Creating a Data Type in C

For this assignment, you will use the \texttt{struct} mechanism in C to implement a data type that represents sets of integers. A set can be modeled using the C \texttt{struct}:

\begin{verbatim}
struct _CSet {
    uint32_t Capacity; // dimension of the set's array
    uint32_t Usage; // number of elements in the set
    int32_t* Data; // pointer to the set's array
};

typedef struct _CSet CSet;
\end{verbatim}

So, a CSet object \texttt{A} that represents the set \{17,9,23,44\} could be initialized like this:

\begin{center}
\begin{tabular}{l|c|c|c|c|c|c|c}
\hline
A & Capacity: 10 & Usage: 4 & Data: & 17 & 9 & 23 & 44 \\
\hline
\end{tabular}
\end{center}

We will say that a CSet object \texttt{A} is \textit{proper} if and only if it satisfies all of the the following conditions:

1. \texttt{A.Data} points to an array of dimension \texttt{A.Capacity} or is NULL if \texttt{A.Capacity} == 0.
2. \texttt{A.Data[0 : A.Usage-1]} are the values stored in the set (in unspecified order).
3. \texttt{A.Data[A.Usage : A.Capacity-1]} equal \texttt{INT32_MAX}.

A \textit{data type} consists of a collection of values and a set of operations that may be performed on those values. For a set type, it would make sense to provide the common set operations; for example:

\begin{verbatim}
/**
* Sets *pUnion to be the union of the sets *pA and *pB.
* Pre: *pUnion, *pA and *pB are all proper
*      *pUnion != pA and pUnion != pB
* Post: *pA and *pB are unchanged
*       *pUnion is proper
*       If successful:
*       For every integer x, x is contained in *pUnion iff x is contained in
*       *pA or x is contained in *pB (or both).
*       *pUnion->Capacity == *pA->Capacity + *pB->Capacity
*       *pUnion->Usage    == *pA->Usage + *pB->Usage - number of elements that
*                       occur in both *pA and *pB
*       *pUnion is proper
*       else *pUnion is unchanged
* Returns: true if the union is successfully created; false otherwise
*/
bool CSet_Union(CSet* const pUnion, const CSet* const pA, const CSet* const pB);
\end{verbatim}
A few design decisions should be explained.

The functions we will implement are designed to pass pointers to `CSet` objects, rather than to pass `CSet` objects by copy. While that would not duplicate the array (see the deep copy discussion later in this specification), passing a `CSet` object would still entail copying the actual members of the object, and that's wasteful of both time and space.

The interface implies that the union is a different `CSet` object than either of the `CSet` objects being combined. That has semantic integrity, since in mathematics writing $A \cup B$ does not imply that either of the two sets will be changed. That is, $A \cup B$ is the name of a different set than $A$ or $B$ (although the union could equal one of them).

The interface of the function `CSet_Union()` implies that `pA` and `pB` point to different `CSet` objects that `pUnion`. This alleviates some potential problems in implementing the computation of the union.

This provides an excuse to make good use of the `const` qualifier, applied to the pointer and/or its target, as appropriate. In the case of the function above, there is no logical reason the function should modify either of the `CSet` objects whose union is being found, so the pointers to those are both qualified as `const CSet* const`. But the function does need to modify the `CSet` object that will hold the union, so the pointer to that object is qualified as `CSet* const`.

Note that all three pointers are qualified so that the function cannot make any of them point to a different location in memory. Obviously there's no reason for the function to give any of those pointers a new target; qualifying them as `const` pointers ensures that we won't make any mistakes of that sort within the definition of the function.

The first post-condition might seem to be superfluous; since the use of `const` would seem to prevent any modifications to `*pA` or `*pB`. However, C allows many typecasts that are potentially unwise, including casting away `const`-ness. The stated post-condition assures the caller that the implementation of the function does not violate the intent indicated by the way `const` was applied to the parameters.

The `CSet` type raises a deep-copy issue, since the data array is allocated dynamically. Since a user might very well need to make a copy of a `CSet` object, we will provide a deep-copy function to support that. The cost of making a deep copy of a `CSet` object may also be significant, due to the potential size of that array, so we should take care to optimize the implementation:

```c
/**
 * Makes a deep copy of a CSet object.
 * 
 * Pre:
 *   *pCopy and *pSet are proper
 * Post:
 *   *pSet is unchanged
 *   If successful:
 *     *pCopy->Capacity == pSet->Capacity
 *     *pCopy->Usage == pSet->Usage
 *     *pCopy[0:*pCopy->Capacity] == pSet[0:pSet->Capacity]
 *     *pCopy->Data != pSet->Data
 *     *pCopy is proper.
 *   else:
 *     *pCopy is unchanged
 *   Returns:
 *     true if successful, false otherwise
 */
bool CSet_Copy(CSet* const pTarget, const CSet* const pSource);
```

In this case, there is no requirement that `*pTarget` and `*pSource` are different. The rationale is that it's easy to handle that case (do nothing), and while it's strange to specify copying an object to itself, a user may choose to do so.
Here's another support function we'll include:

```c
/**
 * Adds Value to a pSet object.
 *
 * Pre:
 *   *pSet is proper
 *   Value has been initialized
 * Post:
 *   If successful:
 *     Value is a member of *pSet
 *     pSet->Capacity has been increased, if necessary
 *     *pSet is proper
 *   else:
 *     *pSet is unchanged
 * Returns:
 *   true if successful, false otherwise
 */
bool CSet_Insert(CSet* const pSet, int32_t Value);
```

One of the post-conditions deserves some comments. If the CSet object *pSet is full, its Capacity is supposed to be increased. That requires allocating a new array dynamically, which could fail; that's why the return type is bool. Once the new array is created, the values must be copied into it from the old array, the new element must be added, and the old array must be deallocated in order to avoid a memory leak. This is a good place to consider using the C Standard Library function realloc().

How should you determine the size of the new array? On the one hand, it is good to avoid wasting memory. On the other hand, the resizing operation described above is fairly expensive, and it is desirable to avoid doing it often.

If we double the size of the array, we can show that the average cost of an insertion is \(O(1)\). In essence, if we currently have a full array of size \(N\), we get one very expensive insertion as we create a new array of size \(2N\), and then we get up to \(N\) more insertions that are very cheap. So, the average cost is quite good.

Here are the (mostly mathematical) set operations we will support:

- inserting a new value into a set
- removing a value from a set
- determining if a value is a member of a set
- forming the union of two sets
- determining if two sets are equal
- determining if one set is a subset of another set
- determining if a set is empty
- make a set empty
- making a copy of a set

There are many other operations that would be included in a production version of a set type (intersection, difference, cardinality, is-proper-subset-of, etc.).

Pay attention to the comments in the header file. All the stated pre- and post-conditions are part of the assignment. Pay particular attention to avoiding memory leaks.

A header file is supplied on the course website. You should use that as your starting point. Your solution will be compiled with a test driver, using the supplied header file, so if you do not conform to the specified interfaces there will very likely be compilation and/or linking errors.
What to Submit

You will submit a single .c file, containing nothing but the implementation of the specified C functions. Be sure to conform to the specified function interfaces.

Your submission will be compiled with a test driver and graded according to how many cases your solution handles correctly.

This assignment will be graded automatically. You will be allowed up to ten submissions for this assignment. Test your function thoroughly before submitting it. Make sure that your function produces correct results for every test case you can think of.

Some requirements, especially those related to memory management, will be checked manually by a TA after the assignment is closed. Any shortcomings will be assessed a penalty, which will be applied to your score from the Curator.

The course policy is that the submission that yields the highest score will be checked. If several submissions are tied for the highest score, the earliest of those will be checked. So, you must implement your solution to handle these requirements before you make submissions, not retrofit the requirements later.

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,
    //   or any other unauthorized source, either modified or unmodified.
    //
    // - If any C language code or documentation used in my program
    //   was obtained from another 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>

Failure to include this pledge in a submission is a violation of the Honor Code.
Appendix: Deep Copy

When an assignment is performed with C struct variables, the operation is carried out (logically) by copying the values of the data members from the object on the right side to the corresponding data members of the object on the left side:

```c
struct Point {
    int x;
    int y;
};
struct Point P = {19, 71};
struct Point Q;
Q = P;
```

So, what's wrong with that? Nothing. But, suppose we have a struct type that includes a data member that is a pointer to something that's allocated dynamically (like a CSet object); assume we have the CSet object A from page 1 and we perform an assignment:

```c
CSet B;
B = A;
```

The result will be that B and A will "share" the dynamically-allocated array, because the value of A.Data (which is the address of the array) will have been copied to B.Data:

```
A
  Capacity: 10
  Usage: 4
  Data: 17 9 23 44
B
  Capacity: 10
  Usage: 4
  Data: 17 9 23 44
```

The effect of the assignment is to make a shallow copy of A. That is, the data members of A are duplicated, but if A contains a pointer, the target of that pointer is not duplicated. In most cases, we would expect that performing an assignment like the one above would result in producing a full (or deep) copy of A. That is, we'd expect the result would have been:

```
A
  Capacity: 10
  Usage: 4
  Data: 17 9 23 44
B
  Capacity: 10
  Usage: 4
  Data: 17 9 23 44
```

The possible consequences of making a shallow copy are severe. Since the two objects share their dynamic content, changes to that content via one object will be visible via the other object. (Of course, there may be a circumstance in which that's exactly what we would want.) Worse, if the dynamic content is deallocated via one of the objects, the other object...
will have lost its dynamic content, but that object will still hold a pointer to the no-longer-accessible memory. That's likely to lead to runtime errors.

In C solutions, we fix the shallow copy problem by implementing a specialized copy function that makes a deep copy of an object. For the \texttt{CSet} type, we would implement the following function:

```c
/** *
 * Makes a deep copy of a CSet object.
 * *
 * Pre:
 * *pCopy and *pSet are proper
 * Post:
 * *pSet is unchanged
 * If successful:
 * pCopy->Capacity == pSet->Capacity
 * pCopy->Usage == pSet->Usage
 * pCopy[0:pCopy->Capacity] == pSet[0:pSet->Capacity]
 * pCopy->Data != pSet->Data
 * *pCopy is proper.
 * else:
 * *pCopy is unchanged
 * Returns:
 * true if successful, false otherwise
 */
bool CSet_Copy(CSet* const pTarget, const CSet* const pSource);
```

A deep copy function must:

- deallocate any previous deep content in the target object
- duplicate the purely shallow content from the source object into the target object
- allocate memory for the necessary dynamic content in the target object
- duplicate the deep content from the source object into the target object

It should be noted that the deep copy problem is not unique to C. The same scenario would arise in the other languages you are likely to use, including Java (where it is most commonly handled by supplying a \texttt{clone()} method) and C++ (where it is most commonly handled with a copy constructor and an assignment operator overload). In all cases, the same basic steps must be carried out.