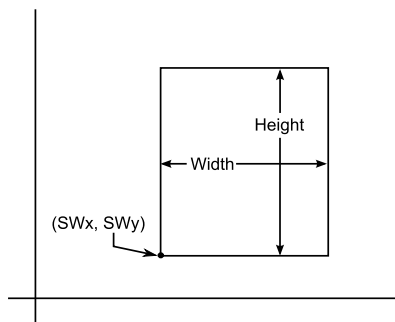


## Decision-making in C

## (Possibly) Intersecting Rectangles

A rectangle in the xy-plane, whose sides are parallel to the coordinate axes can be fully specified by giving the coordinates of one corner and the height and width; for example:



So, we may consider the SW corner, height and width to form a set of defining attributes of such a rectangle. Given that much information about two such rectangles, it is possible to compute the attributes of their intersection, if any.

For this assignment, you will use very basic C techniques to implement a C function to determine whether two rectangles (as described above) do intersect. For our purposes, consider each rectangle to be closed; i.e., the points on the border of the rectangle are considered to be part of the rectangle. So, it's possible for two rectangles to not intersect, or to intersect in many ways, from a single point up to being entirely identical.

You will provide an implementation for a C function that satisfies the conditions stated in the header comment:

```
/** Determines whether two rectangles, A and B, intersect, computes the attributes
 * of the intersection (if any), and returns true or false accordingly.
 *
 * Pre:
 *     aSWx and aSWy specify the SW (lower, left) corner of A
 *     aHeight specifies the vertical dimension of A
 *     aWidth specifies the horizontal dimension of A
 *     bSWx and bSWy specify the SW (lower, left) corner of B
 *     bHeight specifies the vertical dimension of B
 *     bWidth specifies the horizontal dimension of B
 *     iSWx and iSWy point to variables the client will use to store the
 *         SW corner of the intersection, if it exists
 *     iHeight and iWidth point to variables the client will use to store
 *         the height and width of the intersection, if it exists
 * Returns:
 *     true iff the given rectangles share at least one point
 *
 * Restrictions:
 *     - uses only its parameters and local automatic variables
 *     - does not read input or write output
 */
bool Intersection(int32_t aSWx, int32_t aSWy, uint32_t aHeight, uint32_t aWidth,
                 int32_t bSWx, int32_t bSWy, uint32_t bHeight, uint32_t bWidth,
                 int32_t* const iSWx, int32_t* const iSWy,
                 uint32_t* const iHeight, uint32_t* const iWidth);
```

Note that it's perfectly valid for some, or all, of the coordinates to be negative. To a mathematician, it's also perfectly valid for the height and/or width of the intersection to be zero (but not negative).

Be warned: this problem may be harder than it appears. You should think carefully about your analysis of the possible relationships the two rectangles could have in the plane. That does not mean that a solution needs to be complicated. Mine involves a total of four C functions and about 40 lines of code.

We are taking this assignment as an opportunity to introduce a new C-language feature, pointers. A pointer is simply a variable whose value is the address of another variable. There's a very brief appendix on pointers at the end of this specification, which will give you all the information you need to deal with the pointers used in this function.

In this case, we are using pointers as parameters passed into the function `Intersection()`. This allows `Intersection()` to modify variables that are declared in the calling code. Otherwise, we would have some real difficulties, since a function can only return a single variable. Later, we'll see another way to deal with such a situation.

You should consider writing helper functions, even for a short program like this; they allow you to think about the solution from a higher-level perspective, and they clarify your logic.

Make useful comments in your implementation of `Intersection()` and any helper functions that you write. The same general guidelines for commenting that you have been taught in your Java courses should provide sufficient guidance.

Download the tar file posted with this assignment and unpack it on your CentOS VM or on rlogin. The tar contains several files you should use to implement and test your solution:

<code>driver.c</code>	C source code that manages the testing
<code>Intersection.h</code>	declaration for intersection detector (do not modify)
<code>Intersection.c</code>	shell for implementing intersection detector (modify!)
<code>checkAnswer.h</code>	declaration for answer-checking utility (do not modify)
<code>checkAnswer.o</code>	64-bit object file for answer-checking utility
<code>Generator.h</code>	declaration for test-case generator (do not modify)
<code>Generator.o</code>	64-bit object file for test-case generator

You can compile your solution with the given code by using the command:

```
gcc -o driver -std=c99 -Wall -ggdb3 driver.c Intersection.c Generator.o checkAnswer.o
```

Run the test/grading code by using the command:

```
./driver <test data file> <results file> [-repeat]
```

For example, you might use the command: `./driver Data01.txt Results01.txt`

The driver will generate test cases, written to the file you specify for the test data, and write results to the file you specify for results, showing how your implementation of `Intersection()` fared on the test cases that were generated. The results will indicate your answer for each test case, whether your answer was correct, and a total score for the test run.

Execute the driver with the optional `-repeat` switch if you want to reuse the test data file created by a previous run. That allows you to focus on a fixed set of test cases while you are debugging. In fact, you could even edit an existing test case file and create test cases as you like, and then use the `-repeat` switch to use those test cases.

## What to Submit

For this part, you will submit your completed `Intersection.c`, containing your solution. That file must include any helper functions you have written and called from your version of `Intersection()`; any such functions must be declared (as `static`) in the file you submit. You must not include any extraneous code (such as an implementation of `main()` in that file).

Your submission will be graded by running the supplied test/grading code on it. A TA will also check to see if your solution violates any of the restrictions given in the header comment for the function; if so, your submission will be assigned a score of zero (0), regardless of how many tests it passes.

## Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
// On my honor:
//
// - I have not discussed the C language code in my program with
//   anyone other than my instructor or the teaching assistants
//   assigned to this course.
//
// - I have not used C language code obtained from another student,
//   the Internet, or any other unauthorized source, either modified
//   or unmodified.
//
// - If any C language code or documentation used in my program
//   was obtained from an authorized source, such as a text book or
//   course notes, that has been clearly noted with a proper citation
//   in the comments of my program.
//
// - I have not designed this program in such a way as to defeat or
//   interfere with the normal operation of the Curator System.
//
//   <Student Name>
//   <Student's VT email PID>
```

**We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.**

## Change Log

Version	Posted	Pg	Change
3.00	Feb 8		Base document.

**Appendix: Quick Introduction to Pointers**

Suppose you needed to write a function to swap the values of two integer variables. Here's an approach that will not work:

```
// function
void Swap(int32_t X, int32_t Y) {
    int32_t temp = X;
    X = Y;
    Y = temp;
}

// caller
int32_t A = 10;
int32_t B = 20;

Swap(A, B);

// Alas, A is still 10, B is still 20
```

This fails because parameters are passed by value in C (as in other languages). So, `Swap()` gets copies of the caller's variables, and the assignments in `Swap()` have absolutely no effect on the caller's variables. That won't do...

We need to give the function direct access to the caller's variables, and we can do that by using pointer variables:

```
// function
void Swap(int32_t* pX, int32_t* pY) {
    int32_t temp = *pX;
    *pX = *pY;
    *pY = temp;
}

// caller
int32_t A = 10;
int32_t B = 20;

Swap(&A, &B);

// Now, A == 20 and B == 10
```

In the function `Swap()`, we now have two pointer variables (`pX` and `pY`). You declare a pointer variable by adding an asterisk between the type name and the name of the variable:

```
int32_t* pX;    // Creates a pointer variable to point to an int32_t variable;
               // however, pX does not actually point to anything yet.
```

When we call this version of `Swap()`, we do not pass the values of `A` and `B` to `Swap()`. Instead, we use the *address-of operator*, `&`, to get the addresses of `A` and `B`, and we pass those addresses to `Swap()`.

When `Swap()` is called, the variable `pX` in `Swap()` receives the address of `A` (back in the caller), and the variable `pY` in `Swap()` receives the address of `B`.

We say that `pX` *points to* `A`, or that `A` *is the target of* `pX`.

The beauty of this is that if you know the address of a variable, then you can access that variable, and even modify that variable. (In fact, this is closely related to the way Java references are used to access objects in Java code.)

To access the target of a pointer, you use the *dereference operator*, `*`. The statement below dereferences `pX`, gets the value of its target (which is `A`), and assigns that value to the local variable `temp`:

```
int32_t temp = *pX;
```

So, the three assignment statements in the second version of `Swap()` actually swap the values of the variables `A` and `B` that were declared in the caller.

Now, this is all you need to understand about pointer variables to correctly implement the function `Intersection()`, and the code that calls it. But do understand that there's much more to be said about pointers...