The University Of Sheffield.

ReMoDeL

Translation Strategy for the C++ Programming Language

White Paper

Version: 0.5

Date: 09 September 2009

Anthony J H Simons
Department of Computer Science
University of Sheffield

Implementing the Common Semantic Model

The CSM deliberately adopts the more dynamic view of objects, which are freely created and deleted, with features like polymorphic aliasing, safe type downcasting and garbage collection. Each of the target languages must support this model. In certain target languages, such as Java, these features are already built-in to the native programming model. In other languages, this requires the provision of a base library of some sophistication. An example of this in C++ is given below.

1 Dynamic objects with memory management

To implement dynamic objects with uniform reference semantics in C++ requires the use of object allocation on the heap, with some form of memory management to ensure that unused objects are deleted afterwards. One possibility is to include a garbage collector for C++ (several implementations exist). In this translation model, all field and variable types become C++ pointer types:

```
SomeClass* ptr = new SomeClass();
```

and the garbage collector performs occasional sweeps through heap memory to determine whether variables are still reachable from current stack memory. This is a good option if a garbage collector with predictable performance characteristics is available. Mark-and-sweep collectors may execute at unpredictable intervals and may suspend normal execution while collecting.

Alternatively, a reference-counted memory management policy is possible, through the creation of smart pointers. In this translation model, all field and variable types become smart pointer types, and objects are assigned immediately to smart pointers upon creation, so that their lifetime may be managed:

```
Ref<SomeClass> ptr = new SomeClass();
```

The smart pointer type *Ref*<*T*> is a template class, parameterised by the type *T* of the object stored in it. In this model, all objects contain a reference count, which is incremented or decremented as that object is assigned to, or forgotten by, a smart pointer. When the reference count reaches zero, the object is deleted. This model is good for incremental garbage collection. The only disadvantage is the need to handle circular referencing explicitly, to ensure that mutually-supporting objects are eventually deleted.

2 Reference counted object Body base class

The following is an implementation of the *Handle-Body* pattern for smart reference counting. A root *Body* class may be provided as the (secret) root class of a reference-counted library, in the header file *Body.h:*

```
class Body {
  private:
    int refcount;         // reference count
  public:
    Body();               // sets refcount = 0
    virtual ~Body();      // makes Body polymorphic
    void acquire();       // increments refcount
```

```
    int release();        // decrements refcount
};

inline Body::Body() : refcount(0) {
}

inline Body::~Body() {
}

inline void Body::acquire() {
   ++refcount;
}

inline int Body::release() {
   return --refcount;
}
```

Body defines the secret *refcount* and provides public methods *acquire()* and *release()* to allow smart pointer types to modify the reference count. Making these public also allows special code to handle circular referencing. Only *release()* need return a value, so that smart pointers can detect when the reference count has reached zero. Notice also that all methods are declared *inline* for efficiency – the code for *acquire()* and *release()* will expand inline inside the smart pointer code that uses these. All other classes (e.g. the system root class *Object*) now inherit from *Body*. The default constructor ensures that the reference count is properly initialised in every object; and the *virtual* destructor *~Body()* is required, to indicate that *Body* and its descendants are polymorphic, subject to dynamic binding.

3 Smart pointer Handle base class

A smart pointer class *Ref<T>* may be provided to manage objects of type *T*, and count references to them. Since this is a template class in C++, there is a risk of code-bloat when multiple versions of the compiled code are generated, one copy for each type that replaces the parameter *T*. To mitigate this, it is possible to move most of the functionality of the smart pointer to a non-template base pointer class, for which the compiler will only generate one copy of the object code:

```
class Handle {
  private:
    Body* body;               // store basic pointer
  protected:
    Handle();                 // set body to null;
    Handle(Body*);            // acquire basic pointer
    ~Handle();                // release body, maybe delete
    void assign(Body*);       // assign from basic pointer
    Body* access() const;     // null-checked body access
    Body* pointer() const;    // fast unchecked body access
};

inline Handle::Handle() : body(0) {}

inline Body* Handle::pointer() const {
   return body;
}
```

The base *Handle* class maintains a private pointer to the *Body*, and provides methods to construct, and to assign, from a primitive *Body* pointer. Each of these increments the reference count of a (non-null) object. The destructor for *Handle* decrements the reference count of a (non-null) object, and deletes the object on a count of zero.

The very short methods are provided *inline* in the *Handle.h* file, while more complicated methods, which perform reference counting or error checking, should be provided in the *Handle.cpp* file, which will be compiled once, and for which the object code will be included once in the final system:

```
Handle::Handle(Body* ptr) : body(ptr) {
    if (body) body->acquire();
}

Handle::~Handle() {
    if (body && body->release() == 0)
          delete body;
}

void Handle::assign(Body* ptr) {
    if (ptr) ptr->acquire();
    if (body && body->release() == 0)
          delete body;
    body = ptr;
}

Body* Handle::access() const {
    if (body) return body;
    else throw NullPointerException();
}
```

All reference counting is expanded inline inside these *Handle* methods, giving the fastest possible coding for separately-compiled functions.

4 Typed smart pointer Ref<T> derived class

The typed smart pointer *Ref<T>* inherits from the base *Handle* class and attaches type information to its inherited methods. All of the methods of *Ref<T>* are inlined, since they merely delegate by super-invocation to the methods inherited from *Handle*. The whole definition of the template class is in the header file, *Ref.h*:

```
template <class T> class Ref : protected Handle {
  public:
    Ref();                 // set body to null;
    Ref(T*);               // acquire body from pointer
    Ref(Ref<T>&);          // acquire body from reference
    ~Ref();                // release body, maybe delete
    T& operator=(T*);           // assign body from pointer
    T& operator=(Ref<T>&);      // assign body from reference
    T* operator->();       // access object field
  ...                      // more to follow below
};
```

```
template <class T>
inline Ref<T>::Ref() : Handle() {}

template <class T>
inline Ref<T>::Ref(T* ptr) : Handle(ptr) {}

template <class T>
inline Ref<T>::Ref(Ref<T>& ref) : Handle(ref.pointer()) {}

template <class T>
inline Ref<T>::~Ref() {}
```

These constructors and the destructor delegate to inherited methods. For each type *T* that inherits from *Body*, the argument *T** passed back in construction is converted to a *Body** pointer for free, by type upcasting. Assignment is only slightly more complex:

```
template <class T>
inline Ref<T>& Ref<T>::operator=(T* ptr) {
    assign(ptr);
    return *this;
}

template <class T>
Ref<T>& Ref<T>::operator=(Ref<T>& ref) {
    assign(ref.pointer());
    return *this;
}
```

These both return the typed smart pointer as a result (standard in C++). Notice how versions of construction and assignment must be provided both for the raw pointer type *T** and for the smart pointer type *const Ref<T>&,* since construction and assignment may be from raw or smart pointer arguments. Notice how the smart pointer argument-handling functions all pass back the raw pointer to the inherited method, using the unchecked *ref.pointer()* access method. *Handle* need only deal with the raw pointer cases, as a result. Finally, we access objects from smart pointers using a checked dereferencing operator:

```
template <class T>
inline T* Ref<T>::operator->() {
    static_cast<T*>(access());
}
```

This invokes the checked *access()* method to obtain the raw pointer, then performs a static type downcast to convert the *Body** to the *T** type (which is guaranteed to be correct). This arrow operator now allows us to invoke the stored object's methods, in the same style as if through a primitive pointer in C++:

```
Ref<SomeClass> ref = new SomeClass();
ref->someMethod();
```

However, any attempt to access a null pointer will result in a *NullPointerException* (see definition of *access()* above, which checks for null).

5 Smart type upcasting from Ref<S> to Ref<T>

One of the hardest things to obtain is safe polymorphic aliasing among smart pointers. The smart pointer as described so far will allow some degree of polymorphic aliasing, for example, assuming *Derived* is a subclass of *Base*, we already have safe automatic upcasting in some situations, but not in others:

```
Ref<Base> ref = new Base();
Ref<Derived> sub = new Derived();
ref = new Derived();      // aliasing OK
ref = sub;                // type error
```

The reason *ref* can receive a subtype object pointer is because C++ can automatically convert between primitive pointer types, from *Derived** to *Base**. However, automatic conversion between two smart pointers *Ref<Derived>* and *Ref<Base>* is still blocked, since these are non-scalar types. Conversion requires a constructor having a signature of the form: *Ref<T>::Ref(Ref<S>&),* for some other type *S* which is a subtype of *T*.

We can obtain this free upcasting, from *Ref<S>* to *Ref<T>*, if the smart pointer class *Ref<T>* provides the following additional operations:

```
template <class T> class Ref : protected Handle {
    ...
  public:
    operator T*();                    // access T* pointer
    template <class S> Ref(Ref<S>&); // convert from Ref<S>
};

template <class T>
inline T* Ref<T>::operator T* () {
   return static_cast<T*>(pointer());
}

template <class T> template <class S>
inline Ref<T>::Ref(Ref<S>& ref) :
   Handle(static_cast<S*>(ref)) {}
```

The first of these converts a smart pointer back into a basic pointer. This is essentially the public typed version of *pointer()*. We use this inside the second operation, a special conversion constructor, when we ask to cast *Ref<S>* back to *S**, which is then a suitable subtype of the *T** body. This now allows a *Ref<S>* to be converted to a *Ref<T>*, provided that *S* is a subtype of *T*.

6 Smart type downcasting from Ref<T> to Ref<S>

Ideally, we want type upcasting to happen for free; whereas type downcasting should be dynamically checked to allow safe downcasts and report an error in other cases. The best approximation you can manage in C++ is free binding among identical types, but dynamic checking between all pairs of different types. Unfortunately, this checks upcasts as well as downcasts; but has the advantage of built-in checking of all type casts of any kind.

To implement checked downcasts, we first add an extra checking-method to the base pointer class *Handle*:

```
class Handle {
    // other members as before
  protected:
     Body* check(Body*, Body*);
};
```

This has the following implementation in the *Handle.cpp* file:

```
Body* Handle::check(Body* ptr, Body* org) {
    if (!ptr && org)
          throw TypeCastException();
    return ptr;
}
```

The notion is that the first argument, *ptr*, is the result of downcasting a pointer-type, which should be *null* if the cast failed, and that the second argument *org* is the original pointer, before downcasting. The exception is only raised if the original pointer was non-*null*, but the failed conversion resulted in a *null* pointer. Conversion of an original *null* pointer value will therefore always succeed.

Now, the type conversion constructor is implemented differently in *Ref<T>,* using the *dynamic_cast* operator to attempt to convert from one pointer type to the other:

```
template <class T> template <class S>
inline Ref<T>::Ref(Ref<S>& ref) : Handle(
    check(dynamic_cast<T*>(static_cast<S*>(ref)),
          static_cast<S*>(ref))) {}
```

Whereas before, the relationship between *Ref<T>* and *Ref<S>* was checked statically at compile-time, this time, the check is deferred until run time. The *dynamic_cast* operator either returns a valid pointer (if the typecast succeeds) or null if it fails. The *Handle* special constructor therefore reports an error in this case.

A similar type-converting assignment operator may also be provided:

```
template <class T> template <class S>
inline Ref<T>& Ref<T>::operator=(Ref<S>& ref) {
    assign(check(dynamic_cast<T*>(static_cast<S*>(ref)),
          static_cast<S*>(ref)));
    return *this;
}
```

This is not strictly required, since assigning a *Ref<S>* value to a *Ref<T>* pointer will automatically invoke the *Ref<S>* to *Ref<T>* conversion constructor. However, providing the assignment operator is slightly more efficient, since it saves constructing a temporary *Ref<T>* object during the conversion.

It is also possible to implement a version of this without the extra template for *S*. In this case, *Ref<T>* supplies typed operations for *Ref<T>&* arguments, and "untyped" operations for *Handle&* arguments, which are subject to a *dynamic_cast* check, as above.