Doc No: SC22/WG21/N1469

J16/03-0052

Date: 03/25/2003

Project: JTC1.22.32

Reply to: Daniel F. Gutson

danielgutson@hotmail.com

Inline Constants

1. The Problem

Constants occupy space during the their life, even if they are intended to be used temporary in assignments or as parameters.
Moreover, if they are going to be used in few places and/or few times (esporadically), which doesn´t justify their co-existance with other objects which existance is required more frequently, or don´t need to hold a ´state´, just be used and discarded (typically POD types).

-Why is the problem important?
Because of unefficient memory usage, and [as consecuence] when avoided, other solutions (mentioned below) carry undesired consequences, neutralizing strong C++ language features and priniples.
Overhead.

-Whom does it affect?
Embedded community, low resources system programming.

-What are the consequences of not addressing it?
Forces constant objects -under the circumstances mentioned above- to steel resources to other program elements (constants or not).

-How are people addressing, or working around, the problem today?
When avoided, people use the ´#define´ or the ´enum´ idioms, or inline functions returning a value.
a)The problem of using the #define preprocessor directive is:
* may have no type information
* has no scope validation
* may prevent optimizations as far as the compiler ignores that the same [logical] concept is referred
(because the original information was lost during preprocessing)

b) The problem of using the ´enum´ idiom is:
* only integral types can be used
* ´enum´ essence is referred to type-safety and self-documentation rather than their ´numeric´ implementation fact
(it´s used the fact that they are represented (implemented) with numbers rather than the concept of having enumerated elements)
* ´enum´ groups elements under a common design and logical assertion. Constants might not be related at all, so no logical grouping is needed, which might lead to confusion while reading.

c) by placing the value inside a function, a conceptual violation is made as far as the value is ‘hidden’ inside the implementation (despite it might be ‘visible’ by the fact of being inlined). Additionally, readability is decreased and coding overhead occurs.

Many compiler implementations already perform a ‘instantiation on demand’ for global variables, but the behavior is not the proposed in this paper (as far as they are not deterministically ‘discardable’).
This paper proposes a STANDARD and warranted way of declaring constants that deterministically will not occupy space unless referenced.
This is extremely important when programming **embedded systems with hard space restrictions**, in which an exact amount of memory usage is determined by design.

-Which of the categories that we're interested in addressing does this fit into?
The major initial categories are:
* improve support for systems programming (performance, scoping)
* features from other languages (would be the C++´ version of the C´s ´#define´)
* remove embarrassments (refer to the enum idiom argument above)

2. The Proposal
Enable the ´inline´ modifier for constants definition, meaning that the instantiation will occur on-demand whenever the constant is referred.

2.1 Basic Cases
Typical POD types:

```
//in a header file
namespace MyConstants
{
      inline const float
            PI = 3.1415926,
            E = 2.78;
      inline const char* MY_NAME = "Daniel";
      inline const unsigned int MAX_RETRIES = 3;
      inline const size_t HEAP_SIZE = 512;
      inline const unsigned char N_BITS = 1;
}

f(PI);      // PI is temporal
```

2.2 Advanced Cases
User-defined types and objects.

```
inline const MyIntWrapper myFive(5);
inline const Direction Up(0,1), Down(0,-1), Left( -1,0), Right( 1,0);

move(Up); // Up is constructed and [might be] destroyed after the function call
```

3. Interactions and Implementability
3.1 Interactions
a) The address of an inline constant may not be constant. The following represents an unspecified behavior:

```
      bool result = (&PI == &PI);
```

similar to the address of literals and temporal objects.

b) References should not be allowed to be constant inlined.

```
      inline const MyObject& myConst(obj);   // error
```

c) inline constant class attributes should not occupy space, behaving as static constants when used.
The result of a sizeof operator applied to an object containing one or more inline constants should not be

affected by them.
The difference with a static const attribute, is that such attribute is [conceptually] allocated during the program life,
while inline constants are temporally allocated on-demand.
Therefore, static and inline const should not be combined in class declarations:

```
class A{
      static inline const int x = 1;      // error
};
```

c.1) static and inline constants in a function-scope should also be avoided, as far as they are a semantic contradiction.

d) the declaration of inline constants do not occupy space, as far as it baheaves just as declaratory, allowing to
place (define) them in the header files, without ´extern´ declaration.

3.2 Implementability
-This feature does not affect compatibility due to its [current] syntactic indefinition.
-inline constant declarations of objects using non-default constructors should be considered as a
´future construction´, that is, how the object will be constructed temporally.
-Additionally, those non-default constructions shall only allow literals or constant parameters.
-inline constants should be solved during compilation, not linkage, therefore they should be defined
prior to be used (that´s why the header file is a good place to define them).