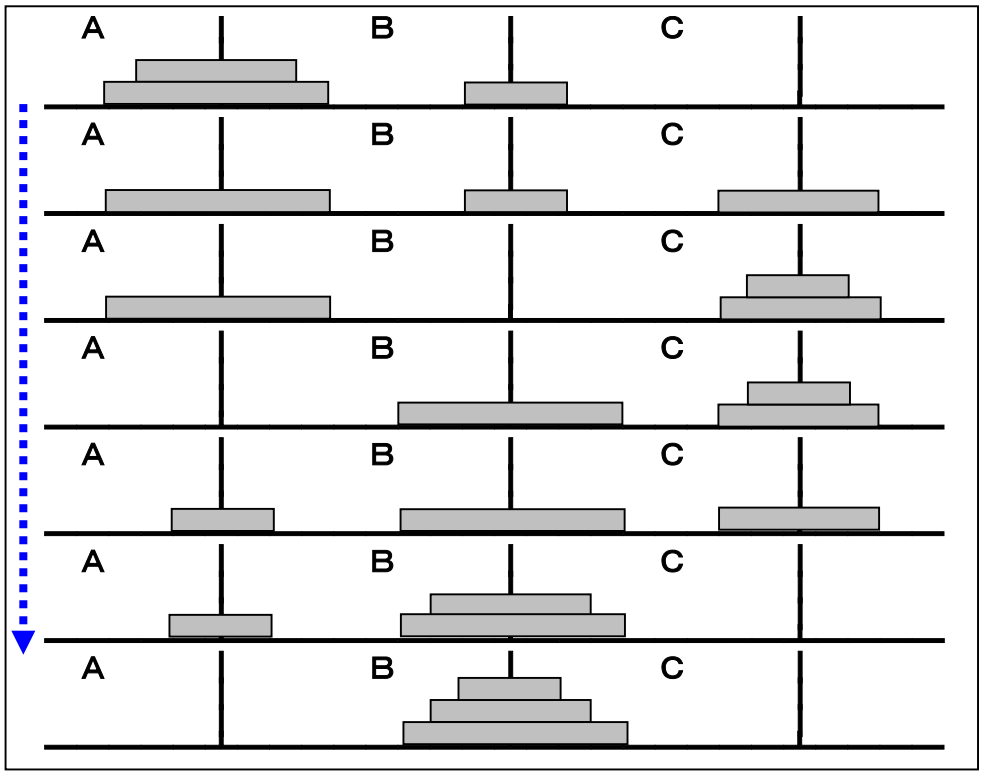
```
階乗(再帰)     = 3628800
階乗(非再帰) = 3628800
```

```
/*************************************************************
     「アルゴリズムとデータ構造」
     サンプルプログラム
     <<再帰プログラムの例：ハノイの塔>>　hanoi.c
     copyright (c)　T.Mori <mori@forest.dnj.ynu.ac.jp>

【ハノイの塔】
初期状態
              3 枚
          ＝｜＝              |                |
        ＝＝｜＝＝            |                |
      ＝＝＝｜＝＝＝          |                |
―――――――――――――――――――――――――――――――
           棒 A              棒 B             棒 C
目標状態
             |                ＝｜＝           |
             |              ＝＝｜＝＝         |
             |            ＝＝＝｜＝＝＝       |
―――――――――――――――――――――――――――――――
           棒 A              棒 B             棒 C

条件 1: 一度に一枚の円盤だけが移動できる.
条件 2: どの円盤もその円盤より小さい円盤の上においてはいけない.
条件 3: 棒 C を補助的な場所として使用してよい.

*************************************************************/
#include <stdio.h>
void hanoi(int height,   char *src, char *dst, char *work);
void move(char *src, char *dst);

int main(void)
{
  /* "棒 A" から 3 枚の円盤を "棒 B" に移動する. */
  /* このとき "棒 C" を円盤の一時保持場所として利用する. */
  hanoi( 3, "棒 A", "棒 B", "棒 C" );
  return 0;
}

/* src にある ndisk 枚の円盤を dst に移動する. */
/* このとき work を円盤の一時保持場所として利用する. */
void hanoi( int ndisk,   char *src,   char *dst,   char *work)
{
  if ( ndisk >= 1 ) {                    /* 移動するのが 1 枚以上の場合は, */
    hanoi(ndisk-1, src, work, dst);      /* まず, ndisk-1 枚を作業領域に移し */
    move(src,dst);                       /* ndisk 枚めを目標の棒に移し */
    hanoi(ndisk-1, work, dst, src);      /* 作業領域の ndisk-1 枚を目標の棒 */
                                         /* に移動する */
```

全てコメントです.

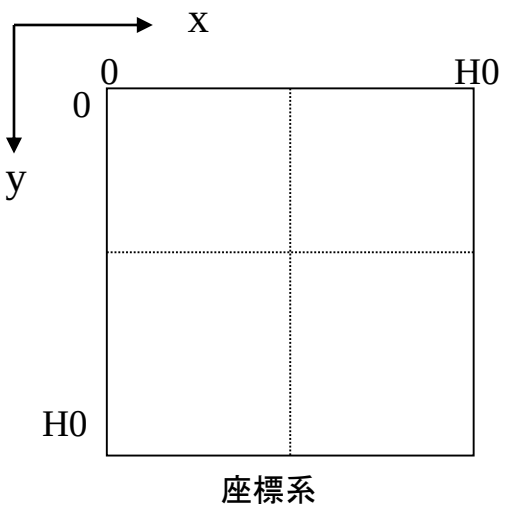
```
48      }
49    }
50
51    /* src の円盤を dst に移動する手続き */
52    /* ここでは「移動する」というメッセージを印刷するだけ */
53    void move(char *src, char *dst)
54    {
55      printf("%s にある一番上の円盤を%s へ移動する¥n", src, dst);
56    }
57
58    【実行結果】
59    棒 A にある一番上の円盤を棒 B へ移動する
60    棒 A にある一番上の円盤を棒 C へ移動する
61    棒 B にある一番上の円盤を棒 C へ移動する
62    棒 A にある一番上の円盤を棒 B へ移動する
63    棒 C にある一番上の円盤を棒 A へ移動する
64    棒 C にある一番上の円盤を棒 B へ移動する
65    棒 A にある一番上の円盤を棒 B へ移動する
66
```

図で書くと
こうなります.

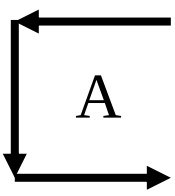


超ムズいパズルが
超楽に解けちゃっ
た！　再帰って，
スゴくない？↑

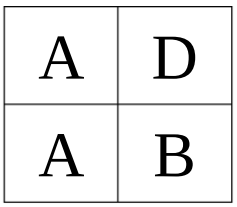
```
/*************************************************************
     「アルゴリズムとデータ構造」
     サンプルプログラム
     <<再帰の例: Hilber 曲線>>   hilbert.c
     copyright (c)   T.Mori <mori@forest.dnj.ynu.ac.jp>
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#define H0 16       /* 画面の大きさ */
#include "plot.c"   /* 簡易プロッタルーチン(H0 の値を使用する) */
#define N  3        /* ヒルベルト曲線の位数 */

void pat_A( int i, int h );
void pat_B( int i, int h );
void pat_C( int i, int h );
void pat_D( int i, int h );

int main(void)
{
    int i, h, x0, y0;

    init_screen();
    i=0;   h = H0;   x0 = h/2;   y0 = x0;
    do {
        i++;
        h /= 2;   x0 += h/2;   y0 -= h/2;
        set_pen(x0,y0);
        pat_A(i,h);
    } while (i != N);
    print_screen();
    return 0;
}

/* パターン A の描画 */
void pat_A(int i, int h)
{
    if (i>0) {
        pat_D(i-1,h); move_pen(-h,0);
        pat_A(i-1,h); move_pen(0,h);
        pat_A(i-1,h); move_pen(h,0);
        pat_B(i-1,h);
    }
}

/* パターン B の描画 */
void pat_B(int i, int h)
{
    if (i>0) {
        pat_C(i-1,h); move_pen(0,-h);
```



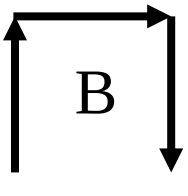
座標系



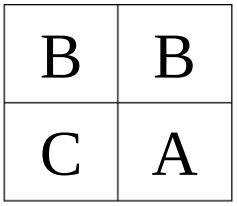
基本型



分割法

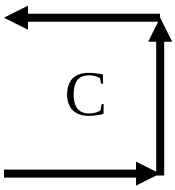


基本型



分割法

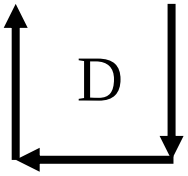
```
50          pat_B(i-1,h); move_pen(h,0);
51          pat_B(i-1,h); move_pen(0,h);
52          pat_A(i-1,h);
53      }
54  }
55
56  /* パターン C の描画 */
57  void pat_C(int i, int h)
58  {
59      if (i>0) {
60          pat_B(i-1,h); move_pen(h,0);
61          pat_C(i-1,h); move_pen(0,-h);
62          pat_C(i-1,h); move_pen(-h,0);
63          pat_D(i-1,h);
64      }
65  }
66
67  /* パターン D の描画 */
68  void pat_D(int i, int h)
69  {
70      if (i>0) {
71          pat_A(i-1,h); move_pen(0,h);
72          pat_D(i-1,h); move_pen(-h,0);
73          pat_D(i-1,h); move_pen(0,-h);
74          pat_C(i-1,h);
75      }
76  }
77
78  【実行結果】
79
```



基本型

D	C
B	C

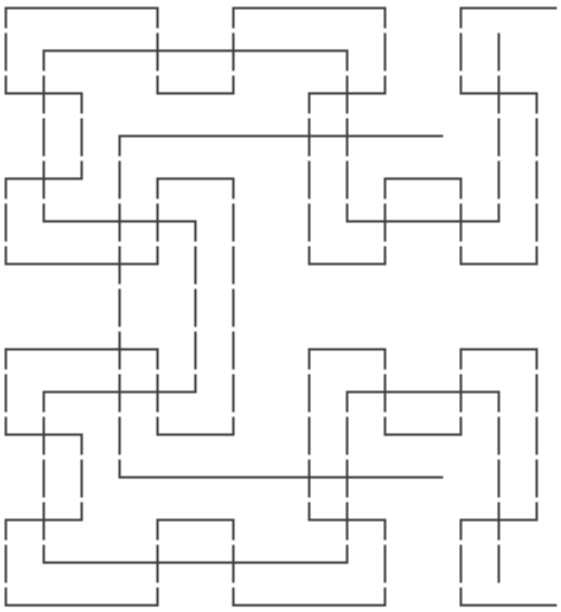
分割法



基本型

C	A
D	D

分割法



```
/*************************************************************
     「アルゴリズムとデータ構造」
     サンプルプログラム
     <<簡易プロッタルーチン>>　plot.c
     copyright (c)　T.Mori <mori@forest.dnj.ynu.ac.jp>
*************************************************************/
/*************************************************************
 【使い方】
 ・画面は正方形
 ・画面の一辺を H0 というマクロで定義する.
 ・画面は以下の座標系.
   +------>x
   |
   |
   V
   y

 ・使用できる関数
void init_screen(void)          画面を初期化する.
void set_pen(int sx, int sy)    ペンをアップし，ペンの位置を（sx,sy)
                                に移動する.
void move_pen(int dx, int dy)   ペンをダウンさせて，ペンの位置を現在の座標
                                から x 座標を dx, y 座標を dy だけ移動する.
void print_screen(void)         画面を印刷する.
*************************************************************/

/* 一つのマスからどの方向に線が出ているかの定義 */
#define UP      1
#define DOWN    2
#define RIGHT   4
#define LEFT    8
/* 画面データ */
int screen[H0][H0];
/* ペンの座標 */
int x, y;

/* 画面ならびにペン座標の初期化 */
void init_screen(void)
{
    int i, j;

    x = 0; y = 0;
    for( i=0; i<H0; i++ )
        for( j=0; j<H0; j++ )
            screen[i][j] = 0;
}
```

この関数はちょっと難しいですから，無理して理解しなくても大丈夫です.

2 次元配列 screen

UP　0001（2 進数）
DOWN　0010（2 進数）
RIGHT　0100（2 進数）
LEFT　1000（2 進数）

大域変数

ゼロ“0”が何もない状態を表しています.

```
49  /* ペン座標設定 */
50  void set_pen( int sx, int sy )
51  {
52      x = sx;   y = sy;
53  }
54
55
56  /* ペンを相対移動し，描画する */
57  void move_pen( int dx, int dy )
58  {
59      int i, x_start, x_end, y_start, y_end;
60
61      if ( dx != 0   &&   dy != 0 ) {
62          printf("Plot Error¥n");
63          exit(1);
64      }
65      if ( dx != 0 ) {
66          if ( dx < 0 ) {
67              x_start = x + dx;
68              x_end = x;
69          } else {
70              x_start = x;
71              x_end = x + dx;
72          }
73          for ( i=x_start+1; i<x_end; i++ )
74              screen[i][y] |= RIGHT|LEFT;
75          screen[x_start][y] |= RIGHT;
76          screen[x_end][y] |= LEFT;
77      }
78      if ( dy != 0 ) {
79          if ( dy < 0 ) {
80              y_start = y + dy;
81              y_end = y;
82          } else {
83              y_start = y;
84              y_end = y + dy;
85          }
86          for ( i=y_start+1; i<y_end; i++ )
87              screen[x][i] |= UP|DOWN;
88          screen[x][y_start] |= DOWN;
89          screen[x][y_end] |= UP;
90      }
91      x += dx;
92      y += dy;
93  }
94
95
96
97
```

この x と y は，あらゆる関数から参照することができる大域変数です．本来なら，大域変数はあまり使わない方が良いですが，ここではプログラムを分かり易くするために用いています．

強制終了

| はビットごとの OR 演算子
screen[i][y] |= RIGHT|LEFT
∴screen[i][y] |= 1100
∴screen[i][y] = screen[i][y] | 1100
これは，screen[i][y] で 1100 のビットが立っているビットを立てる処理を表しています．以下同様です．

UP	0001（2 進数）
DOWN	0010（2 進数）
RIGHT	0100（2 進数）
LEFT	1000（2 進数）

```
 98  /* 画面描画 */
 99  void print_screen(void)
100  {
101      int i,j;
102
103      for ( i=0; i<H0; i++) {
104          for ( j=0; j<H0; j++){
105              switch(screen[j][i]) {
106                  case UP: case DOWN: case UP|DOWN:
107                      printf("│"); break;
108                  case RIGHT: case LEFT: case RIGHT|LEFT:
109                      printf("―"); break;
110                  case LEFT|DOWN:
111                      printf("┐"); break;
112                  case RIGHT|DOWN:
113                      printf("┌"); break;
114                  case LEFT|UP:
115                      printf("┘"); break;
116                  case RIGHT|UP:
117                      printf("└"); break;
118                  case LEFT|RIGHT|DOWN:
119                      printf("┬"); break;
120                  case LEFT|UP|DOWN:
121                      printf("┤"); break;
122                  case LEFT|RIGHT|UP:
123                      printf("┴"); break;
124                  case RIGHT|UP|DOWN:
125                      printf("├"); break;
126                  case LEFT|RIGHT|UP|DOWN:
127                      printf("┼"); break;
128                  default:
129                      printf("　");
130              }
131          }
132          printf("¥n");
133      }
134  }
```

screen[j][i]のどのビットが立っているかで処理を分岐しています．例えば，LEFT|DOWN は 1010 を表しています．その場合は「左方向，下方向に線が出ている」ですから，┐と表示するということになります．以下同様です．

UP	0001（2 進数）
DOWN	0010（2 進数）
RIGHT	0100（2 進数）
LEFT	1000（2 進数）