Assignment 1: `grid`

Due November 20, 11:59 PM

Introduction

The STL container classes encompass a wide selection of associative and sequence containers. However, one useful data type that did not find its way into the STL is a multidimensional array class akin to the CS106X `Grid`. In this assignment, you will implement an STL-friendly version of the CS106X `Grid` class, which we'll call `grid`, that will support deep-copying, STL-compatible iterators, intuitive element-access syntax, and relational operators. Once you've done, you'll have an industrial-strength container class that will almost certainly be of assistance in your future programming endeavors.

The Assignment

In this assignment, you will implement the `grid` container from scratch and will get to practice the techniques we've explored over the past several lectures. The resulting class is relatively large, and to simplify the task of creating it I've broken the assignment down into six smaller steps. While you're free to implement the class features in any order you'd like, I strongly encourage you to follow the steps outlined below, since they greatly simplify implementation.

Step 0: Implement the Basic `grid` Class.

Before diving into some of the `grid`'s more advanced features, it's probably best to warm up by implementing and testing the `grid` basics. Below is a partial specification of the `grid` class that provides core functionality:

Figure 0: Basic (incomplete) interface for the `grid` class

```
template <typename ElemType> class grid {
public:
    /* Constructors, destructors. */
    grid();                                 // Create empty grid
    grid(int width, int height);            // Construct to specified size
    ~grid();

    /* Resizing operations. */
    void clear();                           // Empty the grid
    void resize(int width, int height);     // Resize the grid

    /* Query operations. */
    int getWidth() const;                   // Returns width of the grid
    int getHeight() const;                  // Returns height of the grid
    bool empty() const;                     // Returns whether the grid is empty
    int size() const;                       // Returns the number of elements

    /* Element access. */
    ElemType& getAt(int x, int y);              // Access individual elements
    const ElemType& getAt(int x, int y) const; // Same, but const
};
```

These functions are defined in greater detail here:

`grid();`	Constructs a new, empty `grid`.
`grid(int width, int height);`	Constructs a new `grid` with the specified width and height. Each element in the `grid` is initialized to its default value. Note that unlike the CS106X `Grid`, which works in terms of rows and columns, the CS106L `grid` works in terms of width and height. In true STL style, you do not need to worry about error handling if the dimensions are negative.
`~grid();`	Deallocates any resources in use by the `grid`.
`void clear();`	Resizes the `grid` to 0x0.
`void resize(int width, int height);`	Discards the current contents of the `grid` and resizes the `grid` to the specified size. Each element in the `grid` is initialized to its default value. As with the constructor, don't worry about the case where the dimensions are negative.
`int getWidth() const;` `int getHeight() const;`	Returns the `width` and `height` of the `grid`.
`bool empty() const;`	Returns whether the `grid` contains no elements. This is true if either the `width` or `height` is zero.
`int size() const;`	Returns the number of total elements in the `grid`.
`ElemType& getAt(int x, int y);` `const ElemType& getAt(int x, int y) const;`	Returns a reference to the element at the specified position. Note that this function is `const`-overloaded. Do not worry about the case where the indices are out of bounds.

You are free to add any number of private data members and helper methods as you like, provided that you do not change the interface provided above.

Because `grid` objects can be dynamically resized, the `grid` you create will need to be backed by some sort of dynamic memory management. Because the `grid` represents a two-dimensional entity, you might think that you would need to use a dynamically-allocated multidimensional array to store `grid` elements. However, working with dynamically-allocated multidimensional arrays is extremely tricky and greatly complicates the assignment. Fortunately, we can sidestep this problem by implementing the two-dimensional `grid` object using a single-dimensional dynamically-allocated array. To see how this works, consider the following 3x3 grid:

1	2	3
4	5	6
7	8	9

We can represent all of the elements in this grid using a one-dimensional array by laying out all of the elements sequentially, as seen here:

1	2	3	4	5	6	7	8	9

If you'll notice, in this ordering, the three elements of the first row appear in order as the first three elements, then the three elements of the second row in order, and finally the three elements of the final row in order. Because this one-dimensional representation of a two-dimensional object preserves the ordering of individual rows, it is sometimes referred to as *row-major order*.

To represent a grid in row-major order, you will need to be able to convert between grid coordinates and array indicates. Given a coordinate (*x*, *y*) in a grid of dimensions (*width*, *height*), the corresponding position in the row-major order representation of that grid is given by *index* = *x* + *y* * *width*. The intuition behind this formula is that because the ordering within any row is preserved, each horizontal step we take in the grid translates into a single step forward or backward in the row-major order representation of the grid. However, each vertical step in the grid requires us to advance forward to the next row in the linearized grid, which can be found *width* steps further in the array.

Similarly, given an index *index* into the array, we can compute the (*x*, *y*) position for this element using the formula (*index* % *width*, *index* / *width*), where *index* / *width* uses integer division.

For this assignment, you should implement the `grid` class using a single-dimensional array in row-major order. This will greatly simplify the implementation, especially when we come to iterator support later in the assignment.

Because the `grid` will be implemented as a single-dimensional array in row-major order, in practice we would implement `grid` in terms of `vector`. However, for the purposes of this assignment, you **must** implement the `grid` class using raw, dynamically-allocated C++ arrays. This will help you practice writing copy constructors and assignment operators.

Step 1: Add Support for Iterators

Now that you have the basics of a `grid` class, it's time to add iterator support. Like the `map` and `set`, the `grid` class does not naturally lend itself to a linear traversal – after all, `grid` is two-dimensional – and so we must arbitrarily choose an order in which to visit elements. In this assignment, we'll have `grid` iterators traverse the grid row-by-row, top to bottom, from left to right. Thus, given a 3x4 `grid`, the order of the traversal would be

0	1	2
3	4	5
6	7	8
9	10	11

Fortunately, this order of iteration maps quite naturally onto the row-major ordering we've chosen for the `grid`. If we consider how the above grid would be laid out in row-major order, the resulting linear array would look like this:

0	1	2	3	4	5	6	7	8	9	10	11

Thus this iteration scheme maps to a simple linear traversal of the underlying representation of the `grid`. Because we've chosen to represent the elements of the `grid` as a pointer to a raw C++ array, we can iterate over the elements of the `grid` using pointers. We thus add the following definitions to the `grid` class:

Figure 1: Iterator interface for the `grid` class

```
// iterators are pointers to elements of the stored type.
typedef ElemType* iterator;

// const_iterators are pointers-to-const elements of the stored type.
typedef const ElemType* const_iterator;

// Basic STL iterator functions.
      iterator begin();
const_iterator begin() const;
      iterator end();
const_iterator end() const;

// Useful helper functions to get iterators to locations.
      iterator getIteratorToElement(int x, int y);
const_iterator getIteratorToElement(int x, int y) const;

// Convert from iterator to grid coordinates.
pair<int, int> getCoordinatesForIterator(const_iterator itr) const;
```

Here, the first two lines use `typedef` to export the `iterator` and `const_iterator` types. The other functions are documented below:

`iterator begin();` `const_iterator begin() const;`	Returns an iterator to the first element in the `grid`. Since the `grid` is implemented in row-major order, this is an iterator to the element in the upper-left corner of the `grid`.
`iterator end();` `const_iterator end() const;`	Returns an iterator to the element one past the end of the `grid`.

`iterator`
`getIteratorToElement(int x, int y);`
 	
`const_iterator`
`getIteratorToElement(int x, int y) const;`
	Given a coordinate inside the grid, returns an iterator to that element. Do not worry about bounds-checking.

`pair<int, int>`
`getCoordinatesForIterator`
`(const_iterator itr) const;`
	Given a `const_iterator`, returns a `pair<int, int>` representing the *x* and *y* coordinates of the element pointed to by the iterator. Note that this function only accepts elements of type `const_iterator`, but will still work for regular `iterator`s because `iterator`s are implicitly convertible to `const_iterator`s. Do not worry about handling the case where the iterator is not contained within the `grid`.

When implementing these functions, there's a small C++ template caveat to be aware of. Suppose that you want to implement the function `begin` outside of the body of the `grid` class. Consider the following code:

```
template <typename T> grid<T>::iterator grid<T>::begin()
{
    /* ... Your code here ... */
}
```

This code looks fairly innocuous, but unfortunately is not legal C++ code. The reason has to do with the fact that C++ will misinterpret the return type `grid<T>::iterator` as the name of a class constant inside of the `grid<T>` class, rather than the name of a type exported by the `grid<T>` class.* Thus, if you want this function to return a `grid<T>::iterator`, you must explicitly indicate to C++ that `iterator` is the name of a type, not a constant. To do so, you use a second form of the `typename` keyword that tells C++ that the next expression should be treated as the name of a type. The syntax is:

```
template<typename T>
typename grid<T>::iterator grid<T>::begin()
{
    /* ... Your code here ... */
}
```

Here, we've prefaced the full type `grid<T>::iterator` with the `typename` keyword, which tells C++ that the next term is a type. When working on this assignment, if you ever use the fully-qualified name of a type nested inside of `grid<T>` inside a template definition, make sure that you preface it with the `typename` keyword. Otherwise, you might get some very strange compiler errors.

Step 2: Add Deep-Copying Support

Now that we have a basic `grid` class with iterator support, it's time to write a copy constructor and assignment operator. Adding these functions to the `grid` shouldn't be particularly difficult. If you're particularly clever, you can simplify the task of writing the code to deep-copy an array from one `grid` to another using the `grid`'s `iterator` interface and the STL `copy` algorithm.

Step 3: Add Support for Bracket Syntax

When using regular C++ multidimensional arrays, we can write code that looks like this:

```
int myArray[137][42];
myArray[2][4] = 271828;
myArray[9][0] = 314159;
```

However, with the current specification of the `grid` class, the above code would be illegal if we replaced the multidimensional array with a `grid<int>`, since we haven't provided an implementation of `operator []`.

As you saw in lecture, adding support for the brackets operator to linear classes like the `vector` is extremely simple – simply have the brackets operator return a reference to proper array element. Unfortunately, the same is not true of `grid` and it is somewhat tricky to design the `grid` class such that

---

* The reason for this ambiguity is technical and has to do with how the C++ compiler is implemented. If you're interested in learning why this is the case, consider taking CS143 or the new CS107.

the bracket syntax will work correctly. The reason is that if we write code that looks like this:

```
grid<int> myGrid(137, 42);
int value = myGrid[2][4];
```

By replacing the bracket syntax with calls to `operator []`, we see that this code expands out to

```
grid<int> myGrid(137, 42);
int value = (myGrid.operator[] (2)).operator[] (4);
```

Here, there are *two* calls to `operator []`, one invoked on the `myGrid` object, and the other invoked on the value returned by `myGrid.operator[]`. Thus the object returned by the `grid`'s `operator[]` function must *itself* define an `operator []` function so that the above code will compile. It is this returned object, rather than the `grid` itself, which is responsible for retrieving the requested element from the `grid`. Since this temporary object is used to perform a task normally reserved for the `grid`, it is sometimes known as a *proxy object*.

Now that we know that the object returned from the `grid`'s `operator []` is supposed to retrieve the stored element, how should we go about making the necessary changes to the `grid` such that the above code will compile? First, we will need to define a new class that represents the object returned from the `grid`'s `operator []` function. In this discussion, we'll call it `MutableReference`, since it represents an object that can call back into the `grid` and mutate it. We'll define `MutableReference` inside of the `grid` class for simplicity. The interface of `MutableReference` looks like this:

```
class MutableReference
{
public:
    friend class grid;
    ElemType& operator[] (int yCoord);
private:
    /* Implementation specific.  Contains some way of referring back to the
     * original grid object, as well as the x coordinate specified in the
     * grid's operator[] function.
     */
};
friend class MutableReference;
```

While I've left out the implementation details that make `MutableReference` work correctly (that's for you to figure out!), the basic idea should be clear from the above comments. The `MutableReference` object stores some means of referring back to the `grid` from which it was created, along with the index passed in to the `grid`'s `operator []` function when the `MutableReference` was created. That way, when we invoke the `MutableReference`'s `operator []` function specifying the *y* coordinate of the grid, we can pair it with the stored *x* coordinate, then query the grid for the element at (*x*, *y*).

Now, we'll define an `operator []` function for the `grid` class that looks something like this:

```
MutableReference operator[] (int xCoord);
```

This function is responsible for taking in an *x* coordinate, then returning a `MutableReference` that stores this *x* coordinate, along with some way of referring back to the original `grid` object. That way, if a class client writes

```
int value = myGrid[1][2];
```

The following sequences of actions occurs:

1. `myGrid.operator[]` is invoked with the parameter 1.
2. `myGrid.operator[]` creates a `MutableReference` storing the *x* coordinate 1 and a means for communicating back with the `myGrid` object.
3. `myGrid.operator[]` returns this `MutableReference`.
4. The returned `MutableReference` then has its `operator[]` function called with parameter 2.
5. The returned `MutableReference` then calls back to the `myGrid` object and asks for the element at position (1, 2).

This sequence of actions is admittedly complex, but fortunately it is transparent to the client of the `grid` class and is rather efficient.

The resulting changes to the `grid` interface that add support for the bracket operator are as follows:

Figure 2: `operator []` interface for the `grid` class

```
class MutableReference
{
public:
    friend class grid;
    ElemType& operator [](int y); // Return element from the original grid
private:
    /* ... any data members and helper functions you see fit ... */
};

class ImmutableReference // Like MutableReference, but for const grids.
{
public:
    friend class grid;
    const ElemType& operator [](int y) const;
private:
    /* ... any data members and helper functions you see fit ... */
};

friend class MutableReference;
friend class ImmutableReference;
MutableReference   operator[] (int x);       // For non-const grids
ImmutableReference operator[] (int x) const; // For const grids.
```

Note that we've introduced another helper class here, `ImmutableReference`, that is returned by the `const`-overloaded version of the `grid`'s `operator []` function. `ImmutableReference` works the same way as `MutableReference`, except that the element it returns is `const`.

As with `iterator`s, when implementing the `grid`'s `operator []` function outside of the body of the `grid` class, you will need to use the `typename` keyword when stating that the function's return type is an object of type `grid<T>::MutableReference`. Feel free to use the following code as a starting point:

```
template<typename ElemType>
typename grid<T>::MutableReference
   grid<T>::operator [](int x) { /* ... */ }
```

Another important point to note is that the `MutableReference` and `ImmutableRefernce` classes declare `grid` as a `friend` class and vice-versa. Even though the `MutableReference` and `ImmutableReference` classes are declared inside of `grid`, C++ prevents `grid` from modifying `MutableReference` data members and `MutableReference` from modifying `grid` data members. Because the three classes together form one logical unit of encapsulation, in the above interface I've marked the classes as `friend`s as one another so that they can freely access each other's internals. Your implementation may or may not require these declarations, so feel free to remove them.

Step 4: Define Relational Operators

Now that our `grid` has full support for iterators and a nice bracket syntax that lets us access individual elements, it's time to put on the finishing touches. As a final step in the project, you'll provide implementations of the relational operators for your `grid` class.

Add the following functions to your `grid` class and define the appropriate support for them:

Figure 3: Relational operator interface for the `grid` class

```
/* Relational operators */
bool operator <  (const grid& other) const;
bool operator <= (const grid& other) const;
bool operator == (const grid& other) const;
bool operator != (const grid& other) const;
bool operator >= (const grid& other) const;
bool operator >  (const grid& other) const;
```

Note that of the six operators listed above, only the `==` and `!=` operators have intuitive meanings when applied to `grid`s. However, it also makes sense to define a `<` operator over `grid`s so that we can store them in STL `map` and `set` containers, and to ensure consistency, we should define the other three operators as well.

Because there is no natural interpretation for what it means for one `grid` to be "less than" another, you are free to implement these functions in any way that you see fit, provided that the following invariants hold for any two grids `one` and `two`:

1. `one == two` if and only if the two grids have the same dimensions and each element compares identical by the `==` operator.
2. All of the mathematical properties of `<`, `<=`, `==`, `!=`, `>=`, and `>` hold. For example if `one < two`, `!(one >= two)` should be true and `one <= two` should also hold. Similarly, `one == one`, and if `one == two`, `two == one`.
3. For any `one` and `two`, exactly one of `one == two`, `one < two`, or `one > two` should be true.

The final item on this list is perhaps the most difficult to get correct but is very important. Given any two `grid`s of arbitrary dimensions and elements, those `grid`s must be comparable. You might want to use the code from the `DebugVector` that we covered in Wednesday's lecture as a starting point.

Step 5: Test Your grid!

Now that the `grid` class is fully specified and implemented, it's time to test it in isolation to see that you've ironed out all of the bugs. To help get you started, I've provided basic testing code on the CS106L website that covers some of the more common cases. This testing framework is far from complete, but should be a good start on your way to well-debugged code.

More to Explore

While the `grid` class you'll be writing is useful in a wide variety of circumstances, there are several extra features you may want to consider adding to the `grid` class. While none of these extensions are required for the assignment, you might find them interesting challenges:

1. **Row and column iterators**: The iterators used in this `grid` class are only useful for iterating across the entire `grid` class and are not particularly suited to iterating across the `grid`'s rows or columns. Consider defining two custom classes, `row_iterator` and `column_iterator`, that can iterate over `grid` rows and columns using iterator syntax. Make sure to provide `const` overloads!
2. **Smart resizing**: Right now, the only way to resize a `grid` is by discarding the old contents entirely. Wouldn't it be nice if there was a way to resize the `grid` by inserting or deleting rows and columns in the existing structure? This is not particularly difficult to implement, but is a nice feature to add.
3. **Subgrid functions**: Just as the C++ `string` exports a `substr` function to get a substring from that string, it might be useful to define a function that gets a "subgrid" out of the `grid`.
4. **Templatized Conversion Functions**: When working with two `grid`s of different types that are implicitly convertible to one another (say, a `grid<int>` and a `grid<double>`), it might be useful to allow expressions like `myDoubleGrid = myIntGrid`. Consider adding a templatized conversion constructor to the `grid` class to make these assignments legal.

If you implement any additional features in your `grid` class, make a note of it in your submission and I'll gladly look over it and offer feedback!

Submission Instructions

Once you've completed the assignment, email all of your source files to htiek@cs.stanford.edu. If you've added any extensions or made any changes that I should be aware of, please make a note of it in your submission email. Then pat yourself on the back – you've just written an STL-compatible container class!

Figure 4.1: The complete interface for the `grid` class, part 1/2

```
template <typename ElemType> class grid {
public:
    /* Constructors, destructors, and copying support. */
    grid();
    grid(int width, int height);
    grid(const grid& other);
    grid& operator= (const grid& other);
    ~grid();

    /* Resizing operations. */
    void clear();
    void resize(int width, int height);

    /* Query operations. */
    int getWidth() const;
    int getHeight() const;
    bool empty() const;
    int size() const;

    /* Element access. */
    ElemType& getAt(int x, int y);
    const ElemType& getAt(int x, int y) const;

    /* Iterator types. */
    typedef ElemType* iterator;
    typedef const ElemType* const_iterator;

    /* Basic iterator functions. */
          iterator begin();
    const_iterator begin() const;
          iterator end();
    const_iterator end() const;

    /* Iterator utility functions. */
          iterator getIteratorToElement(int x, int y);
    const_iterator getIteratorToElement(int x, int y) const;
    pair<int, int> getCoordinatesForIterator(const_iterator itr) const;

    /* Bracket operators. */
    class MutableReference {
    public:
        friend class grid;
        ElemType& operator [](int y);
    private:
        /* ... implementation specific ... */
    };

    class ImmutableReference {
    public:
        friend class grid;
        const ElemType& operator [](int y) const;
    private:
        /* ... implementation specific ... */
    };
```

*Figure 4.2: The **complete** interface for the `grid` class, part 2/2*

```
    friend class MutableReference;
    friend class ImmutableReference;
    MutableReference   operator[] (int x);
    ImmutableReference operator[] (int x) const;

    /* Relational operators */
    bool operator <  (const grid& other) const;
    bool operator <= (const grid& other) const;
    bool operator == (const grid& other) const;
    bool operator != (const grid& other) const;
    bool operator >= (const grid& other) const;
    bool operator >  (const grid& other) const;

private:
    /* ... implementation specific ... */
    /* Remember that grid MUST be backed by a dynamically-managed array. */
};
```