TDDD38 - Extra lecture

Pointers

Eric Elfving

Department of Computer and Information Science
Linköping University

LINKÖPING UNIVERSITY

Spot the error

```
void fun() { /* ... */ }

class X { /* ... */ };

void bar() {
    X * x = new X;
    fun();
    delete x;
}
```

Possible memory leak!

(if `fun` throws)

use unique_ptr

```
void fun() { /* ... */ }

class X { /* ... */ };

void bar() {
    auto x = make_unique<X>( /* args forwarded to X's constructor */ );
    fun();
    /* x is automatically destroyed */
}
```

Helper function `make_unique` was added in C++14.
C++11:

```
unique_ptr<X> x { new X };
```

`unique_ptr` is a class made for single ownership with pointer semantics[1]. Cannot be copied but can be moved. Has the following intresting member functions:

- `get` Returns the stored raw pointer
- `release` Releases the resource
- `reset(p)` Releases the resource and stores p instead.

[1] Has overloads for standard pointer operations

shared_ptr

`shared_ptr` is a reference counted object where several can share the same stored pointer.

```
{
    auto x = make_shared<X>();
    {
        auto x2 = x;
    }
}
```

`make_shared` is available in C++11...
`shared_ptr` has an internal `use_count` - number of `shared_ptrs` sharing this resource.

weak_ptr

A `weak_ptr` can share a resource with a `shared_ptr` without increasing the use count - the resource will still be released when all `shared_ptrs` using it have been destroyed. Has two intresting members:

- `expired` - is the resource available.
- `lock` - return a `shared_ptr` if not expired.

Custom deleter

Both `unique_ptr` and `shared_ptr` uses standard `delete` to release the resource. What happens if we want something else?

```
void deleter(int * ptr) {
    cout << "Deletes " << *ptr;
    delete ptr;
}
...
shared_ptr<int> p { new int{234}, deleter };
```

Note: `unique_ptr` takes the type of the deleter as a template parameter as well.

Watch Stephan T. Lavavejs introduction for a great presentation. It's a bit old (2010), but has great content. The smart pointers starts at approx 15 min.

`https://channel9.msdn.com/Series/`

`C9-Lectures-Stephan-T-Lavavej-Standard-Template-Library-STL-/`

`C9-Lectures-Stephan-T-Lavavej-Standard-Template-Library-STL-3-of-n`

His entire series is highly recommended (even though it's old).

Prefer usage of smart pointers instead of raw ponters! Sadly, sometimes we might have restrictions so that this can't be done... The GSL[2] defines a wrapper to mark your raw pointers as owning its resouce to make resourse handling and sharing easier.

[2]Guideline Support Library, part of the https://github.com/isocpp/CppCoreGuidelines/

```
template <typename T>
using owner = T;
...
owner<int *> p = new int{123};
```

It won't change anything except your statical code which means no overhead.

By using owner or some real container / smart pointer everywhere where ownership is implied, usage of normal pointers are ok! - A normal pointer is a non-owning reference to an object someone else manages.

```
owner<X*> compute(args)    // It is now clear that ownership is transferred
{
    owner<X*> res = new X{};
    // ...
    return res;
}
```

www.liu.se

LINKÖPING UNIVERSITY