Smart Pointers in C++

Aleksandra R. Glesaaen
aleksandra@glesaaen.com

September 24th 2014

Literature

[1] Boost c++ library.
`http://www.boost.org`.

[2] C++ reference.
`http://cppreference.com`.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Pearson Education, 1994.

[4] S. Meyers.
More Effective C++: 35 New Ways to Improve Your Programs and Designs.
Pearson Education, 1995.

[5] S. Meyers.
Effective C++: 55 Specific Ways to Improve Your Programs and Designs.
Pearson Education, 2005.

[6] H. Sutter.
Gotw #89 solution: Smart pointers.
http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/.

What are smart pointers?

Objects designed to act like pointers, but provide extended functionality. Example of the proxy pattern [3].

Standard pointer use example:

```
MyClass * ptr = new MyClass();
ptr->Function();
delete ptr;
```

Smart pointers can manipulate three aspects of pointer behaviour:

- Construction
- Dereferencing
- Destruction

Why use smart pointers?

Primarily to avoid memory leaks, which can come from a myriad of different sources;

Memory leak sources

```
MyClass * ptr = new MyClass();
//... (1)
ptr->Function(); //(2)
//...
delete ptr; //(3)
```

1. Might have multiple return paths
2. Might throw an exception
3. One might simply forget to free the resource

Why use smart pointers?

Primarily to avoid memory leaks, which can come from a myriad of different sources;

Memory leak sources

```
MyClass * ptr = new MyClass();
//... (1)
ptr->Function(); //(2)
//...
delete ptr; //(3)
```

1. Might have multiple return paths
2. Might throw an exception
3. One might simply forget to free the resource

Solution: Wrap the resource in a class which frees it on destruction.

Types of smart pointers

Smart pointers where one object singularly owns a resource

Smart pointers where the resource is shared by multiple objects.

Shared smart pointers utilising the *copy-on-write* technique.

Smart pointer implementations

All following smart pointers do "garbage collection", but they differ in how they are assigned:

Assignment

```
SmartPtr<MyClass> p(new MyClass());
SmartPtr<MyClass> q = p;
```

What happens here?

std	boost	Qt
`std::unique_ptr`		
`std::shared_ptr`	`boost::shared_ptr`	`QSharedPointer`
`std::weak_ptr`	`boost::weak_ptr`	`QWeakPointer`
`std::auto_ptr`	`boost::scoped_ptr`	`QScopedPointer`

Example: std::unique_ptr

```
MyClass * CreateObject()
{
  std::unique_ptr<MyClass> new_object(new MyClass());

  //...

  return new_object.release();
```

Release ownership of resource and sets own pointer to `nullptr`

```
};

int main()
{
  std::unique_ptr<MyClass> p1( CreateObject() );
  std::unique_ptr<MyClass> p2 = CreateObject();
```

Compilation error!

```
  //...

  std::unique_ptr<MyClass> q1 = p1;
```

Compilation error!

```
  std::unique_ptr<MyClass> q2 = std::move(p1);
```

OK!

```
  //...

  q1.reset(new MyClass());
```

Deletes previously owned resource and takes ownership of the new one.

```
}
```

Example: std::shared_ptr and std::weak_ptr

```
std::shared_ptr<MyClass> p1(new MyClass());

std::shared_ptr<MyClass> p2 = p1;

{
  std::shared_ptr<MyClass> p3 = p2;

  std::weak_ptr<MyClass> wp = p2;

  if(auto p = wp.lock()) {
    // ...
  }
}

// ...
```

Example: std::shared_ptr and std::weak_ptr

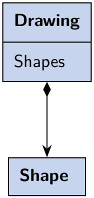
```
std::shared_ptr<MyClass> p1(new MyClass());     use count: 1

std::shared_ptr<MyClass> p2 = p1;               use count: 2

{
  std::shared_ptr<MyClass> p3 = p2;             use count: 3

  std::weak_ptr<MyClass> wp = p2;               use count: 3

  if(auto p = wp.lock()) {
    // ...                                      use count: 4
  }
}

// ...                                          use count: 2
```

Example: std::shared_ptr and std::weak_ptr

```
std::weak_ptr<MyClass> wp;

{
  std::shared_ptr<MyClass> sp =
        std::make_shared<MyClass>();

  wp = sp;

  if(auto wsp = wp.lock()) {
    // ...
  }

  // ...
}

if(wp.expired()) {
  // Managed resource has been deleted
}
```

Choosing the right smart pointer

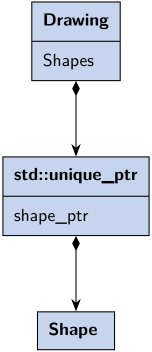
`std::unique_ptr` symbolises owning a resource.



The resource can be shared through references or raw pointers

Choosing the right smart pointer

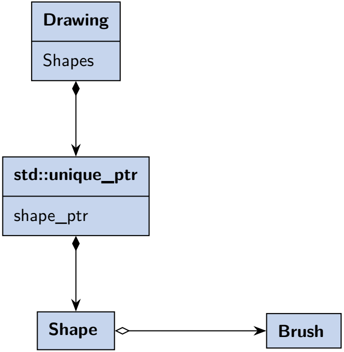
`std::unique_ptr` symbolises owning a resource.



The resource can be shared through references or raw pointers

Choosing the right smart pointer

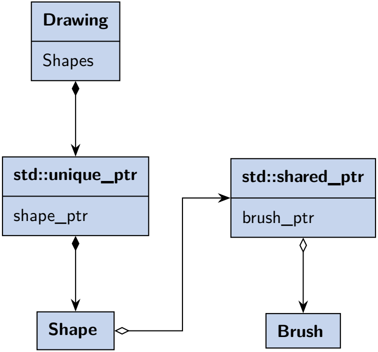
`std::shared_ptr` on the other hand symbolises sharing a resource with other objects.



The resource can still be shared through pointers and references, but also using the `std::shared_ptr` copy constructor and copy assignment operator.

Choosing the right smart pointer

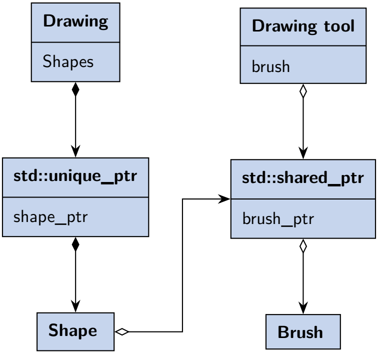
`std::shared_ptr` on the other hand symbolises sharing a resource with other objects.



The resource can still be shared through pointers and references, but also using the `std::shared_ptr` copy constructor and copy assignment operator.

Choosing the right smart pointer

`std::shared_ptr` on the other hand symbolises sharing a resource with other objects.



The resource can still be shared through pointers and references, but also using the `std::shared_ptr` copy constructor and copy assignment operator.

Example: Abstract Factory 1

```
class ShapeFactory
{
  Shape * CreateShape() = 0;
};

class CircleFactory : public ShapeFactory
{
  Shape * CreateShape()
  {
    std::unique_ptr<Shape> shape_ptr(new Circle());

    // ...

    return shape_ptr.release();
  };
};
```

The pointer will be deleted if something happens in between

Hope that whoever takes ownership over the newly created Shape object manages it properly.

Example: Abstract Factory 2

```
class ShapeFactory
{
  std::unique_ptr<Shape> CreateShape() = 0;
};

class CircleFactory : public ShapeFactory
{
  std::unique_ptr<Shape> CreateShape()
  {
    std::unique_ptr<Shape> shape_ptr(new Circle());

    // ...

    return shape_ptr;      ←—— OK, because it is turned into an rvalue.
  };
};
```

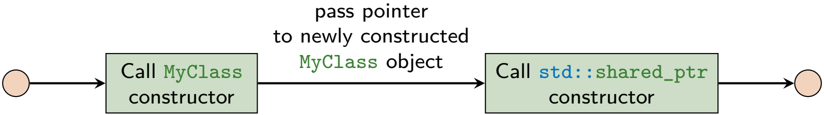
The new owner of the `Shape` object is forced to manage its memory properly.

Problems with explicit `new`'s: #1

Consider creating a `std::shared_ptr` with a `new` statement

Naïve construction

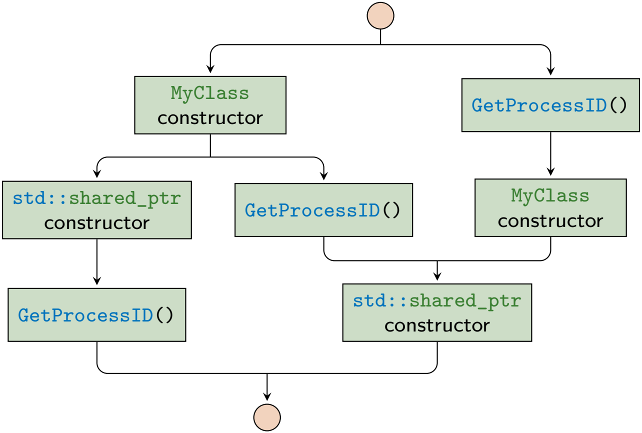
```
std::shared_ptr<MyClass> ptr(new MyClass());
```



The constructors are called separately and the compiler cannot optimise memory location.

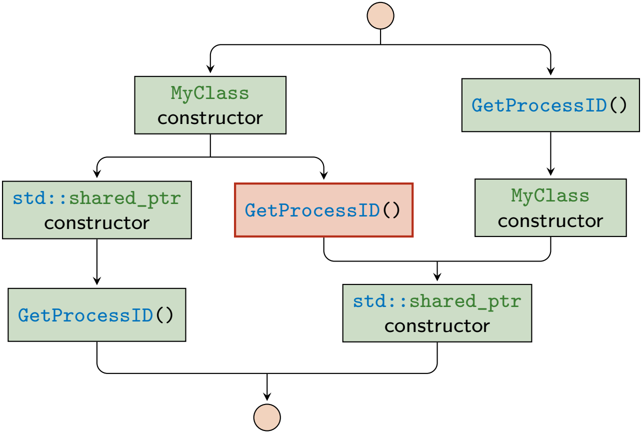
Problems with explicit new's: #2

```
void ProcessObject(std::shared_ptr<MyClass> obj,
                   int process_id);

ProcessObject(std::shared_ptr<MyClass>(new MyClass()),
              GetProcessID());
```



Problems with explicit new's: #2

```
void ProcessObject(std::shared_ptr<MyClass> obj,
                   int process_id);

ProcessObject(std::shared_ptr<MyClass>(new MyClass()),
              GetProcessID());
```



Create using `std::make_unique` and `std::make_shared`

Both these problems can be remedied by using `std::make_shared` and `std::make_unique` (*C++14*).

Replacing the constructor call

```
void ProcessObject(std::shared_ptr<MyClass> obj,
                   int process_id);

ProcessObject(std::make_shared<MyClass>(),
              GetProcessID());
```

Constructor calls cannot be intertwined with the `GetProcessID()` anymore.

Create using `std::make_unique` and `std::make_shared`

Guideline

*Don't use explicit new, delete, and owning * pointers, except in rare cases encapsulated inside the implementation of a low-level data structure.*

Herb Sutter [6]

Match constructors with destructors

Smart pointers have a control block which also keeps track of an allocator and a deleter

std::shared_ptr constructor

```
template <class Type, class Deleter, class Alloc>
std::shared_ptr(Type * p, Deleter d, Alloc a);
```

Custom deleter

```
std::shared_ptr<int> ap(new int[10]);
```

Destructs using `delete`

```
std::shared_ptr<int> ap(new int[10],
  std::default_delete<int[]>());
```

Destructs using `delete[]`

```
std::shared_ptr<int[]> ap(new int[10]);
```

Destructs using `delete[]`

Very important that the deleter doesn't throw.

Passing smart pointers

There are many options for passing smart pointers to functions (and classes).

Passing smart pointers

```
void foo(MyClass *);
void foo(MyClass &);
void foo(std::unique_ptr<MyClass>);
void foo(std::unique_ptr<MyClass> &);
void foo(std::shared_ptr<MyClass>);
void foo(std::shared_ptr<MyClass> &);
```

All of these has a distinct meaning, use them to express yourself.

Smart pointers and polymorphic classes

Using smart pointers and polymorphic classes as template arguments works as expected because one of the smart pointer constructors read:

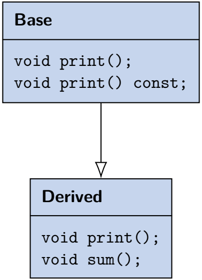
The `std::shared_ptr` constructor

```
template<class T, class U>
std::shared_ptr<T>(const std::shared_ptr<U> &)
```

This costructor can be used to convert between `std::shared_ptr`'s if `U*` is implicitly convertible to `T*`.

Smart pointers and polymorphic classes

Assume we have a class hierarchy:



Where `Derived` overloads the `print()` function but not the `const` variant.

Smart pointers and polymorphic classes

```
std::shared_ptr<Derived> d_ptr =
  std::make_shared<Derived>();

std::shared_ptr<Base> b_ptr = d_ptr;
std::shared_ptr<const Base> b_const_ptr = d_ptr;

std::shared_ptr<Derived> d_err_ptr = b_ptr;      Compilation error
std::shared_ptr<Base> b_err_ptr = b_const_ptr;   Compilation error

b_ptr->print();          Calls Derived::print()
b_const_ptr->print();    Calls Base::print()const

//use_count: 3
```

Smart pointers and polymorphic classes

```
std::shared_ptr<Derived> d_ptr =
  std::make_shared<Derived>();

std::shared_ptr<Base> b_ptr = d_ptr;
std::shared_ptr<const Base> b_const_ptr = d_ptr;

std::shared_ptr<Derived> d_new_ptr =
  std::dynamic_pointer_cast<Derived>(b_ptr); ⟵ OK!

std::shared_ptr<Base> b_new_ptr =
  std::const_pointer_cast<Base>(b_const_ptr); ⟵ OK!

d_new_ptr->sum(); ⟵ Calls Derived::sum()
b_new_ptr->print(); ⟵ Calls Derived::print()

//use_count: 5
```

Smart pointers and the STL

- Smart pointers can be stored in the STL containers.
- However, not all algorithms work with the resulting containers.
  - E.g. `std::unique_ptr` is MoveConstructible and MoveAssignable
  - But not CopyConstructable or CopyAssignable

Thus if an algorithm requires CopyConstructability and a `std::unique_ptr` is given, it should fail to compile.

`std::auto_ptr` on the other hand is a bit more unreliable.

The boost pointer container library

Library inteded to provide a STL-like library for single ownership pointers.

Advantages

- Simplifies the container-of-pointer syntax.
- Notational convenience
  - Dereferencing an iterator returns a dereferenced pointer
- Introduces "Clonability" to do deep copies.
- Faster and has a small memory overhead.

Disadvantages

- Not very compatible with the algorithm library
- Not as flexible as a container of smart pointers

Summary

- Use smart pointers to manage dynamic resources so that they are freed when they aren't used anymore.
  - Use `std::unique_ptr` to signal singular ownership
  - Use `std::shared_ptr` to signal shared ownership
  - Use `std::weak_ptr` to signal uncommitted shared ownership
- Avoid using explicit `new` and `delete` statements, and explicit ownership of raw pointers.