Classes That Manage Dynamic Memory

In this section, we'll implement a simplification of the library `vector` class.

`StrVec` Class Design

Recall that 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 element. 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.

> In essence, the implementation is similar to the call of `realloc` and `malloc` in C.

We'll use a similar strategy in our `StrVec` class. We'll use an `allocator` to obtain raw memory. Beacause the memory an `allocator` allocates is unconstrucoted, 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

In addition to these pointers, `StrVec` have a member named `alloc` that is an `allocator<string>`. The `alloc` member will allocate the memory used by a `StrVec`. Our class 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 the another element, `chk_n_alloc` will call `reallocate` to get more space.
- `reallocate` will reallocate the `StrVec` when it runs out of space.

`StrVec` Class Definition

```
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 std::string &);// copy the element

        size_t size() const{return first_free-elements;}
        size_t capacity() const{return cap-elements;}
```

```
        string * begin()const{return elements;}
        string *end()const {return first_free;}
        //...

    private:
        std::allocator<std::string> alloc;

        void chk_n_alloc(){
            if(size()==capacity())reallocate();
        }
        std::pair<std::string *,std::string *> alloc_n_copy(const std::string
*,const std::strnig *);

        void free(); // destroy the elements and free the space
        void reallocate(); // get more space and copy the existing elements

        std::string *elements;
        std::string *first_free;
        std::string *cap;
};
```

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.construt(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 `consturct`, which will construct an object in that memory.

The first argument to `construct` must be a pointer to unconstructed space allocated by a call to `allocate`. **The remaining arguments determine which constructor to use to construct the object that will go in that space.** In this case, there is only one additional argument. That argument has type `string`, so this call uses the `string` copy constructor.

The `alloc_n_copy` Member

The `alloc_n_copy` member is called when we copy or assign a `StrVec`. Our `StrVec` class has valuelike behavior; when we copy or assign a `StrVec`, we have to allocate independent memory and copy the elements from the original to the new `StrVec`.

This function return a `pair` of pointers, pointing to the beginning of the new space and just past the last element it copied.

```
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,unitialized_copy(b,e,data)};
}
```

> It dose the copy in the return statement, which list initializes the return value. The `first` member of the returned `pair` points to the start of the allocated memory; the second is the value returned from `unitialized_copy`. That value will be pointer positioned one element past the last constructed element.

The `free` Member

The `free` member 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.***

```
void StrVec::free(){

    // may not pass `deallocate` a null pointer;
    if(elements){
        // destroy the old elements in reverse order
        for(auto p=first_free;p!=elements;/*empty*/){
            alloc.destroy(--p);
        }
        // capacity() const {return cap-elements;}
        alloc.deallocate(elements,cap-elements);
    }
}
```

The `destroy` function runs the `string` destructor. The `string` destructor frees whatever storage was allocated by the `string`s themselves.

Once the elements have been destroyed, we free the space that this `StrVec` allocated by calling `deallocate`. The pointer we pass to `deallocate` must be one that was previouly generated by a call to `allocate`. Therefore, we first check that `elements` is not null before calling `deallocate`.

Copy-Control Members

Given our `alloc_n_copy` and `free` members, the copy-control members of our class are straightforward.

```
StrVec::StrVec(const StrVec &s){

    auto newdata=alloc_n_copy(s.begin(),s.end());
```

```
    elements=newdata.first;
    first_free=cap=newdata.second;
}
```

> Because `alloc_n_copy` allocates space for exactly as many elements as it is given, `cap` also points just past the last constructed element.

Then the destructor calls `free()`

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

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

```
StrVec & StrVec::operator=(const StrVec &rhs){
    auto data=alloc_n_copy(rhs.begin(),rhs.end());
    free();
    elements=data.first;
    first_free=cap=data.second;
    return *this;
}
```

Moving, Not Copying, Elements during Reallocation

The `reallocate` functions will:

- allocate memory for a new, larger array of `string`s
- construct the first part of that space to hold the existing elements
- destroy the elements in the existing memory and deallocate that memory

Coping a `string` copies the data because ordinarily after we copy a `string`, there are two users of that `string`. However, when `reallocte` copies the `string`s in a `StrVec`, there will be only one user of these `string`s after the copy. As soon as we copy the elements from the old space to the new, we will immediately destroy the original `string`s.

Coping the data in these `string`s is unnecessary. Our `StrVec`s performance will be ***much*** better if we can avoid the overhead of allocating and deallocating the `string`s themselves each time we reallocate.

Move Constructor and `std::move`

> We can avoid copying the `string`s by using two facilities introduced by the new library, that is move constructor defined in the library classes and `std::move` function.

Move Constructor

The details of how the `string` move constructor works-like any other detail about the implementation-are not disclosed. However, we do know that move constructors typically operate by "moving" resources from the given object to the object being constructed. We also know that the library guarantees that the "moved-from" `string` remains in a valid, destructible state. For `string`, we can imagine that each `string` has a pointer to array of `char`. Presumably the `string` move constructor copies the pointer rather than allocating space for and coping the characters themselves.

The second we'll use is a library function named `move`, which is defined in the `utility` header. There are two important points to know about `move`.

- when `reallocate` constructs the `string`s in the new memory it must call `move` to signal that it wants to use the `string` move constructor. If it omits the call to `move`, the `string` the copy constructor will be used.
- we usually do not provide a `using` declaration for `move`. When we use `move`, we call `sta::move`, not `move`.

The `reallocate` Member

The `reallocate` member will be wrote as below.

```
void StrVec::reallocate(){
    // allocate space for twice as many elements as the current size
    // if the StrVec is empty, we allocate room for one element.
    auto newcapacity=size()?2*size():1;

    auto newdata=alloc.allocate(newcapcity);
    auto dest=newdata;
    auto elem=elements;

    for(size_t i=0;i!=size();++i)
        alloc.construct(dest++,std::move(*elem++));
    // std::move(*elem++) is equivalent to std::move(*elem);++elem;

    free();// free the old space once we've moved the elements

    // update
    elements=newdata;
    first_free=dest;
    cap=elements+newcapacity;
}
```

It's worth noting that the second argument in the call to `construct` is the value returned by `move`.

Calling `move` returns a result that causes `construct` to use the `string` move constructor. Because we're using the move constructor, the memory managed by those `string`s will not be copied. Instead, each `string` we construct will take over ownership of the memory from the `string` to which `elem` points.

After moving the elements, we call `free` to destroy the old elements and free the memory that this `StrVec` was using before the call to `reallocate`. The `string`s themselves no longer manage the memory to which they had pointed;

> We don't know what value the `string`s in the old memory have, but we are guaranteed that it is safe to run the `string` destructor on these objects.