ソフトウェア２
第１回
(2015/11/19)

鶴岡 慶雅

ソフトウェア２

- C言語入門
  - 構造体
  - ファイル入出力
  - データ構造

- アプリケーション
  - 数値計算
  - シミュレーション
  - 探索・最適化
  - etc

連絡用ページ

- URL

  http://www.logos.t.u-tokyo.ac.jp/~tsuruoka/lecture/software2/

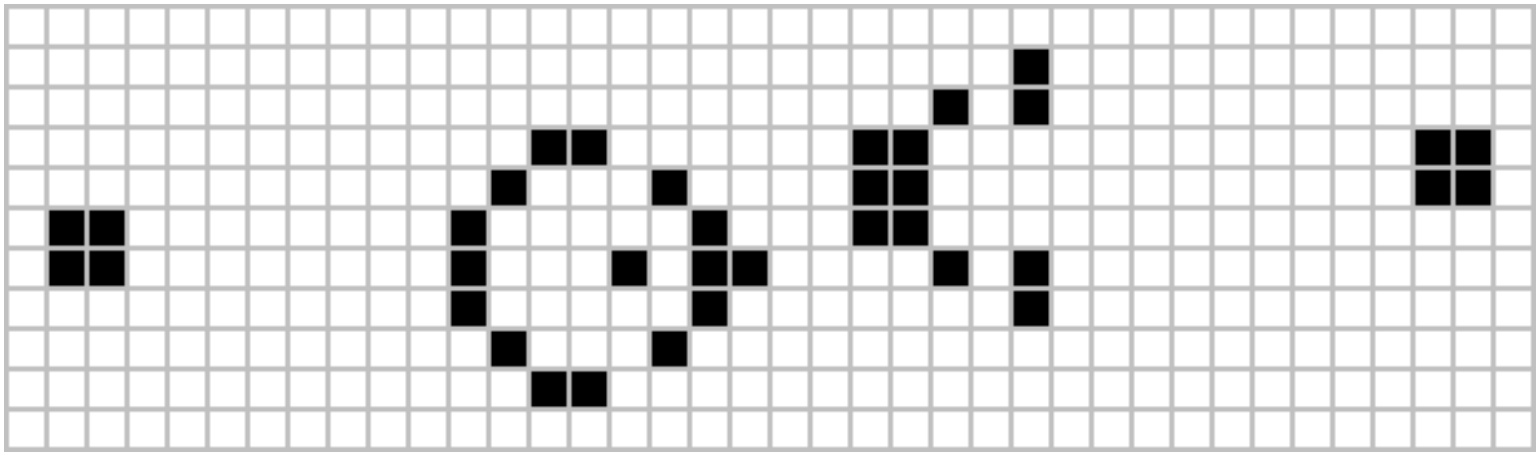
  ユーザ名 :　ee2015

  パスワード :　soft2

- 資料
  - 講義スライド
  - サンプルプログラム

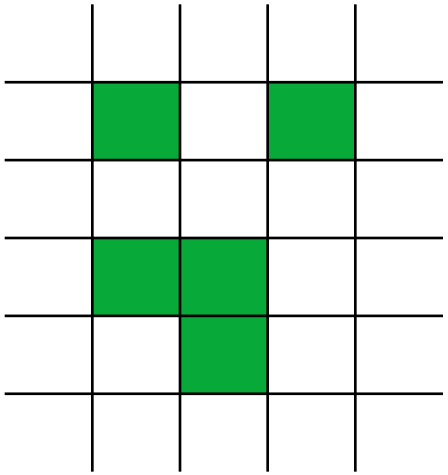
ライフゲーム

- Conway’s Game of Life (Conway 1970)
  - イギリスの数学者 John Conway が提案
  - 生命の誕生・衰亡・変化のシミュレーション



http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

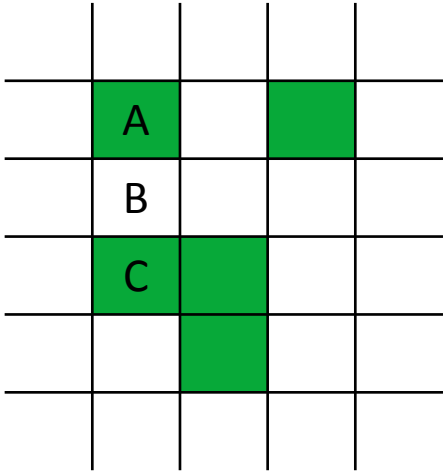
ライフゲーム



- ルール
  - ２次元グリッド上にセル（cell）が存在
  - 次の世代
    - 生きているセルが有る場所
      - 周囲に生きているセルが2個または3個存在するならばそのまま
      - そうでなければセルは消滅
    - 生きているセルが無い場所
      - 周囲に生きているセルがちょうど3個存在するならばセルが誕生
      - そうでなければそのまま

例

- A, B, C の位置にあるセルは、それぞれ次世代にどう変化するか？



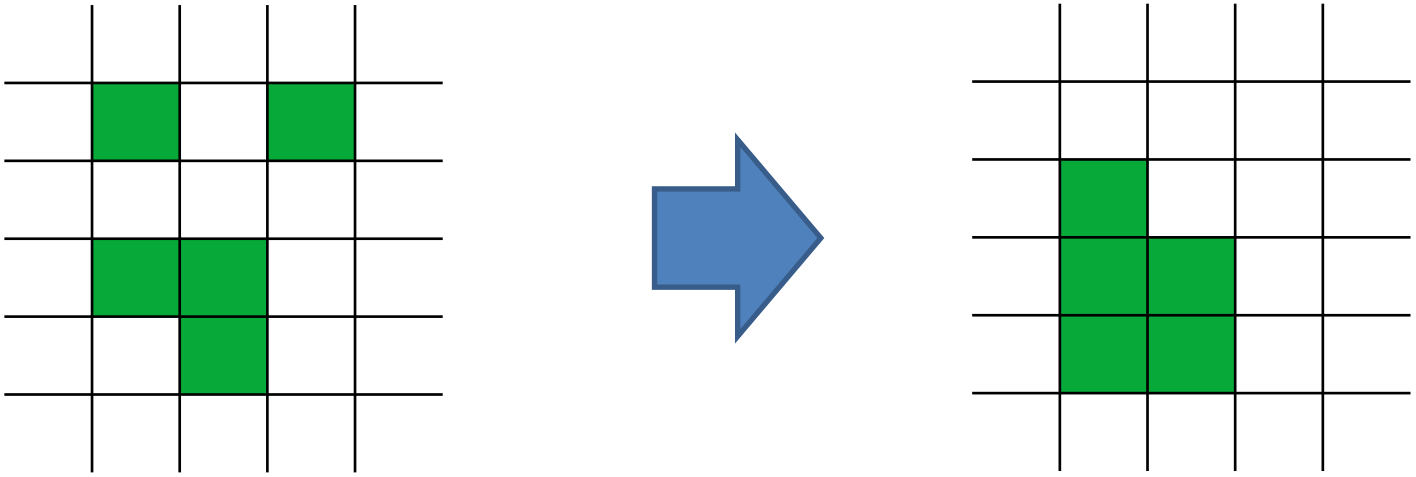
A: 周囲にセルが存在しないので消滅

B: 周囲にセルが3個存在するので新しいセルが誕生

C: 周囲にセルが2個存在するのでそのまま

例

- １世代進むと結局、



これが　　こうなる

サンプルプログラム life.c

- コンパイル＆実行

```
% gcc -Wall life.c
% ./a.out
```

- ターミナルをもうひとつ開く（結果表示用）

```
% tail –f cells.txt
```

ターミナルのサイズをマウスで調整（縦に大きく）して
---------- が左上にくるように

life.c

- ヘッダーファイルの読み込み（インクルード）

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

  - コンパイラ（正確にはプリプロセッサ）が、個々の #include 文をヘッダーファイルで置き換える
  - ヘッダーファイル：
    - 関数の宣言やマクロの定義が記述されているファイル
    - 例えば、printf 関数をプログラム中で使いたい場合、その宣言が記述されているヘッダーファイル stdio.h をインクルードする必要がある

life.c

- 記号定数の宣言

```
#define HEIGHT 50
#define WIDTH 70
```

  - 宣言以降に出現する HEIGHT, WIDTH が、それぞれ 50, 70 に置き換えられる。
  - このようにしておくと、グリッドの高さ（HEIGHT）や幅（WIDTH）を変更したくなった場合、ここだけを変更すれば済むので便利

life.c

- グローバル変数の定義

```
int cell[HEIGHT][WIDTH];
```

  - 整数型の2次元配列でセルの状態を保持
    - 大きさは HEIGHT × WIDTH つまり 50 × 70 = 3500
  - グローバル変数なので、プログラム中のどの関数からも参照・変更が可能

init_cells() 関数

- セルの状態を初期化する

```
void init_cells()
{
  int i, j;

  for (i = 0; i < HEIGHT; i++) {
    for (j = 0; j < WIDTH; j++) {
      cell[i][j] = 0;
    }
  }

  cell[20][30] = 1;
  cell[22][30] = 1;
  cell[22][31] = 1;
  cell[23][31] = 1;
  cell[20][32] = 1;
}
```

2次元配列中のすべての値をゼロで初期化
→ 生きたセルがいない状態

5つの生きたセルを配置

print_cells() 関数

- セルを表示する

```
void print_cells(FILE *fp)
{
  int i, j;

  fprintf(fp, “----------¥n”);

  for (i = 0; i < HEIGHT; i++) {
    for (j = 0; j < WIDTH; j++) {
      char c = (cell[i][j] == 1) ? '#' : ' ';
      fputc(c, fp);
    }
    fputc('¥n', fp);
  }
  fflush(fp);

  sleep(1);
}
```

ファイル構造体へのポインタ

生きたセルが存在するときは # を出力

バッファの内容を強制出力
（tail –f での表示が乱れないように）

1秒停止

print_cells() 関数

- ? を用いた記法について

```
char c = (cell[i][j] == 1) ? '#' : ' ';
```

  - 以下のように書くのと同じ

```
char c;
if (cell[i][j] == 1)
  c = '#';
else
  c = ' ';
```

print_cells() 関数

- '#' とは何か？

```
char c = (cell[i][j] == 1) ? '#' : ' ';
```

  - 以下と同じ

```
char c = (cell[i][j] == 1) ? 35 : 32;
```

  - コンパイル時に、それぞれ、#（シャープ記号）や空白記号のASCII文字コードに置き換えられる

参考） ASCII文字コード

コード	文字
:	:
32	（空白）
33	!
34	″
35	#
36	$
37	%
38	&
39	′
40	(
:	:

コード	文字
:	:
65	A
66	B
67	C
:	:
97	a
98	b
99	c
:	:
126	~
127	

count_adacent_cells() 関数

- 周囲のセルの数を数える

注目する場所を引数として受け取る

```
int count_adjacent_cells(int i, int j)
{
  int n = 0;
  int k, l;
  for (k = i - 1; k <= i + 1; k++) {
    if (k < 0 || k >= HEIGHT) continue;
    for (l = j - 1; l <= j + 1; l++) {
      if (k == i && l == j) continue;
      if (l < 0 || l >= WIDTH) continue;
      n += cell[k][l];
    }
  }
  return n;
}
```

周囲8マス

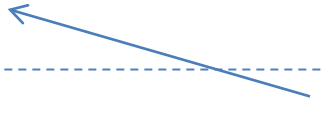
(i-1, j-1)	(i-1, j)	(i-1, j+1)
(i, j-1)	(i, j)	(i, j+1)
(i+1, j-1)	(i+1, j)	(i+1, j+1)

周囲の生きているセルの数を返り値として返す

if 文について

- k がグリッド内かどうかをチェック

```
if (k < 0 || k >= HEIGHT) continue;
```



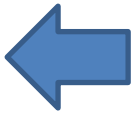
  - k がOより小さいかHEIGHT 以上であれば continue（以降を実行せず次のループへ）
  - もちろん以下のように書いてもよい

```
if (k < 0 || k >= HEIGHT) {
  continue;
}
```

cell を数える

- (k, l) にセルが存在ならば n を1増やす

```
n += cell[k][l];
```



略記

```
n = n + cell[k][l];
```

  - 以下のように if 文を使って書いてもよいが、cell[k][l] の値は0か1のどちらかなので、上のようにシンプルに書ける

```
if (cell[k][l] == 1) {
  n = n + 1;
}
```

update_cells() 関数

- 次世代のセルの状態を計算

```
void update_cells()
{
  int i, j;
  int cell_next[HEIGHT][WIDTH];

  for (i = 0; i < HEIGHT; i++) {
    for (j = 0; j < WIDTH; j++) {
     cell_next[i][j] = 0;
    int n = count_adjacent_cells(i, j);
      ?
    }
  }

  for (i = 0; i < HEIGHT; i++) {
    for (j = 0; j < WIDTH; j++) {
      cell[i][j] = cell_next[i][j];
    }
  }
}
```

周囲8マスのセルの数を数える

ルールに従って次世代のセルの状態を計算する

cell[][]を次世代の状態に更新

cell_next[][] について

- 次世代のセルの状態を計算するためのテンポラリ(一時的)な2次元配列

```
void update_cells()
{
  int i, j;
  int cell_next[HEIGHT][WIDTH];
```

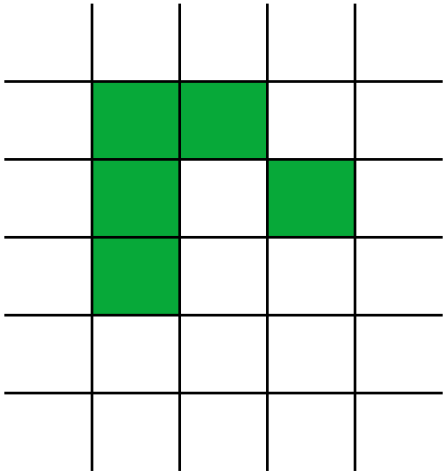
  - この関数の中だけで有効
  - スタック領域に確保され、この関数の実行が終わったら解放される

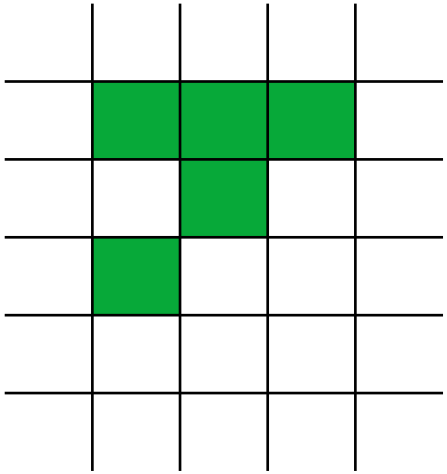
→ アクセス不可になるので注意

実習

1. update_cells() 関数を完成させる。

2. 以下の2つの初期配置を試す。





メイン関数

```
int main()
{
  int gen;
  FILE *fp;

  if ((fp = fopen("cells.txt", "a")) == NULL) {
    fprintf(stderr, "error: cannot open a file.¥n");
    return 1;
  }

  init_cells();
  print_cells(fp);

  for (gen = 1;; gen++) {
    printf("generation = %d¥n", gen);
    update_cells();
    print_cells(fp);
  }

  fclose(fp);
}
```

結果出力用のファイルを開く

セルの初期配置を決めて表示

セルの更新・表示を繰り返す

FILE構造体

- 構造体
  - いくつかの変数をまとめ、ひとかたまりの変数として扱えるようにする仕組み
  - 次回以降で詳しく説明

- FILE構造体
  - ファイルの入出力を行うための構造体
  - ファイルアクセスに関する情報が保存されている
    - どのファイルのどの場所をアクセスしているか等

fopen() 関数について

- ファイルを開く

```
FILE *fp;

if ((fp = fopen("cells.txt", "a")) == NULL) {
  fprintf(stderr, "error: cannot open a file.¥n");
  return 1;
}
```

  - fopen() 関数は、第１引数でファイル名、第２引数でファイルアクセスのモードを指定
    - モード： “r”（読み込み）, “w”（書き出し）, “a” （追記）
  - ファイルのオープンに成功した場合はファイルポインタが返ってくる
    - 以後、そのファイルポインタを使ってそのファイルに対する読み書きを行うことができる
  - ファイルのオープンに失敗した場合は NULL が返ってくる

if 文の書き方について

- ファイルを開く

```
FILE *fp;

if ((fp = fopen("cells.txt", "a")) == NULL) {
  fprintf(stderr, "error: cannot open a file.¥n");
  return 1;
}
```

  - 上記の if 文は以下のように書いたのと同じ

```
fp = fopen("cells.txt", "a");
if (fp == NULL) {
  fprintf(stderr, "error: cannot open a file.¥n");
  return 1;
}
```

  - 代入式の値は代入演算された後の値になるので

stderr について

- ファイルを開くのに失敗した場合、標準エラー出力にメッセージを出力して、プログラムの実行を終了する

```
fprintf(stderr, ”error: cannot open a file. ¥n”);
return 1;
```

- 標準エラー出力
  - 普通はシェル上でメッセージを見ることになる
  - stderr: 標準エラー出力へのファイルポインタ
- main関数の返り値
  - 正常終了ならゼロ、異常終了ならゼロ以外の値を返す

ファイル入出力のための関数

- ファイルアクセス関数など

```
FILE *fopen(char *name, char *mode);
```
ファイルを開く

```
int fprintf(FILE *fp, const char *format);
```
ファイルfpへ書式付で文字列を出力

```
int fputc(int c, FILE *fp);
```
ファイルfpへ文字cを出力

```
int fclose(FILE *fp);
```
ファイルを閉じる

レポート課題（締め切り11/25）

1. 各世代で存在するセルの総数を表示するように “life.c” を拡張せよ
   - プログラムを添付すること（ファイル名は “life1.c”）
   - ``life.c'' を利用せず自分でゼロからプログラムを作成しても構わない

2. 初回配置パターンがランダムになるように修正せよ
   - プログラムを添付すること（ファイル名は “life2.c”）
   - 生きたセルの割合がほぼ 10% になるようにすること

3. 初期配置パターンをファイルから読み込めるように拡張せよ
   - プログラムを添付すること（ファイル名は “life3.c”）
   - ファイルからの入力に関しては “readfile.c” を参照

4. ［発展課題］ ライフゲームのルールを適当に変更しその振る舞いを確認せよ
   - プログラムを添付すること（ファイル名は “life4.c”）
   - どのようなルールに変更し、その結果として、どのような振る舞いが見られたかを簡単に説明すること
   - ルールは大幅に変更しても構わない（生物種を増やす、地形効果を導入、etc）

課題の提出方法

- 宛先
  - software2@logos.t.u-tokyo.ac.jp
- Subject
  - 形式: SOFT-MM-DD-NNNNNNX
    - MM: 月
    - DD: 日（授業が行われた日）
    - NNNNNNX: 学籍番号
- 本文
  - 冒頭に学籍番号、氏名を明記

乱数発生について

- rand() 関数
  - ヘッダーファイル：stdlib.h
  - 0 ～ RAND_MAX（非常に大きな整数）の範囲で疑似乱数を返す
  - 0以上5未満の整数乱数を得たいときは

```
int r = rand() % 5;
```

    とすればよい
  - 浮動小数点数乱数を得たい場合は、たとえば、

```
double r = (double)rand() / (RAND_MAX+1);
```

    とすると [0, 1) の範囲の乱数が得られる

乱数発生について

- 発生する疑似乱数の系列はつねに同じ

- srand() 関数
  - 疑似乱数の seed（種）を設定
  - プログラムの実行のたびに違う乱数系列を発生させたい場合はmain() の先頭で

    ```
    srand(time(NULL));
    ```

    を1度実行すればよい

readfile.c

- ファイル入力のサンプルプログラム
  - ファイルを開く
  - 各行の文字数を表示

- 標準ライブラリ関数

```
fgets(char *s, int size, FILE *fp);
```

  ファイルから1行読み込んでバッファに保存

```
size_t strlen(const char *s);
```

  文字列の長さを取得