Problem 1 Solution: Superheroes Then and Now
State of memory just prior to the call to \texttt{barbarella}:

\begin{itemize}
  \item \texttt{ironman}
  \item \texttt{marineboy[0]}
  \item \texttt{marineboy[1]}
  \item \texttt{9189}
\end{itemize}

\textit{No orphaned memory.}

State of memory just before the call to \texttt{barbarella} exits:

\begin{itemize}
  \item \texttt{ironman}
  \item \texttt{marineboy[0]}
  \item \texttt{marineboy[1]}
  \item \texttt{9189}
  \item \texttt{storm}
  \item \texttt{catwoman}
\end{itemize}

\textit{No orphaned memory.}
Problem 2 Solution: Bloom Filters and Sorted String Sets

Here’s my `SortedStringSet` interface:

```cpp
class SortedStringSet {
public:
    SortedStringSet(const Vector<int (*)(const std::string&, int)>& hashers);
    ~SortedStringSet();

    int size() const { return values.size(); }
    bool contains(const std::string& value) const;
    void add(const std::string& value);

private:
    Set<string> values;
    Vector<int (*)(const std::string&, int)> hashers;
    bool *footprints;
    int allocLength;
    int numFootprints;
    void createEmptyBloomFilter();
    void leaveFootprints(const std::string& value);
    void rehash();
};
```

Everything below the `Set<string> values` line is my own, and all of what’s new helps to manage a Bloom filter. The two instance variables `footprints` and `allocLength` team up to manage the Bloom filter as a manually managed array of Boolean footprints that needs to be reallocated when we congest the filter with lots of `true` values.

The constructor and destructor are algorithmically straightforward. The primary reason I decompose the constructor to call the helper `createEmptyBloomFilter` method is that I need to execute the same exact code within the `add` method.

```cpp
static const int kInitBloomFilterLength = 1001;
SortedStringSet::SortedStringSet(const Vector<int (*)(const string&, int)>& hashers) {
    this->hashers = hashers;
    allocLength = kInitBloomFilterLength;
    createEmptyBloomFilter();
}

SortedStringSet::~SortedStringSet() {
    delete[] footprints;
}

void SortedStringSet::createEmptyBloomFilter() {
    footprints = new bool[allocLength];
    for (int i = 0; i < allocLength; i++) footprints[i] = false;
    numFootprints = 0;
}
```

Note that `createEmptyBloomFilter` assumes that `allocLength` has been initialized to be the desired Bloom filter length before it’s called. As is always the case, we need to manually zero out every entry in the `footprints` array, because C++ doesn’t support
default initialization like some other languages do. Because the Bloom filter is empty (e.g. there are no trues anywhere in the array), numfootprints is set to 0.

The implementation of contains is potentially framed as a call to contains on the encapsulated Set<string>. But before we commit to the (relatively) expensive Set<string>::contains call, we examine the Bloom filter to see if the expected set of footprints have been left by the accumulation of all prior add calls. If they haven’t been, we know there’s no way the supplied string will be in the master Set. If they have been, then and only then is it sensible to examine the master Set to see if the referenced string is truly and officially present.

```cpp
bool SortedStringSet::contains(const string& value) const {
    for (int i = 0; i < hashers.size(); i++) {
        int hash = (hashers.get(i))(value, allocLength);
        if (!footprints[hash]) {
            return false;
        }
    }
    return values.contains(value);
}
```

The implementation of add is more complicated, because we need to check to see if the Bloom filter is congested with a high fraction of footprints. Before we go on stamping down even more footprints, we need to check if there are more trues than falses. If so, we allocate a much larger filter, rehash all existing strings to leave new footprints, and dispose of the old filter. Whether or not we got a new filter, we need to leave some footprints on behalf of the supplied string, and then add it to the master Set.

```cpp
static const double kSaturationFactor = 0.50;
void SortedStringSet::add(const string& value) {
    if (numFootprints > kSaturationFactor * allocLength) rehash();
    leaveFootprints(value);
    values.add(value);
}
```

```cpp
void SortedStringSet::rehash() {
    delete[] footprints;
    allocLength *= hashers.size(); // heuristic: multiply by number of hashers
    createEmptyBloomFilter();
    for (const string& value : values) leaveFootprints(value);
}
```

```cpp
void SortedStringSet::leaveFootprints(const string& value) {
    for (int i = 0; i < hashers.size(); i++) {
        int hash = hashers[i](value, allocLength);
        if (!footprints[hash]) numFootprints++;
        footprints[hash] = true;
    }
}
```