Document Number: N3875
Date: 2014-01-16
Reply to: josedaniel.garcia@uc3m.es

Run-time bound array data members

J. Daniel Garcia and Xin Li
Computer Architecture Group
University Carlos III of Madrid

January 20, 2014

1 Introduction

During the Chicago meeting several options were discussed for *Array of Runtime Bounds*. These alternatives are well summarized in N3810 [1].

This paper explores into more detail the last option mentioned in N3810 (named there as *array constructors*). The goal is to provide a general way for defining automatic storage arrays (i.e. stack allocated or side-stack allocated) whose size is not known at compile time.

2 Problem

Variable Length Arrays (VLAs) have existed in C since C99 [2]. However, the introduction of VLAs in C++ standard as they are, has been identified as problematic.

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 [3]). At the Chicago meeting the committe decided that this solution was controversial and this part was removed from the working draft (N3797 [4]). The committee decided that a separate technical report on *array extensions* should address this issue.

2.1 Runtime-sized arrays with atuomatic storage

As stated by Jens Maurer in *Runtime-sized arrays with automatic storage duration* series (N3366 [5], N3412 [6], N3467 [7], N3497 [8], N3639 [9]) there are situations where a *user wishes to allocate a local array whose size is not known at compile-time, but at runtime only.*

This kind of uses can be be handled by a `std::vector` usage. However, such approach may lead to a serious performance penalty derived from dynamic memory allocation and deallocation.

```
std::string f(int n) {
  std::vector<int> v(n); // std::vector allocation
  std::iota(v.begin(), v.end(), 1);
  std::string result;
  for (auto x : v) {
    result += std::to_string(x);
  }
  return result;
} // std::vector deallocation
```

The use of a run-time sized array with automatic storage allows to avoid dynamic memory allocation by using automatic storage.

```
std::string f(int n) {
  int v[n]; // run-time sized array with automatic storage
  std::iota(v, v + n, 1);
  std::string result;
  for (int i=0; i<n; ++i) {
```

```
    result += std::to_string(x);
  }
  return result;
}
```

This approach promotes the use of *pointer-plus-size* interface when arrays are passed around.

```
std::string g(int * v, std::size_t n) {
  std::string result;
  for (std::size_t i=0; i!=n; ++i) {
    result += std::to_string(v[i]);
  }
  return result;
}

std::string f(int n) {
  int v[n];
  std::iota(v, v + n, 1);
  return g(v, n);
}
```

Stroustrup lists in N3810 [1] reasons why that interface promotion is not desirable including:

- The number of elements of a run-time sized array is not available and the user needs to *remember* it.
- The type used to access elements can implicitly change after the array has decayed to a pointer.

2.2 *dynarray*

A class providing a dynamic array was proposed in N3532 [10] and adopted into the working draft as N3662 [11]. However, this was not first time such a class was proposed dating back to N2648 [12] in 2008.

While the proposal claims that a *pure-library* implementation is possible, such an implementation would be probably based on dynamic memory allocation. To provide stack allocation of automatic variables compiler magic is needed by giving the compiler the freedom of implementing construction and destruction of *dynarray* directly.

While the border between the library and the compiler has been crossed before (e.g. `std::initializer_list`) we believe alternate solutions can be used here.

In any case, it is true that `dynarray` solves the interface problems that run-time sized arrays would be promoting and better code can be written.

```
std::string g(const dynarray<int> & v) {
  std::string result;
  for (auto x : v) {
    result += std::to_string(x);
  }
  return result;
}

std::string f(int n) {
  dynarray<int> v{n}; // Where is this allocated?
  std::iota(v.begin(), v.end(), 1);
  return g(v);
}
```

However, a more serious concern is that the user has no guarantee whether a `dynarray` object will allocated in automatic or dynamic storage. **This is specially important for applications where performance needs to be predictable**.

2.3 *bs_array*

In N3810 [1] a minimal *basic stack-allocated* array is proposed. The idea is that this class relates to an array with dynamic extent as `std::array` relates to an array with static extent.

Here is a possible outline taken from N3810.

```
template<class T>
class bs_array { // basic stack-allocated array
  using value_type = T;

  bs_array(int n); // n elements
  // default copy, no move, maybe copy to vector/other_containers

  T& operator[](int i);
  const T& operator(int i) const;
  T& at(int i);
  const T& at(int i) const;

  T* begin() { return a; }
  const T* begin() const { return a; }
  T* end() { return a+n; }
  const T* end() const { return a+n; }

  int size();

  T* data();

private:
  T a[n]; // for exposition only, a is stack allocated
};
```

This approach addresses our concern about allocation as `bs_array` is guaranteed to be allocated on the stack. Now, the interface promotion problem is solved at the same time that storage allocation is predictable.

```
std::string g(const bs_array<int> & v) {
  std::string result;
  for (auto x : v) {
    result += std::to_string(x);
  }
  return result;
}

std::string f(int n) {
  bs_array<int> v{n}; // allocated on the stack
  std::iota(v.begin(), v.end(), 1);
  return g(v);
}
```

However, `bs_array` shares with `dynarray` the need for compiler support. **While this is acceptable, we beleive that a more general solution can be achieved at the language level**.

3 Proposal: run-time size bounds

In this paper we present a proposal of what N3810 [1] names as the *array constructors* proposal which was originally suggested by J. Daniel Garcia in the *-ext* mailing list. Ellaborations on this idea were later made by others including Bjarne Stroustrup, Daveed Vandevoorde and Lawrence Crowl.

The core idea of this proposal is to allow a class to have one or more run-time bound array data-members. Those array data members will be of unspecified number of elements. The constructor is responsible for setting the size upon object construction.

We present two variations of this proposed ideas:

- **Inline constructors**: Constructors are required to be inline and the size of array data members is set in the constructor initialization list.
- **Sized constructors**: Constructors specify the size of run-time bound array data-members as part of their declaration.

3.1 Inline constructors proposal

We present here the inline constructors proposal, which may be seen as more restrictive.

3.1.1 Run-time bound array data members

We propose a new syntax for introducing array data members of unspecified size. The size will be specified at construction time.

```
class my_vec {
public:
  // ...
private:
  int z;
  double[] v; //rtb array
  int [] w; // rtb array
};
```

While, the initial proposed syntax was the (perhaps more natural) `T var[]`, it was noted that this might clash with C's empty array bounds.

Any class that has one or more run-time bound array data members becomes a *run-time size bound class*.

3.1.2 Constructors

A *run-time size bound class* needs to specify the size of its array data members upon construction.

Array sizes specifications need to be known from any constructor call site. Thus, we require that a run-time size bound class has all its constructors declared `inline`. Any of these constructors needs to specify the size of each *run-time bound array data member*. We propose the following syntax:

```
class my_vec {
public:
  my_vec(int n) : z{n}, v[n]{}, w[n*2]{} {/*...*/}
  my_vec() : z{0}, v[2]{}, w[4]{} {/*...*/}
  // ...
private
  int z;
  double[] v;
  int [] w;
};
```

Another option suggested by Daveed and collected in N3810 was describing underlying storage within constructor declaration. This is illustrated in the following example taken from N3810:

```
struct MyArray {
  MyArray(int n) double storage[n];
  int size() const { return s; }
private:
  MyPtr<double> p;
  int s;
};

MyArray::MyArray(int n): s(n), p(storage) {}
```

We think that the major problem of such syntax is the case where a class needs to have more than one RTB data member. We recognize that this is not a frequent case. Besides, we think that this suggestion makes code more difficult to read by users.

It has also been suggested to simplify the syntax of RTB array data members to the following:

```
class my_vec {
public:
  my_vec(int n) : z{n}, v{n}, w{n*2} {/*...*/}
  my_vec() : z{0}, v{2}, w{4} {/*...*/}
  // ...
private
  int z;
```

```
  double[] v;
  int []  w;
};
```

While this syntax seems simpler, it may be considered confusing that the initializer in this case is the size of the array and not a value. Besides, it also prevents a user of potentially initializing a small array during construction.

```
class vec {
public:
  vec() : v[2]{1.0, 2.0} {/*...*/}
  // ...
private
  double[] v;
};
```

3.1.3 Implementing *bs_array*

One of the advantages of *run-time size bound classes* is that they allow a really *pure-library* implementation of `bs_array`.

```
template <class T>
class bs_array {
public:
  using value_type = T;

  bs_array(int n) : sz{n}, v[n]{} {}
  bs_array(const bs_array & a) = default;
  bs_array(bs_array &&) = delete;

  // Unchecked access
  T & operator[](int i) { return v[i]; }
  const T & operator[](int i) const { return v[i]; }

  // Range checked access
  T & at(int i);
  const T & at(int i) cosnt;

  // begin/end
  T * begin() { return v; }
  const T * begin() const { return v; }
  T * end() { return v+sz; }
  const T * end() { return v+sz; }

  int size() const { return sz; }
  T * data() { return v; }

private:
  int sz;
  T[] v;
};
```

We propose that a class like this is standardized. While we are not proposing a specific name for such class, alternate names could be `auto_array`, `stack_array`, `dynarray` or even `autodynarray`.

3.1.4 Initialization options

The syntax needs to support several options for initialization as a balance needs to be made between performance and flexibility. For this reason, we have kept separate the specification of the data member size and its value in the constructor.

If no values are provided in the initializer for the array data member, each element in the array is *value-initialized*.

```
class vec {
public:
  vec(int n) :
    v[n]{}, // v[i] = 0.0
    w[n]{}  // w[i] = point{}
  {}
private:
  double[] v;
  point[] w;
};
```

If braces are not present after the size specification, each element is *default-initialized* (including no initialization for scalars):

```
class vec {
public:
  vec(int n) :
    v[n], // v[i] is not initialized
    w[n]  // w[i] = point{}
  {}
private:
  double[] v;
  point[] w;
};
```

It is also possible to provide an *initalizer-list* in the data member initializer:

```
class vec {
public:
  vec() :
    v[2]{-1.0, 2.5},
    w[2]{ {1.0,1.0}, {2.5,2.5} }
  {}
private:
  double[] v;
  point[] w;
};
```

If the *initializer-list* provides less initializers than the array data member size, the rest of elements are value initialized.

However, if the number of initializers exceed the number of elements, exceeding initializers are discarded. Note that this condition cannot be checked until run time making any compile time diagnostic not viable. An alternate solution here, could be throwing an exception, but we are not currently proposing this alternative.

```
class vec {
public:
  vec(int n) :
    v[n]{1.0, 2.0, 3.0}
  {}
private:
  double[] v;
};

vec a{4}; // a == {1.0, 2.0, 3.0, 0.0}
vec b{2}; // b == {1.0, 2.0}
```

3.1.5 Run-time size bound objects as data members

If a data member of a class is of a type that is a *run-time size bound class*, the the new class is also a *run-time size bound class* and the same restrictions are applied to it. In particular, its constructor also needs to be defined inline.

```
class A {
public:
```

```
  A(int n) : v[n]{} {}
private:
  double[] v;
};

class B {
public:
  B() : a{4} {}
private:
  A a;
};
```

The same applies to a inheritance.

```
class A {
public:
  A(int n) : v[n]{} {}
private:
  double[] v;
};

class B : public A {
public:
  B() : A{4} {}
};
```

3.1.6 Contexts of use

We propose that the only use of a *run-time size bound class* object is as an automatic variable. In particular, we propose to ban any use implying dynamic memory.

```
class A {
public:
  A(int n) : v[n]{} {}
private:
  double[] v;
};

void f() {
  A a{4}; // OK;
  A * p = new A{4}; // Error
  vector<A> w; // Error
}
```

3.1.7 *sizeof*

Operator `sizeof` is not supported on any type or object of a *run-time size bound class*.

3.1.8 Non-inlined constructors

Imposing the (transitive) requirement that constructors need to be inlined may be seen as very restrictive. We think that the most common use of this feature is defining automatic storage objects at function scope level. Thus, this restriction may be not so critical.

However, an alternate approach could be allowing that a constructor body is first inlined and then its body is refined.

```
class vec {
public:
  vec(int n) : v[n]{} {} // inline definition
private:
  double[] v;
};
```

```
// Definition somehwere else
vec::vec(int n) // Member initiation not allowed here
{
  do_some_additional_thing();
}
```

In this case the inline definition can be used for computing the size of the object at the construction call site. However, after that the constructor body (if one is existing) may be called. For the sake of avoiding redundancy in this case, we do not allow to repeat the member initiation list.

3.1.9 Size determination

The current proposal implies that the size of any *run-time bound array data member* can be derived from the environment where the inline constructor is defined.

In particular, this proposal does not allow the following example (slightly modified from an example provided by Lawrence Crowl).

```
// a.h
struct A {
  bs_array<double> storage;
  A(int n);
};

// a.cc
  extern int config;
  A::A(int n) : storage(n*config) {}
```

As `struct A` has a data member which is a *run-time size bound class* (the `bs_array`), it is considered itself to be also a *run-time size bound class* and the same restrictions apply to it. Thus, it needs that its constructor is defined inline.

In this proposal we have not addressed this problem. We think it should be first clarified if this use case is important enough to be solved.

Possible solutions are:

- Require that the variable (i.e. `config`) is visible at the point of definition of the inline constructor.
- Modify the proposal so that it is not required that the constructor is defined inline.

3.1.10 Multi-dimensional data members

This proposal initially tries to support single-dimension *run-time bound* data members. However, if needed it could be extended to support multiple dimension *run-time bound* data members.

```
class stack_matrix {
public:
  stack_matrix(int r, int c) : v[r][c]{} {}
  // ...
private:
  double[][] v;
};
```

3.2 Sized constructors

Restricting constructors of *run-time size bound classes* to be inline have been seen as problematic. We have outlined some concerns about this issue in the previous section.

Here we outline an alternative where the size of run-time bound array data members is part of the constructor specification. This proposal is based in (what we think is) a simplification over an original idea proposed by Daveed Vandevoorde.

3.2.1 Constructors with size specification

A *run-time size bound class* needs to specify the size of its array data members upon construction. We propose to make such specification part of the constructor declaration.

For this purpose we propose to add to the constructor syntax a *sizeof-specifier*.

```
class my_vec {
public:
  my_vec(int n) sizeof(v[n], w[2*n]);
  my_vec() sizeof(v [2], w [4]);
private:
  int z;
  double[] v;
  int [] w;
};

my_vec::my_vec(int n) : z{n} {}

my_vec::my_vec() : z{0} {}
```

We think this is simpler than the storage specification suggested by Daveed Vandevoorde and quoted in N3810. With that syntax the above example would be:

```
class my_vec {
public:
  my_vec(int n) double storage_v[n], int storage_w[2*n];
  my_vec() double storage_v[2], int storage_w [4];
private:
  int z;
  double * v;
  int * w;
};

my_vec::my_vec(int n) : z{n}, v{storage_v}, w{storage_w} {}

my_vec::my_vec() : z{0}, v{storage_v}, w{storage_w}{}
```

We see storage specification option to look more verbose. Besides, it implies to manage two names per array (one for the storage and another one for the pointer).

3.2.2 Implementing *bs_array*

An implementation of `bs_array` is possible with sized constructors.

```
template <class T>
class bs_array {
public:
  using value type = T;

  bs_array(int n) sizeof(v[n]);
  bs array(const bs array & a) = default;
  bs array(bs array &&) = delete;

  // Unchecked access
  T & operator[](int i) { return v[i]; }
  const T & operator[](int i) const { return v[i]; }

  // Range checked access
  T & at(int i );
  const T & at(int i) cosnt;

  // begin/end
  T * begin() { return v; }
  const T * begin() const { return v; }
  T * end() { return v+sz; }
  const T * end() { return v+sz; }

  int size () const { return sz; }
  T * data() { return v; }
```

```
private:
  int sz;
  T[] v;
};
```

3.2.3 Initialization options

Initialization for *run-time bound array data members* is performed at the constructor through initialization lists as it would have been done for any other array data member.

```
class vec {
public:
  vec(int n) sizeof(v[n], w[n]);
private:
  double[] v;
  point[] w;
};

vec::vec(int n) : v{}, w{} {} // v[i] == 0.0, w[i] == point{}
```

It is possible to *default-initialize* any array data-member by not including it in the initialization list.

```
vec::vec(int n) {} // v and w are default- initialized
```

It is also possible to provide *initializer-lists* for performing array initialization. If the list provides less initializers than the array data member size, the rest of elements are *value-initialized*. If the list provides more elements than the array data member size exceeding elements are discarded.

```
class myvec {
public:
  myvec() sizeof(v[4], w[2]);
  myvec(int n) : sizeof(v[n], w[n]);
private:
  double[] v;
  float [] w;
};

myvec() :
  v{1.0, 2.0}, // {1.0, 2.0, 0.0, 0.0}
  w{1.5, 2.5, 3.5} // {1.5, 2.5}. Third value (3.5) is discarded
{}

myvec{int n) :
  v{1.0, 2.0 }, // Discarding if n<2
  w{1.5, 2.5, 3.5 }, // Discarding if n<3
{}
```

3.2.4 Run-time size bound objects

If a class has a data member which is of a *run-time size bound class* it becomes itself a *run-time size bound class*. In that case we require that the call to the constructor of such sub-object is specified in the *sizeof* specifier of the constructor.

```
class A {
public:
  A(int n) sizeof(v[n]);
private:
  double[] v;
};

class B {
public:
  B() sizeof(a{4});
  B(int m) sizeof(a{m});
```

```
private:
  A a; // A is run-time size bound -> B is run-time size bound
};

B::B() {}
B::B(int m) {}
```

This is much less restrictive that requiring the inlining of the full constructor. This requirement allows for size determination from the call site.

A similar restriction applies to inheritance:

```
class A {
public:
  A(int n) sizeof(v[n]);
private:
  double[] v;
};

class B : public A { // A is run-time size bound -> B is run-time size bound
public:
  B() sizeof(A{3});
  B(int m) sizeof(A{m});
};

B::B() {}
B::B(int m) {}
```

3.2.5 Contexts of use

As with inline constructors alternative, we propose that the only use of a *run-time size bound class* object is as an automatic variable.

3.2.6 *sizeof*

As with inline constructors alternative, we do not plan to support operator `sizeof`.

3.2.7 Non-inlined constructors

In this option there is no requirement to force constructors to be *inline*. This allows constructors to be defined elsewhere and to have complex bodies when needed.

```
class vec {
public:
  vec(int n) sizeof(v[n]);
private:
  double[] v;
};

// Definition somewhere else
vec::vec(int n) : v{}
{
  do_some_additional_thing();
}
```

3.2.8 Size determination

This proposal does not allow writing the example from Lawrence Crowl already cited in Section 3.1.9.

3.2.9 Multi-dimensional data members

Multi-dimensional data members can also be expressed with this alternative.

```
class stack_matrix {
public:
  stack_matrix(int r, int c) sizeof(v[r][c]);
private:
  double[][] v;
};
```

Acknowledgments

Daveed Vandevoorde reviewed a previous draft of this paper and provided very useful comments. We have also benefited from e-mail discussions with Bjarne Stroustrup and Lawrence Crowl.

References

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

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

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

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

[5] Jens Maurer. Runtime-sized arrays with automatic storage duration. Working paper N3366, ISO/IEC JTC1/SC22/WG21, February 2012.

[6] Jens Maurer. Runtime-sized arrays with automatic storage duration (revision 2). Working paper N3412, ISO/IEC JTC1/SC22/WG21, September 2012.

[7] Jens Maurer. Runtime-sized arrays with automatic storage duration (revision 3). Working paper N3467, ISO/IEC JTC1/SC22/WG21, October 2012.

[8] Jens Maurer. Runtime-sized arrays with automatic storage duration (revision 4). Working paper N3497, ISO/IEC JTC1/SC22/WG21, January 2013.

[9] Jens Maurer. Runtime-sized arrays with automatic storage duration (revision 5). Working paper N3639, ISO/IEC JTC1/SC22/WG21, April 2013.

[10] Lawrence Crowl and Matt Austern. C++ Dynamic Arrays. Working paper N3532, ISO/IEC JTC1/SC22/WG21, March 2013.

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

[12] Lawrence Crowl and Matt Austern. C++ Dynamic Arrays. Working paper N2648, ISO/IEC JTC1/SC22/WG21, May 2008.