Item 12: Prefer initialization to assignment in constructors. [Item E12, P1]

Consider a template for generating classes that allow a name to be associated with a pointer to an object of some type T: [Item E12, P2]

```cpp
template<class T>
class NamedPtr {
public:
    NamedPtr(const string& initName, T *initPtr);
...
private:
    string name;
    T *ptr;
};
```

(In light of the aliasing that can arise during the assignment and copy construction of objects with pointer members (see Item 11), you might wish to consider whether NamedPtr should implement these functions. Hint: it should (see Item 27).) [Item E12, P3]

When you write the NamedPtr constructor, you have to transfer the values of the parameters to the corresponding data members. There are two ways to do this. The first is to use the member initialization list: [Item E12, P4]

```cpp
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr )
    : name(initName), ptr(initPtr) {}
```

The second is to make assignments in the constructor body: [Item E12, P5]

```cpp
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr) {
    name = initName;
    ptr = initPtr;
}
```

There are important differences between these two approaches. [Item E12, P6]

From a purely pragmatic point of view, there are times when the initialization list must be used. In particular, const and reference members may only be initialized, never assigned. So, if you decided that a NamedPtr<T> object could never change its name or its pointer, you might follow the advice of Item 21 and declare the members const: [Item E12, P7]

```cpp
template<class T>
class NamedPtr {
public:
    NamedPtr(const string& initName, T *initPtr);
...
private:
    const string name;
```
T * const ptr;
};

This class definition requires that you use a member initialization list, because const members may only be initialized, never assigned. Item E12, P8

You'd obtain very different behavior if you decided that a NamedPtr<T> object should contain a reference to an existing name. Even so, you'd still have to initialize the reference on your constructors' member initialization lists. Of course, you could also combine the two, yielding NamedPtr<T> objects with read-only access to names that might be modified outside the class: Item E12, P9

template<class T>
class NamedPtr {
    public:
        NamedPtr(const string& initName, T *initPtr);
        ...
    private:
        const string& name;               // must be initialized via
                                          // initializer list
        T * const ptr;                    // must be initialized via
                                          // initializer list
};

The original class template, however, contains no const or reference members. Even so, using a member initialization list is still preferable to performing assignments inside the constructor. This time the reason is efficiency. When a member initialization list is used, only a single string member function is called. When assignment inside the constructor is used, two are called. To understand why, consider what happens when you declare a NamedPtr<T> object. Item E12, P10

Construction of objects proceeds in two phases: Item E12, P11

1. Initialization of data members. (See also Item 13.) Item E12, P12
2. Execution of the body of the constructor that was called. Item E12, P13

(For objects with base classes, base class member initialization and constructor body execution occurs prior to that for derived classes.) Item E12, P14

For the NamedPtr classes, this means that a constructor for the string object name will always be called before you ever get inside the body of a NamedPtr constructor. The only question, then, is this: which string constructor will be called? Item E12, P15

That depends on the member initialization list in the NamedPtr classes. If you fail to specify an initialization argument for name, the default string constructor will be called. When you later perform an assignment to name inside the NamedPtr constructors, you will call operator= on name. That will total two calls to string member functions: one for the default constructor and one more for the assignment. Item E12, P16
On the other hand, if you use a member initialization list to specify that `name` should be initialized with `initName`, `name` will be initialized through the copy constructor at a cost of only a single function call. .GetItem E12, P17

Even in the case of the lowly `string` type, the cost of an unnecessary function call may be significant, and as classes become larger and more complex, so do their constructors, and so does the cost of constructing objects. If you establish the habit of using a member initialization list whenever you can, not only do you satisfy a requirement for `const` and reference members, you also minimize the chances of initializing data members in an inefficient manner. .GetItem E12, P18

In other words, initialization via a member initialization list is always legal, is never less efficient than assignment inside the body of the constructor, and is often more efficient. Furthermore, it simplifies maintenance of the class (see Item M32), because if a data member's type is later modified to something that requires use of a member initialization list, nothing has to change. .GetItem E12, P19

There is one time, however, when it may make sense to use assignment instead of initialization for the data members in a class. That is when you have a large number of data members of built-in types, and you want them all initialized the same way in each constructor. For example, here's a class that might qualify for this kind of treatment: .GetItem E12, P20

```cpp
class ManyDataMbrs {
public:
    // default constructor
    ManyDataMbrs();
    // copy constructor
    ManyDataMbrs(const ManyDataMbrs& x);
private:
    int a, b, c, d, e, f, g, h;
    double i, j, k, l, m;
};
```

Suppose you want to initialize all the ints to 1 and all the doubles to 0, even if the copy constructor is used. Using member initialization lists, you'd have to write this: .GetItem E12, P21

```cpp
ManyDataMbrs::ManyDataMbrs()
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
 j(0), k(0), l(0), m(0)
{ ... }
ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
 j(0), k(0), l(0), m(0)
{ ... }
```

This is more than just unpleasant drudge work. It is error-prone in the short term and difficult to maintain in the long term. .GetItem E12, P22

However, you can take advantage of the fact that there is no operational difference between initialization and assignment for (non-`const`, non-reference) objects of built-in types, so you can
safely replace the memberwise initialization lists with a function call to a common initialization routine: "Item E12, P23"

class ManyDataMbrs {
public:
   // default constructor
   ManyDataMbrs();
   // copy constructor
   ManyDataMbrs(const ManyDataMbrs& x);
private:
   int a, b, c, d, e, f, g, h;
   double i, j, k, l, m;
   void init();                  // used to initialize data
       // members
};
void ManyDataMbrs::init()
{
   a = b = c = d = e = f = g = h = 1;
   i = j = k = l = m = 0;
}
ManyDataMbrs::ManyDataMbrs()
{
   init();
   ...
}
ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
{
   init();
   ...
}

Because the initialization routine is an implementation detail of the class, you are, of course, careful to make it private, right? "Item E12, P24"

Note that static class members should never be initialized in a class's constructor. Static members are initialized only once per program run, so it makes no sense to try to "initialize" them each time an object of the class's type is created. At the very least, doing so would be inefficient: why pay to "initialize" an object multiple times? Besides, initialization of static class members is different enough from initialization of their nonstatic counterparts that an entire Item — "Item 47" — is devoted to the topic.