Programming in C/C++
Exercises set three: advanced advanced class templates

Christiaan Steenkist
Jaime Betancor Valado
Remco Bos

February 9, 2017

Exercise 17, Insertable policies

We made a struct that can technically be any container. Here we the define the functions empty to ensure compilation.

Code listings

Listing 1: insertable.h

```
1  #ifndef INSERTABLE_H
2  #define INSERTABLE_H
3
4  #include <vector>
5  #include <deque>
6  #include <memory>
7
8  template <typename DataType,
9    template <typename, typename =
10   std::allocator<DataType>> class Container>
11 struct Insertable : public Container<DataType>
12 {
13   Insertable(){};
14   Insertable(Container<DataType> container){};
15   Insertable(int val){};
```

```
16    Insertable(Insertable const &other){};
17    Insertable(Insertable &&tmp){};
18  };
19  template <typename DataType,
20    template <typename, typename =
21    std::allocator<DataType>> class Container>
22  std::ostream &operator<<(std::ostream &out,
23    Insertable<DataType, Container> &obj)
24  {
25    return out;
26  }
27
28  #endif
```

Listing 2: main.ih

```
1  #include "insertable.h"
2  #include <iostream>
3
4  using namespace std;
```

Listing 3: main.cc

```
1  #include "main.ih"
2
3  int main()
4  {
5    vector<int> vi {1, 2, 3, 4, 5};
6    typedef Insertable<int, vector> InsertableVector;
7
8    InsertableVector iv;
9    InsertableVector iv2(vi);
10   InsertableVector iv3(4);
11   InsertableVector iv4(iv2);
12
13   cout << iv2 << '\n' <<
14       iv3 << '\n' <<
15       iv4 << '\n';
16
17   iv3.push_back(123);
18   cout << iv3 << '\n';
19 }
```

Exercise 21, `Ptr`, the journey

It is pretty hard to make a pointer that is made for a base class but still appropriately frees a derived class, especially if you don't know what you're doing. Fortunately it worked out and we are hopefully a little wiser because of it.

Design

There were two mountains to overcome while designing this class. The first problem was that the derived type would disappear when it is casted to a base pointer in the constructor. When you no longer know the derived type you can't go back (we tried a lot of things) and so the class could do nothing but call the base destructor. The solution to this was to make the constructor a templated function which apparently is a thing that can be done and so pointers of any type (that can be converted to a base pointer) could be supplied and we could know their type (sucess!).

Now that the derived class was gotten in the constructor we needed to secure it somewhere lest it slip away and disappear at the end of construction. Unsurprisingly sticking it as a data member in the main class doesn't work since you need to know what it is going to be beforehand. We settled on storing the derived pointer in a deleter object (using an empty base class and a templated derived class) but it seems that virtual functions don't really want to work with template classes so it wouldn't call our derived destructor. After learning of type erasure we managed to work something out by changing the deleter to a `std::function` and then pass the derived class information in a lambda function that was made in the constructor.

The final small problem was the fact that this lambda function could not take derived pointers as parameters when we designed the `std::function` as taking a base pointer. To solve this we stuck with the base pointer and used a static cast instead and apparently that works. What is more the derived constructor automatically calls the base constructor so the `Ptr` class does not have to do anything else. Everything went better than expected after 8 hours of mucking about.

Output

```
1 Derived's destructor
2 Base destr.
3 Derived2's destructor
4 Base destr.
5 Derived's destructor
6 Base destr.
```

Code listings

Listing 4: ptr.h

```
1  #ifndef PTR_H
2  #define PTR_H
3
4  #include <functional>
5
6  template <typename Base>
7  class Ptr
8  {
9    Base *d_ptr;
10   std::function<void(Base*)> d_deleter;
11
12   public:
13     // Set pointer and deleter
14     template <typename Derived>
15     Ptr(Derived *ptr)
16     :
17       d_ptr(ptr),
18       d_deleter([](Base *ptr) {
19         delete static_cast<Derived*>(ptr);})
20     {}
21
22     // Call deleter on destruction
23     ~Ptr()
24     {
25       d_deleter(d_ptr);
26     }
27
28     // Call deleter and set the new pointer and
29     //  the new deleter
30     template <typename Derived>
31     void reset(Derived *basePtr)
32     {
33       d_deleter(d_ptr);
34
35       d_ptr = basePtr;
36       d_deleter = [](Base *ptr) {
```

```
37          delete static_cast<Derived*>(ptr);};
38      }
39  };
40
41  #endif
```

Listing 5: main.cc

```
1   #include <iostream>
2   #include "ptr.h"
3
4   using namespace std;
5
6   struct Base
7   {
8     // not a virtual destructor
9     ~Base()
10    {
11      cerr << "Base destr.\n";
12    }
13  };
14
15  struct Derived: public Base
16  {
17    // no overriding
18    ~Derived()
19    {
20      cerr << "Derived's destructor\n";
21    }
22  };
23
24  struct Derived2: public Base
25  {
26    // no overriding
27    ~Derived2()
28    {
29      cerr << "Derived2's destructor\n";
30    }
31  };
32
33  int main()
```

```
34 {
35   {
36     Ptr<Base> p1(new Derived);
37   }
38   Ptr<Base> p2(new Derived2);
39   p2.reset(new Derived);
40 }
```