V8: C++ layout FixedArray

Attention: Externally visible, non-confidential

Authors: Leszek Swirski

Status: Under review

Created: Feb 21, 2024 / **Last updated:** Feb 21, 2024

LGTMs needed

Name	Write (not) LGTM in this row
<your name here>	

Background

Porting FixedArray to C++ layout presents some difficulties with the existing templated shared TaggedArrayBase as introduced by Refactoring FixedArray & Friends.

The current approach is to use a CRTP base class to define the shared methods, a Shape class to define layout (both of the fixed header, like length, and any additional fields), and a FixedArray class to define methods:

```
C/C++
template<typename Derived, typename Shape, typename Super>
class TaggedArrayBase : public Super {
 public:
  using ElementT = Shape::ElementT;

  int capacity() { return READ_SMI(*this, Shape::kCapacityOffset); }
  Tagged<ElementT> get(int i);
  ...
};
```

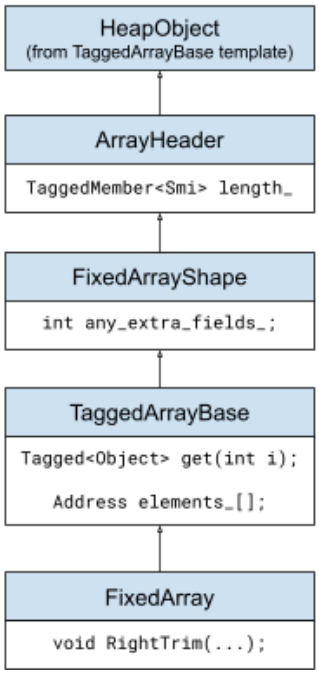
```
class FixedArrayShape final : public AllStatic {
 public:
  static constexpr int kElementSize = kTaggedSize;
#define FIELD_LIST(V)                                                    \
  V(kCapacityOffset, kTaggedSize)                                        \
  V(kMaybeSomeOtherFields, kTaggedSize)                                  \
  V(kUnalignedHeaderSize, OBJECT_POINTER_PADDING(kUnalignedHeaderSize)) \
  V(kHeaderSize, 0)
  DEFINE_FIELD_OFFSET_CONSTANTS(HeapObject::kHeaderSize, FIELD_LIST)
#undef FIELD_LIST
};

class FixedArray : TaggedArrayBase<FixedArray, FixedArrayShape, HeapObject> {
 public:
  void RightTrim(Isolate* isolate, int new_capacity);
  ...
}
```

When trying to translate this to C++ layout presents the difficulty that there is no good owner for the fields defined by the shape. The Shape class is not in the type hierarchy, so it can't define the fields; the FixedArray is defined *after* the TaggedArrayBase it inherits from, so the TaggedArrayBase can't access fields defined on the FixedArray – in particular, it wouldn't be able to access the size of the FixedArray, in order to figure out the offset of the elements of the array.

Proposal

I propose that we restructure this to make the Shape class with a magic templated class that injects itself into the middle of the type hierarchy, which defines any extra fields needed. Specifically, I propose the hierarchy:

```
class FixedArray : public TaggedArrayBase<FixedArray, FixedArrayShape, HeapObject>
```



ArrayHeader will define the fixed header shared between all arrays, FixedArrayShape (actual name TBD) will become a templated class on its base class, and TaggedArray will inject it between the header and the data. The template magic for this would be something like this:

C/C++

```
template<typename Base>
class ArrayHeader : public Base {
 public:
  int capacity() { return capacity_; }
 private:
  int capacity_;
};

template<typename Derived, template<typename> typename Shape, typename Base>
class TaggedArrayBase : public Shape<ArrayHeader<Base>> {
 public:
  using ElementT = Shape<ArrayHeader<Base>>::ElementT;

  Tagged<ElementT> get(int i) { return data_[i].untag(); }
```

```
 private:
  TaggedMember<ElementT> data_[];
};

template<typename Base>
class FixedArrayShape: public Base {
 public:
  using ElementT = Object;

  void RightTrim(Isolate* isolate, int new_capacity);
 private:
  // `int capacity_` will be defined by ArrayHeader before this field.
  int some_extra_field_;
  // `TaggedMember<ElementT> data_[]` will be defined by TaggedArrayBase after
this.
};

class FixedArray : public TaggedArrayBase<FixedArray, FixedArrayShape,
HeapObject> {
 public:
  void RightTrim(Isolate* isolate, int new_capacity);
  ...
}
```

~~We could wrap this up in a helper macro which defines both the shape and the typedef:~~

```
C/C++
#define TAGGED_ARRAY(NAME, BASE)                                 \
    NAME##_Impl {                                                \
        public:                                                  \
          template <typename Base> class Class;                  \
    };                                                           \
    using NAME = TaggedArray<NAME##_Impl::Class, BASE>; \
    template <typename Base>                                     \
    class NAME##_Impl::Class : public Base

class TAGGED_ARRAY(FixedArray, HeapObject) {
  public:
    using ElementT = Object;
```

```
  void RightTrim(Isolate* isolate, int new_capacity);
  static Handle<FixedArray> RightTrimOrEmpty(Isolate* isolate,
                                             Handle<FixedArray> array,
                                             int new_length);

 private:
  int extra_field_;
};
```

This has the nice side benefit that the methods, configuration (e.g. ElementT) and layout of FixedArray are all in the same class (just not the FixedArray class, since there isn't one). The macro & template magic is hopefully manageable.