Doc No: SC22/WG21/N1493
J16/03-0076
Date: 09/18/2003
Project: JTC1.22.32
Reply to: Daniel F. Gutson
danielgutson@hotmail.com

Braces Initialization Overloading

1. The Problem

A goal of C++ is to simulate with classes the same behaviors and capabilities of PODs.
This is possible through operator overloading.
However, it is not possible to provide the initialization abilities of a C-array to a class: the braces initialization.
Also, it is neither possible to initialize (construct) a non-aggregate class with braces, as a simple struct allows.
This paper proposes a mechanism to do both things, by overloading the braces initialization.
This problem specially applies to container classes in libraries -such as the std::vector, which would permit to treat it as an ordinary C-array -.
By no addressing this issue, "manual" (or explicit) initialization must be coded, loosing the simplicity of the wide-used C´s braces initialization.

This proposal may fit in the following categories:
* improve support for systems programming
* improve support for library building
* improve compatibility with C

2. The Proposal

A) Problem partitioning.

This paper considers the following scenarios:

A.1) braces initialization for a fixed number of elements of known types: this is the case of initializing a non-aggregate as an ordinary struct

```
struct S
{
    int a, b;
    char* p;
};
S s = { 1, 2, "hello" };
```

A.2) braces initialization for a NON-fixed number (known at compile time) of elements. This is sub-partitioned in two:

A.2.1) homogeneous elements: a non-fixed number of elements of the same type. This is the case of a C-array:

```
int x[] = {1,2,3,4,100};
```

A.2.2) heterogeneous elements: a non-fixed number of elements, not necessarily of the same type.

B) The proposal.

This paper distinguishes three constructor types, as primitive semantic units for achieving the problem scenarios.

1-the "fixed brace constructor"
2-the "brace initiator constructor"
3-the "brace element constructor" (or “brace individual constructor”)

The "fixed brace constructor" covers the A.1 scenario; the "brace initiator" and the "brace element" constructors are combined together for covering A.2.1 as well as A.2.2.

Characteristics and semantics:
1. The "fixed brace constructor" is defined in the class with a specific signature, telling the compiler that whenever the class is initialized with the form

classname instance = { v_1, v_2, ..., v_n };

will try to match the signature with the v_1..v_n value types, and invoke it, each v_i being a parameter.
This constructor type is multiple-overloadable, and mimics the initialization of an aggregate using braces for assigning a value to each member.

2. The "brace initiator constructor" can be defined once in the class with the unique signature of receiving one ´size_t´-like parameter.
The "brace element" accepts only one parameter, and is multiple-overloadable, with an arbitrary parameter type.

When a class is initialized with the form

classname instance = { v_1, v_2, ..., v_n };

if no "fixed brace constructor" is defined, or no one matches the signature with the n elements, then the compiler invokes the "brace initiator constructor" once passing *n* as the parameter, and then invokes n times the "brace element" constructor (one per each v_i). The restriction is that every type of the v_i has a type-compatible "brace element" available.

If the "brace initiator constructor" is defined, then at least one "brace element" constructor must be defined.
A class having "brace element" constructors, without the "brace initiator" constructor defined, is ill-formed and a compiler error should be thrown.
By providing many "brace element" constructors to a class, the heterogeneous scenario (A.2.2) is covered.
By providing only one "brace element" constructor, the homogeneous scenario (A.2.1) is covered.
For example, if the class is intended to be constructed with a braces initialization like a C-array of type int[], the "brace initiator" constructor must be defined, and one "brace element" receiving an int parameter must be also defined.

The three constructor types are either called by the compiler, or ´transparently´ by a subclass in the initializer list (´transparently´ means that the {} initialization should be used in the initializer list, and then the compiler will either invoke the "fixed braces constructor" or the "braces initiator" + "brace element" constructors).

2.1 Basic Cases

Before entering in the examples, the syntax and alternatives are proposed now. First, a set of candidates are exposed, and then assigned (arbitrary) to the constructor types for developing the examples only. Syntax is not the aim of this paper.

Syntax candidates, and their usage in the examples below:

```
classname {} ( signature );   // used for "braces initiator constructor"
classname ={} ( signature );  // used for "fixed brace constructor"
classname {*} ( signature );  // used for "brace element constructor"
classname {...} ( signature );      // just proposed; not used here
classname []{} ( signature );       // just proposed; not used here
```

The first example covers scenario A.1, exposing the "fixed brace constructor":

```
class Packet
{
public:
        Packet = {} ( char parity, const char* data, size_t len );
        ~Packet();

        int getHeader() const;
        char getParity() const;
        const char* getData() const;
        size_t getLength() const;
        size_t getCRC() const;

private:
        char _parity;
        char* _data;
        size_t _len;
        size_t _crc;

        void _calcCRC();
};

void sendInitialPacket()
{
        Packet p = { 'o', "INIT", 5 };
        sendPacket(p);
}

Packet::Packet = {} (char parity, const char* data, size_t len )
        : _parity(parity), _len(len)
{
        _data = new char[len];
        memcpy( _data, data, len );
        _calcCRC();
}
```

In the example above, the Packet class behaves as a C structure having a char, char*, and a size_t element. The "fixed brace constructor", however, hides the default constructor, assigns the private attributes, copies the data for owning it, and then calculates the CRC. (The destructor would delete [] the data).

The second example covers scenario A.2.1, simulating an array of integers.

```
class IntArray
{
public:
        IntArray {} (size_t count);
        IntArray {*} (int element, size_t position);        //see 3.1
        ~IntArray();

        int operator [] (size_t index) const
        {
                assert( index < _len);
                return _array[index];
        }

        int& operator [] (size_t index)
        {
                assert( index < _len);
                return _array[index];
        }
private:
        int* const _array;
        const int _len;
};

void f()
{
        IntArray arr = { 1,2,3,4,5,6 };
        int x = arr[1];
        arr[x] = 200;
}

IntArray::IntArray {} (size_t count)
        : _array(new int[count]), len(count)
{}

IntArray::IntArray {*} (int element, size_t position)
{
        _array[position] = element;
}

IntArray::~IntArray()
{
        delete [] _array;
}
```

This example shows the usage of the “brace initiator” and the “brace element” constructors. The first allocates the memory and assigns the length. The second assigns the position-$_{th}$ element to the private allocated array. (please see section 3.1 for information about the second parameter of the “brace element constructor”).
The IntArray class cannot be initiated with a list of elements of a type not casteable to int, as far as the “brace element constructor” is not available for other types.

In function ‘f’, the compiler will invoke the “brace initiator” passing 6 as the parameter, and will invoke the “brace element” six times immediately, passing the incremental number in the ‘position’ argument (0..5).

This example just exposes the usage of the two constructors proposed in this paper. The usage for a template class is considered a trivial application and is not exposed.

The third example covers the scenario A.2.2, where the class can be initialized with a list of two classes types derived from a base class.

```
struct Base { … };
struct Der1 : Base { … };
struct Der2 : Base { … };

class BArray
{
public:
        BArray {} (size_t count);
        BArray {*} (Der1 element, size_t position);
        BArray {*} (Der2 element, size_t position);
        …
};

void f()
{
        Der1 d1;
        Der2 d2;
        BArray arr = { d1, d1, d2, d1, d2 };
        …
}
```

This example is similar to the second, but differs in the fact that BArray is an ‘heterogeneous’ container, having two types of elements (Der1 and/or Der2). Implementation is left for the imagination of the reader.

2.2 Advanced Cases

The example here will combine the first and second examples of the previous section, by using a nested braces initialization: an array of packets.
The Packet definition is the same as the defined in the first example.

```
class PacketArray
{
public:
        PacketArray {} (size_t count);
        PacketArray {*} (const Packet& pkt, size_t position);
        …
};

void sendSequence1()
{
        PacketArray seq = {
                                { 'o', "INIT", 5 },
                                { 'e', "DATA1", 6 },
                                { 'n', "END", 4 }
                        };

        sendPacketArray( seq );
}
```

In this example, the compiler will construct three temporary packets by invoking the Packet::Packet ={}, then invoke the PacketArray's initiator, then three times the PacketArray's brace element, and finally destroy the temporary packets.
(Let's assume that the packets will be copied in private memory spaces by the PacketArray implementation).

3. Interactions and Implementability

3.1 Interactions

The A.2 scenarios could be covered by the "fixed braces constructor" by receiving one container and overloading the comma operator, but providing the "brace initiator constructor" / "brace element constructor" complex provides simplicity and efficiency (the number of elements is known at compile time, so no re-allocation must be carried out while appending the elements).

Optionally, the "brace element constructor" might accept an alternative syntax, receiving the element position as the second parameter (of a ´size_t´-like type).

Initializer list can only be defined in the "fixed brace" constructors and the "brace initiator" constructor; the "brace element constructor" behaves as a ´sub-constructor´ of the initiator, meaning that the per-element call is not a constructor call itself, but a subsequent call sequence of the initiator.

The 'operator =', if defined, will be invoked neither in A.1 nor A.2 scenarios, considering that the '=' is syntactically required.

The place (private, public or protected) where the “brace element constructor” is defined is insignificant, as far as the “brace initiator constructor”’s accessibility is considered.

Name hiding rules for constructors is normally applicable. Many constructors can be provided, and ambiguities between the “fixed brace” and the “brace initiator”/”brace element” constructors shall be clarified by prioritizing the fixed brace constructor. (see section 3.2)

This proposal is compatible with the existing syntax, as far as the proposed constructs are currently invalid.

3.2 Implementability

The compiler must prioritize the "fixed brace constructor" signature matching first.
If no one matches (or there is no one defined at all), then the "brace initiator"/"brace element" complex look up should take place.
In this case, all the element types listed within the braces should have a compatible "brace element constructor" defined.