Linköping University
Department of Computer and Information Science (IDA)
The UPP group 2015-09-08

Functions and composed types

Aim

This laboration concerns first and foremost abstraction. Abstraction is about taking something that may be complicated in its detail, and make it easy to use by only revealing *what* it achieve, hiding the complexity of *how*.

You will create a program module, and the user of this module is another programmer that we name Sam[1]. Your goal is to make an abstraction for Sam to use. In this case Sam already have an idea of what she[2] needs, and he[3] has already written a program to test the intended abstraction. You'll have to make it work.

The most important tools a programmer (you) have to create abstractions are *functions* and *composed types*.

With a function you put a nice clear name (right?) on the solution to one single small problem. If the function abstraction is successful you can use your function in several places without bothering about how it works exactly, you just need to know what is does (from the name) and trust that it works. A bonus is that a good name will make your code easy to read and understand by others[4].

Composed data types achieve a similar goal, but for data instead of algorithms. You collect all data needed to describe some object or concept, put a clear name on it, and then you can use it as a single entity. `std::string`, composed of a sequence of characters, is an example.

In C++ you would use a `class` with member functions to solve this laboration. You can however also solve it with the more basic C `struct` and free functions. Using a `struct` is much more explicit about every detail, and converting that solution to a class reveal much about how a C++ class automate and hide things. You may however not have time to do both, so pick one and stick to it. We will cover the C++ class in a later laboration (which may tip the scale either way depending on how you argue). We will provide the test program for both versions.

[1] To avoid any further confusion selecting between *she* or *he*.

[2] I mean it. Sam might be short for Samantha.

[3] It happens. Sam can be a guy.

[4] Actually a bad thing if your plan is to keep your job by being the only one to understand your mission critical code. Just to mention: it's a poor plan. It will, given some time, end with mission critical code not even understood by you. But all blame will go on you.

Reading instructions

- Functions
- Overloading functions
- Return values
- Parameter passing
  - Input parameters for basic types (pass by value)
  - Input parameters for composed types (pass by **const** reference)
  - Output parameters (pass by reference)
- Composed data types (struct, class)
  - Member variables
  - Encapsulation (public, private)
  - Member functions (methods)
  - Constructors (constructor)
- Renaming existing types (using or typedef)
- Header file (declarations to be included)
- Implementation file (definitions to be compiled)

Assignment

Create a program library to handle collision detection. The objects that may collide are encapsulated in a bounding box (rectangle) with sides parallel to x- and y-axis. This kind of rectangle is commonly referred to as an axis aligned bounding box (AABB).

Our coordinate system have its origin at the top left corner of the screen, with x increasing to right (as usual in math) and y increasing down (as usual on computer screens).

Your program should consist of a data type to represent one AABB, coupled with functions to perform the operations listed below.

Although the main focus is to implement the more or less given abstraction, another problem is to arrive at code which actually calculate correct answers (problem solving). Your solution does not have to be efficient at all. If you get stuck, draw a figure to better illustrate your problem, and explain it to your assistant.

(construct) Should an AABB by requiring Sam to specify the coordinates for the top side, left side, bottom side and right side. The left side in the created AABB must be to the left of the right side, and the bottom side must be below the top side. You must however expect Sam to get this wrong, and if so, fix it (without generating an error). For simplicity we will allow zero size boxes.

Some examples of code Sam should be able to write:

```
int top{0}, left{0}, bottom{10}, right{10};
AABB my_aabb(top, left, bottom, right);
AABB my_aabb(5, 10, 43, 12); // Box (x=10,y=5) ==> (x=12,y=43)
```

You may also want a way to create an AABB by specifying two `points` (see **contain** below).

contain A function that operate on an AABB. Should return `true` if input parameter `x` and `y` is a dot inside the AABB.

contain A function that operate on an AABB. Should return `true` if input parameter `pt` is a dot inside the AABB. You need to create a composed type to represent the point `pt`. As a forced restriction you are not allowed any comparison or calculation inside this function.

intersect A function that operates on two AABB's. Return `true` if any part of one AABB is inside the other.

Hint: Start by drawing all cases of how two rectangles may overlap on a piece of paper (at least 10 cases if we count all mirrored and rotated cases). You can solve the problem with no more than 4 comparisons and zero if-statements (it's okay to use more comparisons and if's as long as it's clear what the code do.)

Spoiler: Assuming rectangle A's left side is to the left of rectangle B's right side, you'll (looking at a drawing) see overlap if rectangles A's right side is to right of rectangle B's left side.

min_bounding_box A function that operate on two AABB's. Should return the smallest new AABB surrounding both input boxes.

may_collide A function that determine if moving one AABB's from one position to another may possibly yield a collision with another AABB. Input will be the moving AABB in it's starting position, in it's final position, and the AABB it may possibly collide with. Return `false` if a collision is impossible. Think of this as a fast way of checking when a collision won't happen.

Hint: To check this in a fast and simple way, we determine if the AABB surrounding the entire movement overlap the AABB we think may be colliding. If an overlap exist a collision may exist, and a more fine tuned collision detection must be performed to determine if, and where, the collision occur.

collision_point A function that determine if, and where, an AABB's moving from one position to another will collide another AABB. If a collision occur, the function should return `true` and update an output parameter with the coordinates of the top left corner at the point just before

the collision. If a collision does not occur, the function should return `false` and leave the output parameter untouched.

Make sure to use any abstraction you already have available!

Hint: A simple (but slow) method to arrive at an answer is to move the `AABB` from it's starting point to it's destination in such small steps that the largest movement in x or y direction is 1 pixel, and check for overlap in each step.

Spoiler: To find the number of movement steps you calculate the movement in each direction `Dx` and `Dy`. Then divide each by the larger of the two to get the movement necessary in each step; `dx` and `dy` (the larger will thus be 1.0). If you are clever the signs of the result will be correct without extra work.

(getters) It's often good to have small functions just to retrieve (read only) a certain (member) value from a composed type. In this case it would for example be convenient to retrieve the width and height. Think of what Sam may need and add what you feel suitable.

The final part of this assignment is to verify that your code actually works. You can do this by creating test cases (use your drawing of all cases from before), feed those to your program, and verify you get the result you expect.

To help you and save some time we have prepared a rudimentary test program for you with a few test cases. You can easily add your own test cases to the provided data file.