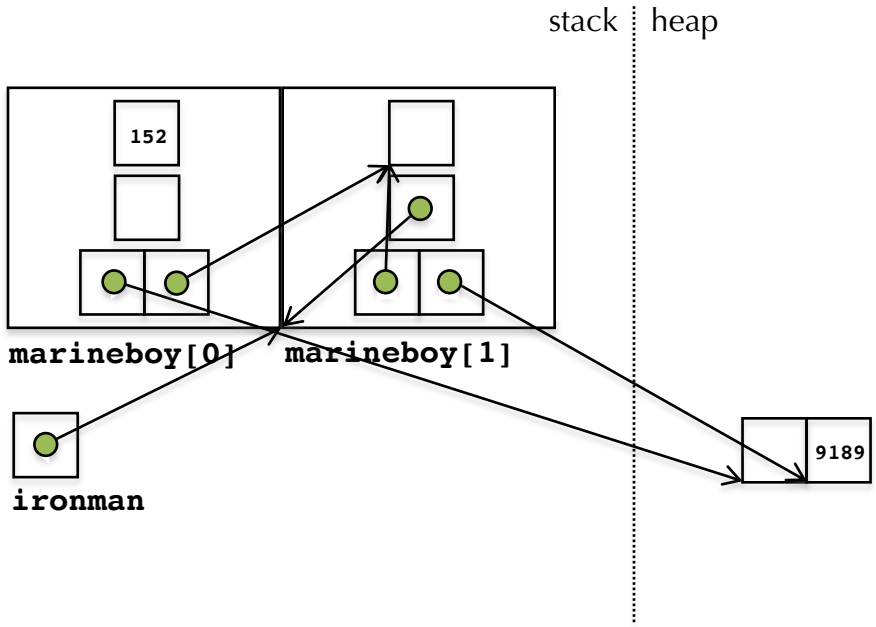
Section Solution

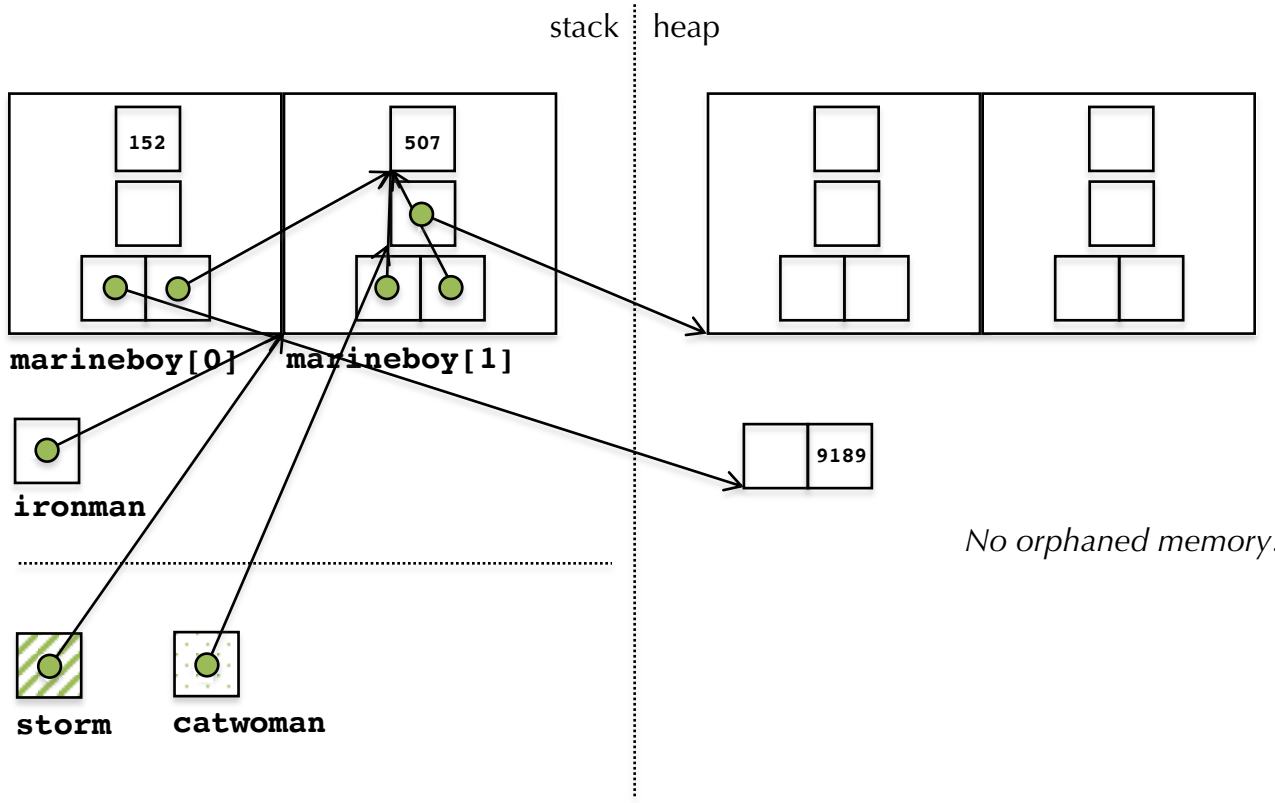
Problem 1 Solution: Superheroes Then and Now

State of memory just prior to the call to **`barbarella`**:



No orphaned memory.

State of memory just before the call to **`barbarella`** exits:



No orphaned memory.

Problem 2 Solution: Bloom Filters and Sorted String Sets

Here's my **SortedStringSet** interface:

```
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.

```
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 **true**s 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.

```
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 **true**s than **false**s. If so, we allocate a much larger filter, rehash all existing **string**s 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**.

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

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

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;
   }
}
```