Accredited Standard Committee[1]
X3, INFORMATION PROCESSING SYSTEMS

Doc. No: X3J16/92-0056
WG21/N0133
Date: June 1, 1992
Project: Programming Language C++
Ref Doc: X3J16/91-0067
Reply To: Bill Gibbons
Taligent Inc.
10725 N. De Anza Blvd.
Cupertino, CA 95014
bgibbons@taligent.com

Proposal for Compile Time Constants in Class Scope

Bill Gibbons and David Goldsmith

This document is a revision of X3J16/91-0067 with additional examples, including problems caused by the absence of this feature; and additional details about semantics and implementation. The document number has been changed to accommodate WG21 numbering.

The Problem

The class scoping features of C++ are very helpful in maintaining large projects, because they allow you to keep most names out of the global namespace. Compile time constants are important for parametrizing constant values which might change, such as array bounds.

However, compile time constants are currently not allowed in class scope:

```
class Foo {
    static const int maxSize = 500;  // illegal
    char localBuffer[maxSize];       // illegal
};
```

It's possible to declare a static const data member and initialize it elsewhere. This is a compile time constant, but it cannot be used within the class declaration (because the initialization appears after the class declaration). The only alternative is to declare such a constant in global scope.

There is a workaround: a similar effect can be had by declaring an appropriate enumeration in class scope:

```
class Foo {
    enum {maxSize = 500};
    ...
};
```

but only needed to hold highest value accept in constant expression

This works, but is highly inelegant. The constant is not an integral type, either. What's more, enumerations are not guaranteed to be able to hold the largest integral types: they may not be able to hold the same range of values as a long.[2]

1. *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 311 First St. NW, Suite 500, Washington, DC 20001

2. §7.2, Enumeration Declarations: "The value of an enumerator must be an `int` or a value that can be promoted to an `int` by integral promotions (§4.1)."

```
class Foo {
    enum {maxSize = 1000000};   // may not be legal
};
```

This makes it impossible to write such declarations in a portable fashion.

Use of an enumerator to represent what is conceptually an unsigned integral type can lead to some surprises. There are places in the language where the promotion of an enumerator to int is not sufficient. For example:

```
void set_delay(unsigned int);   // delay time
void set_delay(char);           // unit code ('h', 'm', or 's')

class Timer {
    enum {initial_delay = 10 };     // initial delay in seconds
    Timer() {
        set_delay(initial_delay);   // fails: two standard conversions
        set_delay('s');
    };
};
```

This fails because the `set_delay` functions are written assuming all delay times will be unsigned, yet there is no way to put an initialized unsigned constant into the class. The enumerator promotes to `int`, and the standard conversions from `int` to `unsigned int` and `int` to `char` are equally valid.

Another example:

```
template<unsigned count> class Flags { ... };   // cheap "set" class

class Sparse_U {
    enum { size = 100 };    // size of sparse array
    U u[size+1];            // includes sentinel
      ...
    Flags<size> exists;     // whether element exists ** DOES NOT WORK **
};
```

The class template instantiation `Flags<size>` does not work because `size` is an enumerator and template arguments must match exactly.

Of course, the examples could have been written as `set_delay((unsigned)initial_delay)` and `Flags<(unsigned)size>`, but this means that the enumeration hack has implications not just at the declaration site but also at each use of the enumerator. If the declaration and use are not adjacent this may be a maintenance problem.

Proposal

We propose that class-scope initialization of static const integral data members be made legal. The file scope definition of the static member would still be required, but the initialization could be placed either in the class or in the file scope definition. For example:

```
class Foo {                               // New form
    static const int maxPath = 500;       // legal
    char name[maxPath];                   // legal
    void f() { char file[maxPath]; }      // legal
    void g();
};

const int Foo::maxPath;                   // required; must not have initializer
void Foo::g() { char file[maxPath]; }     // legal


class Bar {                               // Existing form (still valid)
    static const int maxPath;
    char name[maxPath];                   // illegal
    void f() { char file[maxPath]; }      // illegal
    void g();
};

const int Bar::maxPath = 500;             // required; must have initializer
void Bar::g() { char file[maxPath]; }     // legal
```

Why require the file scope definition? Because some implementations will not be able to create a unique instance of the static member unless there is a unique file scope definition. This problem does not occur with (non-class-member) compile time constants today; if they are declared at file scope, they have internal linkage by default.[3]

Formally, we propose the following addition to the draft (suitably edited) in § 9.4/5: "A const static data member with an integral type must be initialized either in the class declaration (using a constant expression for the initializer) or in the file scope definition of the member. The file scope definition is required even when the member is initialized in the class declaration."

Grammar Changes

The existing definition of *member-declarator* would have to change to accommodate the initializer. The simplest change would be:

old:

member-declarator:
  declarator pure-specifier$_{opt}$
  identifier$_{opt}$: *constant-expression*

pure-specifier:
  = 0

new:

member-declarator:
  init-declarator
  identifier$_{opt}$: *constant-expression*

3. §7.1.6, Type Specifiers: "Unless explicitly declared `extern`, a `const` object does not have external linkage and must be initialized."

This makes initializers for member functions legal in the grammar, but we already have initializers for non-member functions in the grammar (but not the language).

Initializer Semantics

Since class declarations are typically put in header files and included many times, the initialization expression will be compiled in different translation units. The names used in the initializer might have different declarations in different translation units, so the constant would not have the same value.

This problem already occurs with default arguments in member function declarations. The solution is the same, the One Definition Rule: the names in the initializer must bind to the "same" declarations in each translation unit. For example:

First Translation Unit

```
const int x = 5;

---- from include file ----

class T {
    void foo(int value = x);
    static const int y = x;
};
```

Second Translation Unit

```
const int x = 6;

---- from include file ----

class T {
    void foo(int value = x);
    static const int y = x;
};
```

Both the default argument and the const member initializer violate the One Definition Rule.

Implementation

Static members can be initialized today, and must act as compile time constants after they are initialized. The additional implementation work to allow the initializer to appear in the class instead of in the file scope definition is minimal.

Under the new name lookup rules for classes, the initializer would appear to be in the same category as inline function bodies, constructor initializer lists, and default arguments. These constructs can have forward references (never actually called that) to members declared later in the class, including types.

But initialized const members could be used to build types in member declarations, so their values must be known immediately. The initializers should **not** be grouped with inline function bodies, constructor initializer lists and default arguments; rather, the semantics of name lookup in initializers should be the same as in member declarations.

Class-scope initialized constants have been implemented in G++ and in a compiler being developed at Taligent. There were no surprises in the Taligent implementation; it went quite well.

special case extension; paving the way for further extension — and those would be undesirable

Variation: Internal Linkage

yuck.

The file scope definition could be eliminated if static const members initialized in the class had "internal linkage" (suggested by John Armstrong). Then each compilation unit would have its own copy of the constant; usually it would not be allocated space, since the main purpose is to get the value at compile time.

Variation: All Static Data Members

The proposal could be extended to apply to all static data members. For integral const members initialized with a constant expression, the value of the member would be available immediately just as with non-members. For other members, the change would just affect the location of the initializer: in the class or in the file-scope definition.

Putting the initializer in the class would make it much easier for a reader of the class to find the initializer. The file-scope definition would be required only to allocate space; there would never be a need to locate the definition when trying to understand the class.

To implement this variation, a compiler would need to save the initializer expression just as default argument expressions are saved today. The expression would be used when the file-scope definition was seen, just as default arguments are used when an appropriate call is seen. The impact on compiler complexity is probably minimal.

Although it is useful and easy to implement, we do not recommend this variation be adopted at this time.

Variation: All Data Members

Kludge!!

There are some advantages to allowing initializers even for non-static members. The initializers would be used as default member initializers by the constructors. This lets the author of the class write the member initializer with the member declaration, instead of in the constructor definition which might be located in a different file. (This use was suggested by Martin O'Riordan.)

Again, this is a matter of convenience and readability rather than functionality. But removing restrictions on initializers makes the language more consistent.

Implementing this variation would require copying the non-static initializers from the member declarations to the constructor initializer lists. Since these lists cannot be processed until the end of the class (due to the new name lookup rules), there should be no additional interactions.

Although it is useful and easy to implement, we do not recommend this variation be adopted at this time.

C++ allows something like that

context for initializer? order of initialization

Summary

Adding class-scope initialized const members to the language would remedy a deficiency in the use of name scoping, and is consistent with the semantics of static members and compile time constants as defined today. The new construct would simplify class library implementation. The semantics and implementation are both simple and obvious.

The workaround using enumerations is counterintuitive and confusing, especially to a novice; and it requires changes both at the point of declaration and the point of use. If we look at the combination of the language and the idioms and folklore needed to use it, this change might actually reduce the overall complexity.