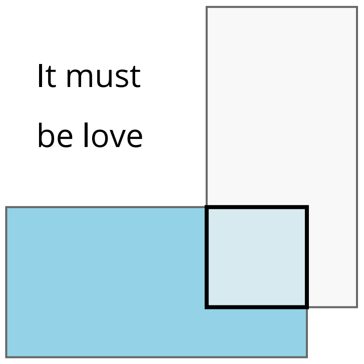
A crack team of love scientists from OkEros (a hot new dating site) have devised a way to represent dating profiles as rectangles on a two-dimensional plane.

They need help writing an algorithm to find the intersection of two users' love rectangles. They suspect finding that intersection is the key to a matching algorithm *so powerful* it will cause an immediate acquisition by Google or Facebook or Obama or something.



Write a function to find the rectangular intersection of two given love rectangles.

As with the example above, love rectangles are always "straight" and never "diagonal." More rigorously: each side is parallel with either the x-axis or the y-axis.

They are defined as instances of the `Rectangle` class:

```cpp
class Rectangle
{
private:
    // coordinates of bottom left corner
    int leftX_;
    int bottomY_;

    // dimensions
    int width_;
    int height_;

public:
    Rectangle() :
        leftX_(0),
        bottomY_(0),
        width_(0),
        height_(0)
    {
    }

    Rectangle(int leftX, int bottomY, int width, int height) :
        leftX_(leftX),
        bottomY_(bottomY),
        width_(width),
        height_(height)
    {
    }

    int getLeftX() const
    {
        return  leftX_;
    }

    int getBottomY() const
    {
        return  bottomY_;
    }

    int getWidth() const
    {
        return  width_;
    }

    int getHeight() const
    {
        return  height_;
    }

    bool operator==(const Rectangle& other) const
    {
        return
            leftX_ == other.leftX_
            && bottomY_ == other.bottomY_
            && width_ == other.width_
            && height_ == other.height_;
    }

    bool operator!=(const Rectangle& other) const
    {
        return !(*this == other);
    }
};
```

Your output rectangle should be a `Rectangle` object as well.

Gotchas

What if there is *no* intersection? Does your function do something reasonable in that case?

What if one rectangle is entirely contained in the other? Does your function do something reasonable in that case?

What if the rectangles don't really intersect but share an edge? Does your function do something reasonable in that case?

Do some parts of your function seem very similar? Can they be refactored so you repeat yourself less?

Breakdown

Let's break this problem into subproblems. How can we divide this problem into smaller parts?

We could look at the two rectangles' "horizontal overlap" or "x overlap" separately from their "vertical overlap" or "y overlap."

Lets start with a helper function `findXOverlap()`.

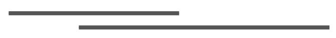
Need help finding the x overlap?

Since we're only working with the x dimension, we can treat the two rectangles' widths as ranges on a 1-dimensional number line.

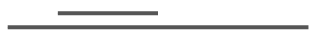
What are the possible cases for how these ranges might overlap or not overlap? Draw out some examples!

There are four relevant cases:

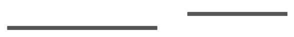
1) The ranges partially overlap:



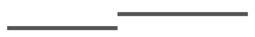
2) One range is completely contained in the other:



3) The ranges don't overlap:



4) The ranges "touch" at a single point:



Let's start with the first 2 cases. **How do we compute the overlapping range?**

One of our ranges starts "further to the right" than the other. We don't know ahead of time which one it is, but we can check the starting points of each range to see which one has the `highestStartPoint`. **That `highestStartPoint` is always the left-hand side of the overlap**, if there is one.

Not convinced? Draw some examples!

Similarly, **the right-hand side of our overlap is always the `lowestEndPoint`**. That *may or may not* be the end point of the same input range that had the `highestStartPoint`—compare cases (1) and (2).

This gives us our x overlap! So we can handle cases (1) and (2). **How do we know when there is *no* overlap?**

If `highestStartPoint > lowestEndPoint`, the two rectangles do not overlap.

But be careful—is it just *greater than* or is it *greater than or equal to*?

It depends how we want to handle case (4) above.

If we use *greater than*, we treat case (4) as an overlap. This means we could end up returning a rectangle with *zero width*, which ... may or may not be what we're looking for. You could make an argument either way.

Let's say a rectangle with zero width (or zero height) isn't a rectangle at all, so we should treat that case as "no intersection."

Can you finish `findXOverlap()`?

Here's one way to do it:

C++

```cpp
class XOverlap
{
private:
    int startPoint_;
    int width_;

public:
    XOverlap() :
        startPoint_(0),
        width_(0)
    {
    }

    XOverlap(int startPoint, int width) :
        startPoint_(startPoint),
        width_(width)
    {
    }

    int getStartPoint() const
    {
        return startPoint_;
    }

    int getWidth() const
    {
        return width_;
    }

    bool overlapExists() const
    {
        return width_ > 0;
    }

    bool operator==(const XOverlap& other) const
    {
        return
            startPoint_ == other.startPoint_
            && width_ == other.width_;
    }
    bool operator!=(const XOverlap& other) const
    {
        return !(*this == other);
    }
};

XOverlap findXOverlap(int x1, int width1, int x2, int width2)
{
    // find the highest ("rightmost") start point and lowest ("leftmost") end point
    int highestStartPoint = max(x1, x2);
    int lowestEndPoint = min(x1 + width1, x2 + width2);

    // return null overlap if there is no overlap
    if (highestStartPoint >= lowestEndPoint) {
        return XOverlap();
    }

    // compute the overlap width
    int overlapWidth = lowestEndPoint - highestStartPoint;

    return XOverlap(highestStartPoint, overlapWidth);
}
```

How can we adapt this for the rectangles' `y`s and `height`s?

Can we just make one `findRangeOverlap()` function that can handle x overlap and y overlap?

Yes! We simply use more general parameter names:

C++ ▼

```
class RangeOverlap
{
private:
    int startPoint_;
    int length_;

public:
    RangeOverlap() :
        startPoint_(0),
        length_(0)
    {
    }

    RangeOverlap(int startPoint, int width) :
        startPoint_(startPoint),
        length_(width)
    {
    }

    int getStartPoint() const
    {
        return startPoint_;
    }

    int getLength() const
    {
        return length_;
    }

    bool overlapExists() const
    {
        return length_ > 0;
    }

    bool operator==(const RangeOverlap& other) const
    {
        return
            startPoint_ == other.startPoint_
            && length_ == other.length_;
    }
    bool operator!=(const RangeOverlap& other) const
    {
        return !(*this == other);
    }
};


RangeOverlap findRangeOverlap(int point1, int length1, int point2, int length2)
{

    // find the highest start point and lowest end point.
    // the highest ("rightmost" or "upmost") start point is
    // the start point of the overlap.
    // the lowest end point is the end point of the overlap.
    int highestStartPoint = max(point1, point2);
    int lowestEndPoint = min(point1 + length1, point2 + length2);

    // return null overlap if there is no overlap
    if (highestStartPoint >= lowestEndPoint) {
        return RangeOverlap();
    }

    // compute the overlap length
    int overlapLength = lowestEndPoint - highestStartPoint;

    return RangeOverlap(highestStartPoint, overlapLength);
}
```

We've solved our subproblem of finding the x and y overlaps! **Now we just need to put the results together.**

Solution

We divide the problem into two halves:

1. The intersection along the x-axis
2. The intersection along the y-axis

Both problems are basically the same as finding the intersection of two "ranges" on a 1-dimensional number line.

So we write a helper function `findRangeOverlap()` that can be used to find both the `x` overlap and the `y` overlap, and we use it to build the rectangular overlap:

C++

```
class RangeOverlap
{
private:
    int startPoint_;
    int length_;

public:
    RangeOverlap() :
        startPoint_(0),
        length_(0)
    {
    }

    RangeOverlap(int startPoint, int width) :
        startPoint_(startPoint),
        length_(width)
    {
    }

    int getStartPoint() const
    {
        return startPoint_;
    }

    int getLength() const
    {
        return length_;
    }

    bool overlapExists() const
    {
        return length_ > 0;
    }

    bool operator==(const RangeOverlap& other) const
    {
        return
            startPoint_ == other.startPoint_
            && length_ == other.length_;
    }
    bool operator!=(const RangeOverlap& other) const
    {
        return !(*this == other);
    }
};

RangeOverlap findRangeOverlap(int point1, int length1, int point2, int length2)
{
    // find the highest start point and lowest end point.
    // the highest ("rightmost" or "upmost") start point is
    // the start point of the overlap.
    // the lowest end point is the end point of the overlap.
    int highestStartPoint = max(point1, point2);
    int lowestEndPoint = min(point1 + length1, point2 + length2);

    // return null overlap if there is no overlap
    if (highestStartPoint >= lowestEndPoint) {
        return RangeOverlap();
    }

    // compute the overlap length
    int overlapLength = lowestEndPoint - highestStartPoint;

    return RangeOverlap(highestStartPoint, overlapLength);
}

Rectangle findRectangularOverlap(
        const Rectangle& rect1,
        const Rectangle& rect2)
{

    // get the x overlap points and lengths
    RangeOverlap xOverlap = findRangeOverlap(
            rect1.getLeftX(), rect1.getWidth(),
            rect2.getLeftX(), rect2.getWidth());

    // get the y overlap points and lengths
    RangeOverlap yOverlap = findRangeOverlap(
            rect1.getBottomY(), rect1.getHeight(),
            rect2.getBottomY(), rect2.getHeight());
    // return "default" rectangle if there is no overlap
    if (!xOverlap.overlapExists() || !yOverlap.overlapExists()) {
        return Rectangle();
    }

    return Rectangle(
        xOverlap.getStartPoint(),
        yOverlap.getStartPoint(),
        xOverlap.getLength(),
        yOverlap.getLength()
    );
}
```

Complexity

$O(1)$ time and $O(1)$ space.

Bonus

What if we had a vector of rectangles and wanted to find *all* the rectangular overlaps between all possible pairs of two rectangles within the vector? Note that we'd be returning *a vector of rectangles*.

What if we had a vector of rectangles and wanted to find the overlap between *all* of them, if there was one? Note that we'd be returning *a single rectangle*.

What We Learned

This is an interesting one because the hard part isn't the time or space optimization—it's getting something that *works* and is *readable*.

For problems like this, I often see candidates who can describe the strategy at a high level but trip over themselves when they get into the details.

Don't let it happen to you. To keep your thoughts clear and avoid bugs, take time to:

1. Think up and draw out all the possible cases. Like we did with the ways ranges can overlap.
2. Use very specific and descriptive variable names.