`rarray`: Design of a Reference-Counted Multidimensional Arrays for C++

Ramses van Zon

January 2024

1 Introduction

While C and thus C++ has some support for multidimensional arrays whose sizes are known at compile time, the support for arrays with sizes that are known only at runtime, is limited. For one-dimensional arrays, C++ has a reasonable allocation and deallocation constructs in the operators `new` and `delete` in the standard. A standard way to allocate a one-dimensional array is as follows:

```
int n = 1000;
float* a = new float[n];
a[40] = 2.4; // as an example on how to use this, access element at offset 40
delete[] a;
```

It is important to note that this code also works if `n` was not known yet at compile time, e.g., if it was passed instead as a function argument or read in as input.

This style of allocation with a "raw" pointer is discouraged in C++ in favor of using "smart" pointers, which is possible since the C++17 standard:

```
int n = 1000;
std::unique_ptr<float[]> a(new float[n]);
a[40] = 2.4; // as an example on how to use this, access element at offset 40
// a gets deallocated automatically, or one can explicitly call a.reset(nullptr)
```

Automatic deallocation happens when the variable `a` goes out of scope. A unique pointer cannot be copied. Instead of `unique_ptr` one can use `shared_ptr`, which can be copied and keeps a reference counter to know when to deallocate the memory. Then, deallocation happens when the life time of all copies of the `shared_ptr` has ended.

In the above code snippets, the `new` construct and the `std::unique_ptr/std::shared_ptr` assign the address of the array to a pointer. These pointers do not remember their size, so they are not really an 'array'. The standard C++ library does provide a dynamically allocated one-dimensional array that remembers its size, in the form of the `std::vector`, e.g.

```
int n = 1000;
std::vector a(n);
a[40] = 2.4; // as an example on how to use this, access element at offset 40
// a gets automatically deallocated, or one can explicitly call a.clear()
```

Multi-dimensional runtime-allocated arrays are currently not supported yet by C++, but there is a proposal for a non-owning multidimensional array in the C++23 standard, and C++26 may have an owning multidimenstional array.

To handle these kinds of arrays in C++, the (early) textbook solution for multidimensional arrays that are dynamically allocated during runtime, would be as follows (here for a three-dimensional array of `dim0`×`dim1`×`dim2`)

```
float*** A;
A = new float**[dim0];
for (int i=0;i<dim0;i++) {
  A[i] = new float*[dim1];
  for (int j=0;j<dim1;j++)
    A[i][j] = new float[dim2];
}
A[1][2][3] = 105; // as an example
```

Apart from the fact this will soon be obsolute, drawbacks of this solution are:

- the elements are not stored contiguously in memory, making this multi-dimensional array unusable for many numerical libraries,
- one has to keep track of array dimensions, and pass them along to functions,
- the intermediate pointers are non-const, so the internal pointer structure can be changed whereas, conceptually, `a` ought to be of type `float*const*const*`.

At first, there seems to be no shortage of libraries to fill this lack of C++ support for dynamic multi-dimensional arrays, such as

- Blitz++;
- The Boost Multidimensional Array Library (`boost::multiarray`);
- Eigen;
- Armadillo
- Nested `vectors` from the Standard Template Library; and
- Kokkos's reference implementation of the C++23 mdspan template.

These typically do have some runtime overhead compared to the above textbook solution, or do not allow arbitrary ranks. In contrast, the purpose of the rarray library is to be a minimal interface for runtime multidimensional arrays of arbitrary rank with *minimal to no performance overhead* compared to the textbook solution. From the above list of library solutions, only the implementation in Kokkos of the non-owning mdspan has virtually no overhead.

Rarray aims to be simpler; all it takes to create a three-dimensional **Example:**

```
#include <rarray>
...
   rtensor<float> A(dim0,dim1,dim2);
   A[1][2][3] = 105; // as an example
...
```

Design Points of rarray

1. To have dynamically allocated multidimensional arrays that combine the ease and convenience of automatic C++ arrays while being compatible with the typical textbook-style dynamically allocated pointer-to-pointer structure.

   The compatibility requirement with pointer-to-pointer structures is achieved by allocating a pointer-to-pointer structure. This structure accounts for most of the memory overhead from using `rarray`.
2. To be as fast as pointer-to-pointer structures.
3. To do shallow copy by default, deep copy possible.
4. To have rarrays know their sizes, so that they can be passed to functions as a single argument.
5. To enable interfacing with libraries such as BLAS, LAPACK and FFTW: this is achieved by guarranteeing contiguous elements in the multi-dimensional array, and a way to get this data out.

   The guarrantee of contiguity means strided arrays are not supported.
6. To allow sharing between components of an application while avoiding dangling references. This is possible by utilizing reference counting.
7. To allow rarrays to hold non-owning views that use an existing buffer, without having to use a separate type.
8. To avoid some of the cluttered sematics around `const` correctness when converting to pointer-to-pointer structures when interfacing with legacy code.

Features of rarray:

- Can use any data type `T` and any rank `R`.
- Elements are accessible using repeated square brackets, like C/C++ arrays.
- Views on pre-allocated contiguous arrays.
- Does shallow, (atomic) reference counted, copies by default, but also has a deep `copy` method.
- Use of C++11 move semantics for efficiency.
- Can be emptied with the `clear` method.
- Can be filled with a uniform value with the `fill` method.
- Can be filled with a nested initializer list.
- Can be formed from a nested initializer list.
- Can be reshaped.
- Automatic C-style arrays can be converted to rarrays.

- A method `empty` to check if the rarray is uninitialized.
- A method to get the number of elements in each dimension (`extent`), or in all dimensions (`shape`).
- A method to obtain the total number of elements in the array (`size`).
- A method to make the data type of the array const (`const_ref`). Used in automatic conversion from `rarray<T,R>` to `rarray<const T,R>`.
- Conversion methods using the member function `data()` for conversions to a `T*` or `const T*`, using the method `ptr_array()` for conversions to `T*const*` or `const T*const*`, and using the method `noconst_ptr_array()` for the conversion to a `T**`.
- Streaming input and output.
- Checks index bounds if the preprocessor constant `RA_BOUNDSCHECK` is defined.

2 Comparison with standard alternatives

Compared to the old textbook method of declaring an array (see above), or the rarray method:

```
#include <rarray>
int main() {
  int n = 256;
  rarray<float,3> arr(n,n,n);
  arr[1][2][3] = 105; // for example
}
```

the more-or-less equivalent automatic array version

```
int main() {
  int n = 256;
  float arr[n][n][n];
  arr[1][2][3] = 105; // for example
}
```

is slightly simpler, but automatic arrays are allocated on the stack, which is typically of limited size. Another big drawback is that this array cannot be passed to functions that do not hard-code exact matching dimensions except for the last one.

Using rarray also has benefits over another C++ solution, i.e. using the `std::vector` class from the Standard Template Library:

```
  #include <vector>
  int main() {
     using std::vector;
     int n = 256;                 // size per dimension
     vector<vector<vector<float>>> v(n);// allocate for top dimension
```

```
   for (int i=0;i<n;i++) {
      v[i].resize(n);         // allocate vectors for middle dimension
      for (int j=0;j<n;j++)
         v[i][j].resize(n);   // allocate elements in last dimension
   }
   v[1][2][3] = 105;           // assign to element (for example)
}
```

which is complicated, is non-contiguous in memory, and likely slower.

C++23 will have a non-owning library, mdspan, which should work roughly as follows:

```
#include <memory>
#include <mdspan>
int main() {
   int n = 256;                    // size per dimension
   std::unique_ptr<float[]> p (new float[n*n*n]); // or vector or a shared_ptr
   using exts = std::extents<size_t,std::dynamic_extent,
                std::dynamic_extent,std::dynamic_extent>;
   std::mdspan<float,exts> (vector.data(), exts(n,n,n));
   v[1,2,3] = 105;                 // assign to element (for example)
}
```

Which is more involved that the rarray solution and does not offer reference counting.

3 Performance

3.1 General performance consideration

`rarray` is written specifically with performance in mind. However, the way it manages to mimic the natural c and c++ way of using multi-dimensional arrays, i.e. with (repeated) square brackets, would come at a high cost of function calls and temporary objects. The rarray library tries to inline most function calls to avoid this cost, but it needs the c++ compiler's optimization capabilities to help out. Even the lowest optimization levels, such as obtained with the `-O1` flag, will enable this for many compilers, though some will need hight levels of optimization.

Rarray is meant to be used with higher levels of optimization. When compiled with higher optimization levels, such as `-O3 -fstrict-aliasing` for `g++`, inlining should make the rarray objects at least as fast as the textbook solution for multidimensional arrays mentioned in the introduction.

3.2 Debugging

When debugging a code with a symbolic debugger (e.g. gdb), in addition to compiling with an option to include "symbols" (i.e., names of functions and other information regarding the code) into the executatble (e.g. `-g`), one usually switches off the compiler's optimization capabilities (`-O0`), because it can change the order of instructions in the code, which makes debugging very

difficult. As explained above, when working with rarray, turning of the compiler's optimization options can cause the program to become quite a bit slower, which can also hamper debugging. If the bug is unlikely to occur in rarray, it may be worth trying build with a mild optimization level such as `-O1` for debugging.

If, on the other hand, the bug is in rarray, one likely needs to switch off optimization to see what is going on. In addition to the `-O0` flag, this also requires switching off forced inlining, using the `-DRA_INLINE=inline` flag.

3.3 Profiling

Sampling profiling tools (such as `gprof`) work by periodically recording the state and call stack of the program as it runs. They can be very useful for performance analysis. When using these tools with rarray, it is advisable to compile with a minimum level of optimization `-O1`. Otherwise, a lot of the internal function calls of rarray that could simply be optimized away, and would pollute the sampling.

3.4 Memory overhead using the rarray class

The memory overhead here comes from having to store the dimensions and a pointer-to-pointer structure. The latter account for most of the memory overhead. A rarray object of $100\times100\times100\times100$ doubles on a 64-bit machine will have a memory overhead of a bit over 1%. In general, the memory overhead as a percentage is roughly 100% divided by the last dimension. Therefore, avoid rarrays with a small last dimension such as $100\times100\times100\times2$.

3.5 Compilation overhead using the rarray class

There is an overhead in the compilation stage, but this is very compiler dependent.

Finally, it is worth noting that many numerical libraries care about the "alignment" of the data. Rarray currently does not have any way of facilitating alignment, but a user can allocate aligned data and pass it as a pre-existing buffer to the rarray constructor.