```
/*=== puzmon9: コンボ処理の実装（完成） ===*/
/*** インクルード宣言 ***/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <string.h>

/*** 列挙型宣言 ***/

// (a)属性
typedef enum Element {FIRE, WATER, WIND, EARTH, LIFE, EMPTY} Element;

/*** グローバル定数の宣言 ***/

// (b)属性別の記号
const char ELEMENT_SYMBOLS[EMPTY+1] = {'$','~','@','#','&', ' '};

// (c)属性別のカラーコード（ディスプレイ制御シーケンス用）
const char ELEMENT_COLORS[EMPTY+1] = {1,6,2,3,5,0};

// (d)バトルフィールドに並ぶ宝石の数
enum {MAX_GEMS = 14};

// (e)属性別の強弱関係
const double ELEMENT_BOOST[EMPTY+1][EMPTY+1] = {
 // FIRE WATER WIND EARTH LIFE EMPTY
  { 1.0, 0.5, 2.0, 1.0, 1.0, 1.0 }, // FIRE
  { 2.0, 1.0, 1.0, 0.5, 1.0, 1.0 }, // WATER
  { 0.5, 1.0, 1.0, 2.0, 1.0, 1.0 }, // WIND
  { 1.0, 2.0, 0.5, 1.0, 1.0, 1.0 }, // EARTH
  { 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 }, // LIFE
  { 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 }  // EMPTY
};

/*** 構造体型宣言 ***/

// (f) モンスター
typedef struct MONSTER {
  char* name;
  Element element;
  int maxhp;
  int hp;
  int attack;
```

```
  int defense;
} Monster;

// (g)ダンジョン
typedef struct DUNGEON {
  Monster* monsters;
  const int numMonsters;
} Dungeon;

// (h)パーティ
typedef struct PARTY {
  char* playerName;
  Monster* monsters;
  const int numMonsters;
  const int maxHp;
  int hp;
  const int defense;
} Party;

// (i)バトルフィールド
typedef struct BATTLE_FIELD {
  Party* pParty;
  Monster* pEnemy;
  Element gems[MAX_GEMS];
} BattleField;

// (j)連続配置宝石の位置情報
typedef struct BANISH_INFO {
  int pos;
  int len;
  Element element;
} BanishInfo;

/*** プロトタイプ宣言 ***/

int goDungeon(Party* pParty, Dungeon* pDungeon);
int doBattle(Party* pParty, Monster* pEnemy);
Party organizeParty(char* playerName, Monster* monsters, int numMonsters);
void showParty(Party* pParty);
void onPlayerTurn(BattleField* pField);
void doAttack(BattleField* pField, BanishInfo bi, int numCombo);
void onEnemyTurn(BattleField* pField);
void doEnemyAttack(BattleField* pField);
void showBattleField(BattleField* pField);
bool checkValidCommand(char* command);
void evaluateGems(BattleField* pField);
```

```
//(10)別解の場合、void evaluateGems(BattleField* pField, int numCombo);
BanishInfo checkBanishable(Element* gems);
void banishGems(BattleField *pField, BanishInfo bi, int numCombo);
void shiftGems(Element* gems);
void spawnGems(Element* gems);
void doRecover(BattleField* pField, BanishInfo bi, int numCombo);

// ユーティリティ関数
void printMonsterName(Monster* monster);
void fillGems(Element* gems, bool emptyOnly);
void printGems(Element* gems);
void printGem(Element element);
void moveGem(Element* gems, int fromPos, int toPos, bool printProcess);
void swapGem(Element* gems, int pos, int step);
int countGems(Element* gems, Element target);
int blurDamage(int base, int percent);
int calcEnemyAttackDamage(Party* pParty, Monster* pEnemy);
int calcAttackDamage(Monster* pAttackMonster, Monster* pEnemy, BanishInfo bi, int
numCombo);
int calcRecoverDamage(BattleField *pField, BanishInfo bi, int numCombo);
void printCombo(int numCombo);

/*** 関数宣言 ***/

// (1)ゲーム開始から終了までの流れ
int main(int argc, char** argv)
{
    srand((unsigned)time(NULL));

    if(argc != 2) {
      printf("エラー: プレイヤー名を指定して起動してください\n");
      return 1;
    }

    printf("*** Puzzle & Monsters ***\n");

    // パーティの準備
    Monster partyMonsters[] = {
      {"朱雀", FIRE,  150, 150, 25, 10},
      {"青龍", WIND,  150, 150, 15, 10},
      {"白虎", EARTH, 150, 150, 20,  5},
      {"玄武", WATER, 150, 150, 20, 15}
    };
    Party party = organizeParty(argv[1], partyMonsters, 4);
```

ポインタ変数　先頭のアドレス

```
    // ダンジョンの準備
    Monster dungeonMonsters[] = {
      {"スライム",    WATER, 100, 100, 10,  5},
      {"ゴブリン",    EARTH, 200, 200, 20, 15},
      {"オオコウモリ", WIND,  300, 300, 30, 25},
      {"ウェアウルフ", WIND,  400, 400, 40, 30},
      {"ドラゴン",    FIRE,  800, 800, 50, 40}
    };
    Dungeon dungeon = {dungeonMonsters, 5};

    // いざ、ダンジョンへ
    int winCount = goDungeon(&party, &dungeon);

    // 冒険終了後
    if(winCount == dungeon.numMonsters) {
      printf("***GAME CLEAR!***\n");
    }  else {
      printf("***GAME OVER***\n");
    }
    printf("倒したモンスター数＝%d\n", winCount);
    return 0;
}

// (2)ダンジョン開始から終了までの流れ
int goDungeon(Party* pParty, Dungeon* pDungeon)
{
    // プレイヤーHP/最大HPの算出とメッセージ表示
    printf("%sのパーティ(HP=%d)はダンジョンに到着した\n",pParty->playerName,
pParty->hp);
    showParty(pParty);

    // そのダンジョンでバトルを繰り返す
    int winCount = 0;
    for(int i = 0; i < pDungeon->numMonsters; i++) {
      winCount += doBattle(pParty, &(pDungeon->monsters[i]));
      if(pParty->hp <= 0) {
        printf("%sはダンジョンから逃げ出した...\n", pParty->playerName);
        break;
      } else {
        printf("%sはさらに奥へと進んだ\n\n", pParty->playerName);
        printf("================\n\n");
      }
    }

    printf("%sはダンジョンを制覇した！\n", pParty->playerName);
```

ポインタ変数

パーティ、ダンジョンのアドレス

main関数はシンプルに!!

もはや自分が書いたんじゃないか.

```
    return winCount;
}

// (3)バトル開始から終了までの流れ
int doBattle(Party* pParty, Monster* pEnemy)
{
  printMonsterName(pEnemy);
  printf("が現れた！\n");

  // バトルフィールドの宝石スロットの準備と初期化
  BattleField field = {pParty, pEnemy};
  fillGems(field.gems, false);

  // 交互ターン繰り返し
  while(true) {
    onPlayerTurn(&field);
    if(pEnemy->hp <= 0) {          // 撃破判定
      printMonsterName(pEnemy);
      printf("を倒した！\n");
      return 1;
    }
    onEnemyTurn(&field);
    if(pParty->hp <= 0) {          // 敗北判定
      printf("%sは倒れた...\n", pParty->playerName);
      return 0;
    }
  }
}

// (4)パーティ編成処理
Party organizeParty(char* playerName, Monster* monsters, int numMonsters)
{
  int sumHp = 0;
  int sumDefense = 0;
  for(int i = 0; i < numMonsters; i++) {
    sumHp += monsters[i].hp;
    sumDefense += monsters[i].defense;
  }
  int avgDefense = sumDefense / numMonsters;

  Party p = {playerName, monsters, numMonsters, sumHp, sumHp, avgDefense};
  return p;
}

// (5)パーティ情報の表示
```

引数3つだけで2つ。
2つでもOK。

構造体をreturnで返すのは可能。
初期化はOKだから。

```
void showParty(Party* pParty)
{
  printf("＜パーティ編成＞----------\n");
  for(int i = 0; i < pParty->numMonsters; i++) {
    printMonsterName(&(pParty->monsters[i]));
    printf("  HP=%4d 攻撃=%3d 防御=%3d\n",
      pParty->monsters[i].hp,
      pParty->monsters[i].attack,
      pParty->monsters[i].defense
    );
  }
  printf("----------------------\n\n");
}

// (6)プレイヤーターン
void onPlayerTurn(BattleField* pField)
{
  printf("\n【%sのターン】\n", pField->pParty->playerName);

  showBattleField(pField);

  char command[3];
  do{
    printf("コマンド？>");
    scanf("%2s", command);
  } while(checkValidCommand(command) == false);

  // 宝石を移動させた上で、「宝石スロットの評価」を機能させる
  moveGem(pField->gems, command[0] - 'A', command[1] - 'A', true);
  evaluateGems(pField);  //(10)別解の場合、引数は(pField, 1);
}

// (7)敵モンスターターン
void onEnemyTurn(BattleField* pField)
{
  printf("\n【%sのターン】\n", pField->pEnemy->name);
  doEnemyAttack(pField);
}

// (8)バトルフィールド情報の表示
void showBattleField(BattleField *pField)
{
  printf("----------------------------\n\n");
  printf("          ");
  printMonsterName(pField->pEnemy);
```

```
  printf("\n       HP= %4d / %4d\n", pField->pEnemy->hp, pField->pEnemy->maxhp);
  printf("\n\n");
  for(int i = 0; i < pField->pParty->numMonsters; i++) {
    printMonsterName(&(pField->pParty->monsters[i]));
    printf("  ");
  }
  printf("\n");
  printf("       HP= %4d / %4d\n", pField->pParty->hp, pField->pParty->maxHp);
  printf("----------------------------\n");
  printf(" ");
  for(int i = 0; i < MAX_GEMS; i++ ){
    printf("%c ", 'A'+i);
  }
  printf("\n");
  printGems(pField->gems);
  printf("----------------------------\n");
}

// (9)入力コマンドの正当性判定
bool checkValidCommand(char* c)
{
  // コマンドの長さは必ず2であるべき
  if(strlen(c) != 2) return false;
  // 1文字目と2文字目が同じであれば不正
  if(c[0] == c[1]) return false;
  // 1文字目がA-Nの範囲で無ければ不正
  if(c[0] < 'A' || c[0] > 'A' + MAX_GEMS - 1) return false;
  // 2文字目もA-Nの範囲で無ければ不正
  if(c[1] < 'A' || c[1] > 'A' + MAX_GEMS - 1) return false;
  // それ以外は有効
  return true;
}

// (10)宝石スロットを評価解決する
void evaluateGems(BattleField* pField)
{
  int numCombo = 1;
  BanishInfo bi = checkBanishable(pField->gems);
  while(bi.len != 0) {
    // 消滅させて、空きスロットを左詰め
    banishGems(pField, bi, numCombo);
    numCombo++;
    shiftGems(pField->gems);
    // 左詰めの結果、コンボ可能かを判定
    bi = checkBanishable(pField->gems);
```

```
    if(bi.len == 0) {
      // 空きスロットに宝石が沸き、コンボ判定
      spawnGems(pField->gems);
      bi = checkBanishable(pField->gems);
      if(bi.len == 0) {
        // 詰めコンボも沸きコンボも起きなかったので
        // whileループを抜けて終了
        break;
      }
    }
  }
}

/*
// (10')宝石スロットを評価解決する　　（別解）
void evaluateGems(BattleField* pField, int numCombo)
{
  BanishInfo bi = checkBanishable(pField->gems);
  if(bi.len != 0) {     // 宝石の消滅可能性がある場合
    // 消滅させて、空きスロットを左詰め
    banishGems(pField, bi, numCombo);
    numCombo++;
    shiftGems(pField->gems);
  } else {              // 宝石の消滅可能性がない場合
    // 空きスロットもなければ即時終了
    if(countGems(pField->gems, EMPTY) == 0) return;
    // 空きスロットがあれば宝石発生
    spawnGems(pField->gems);
  }
  // 沸きや詰めの結果、さらに評価可能か判定（コンボ）
  evaluateGems(pField, numCombo);
}
*/

// (11)敵モンスターの攻撃
void doEnemyAttack(BattleField* pField)
{
    printMonsterName(pField->pEnemy);
    printf("の攻撃!");

    int damage = calcEnemyAttackDamage(pField->pParty, pField->pEnemy);
    pField->pParty->hp -= damage;
    printf("%dのダメージを受けた\n", damage);
}
```

```
// (12)宝石の消滅可能箇所判定
BanishInfo checkBanishable(Element* gems)
{
  const int BANISH_GEMS = 3;    // 消滅に必要な連続数

  for(int i = 0; i < MAX_GEMS - BANISH_GEMS + 1; i++) {
    Element targetGem = gems[i];
    int len = 1;
    if(targetGem == EMPTY) continue;
    for(int j = i + 1; j < MAX_GEMS; j++) {
      if(gems[i] == gems[j]) {
        len++;
      } else {
        break;
      }
    }
    if(len >= BANISH_GEMS) {
      BanishInfo found = {i, len, targetGem};
      return found;
    }
  }

  // 見付からなかった
  BanishInfo notFound = {0, 0, EMPTY};
  return notFound;
}

// (13)指定箇所の宝石を消滅させ効果発動
void banishGems(BattleField *pField, BanishInfo bi, int numCombo)
{
   // 宝石の消滅
   for(int i = bi.pos; i < bi.pos + bi.len; i++) {
     pField->gems[i] = EMPTY;
   }
   printGems(pField->gems);

   // 効果の発動
   switch(bi.element) {
     case FIRE: case WATER: case WIND: case EARTH:
      doAttack(pField, bi, numCombo);
      break;
     case LIFE:
      doRecover(pField, bi, numCombo);
      break;
     default:
      break;
```

```
  }
}

// (14)空いている部分を左詰めしていく
void shiftGems(Element* gems)
{
  // まずEMPTYの数を数える
  int numEmpty = countGems(gems, EMPTY);

  // 先頭からMAX_GEMS-numEmpty-1番目までのEMPTYを右端へ移動
  for(int i = 0; i < MAX_GEMS - numEmpty; i++) {
    if(gems[i] == EMPTY) {
      moveGem(gems, i, MAX_GEMS - 1, false);
      i--;    // 初回右隣もEMPTYだった場合に備えて再実施
    }
  }

  printGems(gems);
}

// (15)空き領域に宝石が沸く
void spawnGems(Element* gems)
{
  fillGems(gems, true);
  printGems(gems);
}

// (16)味方モンスターの攻撃
void doAttack(BattleField* pField, BanishInfo bi, int numCombo)
{
  for(int i = 0; i < pField->pParty->numMonsters; i++) {
    Monster* attacker = &(pField->pParty->monsters[i]);
    if(attacker->element == bi.element) {
      // 攻撃の実行
      printMonsterName(attacker);
      printf("の攻撃！ ");
      printCombo(numCombo);
      printf("\n");

      int damage = calcAttackDamage(attacker, pField->pEnemy, bi, numCombo);

      pField->pEnemy->hp -= damage;
      printf("%sに%dのダメージ！\n", pField->pEnemy->name, damage);
    }
  }
```

```
}

// (17)宝石の消滅による回復
void doRecover(BattleField* pField, BanishInfo bi, int numCombo)
{
  printf("%sは命の宝石を使った!", pField->pParty->playerName);
  printCombo(numCombo);
  printf("\n");

  int damage = calcRecoverDamage(pField, bi, numCombo);
  pField->pParty->hp += damage;
  printf("HPが%d回復した!\n", damage);
}

/*** ユーティリティ関数宣言 ***/

// (A)モンスター名のカラー表示
void printMonsterName(Monster* pMonster)
{
  char symbol = ELEMENT_SYMBOLS[pMonster->element];

  printf("\x1b[3%dm", ELEMENT_COLORS[pMonster->element]);
  printf("%c%s%c", symbol, pMonster->name, symbol);
  printf("\x1b[0m");
}

// (B)スロットをランダムな宝石で埋める
void fillGems(Element* gems, bool emptyOnly)
{
  for(int i = 0; i < MAX_GEMS; i++) {
    if(!emptyOnly || gems[i] == EMPTY) {
      gems[i] = rand() % EMPTY;
    }
  }
}

// (C)スロットに並ぶ宝石を表示する
void printGems(Element* gems)
{
  for(int i = 0; i < MAX_GEMS; i++) {
    printf(" ");
    printGem(gems[i]);
  }
  printf("\n");
}
```

gem配列のアクセス

```
// (D)1個の宝石の表示
void printGem(Element e)
{
  printf("\x1b[30m");       // 黒文字
  printf("\x1b[4%dm", ELEMENT_COLORS[e]); // 属性色背景
  printf("%c", ELEMENT_SYMBOLS[e]);
  printf("\x1b[0m");        // 色指定解除
}

// (E)指定の宝石を指定の位置まで1つずつ移動させる
void moveGem(Element* gems, int fromPos, int toPos, bool printProcess)
{
  // 移動方向（1=右、-1=左）
  int step = (toPos > fromPos) ? 1 : -1;

  printGems(gems);
  for(int i = fromPos; i != toPos; i += step) {
    swapGem(gems, i, step);
    if(printProcess) printGems(gems);
  }
}

// (F)pos番目の宝石を、step個隣の宝石と交換する
void swapGem(Element* gems, int pos, int step)
{
  Element buf = gems[pos];
  gems[pos] = gems[pos + step];
  gems[pos + step] = buf;
}

// (G)指定種類の宝石のカウント
int countGems(Element* gems, Element target)
{
  int num = 0;
  for(int i = 0; i < MAX_GEMS; i++) {
    if(gems[i] == target) num++;
  }
  return num;
}

// (H)ある値を基準に±percent%の幅でランダムなダメージ値を算出
int blurDamage(int base, int percent)
{
  int r = rand() % (percent * 2 + 1) - percent + 100;
  return base * r / 100;
```

```
}

// (I)敵モンスターによる攻撃ダメージを計算
int calcEnemyAttackDamage(Party* pParty, Monster* pEnemy)
{
  int damage = blurDamage(pEnemy->attack - pParty->defense, 10);
  if(damage <= 0) damage = 1;
  return damage;
}

// (J)味方モンスターによる攻撃ダメージ計算
int calcAttackDamage(Monster* pAttackMonster, Monster* pEnemy, BanishInfo bi, int
numCombo)
{
  int damage = (pAttackMonster->attack - pEnemy->defense) *
    ELEMENT_BOOST[pAttackMonster->element][pEnemy->element];
  for(int j = 0; j < bi.len - 2 + numCombo; j++) {
    damage *= 1.25;
  }
  damage = blurDamage(damage, 10);
  if(damage <= 0) damage = 1;
  return damage;
}

// (K)回復ダメージ計算
int calcRecoverDamage(BattleField *pField, BanishInfo bi, int numCombo)
{
  int damage = 20;
  int boost = bi.len - 2 + numCombo;
  for(int i = 0; i < boost; i++) {
    damage *= 1.25;
  }
  damage = blurDamage(damage, 10);

  // 最大HP以上には回復できない
  int recoverble = pField->pParty->maxHp - pField->pParty->hp;
  if(recoverble < damage) {
    damage = recoverble;
  }
  return damage;
}

// (L)コンボ表示
void printCombo(int numCombo)
{
  if(numCombo > 1) {
```

```
    printf("\x1b[37m\x1b[41m");     // 赤背景白文字
    printf("%d COMBO!", numCombo);
    printf("\x1b[0m");              // 色指定解除
  }
}
```