Document Number: WG21/N1959=06-0029
Date: 2006-02-24
Reply to: Michael Spertus
mike_spertus@symantec.com
Symantec Corporation
2015 Spring Rd., #417
Oakbrook IL 60523 USA

Class member initializers

Michael Spertus

Abstract

We propose allowing the use of initializers for non-static `class` and `struct` attributes. The purpose of this is to increase maintainability, reduce the risk of subtle errors in complex program code, and to make the use of initializers more consistent.

1 The proposal

The basic idea is to allow non-static attributes of `class` and `struct` types to be initialized where declared. All of the same initialization syntaxes may be used as for initialization of local variables.[1]

As a simple example,

```
class A {
public:
    int a = 7;
};
```

is equivalent to

```
class A {
public:
    A() : a(7) {}
};
```

The real benefits of member initializers do not become apparent until a class has multiple constructors. For many attributes, especially private attributes, all constructors initialize an attribute to a common value as in the next example:

[1] Other analyses and suggestions for improving initialization of local variables as in N1493, N1509, N1584, N1701, N1806, N1824, and N1919 may also be applied *mutatis mutandis* to non-static `class` and `struct` attributes.

```
class A {
public:
    A(): a(7), b(5), hash_algorithm("MD5"), s("class A example") {}
    A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"), s("Constructor run") {}
    A(int b_val) : a(7), b(b_val),  hash_algorithm("MD5"), s("Constructor run") {}
    A(D d) : a(f(d)), b(g(d)),  hash_algorithm("MD5"), s("Constructor run") {}
    int a, b;
private:
    // Cryptographic hash to be applied to all A instances
    HashingFunction hash_algorithm;
    // String indicating state in object lifecycle
    std::string s;
};
```

Even in this simple example, the redundant code is already problematic if the constructor arguments for `hash_algorithm` are copied incorrectly in one of the `A` constructors or if one of the lifecycle states was accidentally misspelled as `"Constructor Run"`. These kinds of errors can easily result in subtle bugs.

Such inconsistencies are readily avoided using member initializers.

```
class A {
public:
    A(): a(7), b(5) {}
    A(int a_val) : a(a_val), b(5) {}
    A(int b_val) : a(7), b(b_val) {}
    A(D d) : a(f(d)), b(g(d)) {}
    int a, b;
private:
    // Cryptographic hash to be applied to all A instances
    HashingFunction hash_algorithm("MD5");
    // String indicating state in object lifecycle
    std::string s("Constructor run");
};
```

Not only does this eliminate redundant code that must be manually synched, it makes much clearer the distinctions between the different constructors.[2]

Now suppose that it is decided that MD5 hashes are not collision resistent enough and that SHA-1 hashes should be used. Without member initializers, all the constructors need to be updated. Unfortunately, if one developer is unaware of this change and creates a constructor that is defined in a different source file[3] and continues to initialize the cryptographic algorithm to MD5, a very hard to detect bug will have been introduced. It seems better to keep the information in one place.

[2]Indeed, in Java, where both forms of initialization are available, the use of member initializers is invariably preferred by experienced Java programmers in examples such as these.

[3]The above examples show inlined constructors for convenience. In practice, they could well be located in different source files. The programmer will be unaware of her error unless she examines the bodies of the constructors in different files.

It may happen that an attribute will usually have a particular value, but a few specialized constructors will have need to be cognizant of that value. If a constructor initializes a particular member explicitly, the constructor initialization overrides the member initializations as shown below:

```
class A {
public:
    A(): a(7), b(5) {}
    A(int a_val) : a(a_val), b(5) {}
    A(int b_val) : a(7), b(b_val) {}
    A(D d) : a(f(d)), b(g(d)) {}
    // Copy constructor
    A(const A& aa) : a(aa.a),
                     b(aa.b),
                    hash_algorithm(aa.hash_algorithm.getName()),
                    s(aa.s) {}
    int a, b;
private:
    // Cryptographic hash to be applied to all A instances
    HashingFunction hash_algorithm("MD5");
    // String indicating state in object lifecycle
    std::string s("Constructor run");
};
```

A few additional points are worth noting.

- By allowing non-static attributes of classes to be initialized in the same way as non-static local variables (which should be thought of as non-static attributes of the function frame), C++ initialization becomes more consistent. As many of the above mentioned documents point out, this is much needed.

- Because these initializers must be given in the class declaration, which may be included in many files, it is possible that the initializers may vary in value depending on context. Although this is troublesome, the same problem already exists for default arguments, inline methods, member types, etc. Therefore member initializers do not introduce this problem and indeed do not appreciably aggravate it.

- There is some overlap between this proposal and constructor delegation and forwarding (N1445, N1581, N1618, and N1898). However, it is easy to see that neither technique obviates the other. For example, overriding member initializers appears difficult to do in generality through constructor forwarding.[4]

[4]It is worth noting that Java has both member initialization and constructor forwarding. This is not regarded as confusing, and most experienced Java programmers make regular use of both techniques based on applicability.

- Member initializers make possible the use of copy-initialization for class attributes. It seems likely this flexibility will prove useful just as in the case of local variables.