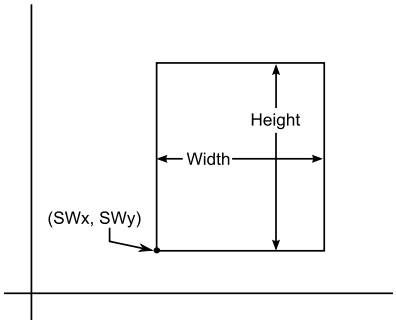
Decision-making in C (Possibly) Intersecting Rectangles

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 determine if the rectangles intersect, and, if they do intersect, to compute the attributes of their intersection (which will be a rectangle).

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
 *
 *   Restriction[1]:
 *     - uses only its parameters and local automatic variables
 */
bool Intersection(int32_t aSWx, int32_t aSWy, int32_t aHeight, int32_t aWidth,
                  int32_t bSWx, int32_t bSWy, int32_t bHeight, int32_t bWidth,
                  int32_t* const iSWx,  int32_t* const iSWy,
                  int32_t* const iHeight, int32_t* const iWidth);
```

Note that it's perfectly valid for some, or all, of the coordinates to be negative, since they just specify points in the xy-plane. To a mathematician, it's also perfectly valid for the height and/or width of the intersection to be zero (but not negative)[2].

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.

Getting Started

A tar file, `c02Files.tar`, is available on the Assignments page, containing the testing/grading code that will be used to grade your solution:

`c02driver.c`	test driver[3]... read the comments!
`Intersection.h`	header file declaring the specified function... do not modify!
`Intersection.c`	C source file for implementing the specified function
`Generator.h`	header file declaring the test case generator... do not modify!
`Generator.o`	64-bit object file containing the test case generator
`checkAnswer.h`	header file declaring the result checking code... do not modify!
`checkAnswer.o`	64-bit object file containing the result checker

Download that tar file and save it, in a suitable directory, on your CentOS 8 Stream installation (or rlogin). Unpack the tar file there. You can do this by executing the command:

```
CentOS> tar xf c02Files.tar
```

The file `Intersection.c` contains a trivial, nonworking implementation of the specified function[4]. You will edit this file to complete the function. As noted above, do not modify any of the given `.h` files; doing so may prevent your solution from compiling when it is graded.

Implementation and Testing

You can compile the given code by executing the command:

```
CentOS> gcc -o c02 -std=c11 -Wall c01driver.c Intersection.c Generator.o checkAnswer.o
```

You can run the test/grading code by executing the command:

```
CentOS> ./c02 <name for test case file> <name for results file> [-repeat]
```

For example, you might use the command: `./c02 TestData.txt Results.txt`

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. Read the comments in `c02driver.c` for more information.

Executing `c02` will create a file holding test data, using the file name you specified on the command line, which might look like this:

```
              A                                  B
  SWx     SWy  Height    Width      SWx     SWy  Height    Width
  -85      24       4        4      -87      22       8        8
  -64       0       4        4      -58       1       5        5
   29      20       5        5       34      20       5        5
   13      20       3       10       14      18       4        6
   . . .
```

Each row specifies two rectangles, as described on the previous page.

The driver will then read each test case from the data file, and call your implementation of the `Intersection()` function. The driver will compare the results computed by your function to the results computed by a reference solution for the function, and write an output file showing how your solution fared on the test data, and generating a score for each test case and an overall score.

For the sample test case file shown above, the driver will produce a results file, using the name you specified on the command line, that might look like this (after you've completed your solution):

```
      Rectangle A                          Rectangle B
      SW corner       Height    Width      SW corner       Height    Width
      ------------------------------------------------------------------------
      (  -85,    24)       4        4     (  -87,    22)       8        8
      (  -85,    24)       4        4
Score:  10 / 10

      (  -64,     0)       4        4     (  -58,     1)       5        5
no intersection
Score:  10 / 10

      (   29,    20)       5        5     (   34,    20)       5        5
      (   34,    20)       5        0
Score:  10 / 10

      (   13,    20)       3       10     (   14,    18)       4        6
      (   14,    20)       2        6
Score:  10 / 10
. . .
```

For each test case, the file shows the attributes of the two given rectangles, and the attributes computed for their intersection, if any. The scoring for each test case is all or nothing, and your total score is shown at the end of the results file.

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.

We are taking this assignment as an opportunity to introduce you to 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.

Submission and Grading

You should not submit your solution to the Curator until you can correctly pass tests with the given testing/grading code.

Submit your completed version of `Intersection.c`, after making changes and testing. Your submission will be compiled, tested and graded by using the supplied code, but that will be done manually after the due date. 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.

If you make multiple submissions of your solution to the Curator, we will grade your last submission.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

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 supplied grading code.
//
//    <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
4.0	Feb 16		Base document.

Notes

[1] The restriction bans the use of non-local variables. In this assignment, there is absolutely no need for doing that, and the use of file-scoped variables is usually a sign of a bad design.

[2] Since height and width cannot, logically, be negative, why not use `uint32_t` for those? Mainly because I do not want to do arithmetic that mixes signed and unsigned integer types, since that can lead to unexpected results.

[3] Study the driver code carefully. It is heavily commented, and shows some C coding patterns that you may find useful in future projects. In particular, pay attention to the loop design, and to the input/output code.

[X] The supplied function simply returns the value `false`, which is incorrect in all cases.

Appendix: Brief and Shallow 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...