Smart pointers

Tom Latham

THE UNIVERSITY OF WARWICK

Dynamic polymorphism in mpags-cipher

- How can we be taking advantage of dynamic polymorphism in our actual program?
- At the moment we have a switch statement in which we construct the concrete instances and then use them to either encrypt or decrypt
- Would be cleaner and far more reusable to have a 'factory' function that constructs an object instance (the concrete type of which depends on a supplied argument) and returns it to us (using the base type)
- For our test suite it could also be advantageous to store various ciphers in a container to be able to loop through them
- Can we do these things with references?

Limitations of references

- For the factory function there is the problem of object lifetime and reference returns (alluded to when we first introduced references in Day 2)

- But also we cannot use references in containers since they cannot be copied or assigned to – these are prerequisites for any object being stored in a container (in fact references are not objects but provide an alias for an object that already exists)

  - (Since C++11 a `reference_wrapper` class is available to allow storage of references in containers but we will not examine that further here)

- In addition to what we mentioned on the previous slide, we also might like to have one of our polymorphic types as a data member of a class

  - But reference data members can only be initialised in the constructor and then they can never be modified to refer to another object, which is rather limiting

Smart pointers

- The solution to these various problems is to be able to do dynamic allocation and to manage the associated memory and ownership issues using smart pointers

- Smart pointers are objects that point to other objects, in particular to objects that have been dynamically allocated

- What is dynamic allocation? Essentially it means to create objects in an area of memory (called the free store or heap) that gives them a lifetime beyond the scope in which the allocation takes place.

- Prior to C++11 the allocation and management was dealt with manually (using `new` and `delete` and raw pointers – more on these next week)

- Since C++11 we have the smart pointers and their helper functions to handle all of that for us – makes for much safer code!

Smart pointers

- There are three types of smart pointer provided in the C++ standard:
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`

- We will discuss the first two of these (the last is only useful in a handful of situations)

- Smart pointers ensure destruction of the managed object at the appropriate time:

  - The `unique_ptr` is used to enforce sole-ownership of an object. So when the that one `unique_ptr` is destroyed (or assigned a new object to manage) it triggers the destruction of the managed object.

  - The `shared_ptr` is used for shared resources, where there is no one owner. So when the reference count (i.e. the number of `shared_ptr`'s referring to this same managed object) falls to zero the managed object is destroyed.

Dynamic allocation with std::make_unique

We want to make a factory function that can construct our objects with dynamic storage duration and return us sole ownership of those objects. Thus we want to use `std::unique_ptr` and its helper function `std::make_unique`.

The return type is a std::unique_ptr to the base type, in this case Vehicle.

The std::unique_ptr's to the concrete types are implicitly converted on the return.

VehicleFactory.hpp (~/cernbox...ching/2015-16/Day5-Slides) - VIM

```cpp
#ifndef VEHICLEFACTORY_HPP
#define VEHICLEFACTORY_HPP

#include <memory>
#include "Vehicle.hpp"

enum class VehicleTypes { CAR, LORRY, SCOOTER };

std::unique_ptr<Vehicle> vehicleFactory( const VehicleTypes type,
                                         int const nGears );

#endif // VEHICLEFACTORY_HPP
```

VehicleFactory.cpp (~/cernbox/Teaching/2015-16/Day5-Slides) - VIM

```cpp
#include <memory>
#include "VehicleFactory.hpp"
#include "Car.hpp"
#include "Lorry.hpp"
#include "Scooter.hpp"

std::unique_ptr<Vehicle> vehicleFactory( const VehicleTypes type,
                                         int const nGears )
{
  switch (type) {

    case VehicleTypes::CAR :
      return std::make_unique<Car>( nGears );

    case VehicleTypes::LORRY :
      return std::make_unique<Lorry>( nGears );

    case VehicleTypes::SCOOTER :
      return std::make_unique<Scooter>( nGears );
  }
}
```

Notes

We select what particular concrete type we want using an enumeration.

Arguments to the `std::make_unique` function are forwarded on as arguments to the constructor.

We need C++14 to use `std::make_unique` – it was accidentally omitted from the C++11 standard!

Either add `cxx_std_14` to the `target_compile_features` for the MPAGSCipher library or if you have CMake version < 3.8 you can instead add the following lines:

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

before the line

set(CMAKE_CXX_EXTENSIONS OFF)

in your top-level CMakeLists.txt file.

Exercise: a Factory Function

- Write a factory function for your cipher classes
  - You can use the example on the previous slide as a guide
- Then modify your main code to use this new function to create your cipher object depending on the corresponding command line option, e.g.

```
auto aVehicle = vehicleFactory( VehicleType::CAR, nGears );
```

- You can then use the `unique_ptr` with the arrow operator "->" instead of the dot operator ".", e.g.

```
double speed { aVehicle->currentSpeed() };
```

Collections of polymorphic types

- We can also use the `unique_ptr<Base>` as the type to store in collections

- For example, an inventory of vehicles:

```
std::vector<std::unique_ptr<Vehicle>> inventory;
inventory.push_back( vehicleFactory( VehicleTypes::LORRY, nGears ) );
...

for ( const auto& v : inventory ) {
  std::cout << v->numberOfGears() << "\n";
}
```

Exercise: Test all ciphers using a collection

- Modify your testCipher.cpp code to:
- Create a collection of ciphers and fill it with one of each type, using your factory function
- Loop through the collection and check that each is encrypting as expected
- Similarly, write another test to check that all are decrypting as expected
- Again, you can use the example on the previous slide as a guide

ADDENDUM

When to use shared_ptr?

- A shared_ptr is used to express a shared ownership of some resource
- More than one client needs to use the resource and it is not obvious that a particular one of them should be the single owner
- For example, an employee has a company car but there is also an inventory of all company vehicles, another list of those that are under a particular service contract, etc.
- This is a case where the resource should be managed by a shared_ptr
    - (There is potentially a case for some of those clients to hold weak_ptr's, which track the object but don't hold a share of the ownership. But this is probably quite rare.)

Abstract Base Classes

- In some cases you can find that you have a lot of duplicated code in (some of) the concrete classes (not an issue for these cipher classes)
- We could move some of this code up into the pABC but then it wouldn't be purely abstract any more and its job is to simply define a type
- So instead we add a new layer in the inheritance structure, an Abstract Base Class (or ABC), which inherits from the pABC and from which (some of) the concrete classes inherit (instead of from the pABC)
- So we have the pABC that defines the type and the ABC that allows some sharing of implementation
- For example, we could have a `MotorVehicle` ABC from which `Car`, `Lorry`, etc. inherit but `Bicycle` does not

Protected access

- This is the third category of access specifier (`public`, `protected`, `private`)
- It is useful in any scenario where you have a utility function in an ABC that needs to be called by the derived classes
- You don't want it to be `public` but making it `private` hides it from the derived classes
- Making it `protected` means that it can be accessed by the ABC itself and any classes that are derived from it