# 【C++】Day60(2)

| | |
|---|---|
| Class | C++ |
| Date | @February 21, 2022 |
| Material | |
| Series Number | |
| Summary | |

## 【Ch13】Copy Control

### 13.5 Classes That Manage Dynamic Memory

Some classes need to allocate a varying amount of storage at run time. Such classes often can(generally should) use a library container to hold their data.

However, this strategy does not work for every class; some classes need to do their own allocation. Such classes generally must define their own copy-control members to manage the memory the allocate.

As an example, we'll implement a simplification of the library `vector` class. Our class will hold `string`s. Thus, we'll call our class `StrVec`.

*StrVec Class Design*

The `vector` class stores its elements in contiguous storage. To obtain acceptable performance, `vector` preallocates enough storage to hold more elements than are needed.

Each vector member that adds elements checks whether there is space available for another elements. If so, the member constructs an object in the next available spot. If there isn't space left, then the vector is reallocated: The vector obtains new space, moves the existing elements into that space, frees the old space, and adds the new element.

We'll use a similar strategy in our `StrVec` class. We'll use an `allocator` to obtain raw memory. Because the memory an `allocator` allocates is unconstructed, we'll use the allocator's `construct` member to create objects in that space when we need to add an element.

Similarly, when we remove an element, we'll use the `destroy` member to destroy the element.

Each `StrVec` will have three pointers into the space it uses for its elements:

- `elements`, which points to the first element in the allocated memory
- `first_free`, which points just after the last actual element
- `cap`, which points just past the end of the allocated memory

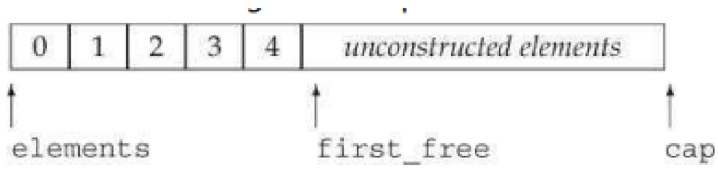


**Figure 13.2. StrVec Memory Allocation Strategy**

In addition to these pointers, `StrVec` will have a member named `alloc` that is an `allocator<string>`. The `alloc` member will allocate the memory used by `StrVec`. Our class will also have four utility functions:

- `alloc_n_copy` will allocate space and copy a given range of elements
- `free` will destroy the constructed elements and deallocate the space
- `chk_n_alloc` will ensure that there is room to add at least one more element to the `StrVec`. If there isn't room for another element, `chk_n_alloc` will call reallocate to get more space
- `reallocate` will reallocate the `StrVe`c when it runs out of space.

### *StrVec Class Definition*

We can now define our `StrVec` class

```
// simplified implementation of the memory allocation stragey for a vector-like class
class StrVec {
public:
  StrVec() : elements(nullptr), first_free(nullptr), cap(nullptr) {}
  StrVec(const StrVec&); // copy constructor
  StrVec &operator=(const StrVec&); // copy assignment
  ~StrVec();
  void push_back(const string&); // copy the element
  size_t size() const { return cap - elements; }
  size_t capacity const { return cap - first_free; }
  string *begin() const { return elements; }
  string *end() const { return first_free; }

private:
```

```
  allocator<string> alloc; // allocates the elements
  // used by the functions that add elements to the StrVec
  void chk_n_alloc() {
    if(size() == capacity())
      reallocate();
  }
  pair<string*, string*> alloc_n_copy(const string*, const string*);
  void free();
  void reallocate();

  string *elements; // pointer to the first element in the array
  string *first_free; // point to the first free element in the array
  string *cap; // pointer to one pasdt the end of array
};
```

### *Using construct*

The `push_back` function calls `chk_n_alloc` to ensure that there is room for an element. If necessary, `chk_n_alloc` will call reallocate.

When `chk_n_alloc` returns, `push_back` knows that there is room for the new element. It asks its allocator member to construct a new last element:

```
void StrVec::push_back(const string& s) {
  chk_n_alloc(); // ensure that there is room for another element
  // construct a copy of s in the element to which first_free points
  alloc.construct(first_free++, s);
}
```

When we use an `allocator` to allocate memory, we must remember that the memory is unconstructed. To use this raw memory we must call `construct`, which will construct an object in that memory.

The first argument to construct must be a pointer to unconstructed

### *The alloc_n_copy Member*

The `alloc_n_copy` member will allocate enough storage to hold its given range of elements, and will copy those elements into the newly allocated space.

```
pair<string*, string*> StrVec::alloc_n_copy(const string* b, const string*e) {
  // allocate space to hold as many elements as are in the range
  auto data = alloc.allocate(e - b);
  // initialize and return a pair constructed from data and the value returned by uninitialized_copy
  return { data, uninitialized_copy(b, e, data) };
}
```

`alloc_n_copy` calculates how much space to allocate by subtracting the pointer to the first element from the pointer one past the last.

### *The free Member*

The `free` member has two responsibilities: It must destroy the elements and then deallocate the space that this `StrVec` itself allocated.

The for loop calls the `allocator` member `destroy` in reverse order, starting with the last constructed element and finishing with the first:

```
void StrVec::free() {
  if(elements) {
    for(auto p = first_free; p != elements;)
      alloc.destroy(--p);
    alloc.deallocate(elements, cap - elements);
  }
}
```

### *Copy-Control Members*

The copy constructor calls `alloc_n_copy`:

```
StrVec::StrVec(const StrVec &s) {
  auto newData = alloc_n_copy(s.begin(), s.end());
  elements = newData.first;
  first_free = cap = newData.second;
}
```

The destructor calls `free`:

```
~StrVec() {
  free();
}
```

The copy-assignment operator calls `alloc_n_copy` before freeing its existing elements. By doing so it protects against self-assignment:

```
StrVec& StrVec::operator=(const StrVec &s) {
  auto data = aloc_n_copy(s.begin(), s.end());
  free();
```

```
  elements = data.first;
  first_free = cap = data.second;
  return *this;
}
```

### *The reallocate Member*

We will double the capacity of the `StrVec` each time we reallocate. If the `StrVec` is empty, we allocate room for one element:

```
void StrVec::reallocate() {
  auto newCapacity = size() ? 2 * size() : 1;
  auto newData = alloc.allocate(newCapacity);

  auto dest = newData;
  auto elem = elements;

  for(size_t i = 0; i != size(); ++i) {
    alloc.construct(dest++, std::move(*elem++));
  }
  free();
  elements = newData;
  first_free = dest;
  cap = elements + newCapacity;
}
```

### *Exercise*

See implementation of the StrVec class in 13_39.cpp