Document Number: N4294
Date: 2014-11-23
Reply to: josedaniel.garcia@uc3m.es

Arrays of run-time bounds as data members

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

1 Introduction

This paper revisits the proposal from N3875 [1] and its associated problems and tries to find a solution to run-time bounds arrays based on *array constructors*. The final goal is to provide a generalized way of 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 committee 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.

Previous attempts to run-time sized arrays include: run-time sized arrays with automatic storage, `dynarray`, and `bs_array`. A summary of such ideas can be found in N3875 [1].

3 Proposal

In this paper we present a simplified proposal based on N3875 [1].

The core idea of this proposal is to allow a class to have a data member of unspecified size and to provide mechanisms to separate size specification from the rest of construction.

3.1 Arrays of run-time bound as data members

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

```
class myvec {
public:
  // ...
private:
  int sz;
  double[] v; //rtb array
};
```

3.2 Constructors

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

This paper proposes a new type of constructor: an array constructor. An array constructor specifies the size of the array data member of unspecified size and my forward a call to the corresponding non-array constructor. The array constructor must always be inline, while the corresponding non-array constructor does not need to be inline.

```
class myvec {
public:
  myvec[]() : v[2], myvec{} {} // Array constructor must be inline
  myvec[](int n) : v[n], myvec{n} {} // Array constructor must be inline

  myvec(); // Need not be inline
  myvec(int n); // Need not be inline

  // ...
private
  int sz;
  double[] v;
};

myvec::myvec() : sz{2} { /* ... */ }
myvec::myvec(int n) : sz{n} { /* ... */ }
```

3.3 Single constructor

An array constructor can be used as single constructor instead of forwarding to a non-array constructor.

```
class myvec {
public:
  myvec[](int n) : sz{n}, v[n] {} // Needs to be inline

  // ...
private:
  int sz;
  double[] v;
};
```

Please, note that not using forwarding construction imposes the additional requirement of being able to inline the constructor. When this is no desirable, still two constructors (an array version and a non-array version) should be used.

3.4 Implementing *bs_array*

The proposed solution allows the implementation of `bs_array` (where bs stands for *bike shed*) as a *pure-library* solution.

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

  bs_array []( int n) : v[n], bs_array{n} {}
  bs_array(int n) : sz{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) const;

  // 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;
};
```

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

3.5 Run-time size bound objects as data members

A class that has one member of an array type of unspecified size is a *run-time size bound object* and its size cannot be determined at compile time as the object size depends on the array size.

A class that has a run-time size bound data member becomes itself also into a run-time size bound class. Such a class may have an array constructor.

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

class B {
public:
  B[](int n) : x{n*1.5}, a{n} {}
private:
  double x;
  A a;
};
```

In general, a class can only have a run-time size bound data member (or an array of unspecified size data member) as its last data member.

Similar restrictions apply to inheritance.

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

class B : public A {
public:
  B[](int n) : x{n*1.5}, A{n} {}
private:
  double x;
};
```

A class inheriting from a run-time sized class cannot have subclasses that add new data members.

In case of multiple inheritance, only the most right base class can be a run-time sized class.

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

```
  double[] v;
};

class B { /* ... */ };

class D1 : public A, public B { /* ... */ }; // Ill-formed
class D2 : public B, public A { /* ... */ }; // Ill-formed
```

3.6 Contexts of use

A run-time size bound object can only be used as an automatic variable. In particular, this paper proposes to ban any use implying direct or indirect use of dynamic memory. Besides an array of such objects is also forbidden.

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

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

When a valid *run-time size bound* object is created both its array constructor and the non-array constructor are used. The array constructor is used to determine the size of the object. The the non-array construcctor is used to initialize the object.

3.6.1 *sizeof*

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

Firstly, on types `sizeof` makes not much sense as different object of such types could require different amounts of storage, due to a member of unspecified size. Secondly, implementing `sizeof` on objects would lead to different objects of the same type having a different size, and requiring additional implementation overhead.

Finally, the size of an object is a constant value and should have the same value for every object of a type. For a run-time size bound class this might not be the case. For the same reason, an array of *run-time size bound* objects is not possible, as subscripting could not be implemented for random access at constant time.

3.7 Parameter passing and return values

Parameters of *run-time size bound classes* can only be passed as parameters to functions (or returned) when such passing mode does not imply an object copy. This restriction avoids problems with size determination.

```
void f(bs_array<int> & a); // OK
bs_array<double> & g(const bs_array<int> & a); // OK
bs_array<char> h(); // Ill-formed
void i(bs_array<long> a); // Ill-formed
```

3.8 Inlining versus non inlining

In contrast with N3875, this proposal does not require constructors to be generally inlined. The only constructor that needs to be inlined is the array constructor. Others may be inlined or not.

3.9 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);
}; // Error A needs an inlined array constructor

// a.cc
  extern int config;
  A::A[](int n) : storage{n*config} {} // Not inline: Ill-formed

// b.cc
void f() {
  A x{10}; // Ill-formed.
  //...
}
```

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 it needs an inlined array constructor.

Acknowledgments

Many thanks to Bjarne Stroustrup and Gabriel Dos Reis for providing feedback on earlier versions of this paper.

References

[1] J. Daniel Garcia and Xin Li. Run-time bound array as data members. Working paper N3875, ISO/IEC JTC1/SC22/WG21, January 2014.

[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.