Now that all things recursion are behind us, we’ll continue our discussion of objects and how they’re defined, and introduce the notion of a pointer, a memory address, and a dynamically allocated figure to implement some of the more basic objects we already have experience with—specifically, the `Vector`, the `Grid`, and the `Stack`. Once I press through those, I’ll spend some time working through an abstraction that’s similar to the `SparseGrid` template in the CS106X library set: the `SparseStringArray`.

**Larger Problem: The Sparse String Array**

A `SparseStringArray` is an array-like data structure providing super-duper fast access to its elements, and near constant-time `operator[]` operations. It layers array semantics over an ordered sequence of C++ strings, with the understanding that most of the strings are _empty_. The `SparseStringArray` is different from our standard C++ `Vector` and other array-like data structures, because its size is set at construction time (as with a traditional array), and its memory footprint is kept to an absolute minimum. In theory, each empty string requires just one bit of storage, which is less than 3% of the memory cost incurred by the allocation of a full empty `string`. The implementation is slower than our `Vector`, but it’s a wise choice when memory is at a premium and the vast majority of its strings are empty.

Our `SparseStringArray` is backed by an array of _groups_, where each _group_ is responsible for managing a contiguous subset of array indices. The programmer specifies not only the logical length of the `SparseStringArray`, but also the group size. If the logical length of the full `SparseStringArray` is, for instance, established as 50,000, and the group size is established to be 100, then group 0 manages indices 0 through 99, group 1 manages indices 100 through 199, group 2 manages indices 200 through 299, and so forth. All search, set, and get operations are passed on to the appropriate group. Here’s a glimpse of our class’s _private_ sector:
Each group contains a bitmap, which is an array of booleans whose length is equal to the group size, and a C++ `Vector<std::string>` to store just the non-empty strings. Search for a particular element amounts to search within a particular group at index $i$. If `bitmap[i]` is `false`, then the string at the $i^{th}$ position is understood to be the empty string. If instead `bitmap[i]` is `true`, then the group needs to find the corresponding string in the C++ `Vector<std::string>`.

Each `true` in a group’s bitmap corresponds to some `string` in the same group’s `Vector<std::string>`. The `true` at the lowest index in the bitmap corresponds to the 0$^{th}$ entry in the `Vector`; the `true` at the second lowest index in the bitmap corresponds to the 1$^{st}$ entry in the `Vector`, and so forth; and the total number of `true`s should be equal to the logical length of the accompanying `Vector`. (In practice, the `bool` array would be compressed to use just one bit of memory for each Boolean value, but for our purposes we won’t implement that optimization, since it requires advanced C++ directives we haven’t covered.) Variations of this data structure are used in industry when insanely large arrays—with lengths in the billions or trillions—contribute to a larger system. It also has the neat feature that individual groups can be distributed across multiple processors or multiple machines.

Here’s the core of the `.h` file for the `SparseStringArray` class:

```cpp
class SparseStringArray {
public:
    SparseStringArray(int length, int groupSize);
    ~SparseStringArray();
    int size() const;
    std::string& operator[](int index);
    const std::string& operator[](int index) const;

private:
    struct group {
        bool *bitmap;  // set to be of size 'groupSize'
        Vector<std::string> strings; // ordered Vector<std::string> on the non-empty strings
    };
    group *groups;
    int numGroups;
    int length;
    int groupSize;
    int getVectorIndex(int groupIndex, int bitmapIndex) const;
};
```
Of course, the **SparseStringArray** presents the illusion that all strings, both empty and nonempty, are stored in a sequential, array-like manner. But we understand the concept of encapsulation enough to understand the smoke and mirrors of the implementation: the internal representation is such that only nonempty strings are usually stored. Our job in lecture will be to cover the implementation of the constructor, the destructor, and the various methods. Here’s a test program that illustrates how a client can interact with a **SparseStringArray**.

```cpp
static void printSerialization(const SparseStringArray& ssa) {
    cout << "Serialization: ";
    for (int i = 0; i < ssa.size(); i++) {
        const string& s = ssa[i];
        if (!s.empty()) {
            cout << s;
        }
    }
    cout << endl;
}

int main() {
    SparseStringArray ssa(70000, 35);
    ssa[33001] = "need";
    ssa[58291] = "more";
    ssa[33000] = "Eye";
    ssa[33000] = "I";
    ssa[67899] = "cowbell!";
    printSerialization(ssa);
    return 0;
}
```

Here’s the output of that test program:

```
Serialization: I need more cowbell!
```

**Implementation**

```cpp
/**
 * File: sparse-string-array.cpp
 *---------------------------------------------------------------------
 * Presents the implementation of the SparseStringArray.
 */
#include "sparse-string-array.h"

SparseStringArray::SparseStringArray(int length, int groupSize) {
    this->length = length;
    this->groupSize = groupSize;
    numGroups = length / groupSize;
    groups = new group[numGroups];
    for (int group = 0; group < numGroups; group++) {
        groups[group].bitmap = new bool[groupSize];
        for (int i = 0; i < groupSize; i++) {
            groups[group].bitmap[i] = false;
        }
    }
}
```
SparseStringArray::~SparseStringArray() {
    for (int i = 0; i < numGroups; i++) {
        delete[] groups[i].bitmap;
    }
    delete[] groups;
}

int SparseStringArray::size() const {
    return length;
}

string& SparseStringArray::operator[](int index) {
    int groupIndex = index / groupSize;
    int bitmapIndex = index % groupSize;
    int vectorIndex = getVectorIndex(groupIndex, bitmapIndex);

    if (!groups[groupIndex].bitmap[bitmapIndex]) {
        groups[groupIndex].bitmap[bitmapIndex] = true;
        groups[groupIndex].strings.insert(vectorIndex, "");
    }
    return groups[groupIndex].strings[vectorIndex];
}

static const string kEmptyString;
const string& SparseStringArray::operator[](int index) const {
    int groupIndex = index / groupSize;
    int bitmapIndex = index % groupSize;
    if (!groups[groupIndex].bitmap[bitmapIndex]) {
        return kEmptyString;
    }
    int vectorIndex = getVectorIndex(groupIndex, bitmapIndex);
    return groups[groupIndex].strings[vectorIndex];
}

int SparseStringArray::getVectorIndex(int groupIndex, int bitmapIndex) const {
    int vectorIndex = 0;
    for (int i = 0; i < bitmapIndex; i++) {
        if (groups[groupIndex].bitmap[i]) {
            vectorIndex++;
        }
    }
    return vectorIndex;
}