Doc No: SC22/WG21/N1464
J16/03-0047
Date: 01-Apr-2003
Project: JTC1.22.32
Reply to: Daniel Gutson
danielgutson@hotmail.com

ANONYMOUS ARRAY MEMBERS

1. The problem

There is no standard way of defining padding spaces in structures (specially aggregates), nor forcing them to have a fixed size (to be completed with padding or to be checked it is not exceeded).

- Why is the problem important?

a) padding: a member must be declared for this purpose, and comments or additional documentation is needed for expliciting its intention of existence, while conceptually padding spaces should not be treated as members. Naming padding spaces (as members) requires a naming convention, documentation, or a "smartly universal" name to be understandable by others.

b) Fixed length: when a structure length is required to be fixed, adding or modifying its members has to be performed carefully, as far as no natural check is performed (whether the size has exceeded or padding space is needed) by the compiler.

- Whom does it affect?

Protocol communications, messaging, embedded. It is common required a packet to have fixed size, or "reserved" padding spaces/bits.

- What are the consequences of not addressing it?

Manual checks, run-time errors or checking, additional maintenance effort, additional documentation required.

- How are people addressing, or working around the problem today?

a) for paddings: usually declared as members of type arrays of chars, with the length as array bound, following some naming convention or explicating the name.

b) For fixed structure length: usually a member as mentioned above (a), with length manually set as fixed-actual-length, requiring an update each time the structure is modified.

- This feature fits in the following subset of categories mentioned in the proposal template:

- improve support for system programming: proposes a native way for supporting paddings (and fixed sizes) with compile-time checking, and without extra maintenance effort. It also serves for self-documentation purposes, expliciting differentiating members from paddings.

2. The proposal

Support anonymous array "members". They are unaccessible, as far as they don't have name; they do not count (participate) in braced-enclosed initializer lists. A structure with an anonymous array only should be considered as an "empty structure" but with non-zero size. This proposal addresses the problem of fixed-size structures using paddings, showing a simple implementation.

2.1. Basic Cases

```
struct Msg1
{
        int a;
        int [3];                // 3 ints of padding
        int b;
};
//      sizeof(Msg1) = 5 * sizeof(int)

        Msg1 m={ 1,3 }; // m.a=1; m.b=3

struct Msg2        // an implementation of fixed-size strucs using paddings
{
        Msg1 m;
        char [128-sizeof (Msg1)]; // makes Msg2 to have a fixed length of 128 bytes
};
```

2.2. Advanced Cases

Bits can also be used as padding

```
struct Field1
{
   int:3  p1;
   int:2;
   int:3   p2;
};
```

A suggested "warning" rule is that a bit-padding field should be contiguous to at least non anonymous (non-padding) bit field.

3. Interactions and implementability

3.1. Interactions:

a) only POD types, POD structures and bit-types can be used as padding base types

b) paddings could be “initialized” according to the syntax proposed in “non-default constructors for arrays”.
   Example
```
        char [10] (0);
```
   It would produce a padding space of 10 zeros

c) padding shall always be arrays, even for size = 1, in order to avoid typing errors:
   example
```
        struct A
        {
                int a;
                int; // error: should be int [1];
                char b;
        };
```

3.2 Implementability

- paddings could be treated by the compiler as common members with a hidden (internal) name.
- compatibility not violated due to current syntax incorrectness
- Alignment issues should be considered: if a padding fits in an “alignment hole”, it should be ignored
  Example: aligning to 4 bytes:
```
        struct S1
        {
                char a;
                int b;
        };

        struct S2
        {
                char a;
                char[1];        //if sizeof(int)=4, there are 3 unused spaces
                int b;
        };

//      sizeof(S1) = sizeof(S2)
```