Multidimensional Arrays in C++: Trying to get it right

Ramses van Zon

SciNet HPC Consortium

June 10, 2015

What's the problem?

- C++ sucks at multidimensional arrays.
- Fortran is excellent at it, but people scold me for using such an old language/it does not get me a job when I'm done with my career/I really like templates/...
- C++ could be great at multidimensional arrays, and there are a plethora of libraries aiming to fill this void. Something must be missing for there not to be a winner.
- Let's look a bit more closely at what is wrong with C++ multidimensional arrays, what we'd want from such entities, and a library that attempts to solve that.

SciNet compute • calcul CANADA

At first it seems so easy

```
#include <iostream>
int main() {
    const long n = 4;
    float a[n] = {1.0,2.0,3.0,10.0};
    for (long i=0; i<n; i++)
        std::cout << a[i] << " ";
}
```

Even easier using C++11 features:

```
#include <iostream>
int main() {
    const long n = 4;
    float a[n] = {1.0,2.0,3.0,10.0};
    for (auto x: a)
        std::cout << x << " ";
}
```

Two issues, even here

1: Arbitrary size limits

```
#include <iostream>
int main() {
    const long n = 1e7;
    float a[n] = {1.0,2.0,3.0,10.0};
    for (auto x: a)
        std::cout << x << " ";
}
```

This segfaults. There 'automatic arrays' are allocated on stack memory, which is usually much smaller than the total memory.

Two issues, even here

2. Function arguments

```
#include <iostream>
void printThis(float a[4]) {
   for (auto x: a)
      std::cout << x << " ";
}
int main() {
   float a[4] = {1.0,2.0,3.0,10.0};
   printThis(a);
}
```

This will not even compile. When passed to as a function argument, the array gets converted to a pointer, that does not know about the length of the array. The 4 in the argument list is ignored.

To solve this, we should:

- Allocate dynamically with `new`
- That means we have to explicitly deallocate with `delete`
- And we should pass the size of the array to functions. Still cannot use the c++11 loop.

Or we can use `std::vector`.

- This solve it all.
- We must be careful not to pass vectors by value to functions though!
- Only for the one dimensional case.

Multidimensional case

At first, just as easy:

```
#include <iostream>
int main() {
    const long n = 2;
    const long m = 3;
    const long k = 2;
    float a[n][m][k]
       = {{{1,2},{3,4},{5,6}},{{7,8},{9,10},{11,12}}};
    for (auto x: a)
       std::cout << x << " ";
}
```

- While the allocation works, the cout statement does not. It prints out two pointers to the two sub-matrices!
- These automatic arrays suffer from the same troubles are their 1d counterparts.

Solution

1. Manual allocation

```
float*** a = new float**[n];
for (int i=0;i<n;i++) {
  a[i] = new float*[m];
  for (int j=0;j<m;j++)
    a[i][j] = new float[k];
}
// Special tuning needed for contiguous elements (needed fo
```

2. Vectors of vectors of vectors

```
vector<vector<vector<float>>> a(n);
for (int i=0;i<n;i++) {
   a[i].reserve(m);
   for (int j=0;j<m;j++)
      v[i][j].reserve(k);
}
// No tuning possible to get contiguous elements
```

Something better

Introducing `rarray`:

A library for runtime-defined multidimensional arrays with none of these issues.

Rarray

- A header-only template solution for multidimensional arrays with dimensions determined at runtime.
- Usually faster than alternatives.
- Has optional bounds checking (no longer fast then, though).
- All allocation on the heap, and contiguous.
- No hidden copies of the array elements.
- Can reuse existing buffers too.

SciNet compute • calcul CANADA

How?

- The header file `rarray.h` provides the type `rarray<T,R>`, where `T` is any type and `R` is the rank.
- Element access uses repeated square brackets.
- Copying rarrays or passing them to functions mean shallow copies, unless explicitly asking for a deep copy.
- For io, use the additional header `rarrayio.h`.
- For element-wise algebraic operations, use `rarrayex.h`

```
git clone https://github.com/vanzonr/rarray
```

SciNet compute • calcul CANADA

Rarray in a Nutshell

Define a n×m×k array of floats:	`rarray<float,3> b(n,m,k);`
Define it with preallocated memory:	`rarray<float,3> c(ptr,n,m,k);`
Element i,j,k of the array b:	`b[i][j][k]`
Pointer to the contiguous data in b:	`b.data()`
Extent in the ith dimension in b:	`b.extent(i)`
Shallow copy of the array:	`rarray<float,3> d=b;`
Deep copy of the array:	`rarray<float,3> e=b.copy();`
A rarray re-using an automatic array:	`float f[10][20][8]={...};`
.	`rarray<float,3> g=RARRAY(f);`
Output a rarray to screen:	`std::cout « h « endl;`
Read a rarray from keyboard:	`std::cin » h;`

SciNet compute • calcul CANADA

Details

Copying and function arguments

- In C++, when we copy a variable of a built-in type to a new variable, the new copy is completely independent of the old variable. Likewise, the default way of passing arguments to a function involves a complete copy for built-in types.
- For C-style arrays, however, only the pointer to the first element gets copied, so you get a reference, not the whole array.
- The latter is called a shallow copy.
- Rarrays use shallow copies much like pointers, but memory allocated by the rarray gets released by the first created reference.

What does this essentially mean? Well:

1. You can pass rarrays by value to function, which is as if you were passing a pointer.
2. When you assign one rarray to another, the other simply points to the old one.
3. If you wish to do a deep copy, i.e., create a new array independent of the old array, you need to use the copy method.

Returning a rarray from a function

Unless you're using C++11 context, the shallow copying causes problems:

```
rarray<double,2> zeros(int n, int m) {
    rarray<double,2> r(n,m);
    r.fill(0.0);
    return r;
}
int main() {
    rarray<double,2> s = zeros(100,100);
    return s[99][99];
}
```

The array would get destroyed just after it is returned in C++03. Solution:

```
rarray<double,2>::return_type zeros(int n, int m) {
    rarray<double,2> r(n,m);
    r.fill(0.0);
    return r;
}
```

Optional Bounds checking

If the preprocessor constant RA_BOUNDSCHECK is defined, an out of bounds exception is thrown if

- an index is too small or too large;
- the size of dimension is requested that does not exist (in a call to extent(int i));
- a constructor is called with a zero pointer for the buffer or for the dimensions array;
- a constructor is called with too few or too many arguments (for R<= 11).

RA_BOUNDSCHECK can be defined by adding the -DRA_BOUNDSCHECK argument to the compilation command, or by `#define RA_BOUNDSCHECK" before the`#include “rarray.h”“‘ in the source.

I/O

- In the header `rarrayio.h`
- Only ascii for now: not great
- Prints it out as you would initialize an automatic array, i.e., with curly braces.
- Doing so means it can read it back in as well

Expressions

- A new and mildly experimental featues allows you to do this:

```
#include "rarray.h"
#include "rarrayex.h"
int main()
{
     rarray<float,2> a(4,4);
     rarray<float,2> b(4,4);
     rarray<float,2> c(4,4);
     float s = 2.0;
     a = b + s*c;
}
```

- Should work, but still under development and needs tuning.

A Dream

This should be in the language

```
#include "rarray.h"
#include "rarrayex.h"
int main()
{
      float[*][*] a(4,4);
      float[*][*] b(4,4);
      float[*][*] c(4,4);
      float s = 2.0;
      a = b + s*c;
}
```