Reference-Counted Multidimensional Arrays for C++: Rarray

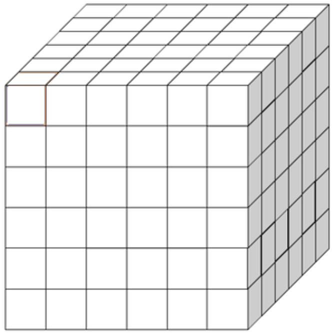
Ramses van Zon

Compute Ontario Colloquium - Nov 8, 2023

An essential data type in scientific computing

Something that is used in code for

- Artifical Intelligence
- Computational Fluid Dynamics
- Optimization
- Molecular dynamics
- and many more computational applications.

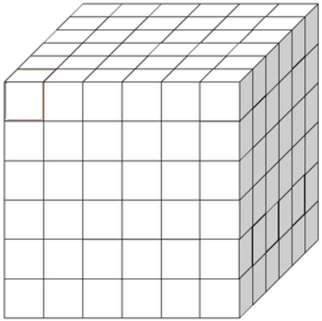


This data type is the **multidimensional array**.

Multidimensional arrays

What are these?

- A set of values of the same type.
- Ordered in a regular grid with multiple dimensions.
- The number of dimensions is called the rank.
  (this is a different from the rank of a matrix!)
- The sizes of the dimensions are called the extents of the array.
- Representations are not standard/universal.



Example

A three-dimensional grid of size 200 by 300 by 20, where each grid point holds a floating point number.
Such a grid may e.g. hold the temperature of the atmosphere over a piece of land.

History

The rise of scientific computing

Electronic computers came about towards the end of WWII. Interest in scientific computing quickly arose.

Initially computers had to be programmed in machine code or assembly language.

This was not ideal because this offered no abstractions and led to lengthy, machine-dependent codes.

FORTRAN (1957-)

Programs written in FORTRAN could be compiled into machine language of different computers.
Even the earliest version of FORTRAN supported multidimensional arrays.

FORTRAN IV (1961)

```fortran
      N=10
      DIMENSION A(10,10)
      DO 20 I=1,N
      DO 10 J=1,N
      A(I,J) = I + N*J
   10 CONTINUE
   20 CONTINUE
```

Fortran 90 (1990)

```fortran
integer :: n = 10, i, j
real(4), allocatable :: a(:,:)
allocate(a(n,n))
do i=1,n
   do j=1,n
      a(i,j) = i + n*j
   enddo
enddo
deallocate(a)
```

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

C (1978-)

- C is a general-purpose computer programming language.
- Static multidimensional arrays but not dynamic; must implement these oneself.
- Used as a system language.
- Interfaces well with software libraries.

Static/automatic array

```
#define n 10
int i,j;
float a[n][n];
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
       a[i][j] = i + n*j;
```

For large `n` this will break, because automatic arrays are allocated on the "stack".

How much stack space is available depends on the machine, its operating system, its configuration, and what other automatic variables you have defined.

Dynamic array

```
int n = 10, i, j;
float** a = malloc(n*sizeof(float*));
*a = malloc(n*n*sizeof(float));
for (i=1;i<n;i++)
   a[i] = a[0] + n*i;
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
       a[i][j] = i + n*j;
free(*a);
free(a);
```

`malloc` allocates on the "heap"
"array of arrays"
same indexing as static

C++ (1985-)

- Started as "C with classes", now much larger language than C.
- Also used outside of scientific computing.
- Can get low-level.
- Can also get high-level.
- Interfaces well with software libraries, but often through pointers.

Static/automatic array

```
int n = 10, i, j;
float a[n][n];
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
       a[i][j] = i + n*j;
```

Dynamic array

```
int n = 10, i, j;
float** a = new float*[n];
*a = new float[n*n];
for (i=1;i<n;i++)
   a[i] = a[0] + 10*i;
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
       a[i][j] = i + n*j;
delete [] *a;
delete [] a;
```

Vectors

```
int n = 10, i, j;
std::vector<
  std::vector<float> > a(n);
for (i=1;i<n;i++) a[i].reserve(n)
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
       a[i][j] = i + n*j;
```

Not contiguous!
Expensive copies!

C++23 will offer non-owning multidimensional views on existing arrays (later).

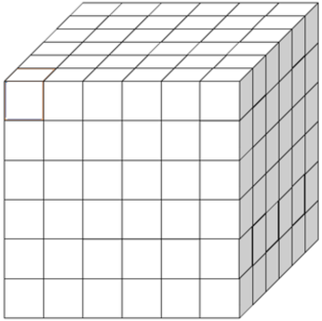
Not ideal, but what do we want?

The concept of multidimensional arrays

In C and C++, multidimesnional arrays are viewed as *arrays of arrays of arrays of*..., and thus if you can do 1D arrays, you can do multidimensional arrays.

There are issues with the array-of-arrays-of-.... idea:

- It is an additional abstraction that is not inherent to the concept. Should one view the athmosphere as an array of layers, and each layer an an array of longitudes, or as an array of lattitude slabs that are an arrays...?It is more natural to think of arrays as *data* with a *shape*.
- Complicates how one creates and works with multidimensional arrays.
- It allows non-contiguous data; contiguous memory accessed contiguously has performance benefits, and interfaces with libraries (e.g. fftw).



Want:

Multidimensional arrays to be objects containing contiguous data with a given shape.

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Usability of multidimensional arrays

- Multidimensional arrays should be easy to create and manipulate.
- Preferably with the same syntax as the rest of the C++ language: square brackets.
- They should be able to have arbitrary sizes, only limited by physical constraints.
- They should be able to work with C++ standard libraries.
- They should be able to work with C libraries that expect pointers.

Wants #2

Easy to use, no limits, square brackets accessors, data pointer access, STL compatibility

Performance of multidimensional arrays

- There should be no/minimal overhead in creating or using them.
- By default, they should not suffer copy overhaead, i.e. use shallow copies.
  Deep copies should be possible.
- They should be sharable between components of a calculations, or not.

Wants #3

Compiler needs to be able to optimize away any overhead. Shallow copy default, deep copy possible. Optional shared ownership.

Solutions

Early-days C++ solution: Write your own

```
class Matrix
{
private:
  int   nrows_, ncols_;
  double* data_;

public:
  Matrix(): nrows_(0), ncols_(0), data_(0) {}
  Matrix(int nrows, int ncols)
  : nrows_(nrows), ncols_(ncols),
   data_(new double[nrows*ncols])
  {}
  ~Matrix()
  { delete[] data_; }

  double& operator()(int i, int j)
  { return data_[i*ncols_ + j]; }
  const double& operator()(int i, int j) const
  { return data_[i*ncols_ + j]; }

  int extent(int i) const
  { return i?ncols_:nrows_; }
};
```

Features

- Construct a matrix:
  `Matrix m(10,10);`
- Access elements with parentheses:
  `m(0,3) = -2.5;`
- Request its size: `ncols = m.extent(1);`

Drawbacks

- Implicit copy constructors do the wrong thing.
  Cannot copy one matrix to another.
  Cannot pass a Matrix by value to a function.
- Requires computation for each element access.
- Why `(i,j)` instead of the standard `[i][j]`?

Write your own #2: Using C++17 features

```cpp
#include <array>
#include <memory>
class Matrix
{
private:
  std::array<size_t,2>        shape_;
  std::shared_ptr<double[]> data_;

public:
  Matrix() = default;
  Matrix(size_t nrows, size_t ncols)
  : shape_{nrows,ncols},
    data_{new double[](nrows*ncols)}
  {}

  double& operator()(size_t i, size_t j)
  { return data_[i*shape_[0] + j]; }
  const double& operator()(size_t i, size_t j)const
  { return data_[i*shape_[0] + j]; }

  size_t extent(int i) const
  { return shape_[i]; }
};
```

Features

- Construct a matrix: `Matrix m(10,10);`
- Access elements with parentheses: `m(0,3) = -2.5;`
- Request its size: `ncols=m.extent(1);`
- Standard library types allow:
  - Trivial constructor implemented trivially.
  - Copying, moving, destructing are implicitly implemented and correct.

Drawbacks

- Requires computation for each element access.
- Why `(i,j)` instead of the standard `[i][j]`?
- And why `std::shared_ptr`?

Write your own #3: Now with repeated brackets

```
#include <array>
#include <memory>
class Matrix
{
private:
  std::array<size_t,2>       shape_;
  std::shared_ptr<double[]> data_;

public:
  Matrix() = default;
  Matrix(size_t nrows, size_t ncols)
  : shape_{nrows,ncols},
    data_{new double[](nrows*ncols)}
  {}

  double* operator[](size_t i)
  { return data_ + i*shape_[0]; }
  const double* operator[](size_t i) const
  { return data_ + i*shape_[0]; }

  size_t extent(int i) const
  { return shape_[i]; }
};
```

Features

- Construct a matrix: `Matrix m(10,10);`
- Access elements with brackets:
  `m[0][3] = -2.5;`
  (Allows a lot of C-like code to be re-used)
- Request its size: `ncols=m.extent(1);`
- Standard library types allow:
  - Trivial constructor implemented trivially.
  - Copying, moving, destructing are implicitly implemented and correct.

Drawbacks

- Requires computation for each element access.
- And why `std::shared_ptr`?

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Reference Counted Pointers

The shared smart pointer `std::shared_ptr` is a reference-counted pointer.

Let's break this down:

- Smart: it will deallocate the memory held by the pointer when the pointer is no longer owned. (note: since C++17 is knows to deallocate an array with `delete[]`.)
- Shared: the ownership of this memory can be held by several parts of the code.
- Reference counted: there is a counter that keeps track of the number of owner.

How does reference counting work?

- A reference-counted object is one that is co-owned by several part of the code.
- When a copy is made, only its reference counter is increased. That copy is now a co-owner.
- When the copy goes out of scope or is deleted, the reference counter is decreaxsed.
- When the reference counter hits zero, the object is deleted.

```cpp
 1 #include <memory>
 2 #include <iostream>
 3 using pdouble = std::shared_ptr<double>;
 4 pdouble adddecimalpart(pdouble x) {
 5     *x += 0.14159;
 6     return x;
 7 }
 8 int main() {
 9     pdouble pi;
10     pdouble b(new double{3.0});
11     pi = adddecimalpart(b);
12     b.reset();
13     std::cout << "pi = " << *pi << std::endl;
14 }
```

Example

- Line 10: assigns `3.0` to `b` as only owner.
- Line 11: calls function `adddecimalpart` with an argument.
- Line 4: argument `x` becomes an owner too.
- Line 6: `x` is returned.
- Line 11: Now owned by pi as well.
- Line 7: `x` goes out of scope, so is no longer an owner.
- Line 12: `b` is no longer an owner.
- Line 13: `pi` remains as an owner, so it writes out the result 3.14159.

More features

Now let's add:

- Precomputation of row pointers
- Move sematics
- A copy() method for a deep copy
- A data() method to get the data out
- A size() method to get the total number of elements
- Iterators for going over all elements
- Conversion from automatic arrays and to pointers
- Non-owning views of existing arrays
- Reshaping functionality
- Bounds checking that can be switch on and off
- Ways to assign values to the whole array easily
- Streaming operators to read and print arrays
- Generalization to any rank (1D, 3D, 4D, ...)
- Generalization to any data type.

Actually, don't do this yourself, use...

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

RARRAY

A library for
dynamically allocated,
reference-counted,
multidimensional arrays
of arbitrary rank and type.

Consists of one header file.

Has all the features of the previous slide.

Download

```
$ git clone https://github.com/vanzonr/rarray
$ cd rarray
$ git checkout v2.6.0 # current version
$ cp rarray WHEREEVERYOUWANT
$ # read rarraydoc.pdf
```

Rarray Details by Example: Initialization

```cpp
#include <rarray>
int main() {
 rarray<int,1> a(5);
 a = 1,2,3,4,5;
 std::cout << a << "\n";
}
```

```
$ g++ -O3 arr1.cpp
$ ./a.out
{1,2,3,4,5}
```

```cpp
#include <rarray>
int main() {
 int adata[] = {1,2,3,4,5};
 rarray<int,1> a(adata);
 std::cout << a << "\n";
}
```

```
$ g++ -O3 arr2.cpp
$ ./a.out
{1,2,3,4,5}
```

```cpp
#include <rarray>
int main() {

 rarray<int,1> a = linspace(1,5);
 std::cout << a << "\n";
}
```

```
$ g++ -O3 arr3.cpp
$ ./a.out
{1,2,3,4,5}
```

```cpp
#include <rarray>
int main() {
 rarray<int,2> a(2,3);
 a = 1,2,3,4,5,6;
 std::cout << a << "\n";
}
```

```
$ g++ -O3 r2d1.cpp
$ ./a.out
{
{1,2,3},
{4,5,6}
}
```

```cpp
#include <rarray>
int main() {
 int adata[][3]={{1,2,3},{4,5,6}};
 rarray<int,2> a(adata);
 std::cout << a << "\n";
}
```

```
$ g++ -O3 r2d2.cpp
$ ./a.out
{
{1,2,3},
{4,5,6}
}
```

```cpp
#include <rarray>
int main() {
 rarray<int,2> a(2,3);
 for (int i=1; auto& x: a) x=i++;
 std::cout << a << "\n";
}
```

```
$ g++ -std=c++20 -O3 r2d3.cpp
$ ./a.out
{
{1,2,3},
{4,5,6}
}
```

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Rarray Details by Example: Methods

```cpp
#include <iostream>
#include <rarray>
int main() {
  rarray<int,2> a(3,4);
  a.fill(13);
  std::cout << "a=" << a << "\n"
    << "a[1][2]       = " << a[1][2] << "\n"
    << "a.empty()     = " << a.empty() << "\n"
    << "a.rank()      = " << a.rank() << "\n"
    << "a.size()      = " << a.size() << "\n"
    << "a.extent(0)   = " << a.extent(0) << "\n"
    << "a.extent(1)   = " << a.extent(1) << "\n"
    << "a.shape()[0] = " << a.shape()[0] << "\n"
    << "a.shape()[1] = " << a.shape()[1] << "\n\n";
  int*       data = a.data();
  int*const* b = a.ptr_array();
  int**      c = a.noconst_ptr_array(); // danger!
  std::cout
    << "data= "     << data << "\n"
    << "data[6] = " << data[6] << "\n"
    << "b[1][2] = " << b[1][2] << "\n"
    << "c[1][2] = " << c[1][2] << "\n";
}
```

```
$ g++ -O3 rprops.cpp

$ ./a.out

a={
{13,13,13,13},
{13,13,13,13},
{13,13,13,13}
}
a[1][2]       = 13
a.empty()     = 0
a.rank()      = 2
a.size()      = 12
a.extent(0)   = 3
a.extent(1)   = 4
a.shape()[0] = 3
a.shape()[1] = 4

data= 0x58b2b0
data[6] = 13
b[1][2] = 13
c[1][2] = 13
```

Rarray Details by Example: Copying

```
#include <iostream>
#include <rarray>

int main()
{
  // create an rarray a
  rarray<double,1> a(4);
  a.fill(1);
  // assign a to another rarray, b
  rarray<double,1> b = a;
  std::cout << "1. " << a << b << "\n";
  // modify b and see he effect
  b[3] = 2;
  std::cout << "2. " << a << b << "\n";
  // copy a to another array, c
  rarray<double,1> c = b.copy();
  // modify c and see the effect
  c[2] = 3;
  std::cout << "3. " << a << b << c << "\n";
}
```

```
$ g++ -O3 rcopying.cpp

$ ./a.out
1. {1,1,1,1}{1,1,1,1}
2. {1,1,1,2}{1,1,1,2}
3. {1,1,1,2}{1,1,1,2}{1,1,3,2}
```

- "=" creates a reference, like `std::shared_ptr`.
- `.copy()` returns a fully independent rarray.
- Similarly for function arguments:

```
void f1(rarray<double,1> b) {
  b[0][0] = 4; //changes original array
}
void f2(const rarray<double,1> b) {
  b[0][0] = 4; //not allowed!
}
void f4(const rarray<double,1>& b) { //no refcount!
  b[0][0] = 4; //not allowed!
}
```

Rarray Details by Example: Reshaping

Reshape an rarray

```
rarray<double,2> r(12,2);
r.fill(20);
r.reshape(4,6);
```

The data stays the same.
The rank must stay the same.

Shrink an rarray

```
r.reshape(2,6, ra::RESIZE::ALLOWED);
```

The superfluous elements become inaccessible.

Flatten an array:

```
r.reshape(1,r.size());
rarray<double,1> rflatref(r[0]);
```

Shapes are independently!

You can reshape an rarray without affecting the other references to it.

```
rarray<double,2> r(12,2);
rarray<double,2> s = r;
s.reshape(4,6);
// only s is reshaped, still shares data with r
```

This can be very useful in functions, e.g.

```
void printflattened(const rarray<double,3>& a)
{
   rarray<double,3> aref = a;
   aref.reshape(1,1,a.size());
   rarray<double,1> aflatref(aref[0][0]);
   std::cout << aflat;
}
```

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Performance details

- Compilation with optimization is needed for intermediate structures to be zero-cost.
- Then, its performance is identical to a good pointer-to-pointer structure, as that is what it uses under the hood.
- Good pointer-to-pointer structure: means it is contiguous in memory. As a result, it usually fits in cache, thus alleviating the dreaded cost of "pointer chasing".
- Reference counting is done atomically.

Bounds checking

You can have rarray check at runtime that indices to not go out of bounds by defining a preprocessor constant `RA_BOUNDSCHECK`

```
g++ -DRA_BOUNDSCHECK -O3 myrarrayprog.cpp
```

Good for debugging, but slows down all access.

Conclusions and outlook

Conclusions

- The Rarray provides multidimensional arrays.
- These are smart, shared arrays.
- They offer reference counting under control of the programmer.
  (and, yes, thread-safe)
- Layout is always row-major, contiguous in memory.
- Only requires C++11.
- C++23 will have `std::mdspan` that is a non-owning, non-reference counting option.

Outlook to the Future

- Allow slicing which does not break contiguity.
- Support for C++20's multidimensional index operator with square brackets:

```
rarray<double,2> a;
a[1,2] = 4.0; // instead of a[1][2] = 4.0;
```

Should `a[1][2]` be 100% equivalent to `a[1,2]`? Maybe. Probably.(already in progress)

- Automatic conversions to c++23's `std::mdspan`.
- Element-wise expressions

```
rarray<double,2> a(2,2),b(2,2),c(2,2);
a.fill(1); b.fill(2);
c.fill(a+2*b);
```

or something like that, would be nice.
(and be very Fortran-like).

- These may find their way into a version 3 series of Rarray.

Thank you for your attention! Questions?

Extra slides

When to use reference counting

1. Copies of the object are stored in different other objects.
   Sometimes it makes sense to rewrite the code, but sometimes shared ownership is natural.

   Example:
   The temperature grid at the beginning may be used by a monitoring object as well as a weather prediction object.
   In this case, it would be unnatural to rewrite the code and design a dedicated owner of this grid, and making sure that owner exists throughout the computation.

2. There is a good chance that case 1 might happen.

3. You're not quite sure if your algorithm properly keeps track of ownership without it.

4. You're sure your algorithm needs it to track of ownership.

5. You need a smart pointer, and `std::unique_ptr` does not fit the bill.

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

When not to use reference counting

- When single ownership suffices.
- When ownership is clear, and it is clear that it will remain unchanged.

  Example: the function `adddecimalpart` does not need x to be an owner.
  x was passed in, thus an owner exists, and cannot cease to exist.
  Make the argument a reference:

```
pdouble& adddecimalpart(pdouble& x) {
    *x += 0.14159;
    return x;
}
```

  Although `x` is reference counted, the counter is now not changed or touched in the call to this function.

- When the cost is too large: but profile first.

  For the large sizes that multidimensional arrays tend to have in scientific computing, the cost of increasing a counter (even atomically) is small.

  Note that if you are not copying, there is no cost.

Rarray API in a Nutshell

`rarray` header provides the type `rarray<T,R>`, where `T` is a type and `R` is the rank.

Define a n×m×k array of doubles:	`rarray<double,3> b(n,m,k);`
Define it with preallocated memory:	`rarray<double,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()`
Total number of elements in b:	`b.size()`
Extent in the ith dimension of b:	`b.extent(i)`
Array of all extents of b:	`b.shape()`
Define an array with same shape as b:	`rarray<double,3> b2(b.shape());`
Shallow copy of the array:	`rarray<double,3> d = b;`
Deep copy of the array:	`rarray<double,3> e = b.copy();`
A rarray re-using an automatic array:	`double f[10][20][8] = {{{...}}};` `rarray<double,3> g(f);`
Output a rarray:	`std::cout << h << endl;`
Read in a rarray:	`std::cin >> h;`