```
// gg.cpp - GUESSING GAME
//
// MODULE INDEX
// NAME                 CONTENTS
// GetConfig            Get configuration based on ball location
// SetConfig            Set ball location from configuration
// PickEnding           Pick ending configuration
// FindPath                     Find non-winning path
// main                 Main line
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 12-09-17     MPF     Original
// 20-10-17 JAS Changed variable naming convention
//-----------------------------------------------------------------------------

#include <cstring>
#include <iostream>
#include <ctime>
#include <vector>
#include <algorithm>
using namespace std;

//-----------------------------------------------------------------------------

// POSSIBLE LOCATIONS OF A BALL

enum Location_t {
        LOC_LEFT,
        LOC_MIDDLE,
        LOC_RIGHT
};

//-----------------------------------------------------------------------------

// IDENTIFIER FOR BALL BASED ON COLOUR

enum Ball_t {
        BALL_RED,
        BALL_GREEN,
        BALL_BLUE,
        BALL_CNT
};

//-----------------------------------------------------------------------------

// CONSTANT DEFINITIONS

#define NBR_OF_LEVELS   5               // Number of level
#define NBR_OF_PLAYERS  5               // Number of players

//-----------------------------------------------------------------------------

// TYPE DEFINITIONS
```

```
// Path taken by a ball. Example:-
// * Path_t[0] is location of the ball after the first crisscross.
// * Path_t[NBR_OF_LEVELS-1] is location of the ball at the end.

typedef Location_t      Path_t[NBR_OF_LEVELS];  // Path taken by a ball

// Collection of paths taken by all balls. Also the answer. Example:-
// * Answer_t[BALL_RED] is the path taken by red ball.
// * Answer_t[BALL_BLUE] is the path taken by blue ball.

typedef Path_t          Answer_t[BALL_CNT];     // Paths taken by all balls

// Placement of all balls at a particular level or time.
// Contains the location of each ball. Example:
// * Place_t[BALL_RED] is the location of red ball.
// * Place_t[BALL_GREEN] is the location of green ball.

typedef Location_t      Place_t[BALL_CNT];      // Placement of all balls

// Guesses made by all players. Example:-
// * Guess_t[0] is the placement of balls player 1 guessed.
// * Guess_t[NBR_OF_PLAYERS-1] is the placement of balls player N guessed.

typedef Place_t         Guess_t[NBR_OF_PLAYERS];        // Guesses made by all players

// Ball placement at each level. Example:-
// * LevelPlace_t[0] is the placement of balls after the first crisscross.
// * LevelPlace_t[NBR_OF_LEVELS] is the placement of balls at the end.

typedef Place_t         LevelPlace_t[NBR_OF_LEVELS];    // Ball placement at each level

//------------------------------------------------------------------------------

// POSSIBLE BALL CONFIGURATIONS

enum Config_t {
        CONFIG_RGB,
        CONFIG_RBG,
        CONFIG_GRB,
        CONFIG_GBR,
        CONFIG_BRG,
        CONFIG_BGR,
        CONFIG_CNT,
        CONFIG_BAD,
};

//------------------------------------------------------------------------------

// GET CONFIGURATION BASED ON BALL LOCATIONS

Config_t
GetConfig (
        const Place_t           loc)            // Location of balls
```

```
{
        // Check that no two balls share the same location

        if (
                loc[BALL_RED] == loc[BALL_GREEN] ||
                loc[BALL_RED] == loc[BALL_BLUE] ||
                loc[BALL_GREEN] == loc[BALL_BLUE]
        )
                return CONFIG_BAD;

        switch (loc[BALL_RED]) {
        case LOC_LEFT:
                switch (loc[BALL_GREEN]) {
                case LOC_MIDDLE:
                        return CONFIG_RGB;
                case LOC_RIGHT:
                        return CONFIG_RBG;
                }
        case LOC_MIDDLE:
                switch (loc[BALL_GREEN]) {
                case LOC_LEFT:
                        return CONFIG_GRB;
                case LOC_RIGHT:
                        return CONFIG_BRG;
                }
        case LOC_RIGHT:
                switch (loc[BALL_GREEN]) {
                case LOC_LEFT:
                        return CONFIG_GBR;
                case LOC_MIDDLE:
                        return CONFIG_BGR;
                }
        }

        // One of the location must be invalid (not left, middle or right)

        return CONFIG_BAD;
}

//------------------------------------------------------------------------------

// SET BALL LOCATION FROM CONFIGURATION

void
SetConfig (
        const Config_t          cfg,            // Configuration
        Place_t                 loc)            // Locations of balls
{
        const Location_t LOC_LIST[CONFIG_CNT][BALL_CNT] = {
                {LOC_LEFT, LOC_MIDDLE, LOC_RIGHT}, // RGB
                {LOC_LEFT, LOC_RIGHT, LOC_MIDDLE}, // RBG
                {LOC_MIDDLE, LOC_LEFT, LOC_RIGHT}, // GRB
                {LOC_RIGHT, LOC_LEFT, LOC_MIDDLE}, // GBR
                {LOC_MIDDLE, LOC_RIGHT, LOC_LEFT}, // BRG
```

```
                {LOC_RIGHT, LOC_MIDDLE, LOC_LEFT}, // BGR
        };

        loc[BALL_RED] = LOC_LIST[cfg][BALL_RED];
        loc[BALL_GREEN] = LOC_LIST[cfg][BALL_GREEN];
        loc[BALL_BLUE] = LOC_LIST[cfg][BALL_BLUE];
}

//----------------------------------------------------------------------------

// PICK ENDING CONFIGURATION

Config_t
PickEnding (
        const Guess_t           guess)          // Guesses made by friends
{
        vector<Config_t>        possible;       // Possible configurations
        vector<Config_t>::iterator it;          // Iterator
        int                     i;              // General purpose index

        possible.push_back (CONFIG_RGB);
        possible.push_back (CONFIG_RBG);
        possible.push_back (CONFIG_GRB);
        possible.push_back (CONFIG_GBR);
        possible.push_back (CONFIG_BRG);
        possible.push_back (CONFIG_BGR);

        for (i = 0; i < NBR_OF_LEVELS; i++) {
                it = find(possible.begin(), possible.end(),
                        GetConfig(guess[i]));
                if (it != possible.end())
                        possible.erase(it);
        }

        return possible[lrand48() % possible.size()];
}

//----------------------------------------------------------------------------

// FIND NON-WINNING PATH

void
FindPath (
        Answer_t                path,           // Array of paths taken by balls
        const Place_t           start,          // Starting placement of balls
        const Guess_t           guess)          // Guesses made by friends
{
        Config_t                curCfg;         // Current configuration
        Config_t                nxtCfg;         // Next configuration
        Config_t                endCfg;         // Ending configuration
        LevelPlace_t            ball;           // Ball placements at each level
        int                     crossLvl;       // Crisscross level

        // Set the randomizer seed
```

```
        srand48(time(NULL));

        // Set any configuration up till last 2 level

        curCfg = GetConfig(start);
        for (crossLvl = 0; crossLvl < NBR_OF_LEVELS - 2; crossLvl++) {
                nxtCfg = static_cast<Config_t>(lrand48() % CONFIG_CNT);

                // Make sure location swap happens

                if (curCfg == nxtCfg)
                        nxtCfg = static_cast<Config_t>(
                                (nxtCfg + 1 + lrand48() % (CONFIG_CNT - 1)) %
                                CONFIG_CNT);
                SetConfig (nxtCfg, ball[crossLvl]);
                curCfg = nxtCfg;
        }

        // Pick last 2 transitions that lead to the desired ending

        endCfg = PickEnding(guess);
        nxtCfg = static_cast<Config_t>(lrand48() % CONFIG_CNT);
        while (curCfg == nxtCfg || endCfg == nxtCfg) {
                nxtCfg = static_cast<Config_t>((nxtCfg + 1 +
                        lrand48() % (CONFIG_CNT - 1)) % CONFIG_CNT);
        }
        SetConfig (nxtCfg, ball[crossLvl]);
        SetConfig (endCfg, ball[crossLvl+1]);

        // Transpose ball to path

        for (crossLvl = 0; crossLvl < NBR_OF_LEVELS; crossLvl++) {
                path[BALL_RED][crossLvl] = ball[crossLvl][BALL_RED];
                path[BALL_GREEN][crossLvl] = ball[crossLvl][BALL_GREEN];
                path[BALL_BLUE][crossLvl] = ball[crossLvl][BALL_BLUE];
        }
}

//------------------------------------------------------------------------

// MAIN LINE

int
main ()
{
        Answer_t                path;           // Array of path taken by balls
        int                     crossLvl;       // Crisscross level
        int                     playerNbr;      // Player number
        char                    buf[512];       // Formatting buffer
        int                     nbrLocChgs;     // Number of location changes
        Place_t                 prev;           // Previous placement of balls

        const Place_t           start =         // Starting placement of balls
```

```
        {LOC_LEFT, LOC_MIDDLE, LOC_RIGHT};
const Guess_t           guess = {       // Guesses made by friends
        {LOC_LEFT, LOC_MIDDLE, LOC_RIGHT},
        {LOC_LEFT, LOC_RIGHT, LOC_MIDDLE},
        {LOC_MIDDLE, LOC_LEFT, LOC_RIGHT},
        {LOC_MIDDLE, LOC_RIGHT, LOC_LEFT},
        {LOC_RIGHT, LOC_LEFT, LOC_MIDDLE}
};

// Test the function

FindPath (path, start, guess);

// Show the result

strcpy (buf, "R G B");
cout << buf << '\n';
prev[BALL_RED] = start[BALL_RED];
prev[BALL_GREEN] = start[BALL_GREEN];
prev[BALL_BLUE] = start[BALL_BLUE];

for (crossLvl = 0; crossLvl < NBR_OF_LEVELS; crossLvl++) {

        // Set the ball colour in the formatting buffer

        buf[path[BALL_RED][crossLvl] * 2] = 'R';
        buf[path[BALL_GREEN][crossLvl] * 2] = 'G';
        buf[path[BALL_BLUE][crossLvl] * 2] = 'B';
        cout << buf;

        // Verify at least 2 swaps happened

        nbrLocChgs = 0;
        if (prev[BALL_RED] != path[BALL_RED][crossLvl])
                nbrLocChgs++;
        if (prev[BALL_GREEN] != path[BALL_GREEN][crossLvl])
                nbrLocChgs++;
        if (prev[BALL_BLUE] != path[BALL_BLUE][crossLvl])
                nbrLocChgs++;
        if (nbrLocChgs < 2)
                cout << " ** No location swap";
        cout << '\n';

        // Save previous placement

        prev[BALL_RED] = path[BALL_RED][crossLvl];
        prev[BALL_GREEN] = path[BALL_GREEN][crossLvl];
        prev[BALL_BLUE] = path[BALL_BLUE][crossLvl];
}

// Verify that no winner is found

for (playerNbr = 0; playerNbr < NBR_OF_PLAYERS; playerNbr++) {
        if (
```

```
                    guess[playerNbr][BALL_RED] == path[BALL_RED][NBR_OF_LEVELS-1] &&
                    guess[playerNbr][BALL_GREEN] == path[BALL_GREEN][NBR_OF_LEVELS-1] &&
                    guess[playerNbr][BALL_BLUE] == path[BALL_BLUE][NBR_OF_LEVELS-1]
            )
                    cout << "** Player " << playerNbr << " wins!\n";
        }

        return 0;
}
```