Lightning Talks

ACCU 2012

Jonathan Wakely – Smarter Than The Average Pointer

Seb Rose – Transformation Priority Premise

Aaron Ridout – Advocating References

Mike Long - Metricide

Anna-Jayne Metcalfe – Don't Let The Big Ball of Mud Sneak Up on You

Roger Orr – Code Critiques

Pete Goodliffe – The C++ Cathedral & The Bizarre

Peter Sommerlad – C++'s "hello, world" Considered Harmful

Smarter than the Average Pointer

Jonathan Wakely

ACCU London

23 Nov 2011

“The future is already here – it's just not very evenly distributed” -- William Gibson

The new C++11 standard includes `std::shared_ptr`, `std::make_shared` and `std::ref`

All came from Boost and versions of them can be found there for C++03 compilers.

```
template<typename T, typename... Args>
   shared_ptr<T>
   make_shared(Args&&...);
```

Calling

```
make_shared<X>(args)
```

is equivalent to

```
shared_ptr<X>(new X(args))
```

but ***better***

What's wrong with this code?

```
void f(shared_ptr<A>, shared_ptr<B>);
...
f(new A, new B);
```

What's wrong with this code?

```
void f(shared_ptr<A>, shared_ptr<B>);
...
f(new A, new B);
```

The order of evaluation is unspecified

If the second constructor throws the first object could be leaked

c.f. GOTW #56: Exception-Safe Function Calls
http://www.gotw.ca/gotw/056.htm

What's wrong with this code?

```
void f(shared_ptr<A>, shared_ptr<B>);
...
f(shared_ptr<A>(new A), shared_ptr<B>(new B));
```

This still has exactly the same problems.

What's wrong with this code?

```
void f(shared_ptr<A>, shared_ptr<B>);
...
f(shared_ptr<A>(new A), shared_ptr<B>(new B));
```

This still has exactly the same problems.

But this solves the problem:

```
f(make_shared<A>(), make_shared<B>());
```

What's wrong with this code?

```
Base* p = new Derived;
shared_ptr<Base> sp(p);
```

What's wrong with this code?

```
Base* p = new Derived;
shared_ptr<Base> sp(p);
```

Maybe nothing, but it depends if it's safe to delete a `Derived` through a pointer to `Base`.

The `shared_ptr` doesn't know the dynamic type of the object it manages.

This is OK:

```
shared_ptr<Base> sp(new Derived);
```

Now the `shared_ptr` knows the dynamic type of the object and will delete it correctly.

But this avoids the problem completely:

```
shared_ptr<Base> sp = make_shared<Derived>();
```

```
shared_ptr<A> sp(new A)
```

There are two memory allocations here.

An `A` is allocated on the heap.

The `shared_ptr`'s reference counting information must also be allocated on the heap.

```
shared_ptr<A> sp = make_shared<A>()
```

There are ??? memory allocations here.

```
shared_ptr<A> sp = make_shared<A>()
```

There is **only one** memory allocation here.

An `A` and the `shared_ptr`'s reference counting information can be allocated as a single block.

The object is allocated right next to its associated reference count.

```
shared_ptr<A>(new A(x, y, z)
```

```
make_shared<A>(x, y, z)
```

Using `make_shared` means less typing too!

So it's:

- Safer

So it's:

- Safer

Good Thing™

So it's:

- Safer
- Faster

Good Thing™

So it's:

- Safer — ***Good Thing™***
- Faster — ***Good Thing™***

So it's:

- Safer — ***Good Thing™***
- Faster — ***Good Thing™***
- Helps fight RSI

So it's:

- Safer ***Good Thing™***
- Faster ***Good Thing™***
- Helps fight RSI ***Good Thing™***

```
#include <memory>
#include <iostream>

struct Base { };

struct Derived : Base {
   Derived(int) { }
   ~Derived() { std::cout << “Bye” << std::endl; }
};

std::shared_ptr<Base> create(int i) {
   return std::make_shared<Derived>(i);
}

int main() {
   std::shared_ptr<Base> p = create(5);
}
```

`std::make_shared` supports *perfect forwarding*

`boost::make_shared` can't for C++03 compilers, takes its arguments by reference-to-const

To pass arguments to a constructor as reference-to-non-const you can use `boost::ref`

```
#include <boost/make_shared.hpp>
#include <boost/ref.hpp>      // <utility> for std::ref
#include <iostream>

struct Base { };

struct Derived : Base {
   Derived(int&) { }
   ~Derived() { std::cout << “Bye” << std::endl; }
};

boost::shared_ptr<Base> create(int& i) {
   return boost::make_shared<Derived>(boost::ref(i));
}

int main() {
   int i = 5;
   boost::shared_ptr<Base> p = create(i);
}
```

```
std::allocate_shared<X>(alloc, args)
```

is like

```
std::make_shared<X>(args)
```

but uses the supplied allocator to obtain the required memory

Go forth and `make_shared` !