```cpp
/*
The SDL2 verison of the collision performance testing code for Games
  Programming

My compiler settings to including SDL2 lib and set optimisation level
g++ -std=c++11 spike26_sdl2.cpp -o "spike26_sdl2" -L/usr/local/lib -lSDL2
  -I/usr/local/include/SDL2 -D_THREAD_SAFE -O3

Try different optimization (letter "O" then number) levels
  -O0 level zero
  -O3 level 3

Look for TODO's and comments for ideas of what things to change.

Note: You REALLY need to
- increase the test time,
- turn the rendering OFF, and
- turn optimisation OFF
to see biggest difference! (Then explain why!)

There are lots of ugly code features.
Enjoy! - Oh i did :)

Clinton Woodward <cwoodward@swin.edu.au>
Updated 2017-10-26  */

#include <iostream>
#include <ctime>
#include "SDL.h"

const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;
const int SCREEN_BPP = 32;

const int BOX_WIDTH = 50;
const int BOX_HEIGHT = 50;
const int BOX_SPEED = 10;
const int BOX_COUNT = 100;

const int TEST_TIME = 15 * 1000; // ie, 3*1000 = 3 seconds

enum BoxState { CONTACT_NO, CONTACT_YES };

struct CrashBox {
    int x, y; // pos
    int dx, dy; // vel
    int w, h; // size
    BoxState state;
};
```

```cpp

// Global variables. (Apparently evil.)
CrashBox boxes[BOX_COUNT];

void (*crash_test_all_ptr)(); // global function pointer!


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

void init_boxes()
{
    // seed value - Set explicitly if you want repeatable results!!
    srand((unsigned)time(0));
    for (int i = 0; i < BOX_COUNT; i++) {
        // position
        boxes[i].x = rand() % SCREEN_WIDTH;
        boxes[i].y = rand() % SCREEN_HEIGHT;
        // size
        boxes[i].w = (rand() % BOX_WIDTH) + 1;
        boxes[i].h = (rand() % BOX_HEIGHT) + 1;
        // velocity (both positive and negative "delta" values)
        boxes[i].dx = (rand() % (BOX_SPEED * 2)) - (BOX_SPEED / 2);
        boxes[i].dy = (rand() % (BOX_SPEED * 2)) - (BOX_SPEED / 2);
    }
}


void render_box(CrashBox& box, SDL_Renderer* renderer, SDL_Color& color)
  {
    SDL_Rect r = { box.x, box.y, box.w, box.h };
    SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, color.a);
    SDL_RenderFillRect(renderer, &r);
}

void render_boxes(SDL_Renderer* renderer) {
    SDL_Color white = { 255,255,255,255 };
    SDL_Color red = { 255,0,0,255 };

    for (int i = 0; i < BOX_COUNT; i++) {
        if (boxes[i].state == CONTACT_NO) {
            render_box(boxes[i], renderer, white);
        }
        else {
            render_box(boxes[i], renderer, red);
        }
    }
}

```

```cpp
//--------------------------------------------------------------------------
-----

bool crash_test_A(int i, int j) // via index
{   //yeap - lazyfoo style!

    //The sides of the rectangles
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    CrashBox A, B;
    A = boxes[i];
    B = boxes[j];

    //Calculate the sides of rect A
    leftA = A.x;
    rightA = A.x + A.w;
    topA = A.y;
    bottomA = A.y + A.h;

    //Calculate the sides of rect B
    leftB = B.x;
    rightB = B.x + B.w;
    topB = B.y;
    bottomB = B.y + B.h;

    //If any of the sides from A are outside of B
    if (bottomA <= topB) return false;
    if (topA >= bottomB) return false;
    if (rightA <= leftB) return false;
    if (leftA >= rightB) return false;

    //If none of the sides from A are outside B
    return true;
}

bool crash_test_B(CrashBox A, CrashBox B) {
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    leftA = A.x;
    rightA = A.x + A.w;
    topA = A.y;
    bottomA = A.y + A.h;

    leftB = B.x;
    rightB = B.x + B.w;
    topB = B.y;
    bottomB = B.y + B.h;

    if (bottomA <= topB) return false;
    if (topA >= bottomB) return false;
    if (rightA <= leftB) return false;
    if (leftA >= rightB) return false;
    return true;
}


bool crash_test_C(CrashBox& A, CrashBox& B) { // via struct (ref!)
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    leftA = A.x;
    rightA = A.x + A.w;
    topA = A.y;
    bottomA = A.y + A.h;

    leftB = B.x;
    rightB = B.x + B.w;
    topB = B.y;
    bottomB = B.y + B.h;

    if (bottomA <= topB) return false;
    if (topA >= bottomB) return false;
    if (rightA <= leftB) return false;
    if (leftA >= rightB) return false;

    return true;
}

bool crash_test_D(CrashBox& A, CrashBox& B) {
    if ((A.y + A.h) <= B.y) return false;
    if (A.y >= (B.y + B.h)) return false;
    if ((A.x + A.w) <= B.x) return false;
    if (A.x >= (B.x + B.w)) return false;
    return true;
}

bool crash_test_E(CrashBox& A, CrashBox& B) {
    return ((A.y + A.h) <= B.y || A.y >= (B.y + B.h)
    || (A.x + A.w) <= B.x || A.x >= (B.x + B.w));
}
```

```

bool crash_test_F(CrashBox& A, CrashBox& B) {
    if ((A.y + A.h) <= B.y) return false;
    else if (A.y >= (B.y + B.h)) return false;
    else if ((A.x + A.w) <= B.x) return false;
    else if (A.x >= (B.x + B.w)) return false;
    else return true;
}

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

//* Extra check for minimal gains == worst performing func thanks to cache
   miss
void crash_test_all_A1() {
    // check i against j
    for (int i = 0; i < BOX_COUNT; i++) {
        for (int j = 0; j < BOX_COUNT; j++) {
            if (crash_test_A(i, j)) {
                if (i != j) {   // <-- difference between A1 and A2.
                    boxes[i].state = CONTACT_YES;
                    boxes[j].state = CONTACT_YES;
                }
            }
        }
    }
}

//* Also garbage
void crash_test_all_A2() {
    for (int i = 0; i < BOX_COUNT; i++) {
        for (int j = i + 1; j < BOX_COUNT; j++) {
            if (crash_test_A(i, j)) {
                boxes[i].state = CONTACT_YES;
                boxes[j].state = CONTACT_YES; }
        }
    }
}

//  Yeh we pass the struct but internal copies are made causing this to be
   hecka slow too
void crash_test_all_B() {
    for (int i = 0; i < BOX_COUNT; i++) {
        for (int j = i + 1; j < BOX_COUNT; j++) {
            if (crash_test_B(boxes[i], boxes[j])) {
                boxes[i].state = CONTACT_YES;
                boxes[j].state = CONTACT_YES;
            }
        }
    }
}

//  Pass via struct reference, very cool and works nice but all the 
  internal
//  copies and assignments cause a performance hit still
void crash_test_all_C() {
    for (int i = 0; i < BOX_COUNT; i++) {
        for (int j = i + 1; j < BOX_COUNT; j++) {
            if (crash_test_C(boxes[i], boxes[j])) {
                boxes[i].state = CONTACT_YES;
                boxes[j].state = CONTACT_YES;
            }
        }
    }
}


//  WOOOOOOO WE DID IT BOIS! this one is pre speed.
void crash_test_all_D() {
    for (int i = 0; i < BOX_COUNT; i++) {
        for (int j = i + 1; j < BOX_COUNT; j++) {
            if (crash_test_D(boxes[i], boxes[j])) {
                boxes[i].state = CONTACT_YES;
                boxes[j].state = CONTACT_YES;
            }
        }
    }
}

void crash_test_all_E() {
    for (int i = 0; i < BOX_COUNT; i++) {
        for (int j = i + 1; j < BOX_COUNT; j++) {
            if (crash_test_E(boxes[i], boxes[j])) {
                boxes[i].state = CONTACT_YES;
                boxes[j].state = CONTACT_YES;
            }
        }
    }
}

void crash_test_all_F() {
    for (int i = 0; i < BOX_COUNT; i++) {
        for (int j = i + 1; j < BOX_COUNT; j++) {
            if (crash_test_F(boxes[i], boxes[j])) {
                boxes[i].state = CONTACT_YES;
                boxes[j].state = CONTACT_YES;
            }
        }
```

```cpp
    }
}

void crash_test_all_G() {
    auto cached_boxes = boxes;
    int count = BOX_COUNT;

    for (int i = 0; i < count; i++) {
        for (int j = i + 1; j < count; j++) {
            if (crash_test_F(cached_boxes[i], cached_boxes[j])) {
                cached_boxes[i].state = CONTACT_YES;
                cached_boxes[j].state = CONTACT_YES;
            }
        }
    }
}

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

void update_boxes()
{
    // First move all boxes
    for (int i = 0; i < BOX_COUNT; i++) {
        // update position using current velocity
        boxes[i].x = boxes[i].x + boxes[i].dx;
        boxes[i].y = boxes[i].y + boxes[i].dy;
        // check for wrap-around condition
        if (boxes[i].x >= SCREEN_WIDTH) boxes[i].x -= SCREEN_WIDTH;
        if (boxes[i].x < 0) boxes[i].x += SCREEN_WIDTH;
        if (boxes[i].y >= SCREEN_HEIGHT) boxes[i].y -= SCREEN_HEIGHT;
        if (boxes[i].y < 0) boxes[i].y += SCREEN_HEIGHT;
    }


    // 1. mark all boxes as not collided
        for (int i = 0; i < BOX_COUNT; i++)
        boxes[i].state = CONTACT_NO;
    // 2. call whatever function has been set to test all i against j boxes
    crash_test_all_ptr();
}


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

int run_test(const char* title, void (*function_ptr)()) {
    // get SDL to setup all subsystems
    if (SDL_Init(SDL_INIT_VIDEO) == -1) {
        std::cout << " Failed to initialize SDL : " << SDL_GetError() << 
          std::endl;
        return -1;
    }

    // window with title and size we want
    SDL_Window* window = SDL_CreateWindow(
        title,
        SDL_WINDOWPOS_UNDEFINED,
        SDL_WINDOWPOS_UNDEFINED,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        SDL_WINDOW_HIDDEN
    );

    if (window == nullptr) {
        std::cout << "Failed to create window : " << SDL_GetError();
        return -1;
    }

    // renderer for the window ...
    SDL_Renderer* renderer = SDL_CreateRenderer(
        window,
        -1,
        0//SDL_RENDERER_ACCELERATED
    );

    if (renderer == nullptr) {
        std::cout << "Failed to create renderer : " << SDL_GetError();
        return -1;
    }

    // some pretty test output to the console
    printf("-- New Test: %s\n", title);
    crash_test_all_ptr = function_ptr;

    // sanity check that the crash test function pointer has been set
    if (crash_test_all_ptr == nullptr) {
        printf("EH? Set the crash_test_all_ptr first!\n");
        return 1;
    }

    // initialise each crashbox
    init_boxes();

    // initialise test count/time values
    Uint32 loop_count = 0;
    Uint32 tick_start = SDL_GetTicks(); // start time == now!
    Uint32 tick_target = tick_start + TEST_TIME; // when to stop

    // CLASSIC GAME LOOP
    bool running = true;
    SDL_Event event;
    while (running) {
        // UPDATE
        // 1. check for quit event
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) running = false;
        }
        // 2. count...
        loop_count++;
        // 3. check for test time finished
        if (SDL_GetTicks() >= tick_target) running = false;
        // 4. move all the crash boxes and check for collisions
        update_boxes();

        // RENDER
        if (false) {
            // 1. clear the background
            SDL_SetRenderDrawColor(renderer, 100, 100, 100, 255);
            SDL_RenderClear(renderer);
            // 2. render all boxes
            render_boxes(renderer);
            // 3. show it
            SDL_RenderPresent(renderer);
        }
        //SDL_Delay(50); // uncomment to see at slow speed.
    }
    // note the end time
    Uint32 tick_end = SDL_GetTicks();

    // CLEAN UP TIME (and close the SDL window)
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();

    // SHOW STATS
    printf("Loops: %d\n", loop_count);
    printf("Time: %d (ms)\n", (tick_end - tick_start));
    printf("Loops/Second: %f\n", (float(loop_count) / (tick_end - 
      tick_start) * 1000.0));

    return 0;
}

int main(int argc, char* args[])
{
    //run_test("Test A1", crash_test_all_A1);
    //run_test("Test A2", crash_test_all_A2);
    //run_test("Test B", crash_test_all_B);
    //run_test("Test C", crash_test_all_C);
    run_test("Test D", crash_test_all_D);
    run_test("Test E", crash_test_all_E);
    run_test("Test F", crash_test_all_F);
    run_test("Test G", crash_test_all_G);

    return 0;
}


```