What's Different: At a Glance

auto type deduction

T* → **shared_ptr**<T>
new → **make_shared**

Then

```
circle* p = new circle( 42 );
vector<shape*> vw = load_shapes();
for( vector<circle*>::iterator i = vw.begin(); i != vw.end(); ++i ) {
   if( *i && **i == *p )
      cout << **i << " is a match\n";
}
for( vector<circle*>::iterator i = vw.begin();
      i != vw.end(); ++i ) {
   delete *i;
}
delete p;
```

not exception-safe
missing try/catch,
__try/__finally

Now

```
auto p = make_shared<circle>( 42 );
vector<shared_ptr<shape>> vw = load_shapes();
for_each( begin(vw), end(vw), [&]( shared_ptr<circle>& s ) {
   if( s && *s == *p )
      cout << *s << " is a match\n";
} );
```

no need for "delete"
automatic lifetime management
exception-safe

for/while/do →
std:: **algorithms**
[&] **lambda** functions

Heap Lifetime: Standard Smart Pointers

```
class gadget;

class widget {
private:
  shared_ptr<gadget> g;
};

class gadget {
private:
  weak_ptr<widget> w;
};
```

shared ownership

still keeps gadget alive w/auto lifetime mgmt

no leak, exception safe

use weak_ptr to break reference-count cycles

```
class node {
  vector<unique_ptr<node>> children;
  node* parent;
  :::
public:
  node( node* parent_)
    : parent(parent_)
  {
    children.push_back( new node(…) );
    :::
  }
};
```

unique ownership

node **owns** its children

no leak, exception safe

node **observes** its parent

plain “new” should immediately initialize another object that owns it, usually a unique_ptr

If/when performance optimization is needed, consider ***well-encapsulated*** uses of owning *’s and delete (e.g., hidden inside objects).

Example: writing your own low-level data structure.