Dynamic Memory, Smart Pointers and Special features in C++

Picked from Lippman 5th edition

Smart Pointers

- Dynamic memory managed thorugh `new` which allocates an object in dynamic memory and returns a pointer, and `delete` which destroys the object and frees the memory.
- The library defines three kinds of smart pointers: `shared_ptr`, `unique_ptr` and `weak_ptr`.
- `shared_ptr` is a template, so you need to supply type, like `shared_ptr<string> p1;`.
- Default initialization gices it the nullptr;
- We should use `make_shared` to generate a new `shared_ptr` to an object with given values, like `auto p1 = make_shared<string>(10, '9');`.
- Copying and assigning of shared pointers is valid, it keeps track of the references.
- `*p` and `p->m` work as for pointers.
- Once the reference count goes to zero, `shared_ptr` will automatically delete the object and free the memory.
- If you put `shared_ptr`s in a container and then reorder and do some stuff, make sure to `erase` the useless ones.
- The syntax of `new` looks like `new type(...)` or `new type{...}`.
- We can also use `auto` on lhs while using a single initializer inside paran-theses.
- We can also allocate new `const` objects.
- You can ask `new` to not throw an error by saying `new (nothrow) int;`. Now if `bad_alloc` happens, `new` just returns a `nullptr`.
- Dynamic memory hangs around utile xplicitly deleted through `delete`.
- After deleting a pointer, `*p` is invalid, called a dnagling pointer. Better set it to `nullptr`.
- But other pointers pointing to this memory location still dangle.
- `shared_ptr` provides an `explicit` constructor from pointer types. This allows initialization from pointers generated by `new`.
- Do not mix plain and smart pointers.
- Do not use `get` to initialize another `shared_ptr`. `get` returns the naked pointer, and this will make the two `shared_ptr` independent. If either reference count goes to zero, the object will be destroyed!
- `reset` will essentially acts to make the `shared_ptr` not point to that object any more, if reference count drops to zero, free it. If a new pointer is passed as an argument, point to that instead otherwise `nullptr`.
- Local object destruction is not bypassed even during exceptions, so using

`shared_ptr` will not lead to memory leaks that way.

- By default, `shared_ptr` will call delete when reference count goes to zero. We can supply a function (pointer) which does what we want instead. For eg:

```
void end_connection(connection *p) { dsconnect (*p); }
void f(destination &d) {
    connection c = connect(&d); // need to disconnect on exit from function, so:
    shared_ptr<connection>(&c, end_connection); // this will ensure
}
```

- We can ask `shared_ptr` to give us the reference count using `.use_count()`.
- `unique_ptr` owns the object to which it points, and you cannot copy or assign it. Only initialize it using naked pointer generated by `new`.
- We stll can transfer ownership, using `release` or `reset`.
- `release` will return the naked pointer, `reset` will free the memory.
- `unique_ptr`s that are about to die can be copied (basically they are moved).
- For giving a deleter to `unique_ptr`, you need to make the template differently:

```
unique_ptr<connection, decltype(end_connection)*> p(&c, end_connection);
// destroys the connection while exitng
```

- `weak_ptr` is a weak pointer to objects pointed to by `shared_ptr`s. It does not alter reference count, and so can be dangling.
- To access the object, you cannot dereference `weak_ptr`, you first need to call `lock()`, which will return a `shared_ptr` if the pointed to object exists, else a null `shared_ptr` which will evaluate to `false` under condition check.
- We can still call `.reset()` and `.use_count()` on a `weak_ptr`, and check for expiration explicitly using `.expired()`.