Proposal for classes with runtime size

L. Deniau and A. Naumann
CERN, Geneva, Switzerland

Date: 2014-10-01

Document number: N4188
E-mail: laurent.deniau@cern.ch

1 Introduction

C99 introduced the concept of Flexible Array Member as a possible definition for the last field of a structure to allow data structures with runtime size. The motivation was to formalize a common practice in C used to emulated this missing feature. The principle was to define an array of a single element at the end of a structure, and to dynamically allocate extra space at runtime to store more elements while ignoring the *undefined behavior*.

The purpose of this proposal is to extend the concept of Flexible Array Member to C++ classes with the same kind of motivation, focusing mainly on the simplicity of the implementation [1]. The proposal remains intentionally limited to the support of Flexible Array Member, and excludes the topic of Variable Length Array considered as more ambitious [2]. The objective is to validate this common practice in C++ with better support from the type system and the compiler. The author is aware that despite of its apparent simplicity, this proposal implies significant work on compilers implementation.

2 Motivation

In many cases, if not all, the C++ type system has to be bypassed when a class wants to handle its own dynamic memory storage efficiently using raw allocation, placement new and `reinterpret_cast<>`. The motivation is to avoid doubled dynamic allocations, one for the object of the implementation class and one for the data storage itself. Implementations of basic containers

holding trivial POD objects in the STL use such hack for the same reason. One can find comments like the following in the implementation of the class `basic_string` distributed with GNU GCC 4.9:

```
[...]
This approach has the enormous advantage that a string object
requires only one allocation.  All the ugliness is confined in
[...]
string::_M_rep(); and the allocation function which gets a
block of raw bytes and with room enough and constructs a _Rep
object at the front.
```

A class wanting to efficiently manage its own data storage will delegate the implementation details to another low-level class. A possible definition of the data members of such class would look like:

```
template <typename T>
struct memblock {
  int     refcount;     // ownership
  int     origin;       // provenance
  size_t  size;         // used memory
  size_t  capacity;     // allocated memory
  T      *data;         // pointer to storage
  T       _data[1];     // extra storage beyond [1]
};
```

The `capacity` is related to the amount of memory allocated, giving the possibility to request more memory than `size` and avoid excessive reallocations each time further elements are added to `data`. The `refcount` member is used for various purposes like sharing read-only storage and applying copy-on-write optimization. The `origin` tag keep track of the provenance and the nature of the storage, and used by the destructor. The `data` and `_data` members may or may not be present in the definition (i.e. anonymous array) depending on the intent of the surrounding class (e.g. `basic_string`). If both are present, the member `data` usually points to the extra storage `_data` or to an external storage provided and tracked by `origin`.

One can observe that all these data members are only useful to support some flexibility and various optimizations except `size`, which is essential to record the number of elements currently stored and write safe code.

3 Proposal

We propose the following simple definition as an example of a templated class with runtime size:

```
template <typename T>
```

```
struct A {
  A(size_t n) : _n{n} {}
  const size_t _n; // runtime size specifier
  T _a[_n];        // runtime sized array
};
```

where language requirements are specified in the following section.

3.1 Properties

P.1 The **runtime sized array** (i.e. `_a`) must be the **last** data member. For any other aspects, it should be treated as an array, that is `decltype(A<T>::_a)` is `T[]`.

P.2 The **runtime size specifier** of the runtime sized array (i.e. `_n`) must be a *const-qualified integral* type[1] and the **first** *non-static* data member, i.e. its value cannot depend on any *non-static* member.

P.3 The `sizeof` operator applied to a class must return its *compile-time* size, i.e. `sizeof(A<T>) == offsetof(A<T>,_a)`.

P.4 The `sizeof` operator applied to an object must return its *runtime* size, i.e. `sizeof A<T>() == sizeof(A<T>) + _n * sizeof(T)`.

The implementation must ensure that the runtime size expression is *evaluated once* and the runtime size specifier is *initialized first*. It is *const-qualified* to prevent any resize of the object during its lifetime. A strong point of our proposal is that the compiler always knows the place of `_n` and `_a` inside an object of type `A<T>`, and their respective positions follow the established data members initialization order.

3.2 Limitations

The following limitations are mandatory to limit the added complexity to the implementation for the support of the proposal. We believe that they do not reduce the usefulness of the proposal.

L.1 Runtime sized class cannot have a runtime sized object as a data member.

L.2 Runtime sized class cannot have virtual base class.

L.3 Runtime sized class must be directly or indirectly the right-most base class of a derived class (i.e. multiple inheritance).

L.4 Runtime sized objects cannot be elements of an array.

[1] The implementation should implicitly convert the type to `const size_t`.

L.5 Runtime sized objects can only have dynamic storage duration.[2]

In the proposal extensions, we examine the added complexity required to remove some of these limitations.

3.3 Construction

The construction of an object with runtime size is the difficult part of our proposal, as it requires to move the evaluation of the runtime size expressions outside the constructors right after the arguments evaluation and before the storage allocation.

The conceptual steps to construct an object of a fixed-size class `B<T>` are *as–if*:

F.1 Allocate correctly aligned `sizeof(B<T>)` bytes.

F.2 Call the placement new on the memory allocated.

In this sequence, the arguments of the constructor can be evaluated anywhere before the call of the placement new in F.2.

The conceptual steps to construct an object of a runtime sized class `A<T>` are *as–if*:

R.1 Compute the values of all the constructor arguments.

R.2 Compute the value of the runtime size expression and store it in a temporary `__tmp_n`.

R.3 If `__tmp_n` is negative, throw `std::bad_array_length` (or equivalent), following the behavior of `operator new[]` described in [3] §5.3.4.

R.4 Allocate correctly aligned `sizeof(A<T>) + __tmp_n * sizeof(T)` bytes, following P.4.

R.5 Call the placement new on the allocated memory, replacing the expression for the initialization of the runtime size specifier by the value of the temporary, that is `_n{`*expr*`}` becomes `_n{__tmp_n}` in the initializer list.

The construction of a runtime sized object needs to calculate *once* the expression involved in the initialization of its runtime size specifier in R.2, that corresponds to the selected constructor in R.5.

If `__tmp_n` is zero, R.4 is equivalent to F.1, which ensures backward compatibility with fixed-size class. If `__tmp_n` is negative before its implicit conversion to `size_t`, the construction throw an exception as for `operator new[]`. If `__tmp_n` is positive, the initialization of the runtime sized array `_a` should work *as–if* it

[2]This limitation could be relaxed to include automatic storage duration if the C99 Variable Length Array are supported by C++ in some future.

were an array of type and size `T[__tmp_n]` (i.e. P.1 & R.5), with the same pointer arithmetic and requirements as for the type `T[]`.

If the constructors of a runtime sized class are not inlined, the implementation is free to generate hidden static member functions to calculate the value of the expressions involved in the initialization of the runtime size specifier, and to pass this value as a hidden argument to the constructors.

4 Effects on existing language features

4.1 Construction and destruction

No problem is foreseen for default construction compared to non-default construction as described in 3.3, nor for the destruction. In particular, runtime size specifier can have default *in-class* initializer as other *non-static* data members; and runtime size array can have extra dimensions with constant sizes as specified in [3] §5.3.4-6, that is `T _a[_n][5]` would be well-formed but not `T _a[5][_n]`.

4.2 Copy construction and move construction

No problem is foreseen for copy construction and move construction. For the move construction, if the runtime size specifiers do not have equal values, a `std::length_error` exception should be thrown [3] §19.2.4.

4.3 Copy assignment and move assignment

No problem is foreseen except the concern reported for move construction.

4.4 auto, decltype and typeid

No problem is foreseen as soon as they fulfils P.1. The runtime size specifier of a runtime sized class is not part of its type (i.e. it is not a template argument) and the runtime size array has a static type (i.e. it cannot be a *glvalue* of polymorphic type).

4.5 sizeof and offsetof

The operator `sizeof` is already treated in 3.1. No problem is foreseen for the macro `offsetof` because of P.1 and L.2.

4.6 Storage duration

No problem is foreseen for dynamic storage duration. Other storage durations are not allowed by L.5.

4.7 Exceptions and stack unwinding

No problem is foreseen because of L.5.

4.8 Single inheritance

No problem is foreseen as soon as it fulfils L.2.

4.9 Multiple inheritance

No problem is foreseen as soon as it fulfils L.3.

4.10 Virtual inheritance

No problem is foreseen as soon as it fulfils L.3.

4.11 Templates

No problem is foreseen as soon as it fulfils L.1.

5 Applications

5.1 String, Vector, etc...

Update `memblock` to use runtime sized array (the proposal) for the underlying implementation class.

5.2 Emulation of multiple Flexible Array Members

The main problem comes from alignment constraints of element types. The following example assumes that `double`s have larger alignment constraint than `TA` and `TB`, but this can be checked at compile-time:

```
#include <iostream>

template <typename TA, typename TB>
struct AB {
AB(size_t na, size_t nb) :
_n{n_size(na,nb)}, _na{na}, _nb{nb},
_pa{reinterpret_cast<TA*>(_a)},
_pb{reinterpret_cast<TB*>(_a+na_size(na))} {}
// ... copy & move ctors and assignment ommitted
const size_t _n, _na, _nb;
TA *_pa;          // points to _a
```

```
TB *_pb;           // points within _a
double _a[_n];     // runtime sized array

private:
enum {  _da = std::alignment_of<double>::value,
_dpa = std::alignment_of<TA    >::value,
_dpb = std::alignment_of<TB    >::value };
static_assert(_dpa<=_da && _dpb<=_da,
"unsupported alignment constraint");

static size_t na_size(size_t na) {
return na*sizeof(TA)/sizeof(double)+1; }
static size_t nb_size(size_t nb) {
return nb*sizeof(TB)/sizeof(double)+1; }
static size_t  n_size(size_t na, size_t nb) {
return na_size(na)+nb_size(nb); }
};


int main() {
AB<char,int> ab(10,20); // (10*1/8+1)+(20*4/8+1)=2+11=13
constexpr size_t AB_sizeof = sizeof(AB<char,int>);
std::cout << "_n =" << ab._n      << "\n"; // _n =13
std::cout << "_na=" << ab._na     << "\n"; // _na=10
std::cout << "_nb=" << ab._nb     << "\n"; // _nb=20
std::cout << "ab =" << sizeof ab << "\n"; // ab =144
std::cout << "AB =" << AB_sizeof << "\n"; // AB=40
}
```

6 Possible extensions to the proposal

6.1 Recursive runtime sized types

If the type `T` is itself a runtime sized class (e.g. `A<A<int>>`), meaning somehow that L.1 and `L.4` are relaxed, then the expression `sizeof(T)` has to be replaced by `sizeof T()`, which must be evaluated (recursively) using a modified version of P.4:

```
sizeof(T) + (_n>0 && _a[0]._n>0 ? _a[0]._n*sizeof _a[0]._a[0] : 0)
```

The idea is that all elements of the runtime size array `_a` must have the same size, hence peeking the runtime size specifier of the first element is enough.

In order to avoid recursive evaluation, we propose to store the value of `sizeof _a[0]` in a hidden data member `size_t __rsa_esize`, standing for *"runtime size array element size"*. In this case, P.4 can be evaluated in one step using:

```
sizeof A<T>() == sizeof(A<T>) + _n*__rsa_esize
```

This would also greatly simplify the pointer arithmetic of `_a`:

```
_a+i == (A<int>*)((char*)_a + i*__rsa_esize)
```

6.1.1 Construction

Unfortunately, the problem of recursive evaluation persists for the construction of such objects as the hidden data member `__rsa_esize` is not yet available, and the calculation of the size must be done outside the constructor once. But again, the implementation can use hidden static member functions to compute recursively once the value of the runtime size specifier of all the encapsulated runtime sized type before construction. Hopefully, in all cases `sizeof T()` does not require to create a temporary object `T()`, only to evaluate the expression(s) belonging to its runtime size specifier(s). An important point is that R.4 and R.5 prevent any change of the runtime size of the elements during the construction of the array `_a` because it is computed only once and substituted in further calls to the elements constructor.

6.1.2 Example

According to the preceding section, the following code should be well-defined:

```
template <typename T>
struct A_tricky {
A_tricky() : _n{rand() % 100} {}
const size_t _n;
T _a[_n];
};

using AAI = A_tricky<A_tricky<int>>;
AAI *aai = new AAI [10];
```

The function `rand()` should be invoked only twice, once for all `aai[]._a[]._n` sharing the same temporary `__tmp_aai_a_n`, and once for all `aai[]._n` sharing the same temporary `__tmp_aai_n`, *in that order* due to the runtime size expressions dependency order.

6.2 Automatic runtime sized objects

Beside that it complicates the stack unwinding mechanism, allowing runtime sized objects with automatic storage duration would make the computation of the address of automatic objects rather complicated and/or slow using linear search or parametrized stack frame. We believe that it is feasible as it should not be more complicated than C99 Variable Length Arrays.

6.2.1 Other storage durations

Runtime sized objects with `static` storage duration could be allocated on the heap during program initialisation and replaced by references in the code.

Runtime sized objects with `thread` storage duration would need to modify the implementation of the Thread Local Storage mechanism. But as for static storage duration, the implementation could rely on the heap for allocating the objects and use references in the code.

7 Acknowledgments

We would like to thank Philippe Canal for his proof reading of this proposal, and his suggestions for some clarification.

References

[1] B. Stroustrup, *"The C++ Programming Language, 4th Edition, C++11"*, Addison Wesley, 2013.

[2] J. Snyder and R. Smith, *"Exploring classes of runtime size"*, doc. N4025, 2014.

[3] S. Du Toit, *"Working Draft, Standard for Programming Language C++"*, doc. N3797, 2013.