【C++】Day54

Class	C++
Date	@February 8, 2022
Material	
Series Number	
Summary	

【Ch12】Dynamic Memory

Freeing Dynamic Arrays

To free a dynamic array, we use a special form of delete that includes an empty pair of square brackets:

```
delete p; // p must point to a dynamically allocated object or be null
delete[] pa; // pa must point to a dynamically allocated rray or be null
```

The second statement destroys the elements in the array to which `pa` points and frees the corresponding memory.

Elements in an array are destroyed in reverse order. That is, the last element is destroyed first, then the second to last, and so on.

Warning: The compiler is unlikely to warn us if we forget the brackets when we delete a pointer to an array or if we use them when we delete a pointer to an object. Instead, our program is apt to misbehave without warning during execution.

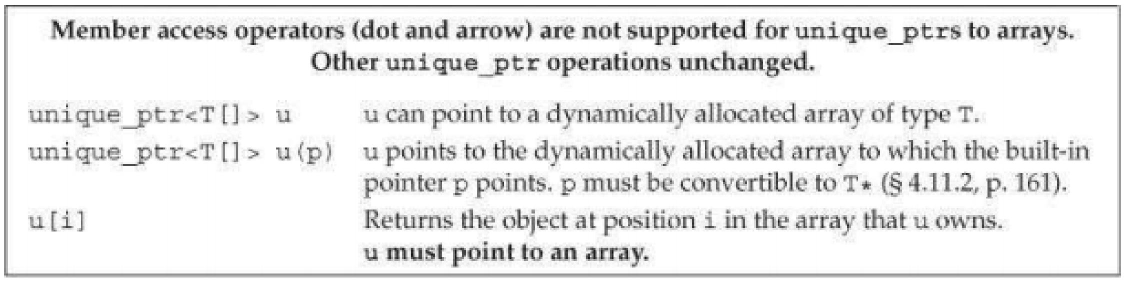
Smart Pointers and Dynamic Arrays

The library provides a version of `unique_ptr` that can manage arrays allocated by `new`. To use a `unique_ptr` to manage a dynamic array, we must include a pair of empty brackets after the object type:

```
unique_ptr<int[]> up(new int[10]);
up.release(); //automatically uses delete[] to destroy its pointer.
```

`unique_ptr`s that point to arrays provide slightly different operations than those we used before. These operations are described below.

Table 12.6. unique_ptrs to Arrays



Member access operators (dot and arrow) are not supported for `unique_ptrs` to arrays. Other `unique_ptr` operations unchanged.	
`unique_ptr<T[]> u`	u can point to a dynamically allocated array of type `T`.
`unique_ptr<T[]> u(p)`	u points to the dynamically allocated array to which the built-in pointer `p` points. `p` must be convertible to `T*` (§ 4.11.2, p. 161).
`u[i]`	Returns the object at position `i` in the array that u owns. **u must point to an array.**

When a `unique_pt`r points to an array, we cannot use the dot and arrow member access operators. After all, the `unique_ptr` points to an array, not an object so these operators would be meaningless.

`shared_ptr` provide no direct support for managing a dynamic array. If we want to use a `shared_ptr` to manage a dynamic array, we must provide our own deleter:

```
// to use a shared_ptr we must supply deleter
shared_ptr<int> sp(new int[10], [] (int *p) -> void { delete[] p; }
```

Had we neglected to supply a deleter, this code would be undefined. By default, `shared_ptr` uses `delete` to destroy the object to which it points. If that object is a dynamic array, using `delete` has the same kinds of problems that arise if we forget to use `[]` when we delete a pointer to a dynamic array.

The fact that `shared_ptr` does not directly support managing array affects how we access the elements in the array:

```
// shared_ptrs don't have subscript operator and don't support pointer arithmetic
for(size_t i = 0; i != 10; ++i) {
  *(sp.get() + i) = i;
}
```

There is no subscript operator for `shared_ptr`s, and the smart pointer types do not support pointer arithmetic. As a result, to access the elements in the array, we must use `get` to obtain a built-in pointer, which we can then use in normal ways.

12.2.2 The allocator Class

An aspect of new that limits its flexibility is that `new` combines allocating memory with constructing objects in that memory. Similarly, `delete` combines destruction with deallocation.

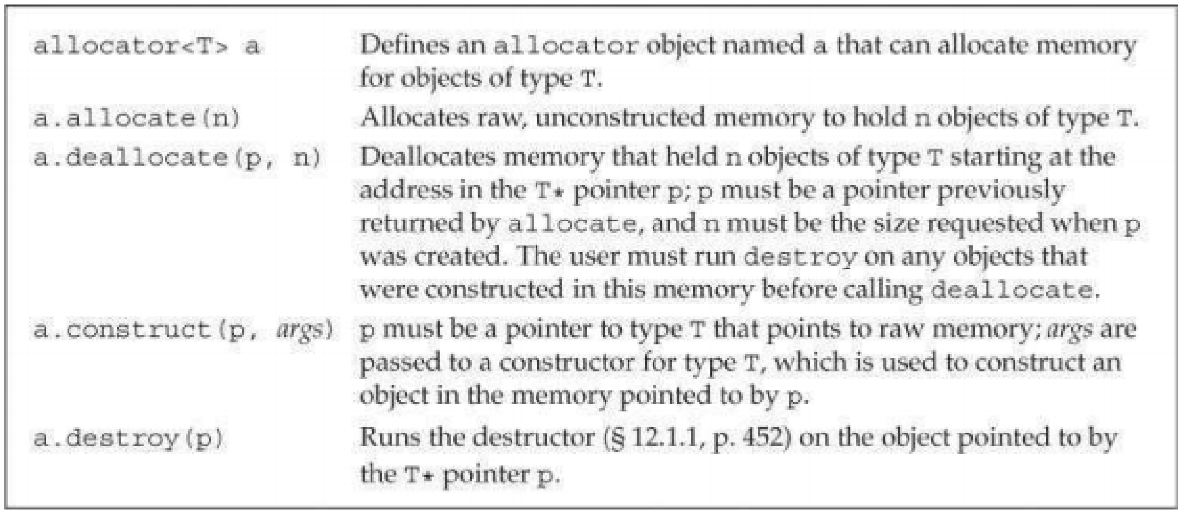
Combining initialization with allocation is usually what we want when we allocate a single object. In that case, we almost certainly know the value the object should have.

When we allocate a block of memory, we often plan to construct objects in that memory as needed. In this case, we'd like to decouple memory allocation from object construction. By doing so, now we can first allocate memory and then construct objects when needed.

The allocator Class

The library allocator class, which is defined in the memory header, lets us separate allocation from construction. It provides type-aware allocation of raw, unconstructed memory. The following table outlines the operations that allocator supports.

Table 12.7. Standard allocator Class and Customized Algorithms



`allocator<T> a`	Defines an `allocator` object named `a` that can allocate memory for objects of type `T`.
`a.allocate(n)`	Allocates raw, unconstructed memory to hold `n` objects of type `T`.
`a.deallocate(p, n)`	Deallocates memory that held `n` objects of type `T` starting at the address in the `T*` pointer `p`; `p` must be a pointer previously returned by `allocate`, and `n` must be the size requested when `p` was created. The user must run `destroy` on any objects that were constructed in this memory before calling `deallocate`.
`a.construct(p, `*args*`)`	`p` must be a pointer to type `T` that points to raw memory; *args* are passed to a constructor for type `T`, which is used to construct an object in the memory pointed to by `p`.
`a.destroy(p)`	Runs the destructor (§ 12.1.1, p. 452) on the object pointed to by the `T*` pointer `p`.

When an allocator object allocates memory, it allocates memory that is appropriately sized and aligned to hold objects of the given type:

```
allocator<string> alloc; //object that can allocate strings
auto const p = alloc.allocate(n); //allocate n unconstructed strings
```

This call to `allocate` allocates memory for n `strings`.

allocators Allocate Unconstructed Memory

The memory an allocator allocates is unconstructed. We use this memory by constructing objects in that memory.

In the new library, the `construct` member takes a pointer and zero or more additional arguments; it constructs an element at the given location.

```
auto q = p; //q will point to one past the last constructed element
alloc.construct(q++);
alloc.construct(q++, 10, 'c');
alloc.construct(q++, "hi");
```

It is an error to use raw memory in which an object has not bee constructed:

```
cout << *p << endl; //ok: uses the string output operator
cout << *q << endl; //disaster: q points to unconstructed memory
```

Warning: We must construct objects in order to use memory returned by allocate. Using unconstructed memory in other ways is undefined.

When we're finished using the objects, we must destroy the elements we constructed, which we do by calling destroy on each constructed element.

The `destroy` function takes a pointer and runs the destructor on the pointed-to object:

```
while(q != p)
  alloc.destroy(--q);
```

Warning: We may destroy only elements that are actually constructed.

Once the elements have been destroyed, we can either reuse the memory to hold other strings ore return the memory to the system. We free the memory by calling deallocate:

```
alloc.deallocate(p, n);
```

The pointer we pass to deallocate cannot be null; it must point to memory allocated by allocate.