Document Number: N4025
Date: 2014-05-23
Reply to: jeff-isocpp@caffeinated.me.uk,
richard@metafoo.co.uk

Exploring classes of runtime size

Jeff Snyder and Richard Smith

May 23, 2014

1 Introduction

During the Chicago meeting, several options were discussed for *Arrays of Runtime Bound*. These alternatives were summarised by Bjarne Stroustrup in N3810 [1]. One of the alternatives discussed, dubbed *Array Constructors*, was to introduce a core language feature that would allow types such as *dynarray* and *bs_array* to be written without special compiler support.

At the Issaquah meeting, J. Daniel Garcia proposed such a language feature, called *Run-time bound array data members*, in N3875 [2]. There was much discussion of this proposal, and of runtime-size types in general, with an overall feeling that it was worth pursuing a core language feature in this area.

This paper proposes an alternative way of allowing classes with runtime-bound array data members (aka *runtime-size types*) in C++ that aims to avoid various issues that were discussed at the Issaquah meeting, as well as exploring the impact that adding runtime-size types would have on the rest of the C++ language.

2 Problems addressed

2.1 Overview

With the introduction of C99 [3], the C language gained *Variable-Length Arrays* (VLAs) of automatic storage duration. These were not adopted into C++11 [4], in part due to concerns over the impact on C++'s type system.

Several attempts have been made to introduce a similar facility in the C++ programming language. A solution was incorporated in the working draft of C++14 in the Bristol meeting (N3691 [5]), along with a corresponding standard library container called *dynarray* (N3532 [6]), but both were then removed from the working draft at the Chicago meeting (N3797 [7]).

At the Issaquah meeting the committee discussed N3875, which proposed a core language feature that required constructors of runtime-size types to be inline. N3875 also acknowledged that this may be seen as very restrictive and proposed a second syntax (*sized constructors*) which included the computation of array data member bounds and initialization of runtime-size subobjects in the constructor's declaration.

Several questions were raised during the discussion of runtime-size types in Issaquah, including:

- Whether requiring constructors to be inline was acceptable or not
- Whether having the implementation resort to heap allocation in some scenarios is acceptable and/or desirable
- Whether being able to place runtime-size types on the heap is necessary, or whether allowing their use only by variables of automatic storage duration is sufficient

2.2 Hard abstraction boundaries

> *"At present, programmers can allocate a substructure behind a hard (i.e. non-inline or externally compiled) abstraction boundary. Proposals must be clear about what happens at hard boundaries, and why that does or does not make programming difficult."* —Lawrence Crowl [8]

Today's C++ ABIs make the assumption that the size of a class is fixed, in that the size of a class is baked into all code that creates an object of that class type. For example, when a call to '*new T*' is compiled, *sizeof(T)* will be determined by the compiler and a call to malloc will be emitted with the result of *sizeof(T)* as its argument. No part of determining the class' size is left until runtime.

This style of ABI cannot work for runtime-size types unless the implementations of the constructors for those types are available when creating new objects, i.e. unless those constructors are inline. To avoid such a restriction on constructors of runtime-size classes, we need a 3-phase approach to creating objects of runtime-size type:

1. Determine how much space the object requires, possibly by making a function call into the translation unit where the constructor is defined
2. Allocate memory
3. Call the constructor

This proposal aims to give the compiler enough information to emit a minimal *size-function* alongside each constructor of each runtime-size type. This would make such an ABI possible, allowing all of the size-determination and construction code to reside behind a hard abstraction boundary.

We believe that this approach minimises the difference between fixed-size and runtime-size types for programmers, making programming with runtime-size types as easy as can reasonably be expected.

2.3 Double-evaluation of expressions not used in size computation

Given the three-phase approach to object creation outlined above, we have to consider what expressions may end up being compiled into the *size-function* as well the constructor, and may therefore be evaluated twice during object creation instead of once.

Ideally, only expressions that are used in the computation of the object size will be evaluated when the *size-function* is invoked. This would give users the same behaviour they see today for expressions that are not used in size computation, i.e. they are only evaluated once.

The *sized-constructors* proposal does not distinguish constructor parameters that are required for size computation from those that are not. This forces the parameter list of the size-function to be the same as the parameter list of the full constructor, which in turn forces the evaluation of expressions used as arguments for those parameters during size computation.

```
// N3875 sized constructors syntax

struct A {
  A(int n, double d) sizeof(v[n]);
  // ...
};

struct B {
  // do the side effects of get_d() occur before or after the memory allocation?
  // is an implementation allowed to evaluate get_d() twice?
  B() sizeof(a{4, get_d()});
  A a;
};
```

3 Design Goals

This proposal aims to maximise integration of runtime-size types with existing language features, and maximise the amount of control the user has over the storage of array bounds and offset computations.

The major differences between this and the sized-constructors proposal from N3875 are as follows:

- Members used to store the bound of a variable-length array data member must be explicitly declared
- Constructor parameters that are involved in size computation must be marked with extra syntax
- No part of the constructors for runtime-size types need to be inline
- Objects of runtime-size type are not limited to having automatic storage duration
- The operator **sizeof** can be applied to variables of runtime-size type, and can be used to determine how large an object would be for a specific set of constructor arguments
- A proposal for how we could support unions and placement new with runtime-size types has been included

This proposal does not include any way to have a trailing array of unspecified bound in a struct, c.f. C99 flexible array members. We consider this to be a distinct feature, which is not mutually exclusive with the feature proposed here.

4 Proposal

4.1 Syntax

We describe a possible syntax below in order to facilitate discussion; we do not wish to spend significant time on syntax discussions at this early stage.

4.2 Declaration of array data members of runtime bound

Declaring a runtime-bound array data member differs from declaring a normal array data member only in that the bound of the array is not a constant expression. Instead, a *bound expression* must be provided. The content of bound expressions is subject to some restrictions.

struct *A* {
    // ... constructors ...
    double *d_array*[*bound expression*];
};

A member variable that is used to store the bound of a runtime-bound array data member (or other information from which the bound can be derived) is called an *array bound data member*, and must be marked with the **sizeof** keyword. Typically, a bound expression will just reference an array bound data member:

struct *B* {
    // ... constructors ...
    const int *d_array_bound* **sizeof**;
    double *d_array*[*d_array_bound*];
};

Bound-expressions and array bound data members are subject to the following restrictions:

- Array bound data members must be of const qualified type or of reference type
- The values of data members other than array bound data members may not be used in a bound expression

- Member functions of the containing object may not be used in a bound expression
- If the bound expression for a runtime-bound array data member *M1* uses the value of an array bound data member *M2* from the same complete object, and &M2 > &M1, then the program is ill-formed, no diagnostic required
- If the bound expression for a runtime-bound array data member *M1* refers (possibly indirectly, through the initializer of an array bound data member) to the address of a subobject *M2* of the same complete object, and &M2 > &M1, then the program is ill-formed, no diagnostic required
- After all array bound data members referenced in a bound expression have been initialized, and before any of those array bound data members have been destroyed, the result of evaluating that bound expression must not change, otherwise behaviour is undefined
- If a bound expression is evaluated for the purposes of size computation "as-if" all referenced array bound data members had been initialized to the values that they will be initialized to during construction, then the result of that evaluation must be the same as the result of evaluating the bound expression during the lifetime of the object, otherwise behaviour is undefined
- Whilst array bound data members and bound expressions may make use of the 'this' pointer and the addresses of some data members, if the bound of any runtime-bound array data member depends upon the absolute value of 'this' (i.e. constructing two objects of the same type with the same arguments at two different addresses can produce two objects with different bounds for one or more of the type's runtime-bound array data members), then the program is ill-formed, no diagnostic required

```
struct C {
    // ... constructors ...
    const int a;
    int b sizeof; // Error, 'b' is 'sizeof' but not const
    const int c sizeof;
    double aa[a]; // Error, 'a' is not declared with 'sizeof'
    double cc[c]; // OK
    double dd[d]; // Error, &d > &dd
    const int d sizeof;
};
```

It may be possible to have no extra syntax on array bound data members and rely on compilers inferring that particular members are array bound data members because they are referenced in bound expressions. We have kept the extra syntax in this proposal for clarity.

4.3 Declaration and definition of constructors

To allow the compiler to generate a *size-function* that performs the minimal computation necessary to determine a not-yet-created object's size, we propose that constructor parameters that are necessary to determine how large the object will be are marked with the **sizeof** keyword. We will refer to such parameters as *array bound constructor parameters*.

Array bound data members will typically be initialized from array bound constructor parameters:

```
struct A {
    A(int n sizeof);
    const int a sizeof;
    double array[a];
};
```

$A::A(\textbf{int}\ n\ \textbf{sizeof})$:
    $a\{n\}$
{}

The *member-initializer* for an array bound data member differs from a normal member initializer in the following ways:

- The values of constructor parameters that are not array bound constructor parameters may not be used
- The values of data members that are not array bound data members may not be used
- Member functions of the object being initialized may not be used
- If the member initializer for an array bound data member *M1* uses the value of an array bound data member *M2* from the same complete object, and &M2 > &M1, then the program is ill-formed, no diagnostic required
- If the member initializer for an array bound data member *M1* refers (possibly indirectly, through the initializer of another array bound data member) to the address of a subobject *M2* of the same complete object, and &M2 > &M1, then the program is ill-formed, no diagnostic required
- The compiler may evaluate member-initializers for array bound data members an unspecified number of times, and may do so in any phase of object creation
- If a member-initializer expression for an array bound data member is evaluated more than once during the creation of an object, and the result of evaluating that expression is not the same all evaluations, behaviour is undefined

4.4 Composition of runtime-size types

Declaring a data member of runtime-size type is syntactically no different from declaring a normal data member. If a class contains a data member of runtime-size type, then that class also has runtime size. Member-initializers for data members of runtime-size are subject to the following restrictions:

- Expressions used as arguments to array bound constructor parameters are subject to the same restrictions as expressions that are used as member-initializers for array bound data members

5 Rationale

In order to support copy-construction, move-construction and destruction of runtime-size types, it is necessary for the bound of each runtime-bound array data member to be stored in the object somehow. One approach to this is for the compiler to introduce a hidden member variable for each runtime-bound array, storing either the bound of the array, its total size, or a pointer to the end of the array. Each of these has its own benefits and drawbacks. This proposal differs from the "implicit variable" approach in two key ways:

- Making bound members explicit gives the user control over the size, position and alignment of the bound member, instead of having a certain size/position/alignment predetermined by the ABI
- Allowing the bound of a runtime-bound array data member to be specified as an expression allows the user to control whether the bound variable stores the actual bound of the array, or some other information that can be used to derive the bound.

The following examples illustrate the flexibility that this approach affords us. A, B, C and D have one runtime-bound array data member each, but use different ways of storing the bound of the array. E and F explore some of the possibilities when multiple runtime-bound array data members are present.

```
struct A {
    A(unsigned int n sizeof) : a{n} {}
    const unsigned int a sizeof; // a stores the bound of aa
    double aa[a];
};

struct B {
    B(unsigned int n sizeof) : b{n*sizeof(double)} {}
    const size_t b sizeof; // b stores the total size of bb
    double bb[b/sizeof(double)];
};

struct C {
    C(unsigned int n sizeof) : c{&cc[0]+n} {}
    double *const c sizeof; // c stores a pointer to end(cc)
    double cc[c - &cc[0]];
};

struct D {
    D(unsigned int n sizeof) : d{reinterpret_cast<char*>(&dd[0]+n)-reinterpret_cast<char*>(this)} {}
    const size_t d sizeof; // d stores the offset of end(dd) from 'this'
    double dd[(d-(reinterpret_cast<char*>(&dd[0])-reinterpret_cast<char*>(this)))/sizeof(double)];
};
```

Note that the constructor of *D* is only valid because there are no runtime-size class members or runtime-bound array data members whose bound expressions reference not-yet-initialized array bound data members between the declarations of *d* and *dd*.

```
struct E {
    E(unsigned int n sizeof) : e{n} {}
    const unsigned int e sizeof; // one bound member is used for two arrays
    float ee1[e];
    double ee2[2*e];
};

struct F {
    F(unsigned int n1 sizeof, unsigned int n2 sizeof) :
        f1{n1},
        f2{reinterpret_cast<char*>(&ff2[0]+n2)-reinterpret_cast<char*>(this)} {}
    const unsigned int f1 sizeof;
    const size_t f2 sizeof; // stores the offset of the end of ff2 from 'this'
    float ff1[f1];
    double ff2[(f2-(reinterpret_cast<char*>(&ff2[0])-reinterpret_cast<char*>(this)))/sizeof(double)];
    // it is necessary to load both f1 and f2 to determine the bound of ff2
    char f3; // but it is not necessary to load f1 to determine the offset of f3 from 'this'
};
```

6 Impact on exiting language features

6.1 sizeof

When used on a runtime-size type, use of the *sizeof* operator is ill-formed. When used on an object of runtime-size type, the *sizeof* operator will return the size of that object. Example:

```
struct A {
    A(int n sizeof) : a{n} {}
    const int a sizeof;
    uint16_t aa[a];
};

void f() {
    A a{42};
    sizeof(A); // ill-formed
    sizeof(a); // == sizeof(const int) + 42 * sizeof(uint16_t) == 4 + 84 = 88
}
```

Additionally, in order to facilitate the use of placement new with runtime-size types, *sizeof* can be applied to an expression that is the direct construction of an object of runtime-size type (see 6.7 for details).

6.2 Default construction and destruction

Given that the bound of all runtime-bound array data members can be computed by the implementation, we see no problems with implementing default construction and destruction of runtime-size types.

Note that it is possible to define a class where the value given to an array bound data member by the defaulted default constructor causes a bound expression to return a nonsensical value—i.e. some value that is either negative or causes implementation limits to be exceeded. The consequences of defining and using such a class would be the same as the consequences of calling operator **new**[] with a nonsensical bound.

6.3 Copy construction and move construction

We see no problems with the generation of implicit copy and move constructors for runtime-size types. However, it is possible to define a class with a runtime-bound array data member where if all array bound data members are copied into the new object, the runtime-bound array data member's bound expression would not yield the same value for the new object as it did for the original. For example, consider an array bound data member which holds a pointer to the end of the array (c.f. struct C in section 5).

We have two main options for handling this—either we define a subset of the possible bound expressions for which we can guarantee that a simple copy/move of the array bound data members is correct, or we accept that the defaulted copy/move constructors can produce an inconsistent object and thus invoke undefined behaviour.

Another concern is that these generated copy/move constructors would perform an element-by-element copy/move of some number of array elements determined at runtime, and this is an operation which is impossible to write explicitly.

This is already an issue for array data members, although it is possible to work around it when the bound is fixed:

```
struct B {
    int bb[4];
//  B(const B& o) : bb{o.bb} {} // invalid, cannot initialize from an array
    B(const B& o) : bb{o.bb[0], o.bb[1], o.bb[2], o.bb[3]} {} // OK, but ugly
};
```

For classes where a simple copy/move of the array bound data member(s) is not sufficient (e.g. struct C in section 5), this is a significant problem. Since the copy/move constructor must be written out explicitly in order to initialize the array bound data member(s) correctly, it is impossible to take advantage of the defaulted copy/move constructor. If the array element type is move-only, then this makes it impossible to implement the move constructor for classes that use a bound expression like the one used by struct C in section 5.

In order to resolve this, we may need to look at supporting copy/move initialization of array types in C++. However, that is beyond the scope of this proposal.

6.4 Copy assignment and move assignment

The issues that affect copy/move construction of runtime-size types also affect copy/move assignment. In addition to those issues, we must must consider what happens when the bounds of the runtime-bound array data members in the objects being assigned from/to differ. If we choose to specify a compiler-generated copy assignment operator, then behaviour would be undefined if they differ.

6.5 Pointer arithmetic

There will be no built-in arithmetic operators for pointers to runtime-size types, except for unary plus. Code that attempts to perform arithmetic on such pointers is ill-formed. The exception for unary plus is made because it is implementable (unlike other pointer arithmetic operations on runtime-size types) and we see no reason to disallow it.

6.6 Unions

To allow the use of runtime-size classes and arrays of runtime bound in unions, we also need to allow runtime-size union types. This opens up some difficult questions such as "what happens when the active member of a runtime-size union is changed?".

It is tempting to simply disallow runtime-size unions. However, *dynarray*, one of the motivating use cases for this core language feature, is a container whose obvious implementation (in terms of this proposal) is a union of an array of runtime bound and a pointer to a heap-allocated array. Therefore, we propose that unions are supported to the extent necessary for this use case.

Specifically, we propose that unions containing one or more runtime-size members differ from normal unions as follows:

- Runtime-size union constructors must initialize a member
- The size of a runtime-size union is the maximum of the runtime size of the member that is initialized at construction and the size of the largest non-runtime-size member
- The active member of a runtime-size union may not be changed

6.7 Placement new

Placement new should work for runtime-size types as it does for other types, since the onus is on the caller of placement new to provide enough memory to construct the object in.

Determining how much memory will be used by placement new is more difficult for runtime-size types than fixed-size types. In order to make use of placement new easier, we need to offer some way to efficiently determine the size of an object before constructing it. In practice, this means exposing a way of calling the *size-function* for a constructor without actually creating a new object.

One approach to this is to specify that **sizeof** can be applied to an expression that is the direct construction of an object of runtime-size type. Use of **sizeof** in this manner would result in a call to the appropriate *size-function*, and the value returned by that call would be returned by **sizeof**. In this context, expressions used as arguments to array bound constructor parameters are evaluated, even though they are subexpressions of an operand of **sizeof**. Expressions used as arguments to other constructor parameters are unevaluated.

```
struct C {
    C(int n sizeof) : c{n} {}
    const int c sizeof;
    uint16_t cc[c];
};

void f() {
```

```
    auto sz = sizeof(C{3}); // does not actually construct a temporary of type
                            // C, instead only calls the size function
                            // corresponding to C(int)
}
```

6.8 Global variables

We believe that global objects of runtime-size types can be supported, but this area needs more analysis. We welcome feedback from implementors regarding this.

6.9 Arrays

Arrays of objects of runtime-size type are ill-formed.

6.10 Templates

There are no special interactions between templates themselves and runtime-size types that need consideration, as far as we are aware.

However, certain constructs that are currently valid for all types will become ill-formed with the introduction of runtime-size types (c.f. *sizeof*, pointer arithmetic, arrays). As Jens Maurer points out [9], template authors may wish to enable, disable or specialise their templates for runtime-size types, and we should provide a type trait to support this. Such a trait could be implemented without special compiler support via SFINAE on **sizeof**.

Acknowledgements

This proposal evolved from the *sized constructors* proposal by J. Daniel Garcia and Xin Li, and was influenced by conversations with Daveed Vandevoorde as well as email discussions with Jens Maurer and Lawrence Crowl.

References

[1] Bjarne Stroustrup. Alternatives for Array Extensions. Working paper N3810, ISO/IEC JTC1/SC22/WG21, October 2013.

[2] J. Daniel Garcia and Xin Li. Run-time bound array data members. Working paper N3875, ISO/IEC, January 2014.

[3] ISO/IEC JTC1/SC22/WG14. Programming Languages – C. ISO Standard ISO/IEC 9899:1999, ISO/IEC, December 1999.

[4] ISO/IEC JTC1/SC22/WG21. Programming Languages – C++. ISO Standard ISO/IEC 14882:2011, ISO/IEC, November 2011.

[5] ISO C++ Standards Committee. Working Draft, Standard for Programming Language C++. Working Draft N3691, ISO/IEC JTC1/SC22/WG21, July 2013.

[6] Lawrence Crowl and Matt Austern. C++ Dynamic Arrays. Working paper N3662, ISO/IEC JTC1/SC22/WG21, April 2013.

[7] ISO C++ Standards Committee. Working Draft, Standard for Programming Language C++. Working Draft N3797, ISO/IEC JTC1/SC22/WG21, October 2013.

[8] Lawrence Crowl. *[c++std-ext-14769] Re: Runtime-sized types (dynarray / bs_array revisited)*, April 2014. http://accu.org/cgi-bin/wg21/message?wg=ext&msg=14769.

[9] Jens Maurer. *[c++std-ext-14818] Re: Runtime-sized types (dynarray / bs_array revisited)*, April 2014. http://accu.org/cgi-bin/wg21/message?wg=ext&msg=14818.