AIPS++ NOTE 173: Initialization of Static and Global Variabless in C++

J. E. Horstkotte 02 March 1995

The following was initially written as advice regarding a very nasty static data member initialization problem for some classes in the aips/Measures module. I have tried to make it more general.

Under certain circumstances, one must be very careful about how one initializes global variables, class static data members, and file (static) variables.

If there is an initialization hierarchy of objects with global linkage (typically static data members of classes and global variables) and objects with file linkage (typically static objects), spread across several compilation units (.cc files), they are not by default guaranteed to be initilized properly, in the proper order. One must take specific steps to ensure that they are initialized in the proper order. An example: If global variable "b" uses the value of global variable "a" when it is initialized, then one must ensure that global variable "a" is initialized before global variable "b".

I have spent alot of time trying to understand how initialization works, reading, talking to Darrell Schiebel, and experimenting. I believe that I now do understand it, how to do it, and what the constraints are. I will attempt to pass this on, though it is complex, and I may not be able to communicate all of the necessary details and subtleties.

The reading I did was in:

1. Meyers, "Effective C++", Item 47, p. 178.

2. Stroustroup, "The Annotated C++ Reference Manual", (ARM),
   Section 3.4, p. 19,
   Section 3.3, p. 17,
   Section 3.5, p. 21,
   Section 8.4, p. 149,
   Section 9.4, p. 179,
   Section 6.7, p. 91.

The set of rules regarding initilization is:

1. There is no guaranteed order of initilization between separate compilation units (i.e. .cc files).

2. Within a compilation unit, initilization is done in order.

To get around 1) above, one uses the technique, discussed in Meyers Item 47 and in Stroustroup Section 3.4.

Let's use the class QC, defined in aips/implement/Measures/QC.h, as an example:

```
class QC {
friend class QC_init;
public:
```

```
    static Quantum<Double> c;           // vel of light
    static Quantum<Double> G;           // Gravitational constant
    static Quantum<Double> h;           // Planck
    static Quantum<Double> R;           // Gas constant
    static Quantum<Double> NA;          // Avogadro
    static Quantum<Double> e;           // electron charge
    static Quantum<Double> mp;          // proton mass
    static Quantum<Double> mp_me;       // mp/me
    static Quantum<Double> mu0;         // permeability vacuum
    static Quantum<Double> epsilon0;    // permittivity vacuum
    static Quantum<Double> k;           // Boltzmann
    static Quantum<Double> F;           // Faraday
    static Quantum<Double> me;          // mass electron
    static Quantum<Double> re;          // radius electron
    static Quantum<Double> a0;          // Bohr's radius
    static Quantum<Double> R0;          // Solar radius
    static Quantum<Double> k2;          // IAU Gaussian grav. const **2
private:
// The following dummy function is used, in conjunction with the QC_init
// class to instantiate statics (see ARM 3.4)
    static void init();
};

// The following class and static object are used to make sure the above
// is instantiated.

class QC_init {
  public:
    QC_init();
    ~QC_init();
  private:
    static uShort count;
};

static QC_init qc_init;


The actual initialzation code, from aips/implement/Measures/QC.cc, is:


Quantum<Double> QC::c(C::c,"m/s");
Quantum<Double> QC::G(C::Gravity,"N.m2/kg2");
Quantum<Double> QC::h(C::Planck,"J.s");
Quantum<Double> QC::R(C::GasConst,"J/K/mol");
Quantum<Double> QC::NA(C::Avogadro,"mol-1");
Quantum<Double> QC::e(C::qe,"C");
Quantum<Double> QC::mp(C::mp,"kg");
Quantum<Double> QC::mp_me(C::mp_me,"");
Quantum<Double> QC::mu0(C::mu0,"H/m");
Quantum<Double> QC::epsilon0(C::epsilon0,"F/m");
Quantum<Double> QC::k(C::Boltzmann,"J/K");
Quantum<Double> QC::F(C::Faraday,"C/mol");
Quantum<Double> QC::me(C::me,"kg");
Quantum<Double> QC::re(C::re,"m");
Quantum<Double> QC::a0(C::a0,"m");
Quantum<Double> QC::R0(C::R0,"m");
Quantum<Double> QC::k2(IAU_k*IAU_k,"AU3/d2/S0");

uShort QC_init::count;

QC_init::QC_init() {
    if (count++ == 0) {
        QC::init();             // make sure statics initialized
    }
}
```

```
QC_init::~QC_init() {
    if (--count == 0) {
    }
}

void QC::init() {
}
```

Suppose some other compilation unit, OtherUnit.h, required that QC be initilized before program execution, say by the definition of a file static object:

```
a)      static Quantum<Double> localc = QC::c;
```

The inclusion of QC.h in OtherUnit.h, before the above definition a), puts the following statement, from QC.h, ahead of the a):

```
b)      static QC_init qc_init;
```

Thus, in the compilation unit where "localc" is used, and needs "QC::c" to be initialzed, "qc_init" is ahead of "localc", so is initilized first, thus by construction guaranteeing that "QC" is initialized.

The problem is this:

The static data members of QC must be defined once, outside of the definition of QC. This was done, an example being:

```
c)      Quantum<Double> QC::c(C::c,"m/s");
```

However, this constructor will only be run when the compilation unit QC.o is initialized. This may very well be after the compilation unit requiring "localc" is initialized, in turn initializing "qc_init". Thus, when "qc_init" is initialized, "QC::c" has not been initialized. Further, it will be initialzed later, by c), when QC.o is initialized. Thus, any initilization done in qc_init will be overwritten by c).

Thus one needs that:

1. the constructor in c) do nothing (recursively).

2. all initialization be done in qc_init (class QC_init).

One way to do this is to have all static data members be pointers. The default constructor for a pointer behaves in the proper way, and one can set up the "init" class, or the constructor for the class itself, to do the appropriate "new", plus other initialization, to initialize things properly. Similarly, the destructor for the "init" class or the class itself can do the proper cleanup, including calling "delete".

These data members can be used directly as pointers (internally to the class?), and the objects they point to can be made available through public accessor member functions which dereference them. However, note that static member functions cannot be inlined, so there will be a cost: either they are out of line, or they are not static and must be associated with an object.

Simple types, with default constructors, also have the proper behavior (actually, they don't have constructors).

In general, for *every* class with static data members, and probably for *every* static or global object which is, or can be, used to initialize some other static or global object, one must make an "init" class and a

```
static instance of this class, in the include file, to guarantee initialization
order.  And 1) above must hold:  the constructor used in the definition
of the global object must do nothing, recursively.
```