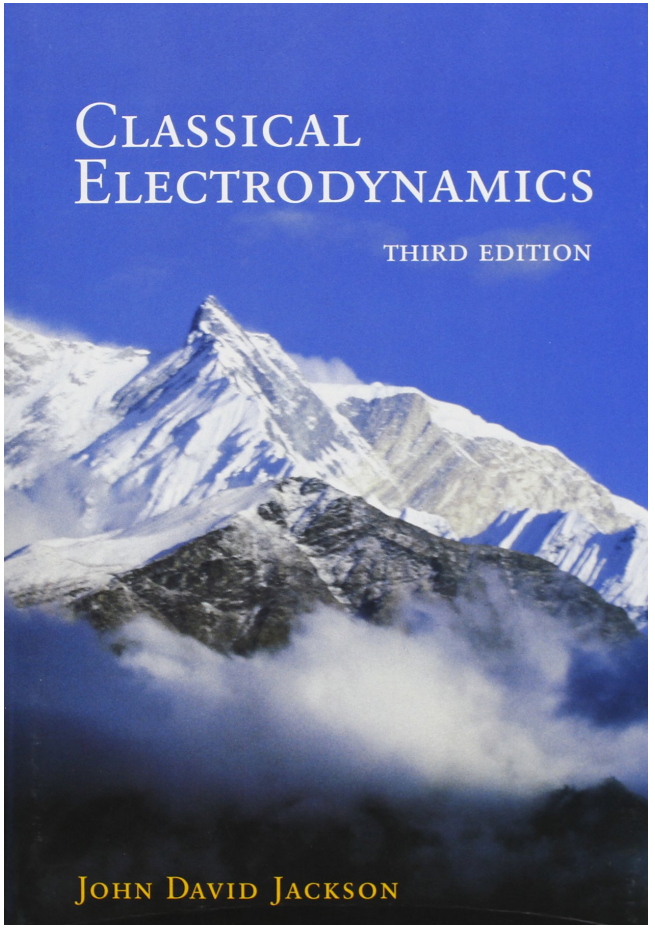
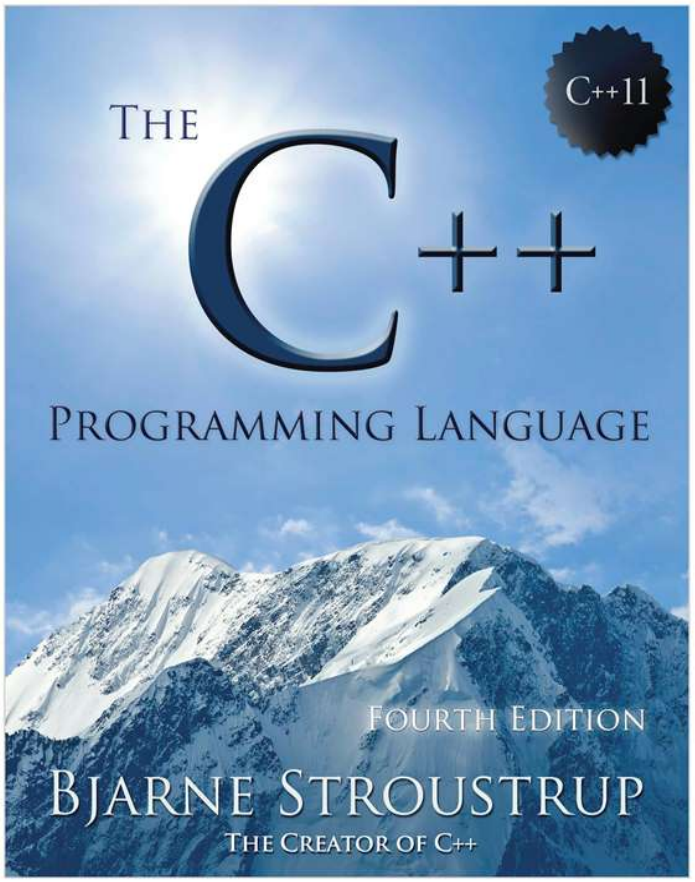
C++ Multi-dimensional Arrays for Computational Physics and Applied Mathematics

Pramod Gupta
Research Scientist
Department of Astronomy
University of Washington


CLASSICAL
ELECTRODYNAMICS
THIRD EDITION
JOHN DAVID JACKSON


C++11
THE
C++
PROGRAMMING LANGUAGE
FOURTH EDITION
BJARNE STROUSTRUP
THE CREATOR OF C++

Multi-Dimensional Arrays and Standard C++

Multi-dimensional arrays in Physics

- Temperature field
  double Temperature[Lx][Ly][Lz];

- Ising model spin at lattice point i, j, k
  int Spin[Lx][Ly][Lz];

- Three components of velocity at a point in a fluid
  double vx[Lx][Ly][Lz];
  double vy[Lx][Ly][Lz];
  double vz[Lx][Ly][Lz];

Passing Multi-dimensional arrays to functions

- The language feature of passing a multi-dimensional array to a function without specifying all its dimensions at compile time is crucial for computational physics, applied mathematics and other fields in scientific computing.
- For example, in linear algebra, a matrix is a two dimensional array and a matrix inversion function which needs to know the size of the matrix at compile time would be of limited use.

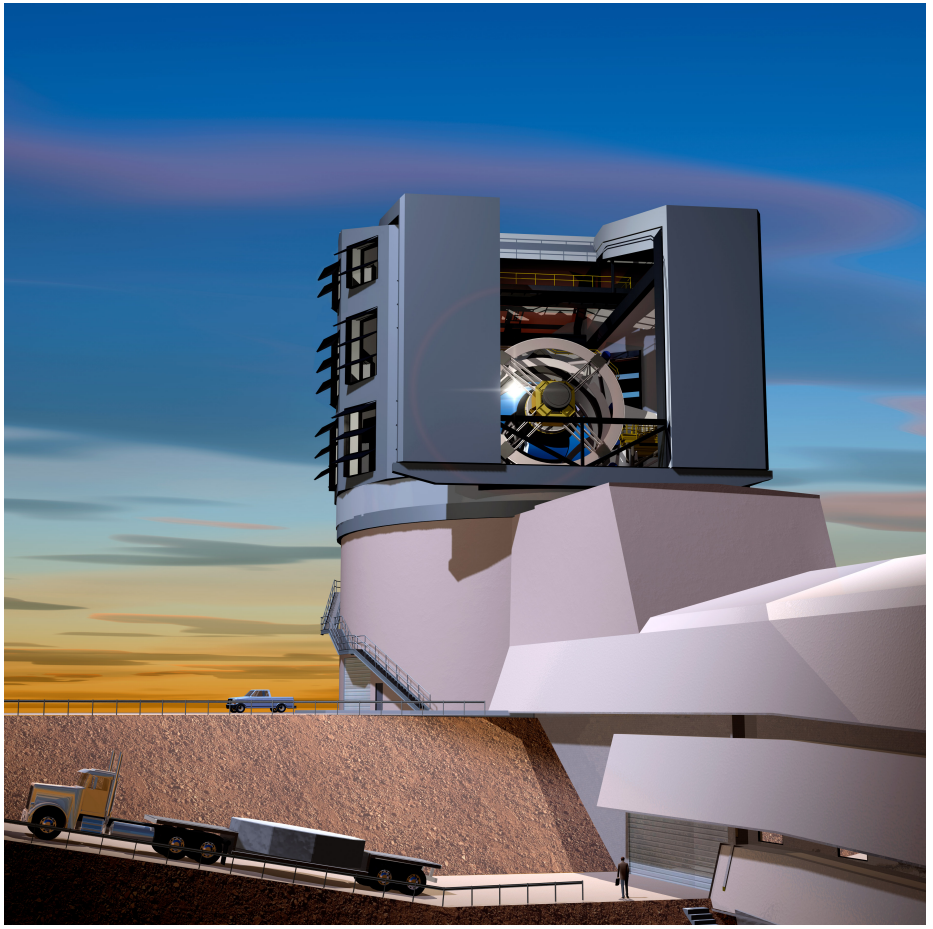
Support in other languages

- General purpose languages such as C99, Java and C# have supported this feature for at least 15 years.
- Scientific programming languages like Fortran, Matlab and R have supported this feature for decades.
- C++ is the only major programming language which does not support this feature.
- The absence of this feature increases the barrier for moving from C etc. to C++. For small teams and grad students it limits the usage of C++.

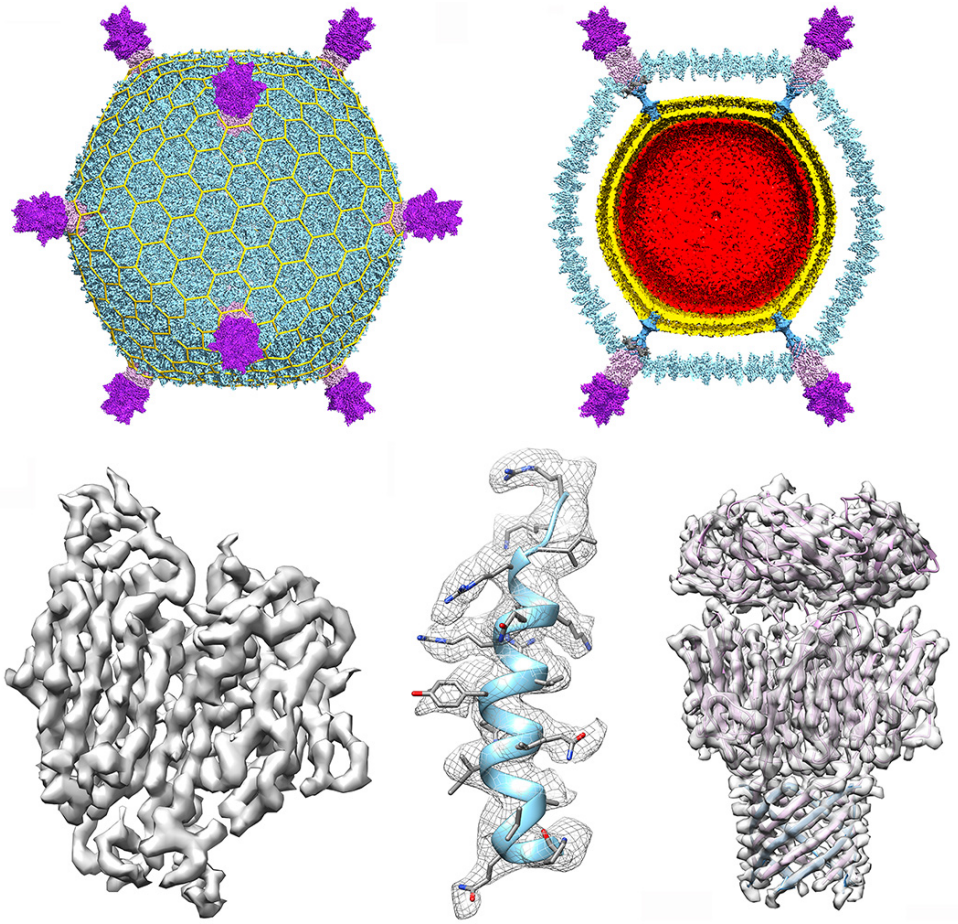
C++ and Scientific Computing

- C++ is used in large scientific organizations e.g. CERN, JPL/NASA (Jet Propulsion Lab), LSST (Large Scale Synoptic Telescope), LMB (Lab of Molecular Biology, Cambridge).
- If C++ is used for Scientific Computing, one of the first tasks is to make a class for multi-dimensional arrays.
- For example, LSST code for astronomy and Relion code for cryo-electron microscopy use custom multi-dimensional array classes.

LSST (Large Synoptic Survey Telescope)



Cryo-electron microscopy



```
//Valid C99.  Not valid C++.
//Passing multi-dimensional array to a
//function is common in scientific
//computing.
//Sum all elements of 2D array.
int sum(int n, int m, int b[n][m]){

int i, j, s;

s=0;
for(i=0;i<n;i++){
for(j=0;j<m;j++){
    s=s+b[i][j];
}
}

return s;
}
```

```
//Valid C99.
//Valid C++.
//Correct if and only if m=3.
//Sum all elements of 2D array.
int sum(int n, int m, int b[][3]){

int i, j, s;

s=0;
for(i=0;i<n;i++){
for(j=0;j<m;j++){
    s=s+b[i][j];
}
}

return s;
}
```

C Variable Length Arrays

- C variable length arrays (VLA) are not allowed by a compiler for C++ even though the same compiler supports them for C.
- gcc, clang, icc compilers support VLA for .c files but not for identical .cpp files.

- C variable length arrays are more general than needed for passing multi-dimensional arrays to functions.

```
//Valid C99.
//Not valid C++.

int try_vla(int n,int m){
int i, j, s;
int b[n][m];

. . . . . .


}
```

Argument against VLA

- If n and/or m are large, below code may crash program (without throwing exception or returning an error code) :

```
void try_vla(int n,int m){
int i, j, s;
int b[n][m];
. . . .
}
```

Important Subset of Variable Length Arrays

- Previous argument does not hold for below code since memory for array b is allocated elsewhere.

```
//Valid C99.

//Not valid C++.

int sum(int n, int m, int b[n][m]){

. . . .

}
```

- If a future C++ standard allows above subset of VLA, then it would reduce barrier in moving from C and other languages to C++ .

C++ Standard Library

- From Boost.MultiArray manual:
- “The C++ standard library provides several generic containers, but it does not provide any multidimensional array types. The std::vector class template can be used to implement N-dimensional arrays, for example expressing a 2-dimensional array of double elements using the type std::vector<std::vector<double>>, but the resulting interface is unwieldy and the memory overhead can be quite high.”

Multi-Dimensional Arrays and non-standard C++ Libraries

Non-standard Libraries

- Various non-standard libraries have been developed for using multi-dimensional arrays in C++
- Armadillo, Eigen, Blitz++, boost.multi_array
- Lots of features
- Very large and very complex
- Good for Matrix (2D array) and Linear Algebra

Armadillo

- C++ linear algebra library
- http://arma.sourceforge.net/
- From Australia http://www.nicta.com.au/
- Focused on matlab-like linear algebra.
- **Only Matrix (2D arrays) and Cube (3D arrays)**

Eigen

- Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
- http://eigen.tuxfamily.org/index.php?title=Main_Page
- From France INRIA http://www.inria.fr/
- **Only Matrices (2D arrays)**

Blitz++

- http://blitz.sourceforge.net/
- Lots of features
- **Very large and very complex**
- **100 page manual**
- Max up to 11 dimensions which is enough for almost all physics or applied math applications.

Boost.multi_array (Boost.MultiArray)

- N-dimensional arrays
- http://www.boost.org/doc/libs/1_58_0/libs/multi_array/doc/index.html
- Inspired by Blitz++
- Lots of features
- Very large and very complex
- **Complicated syntax**

```
#include "boost/multi_array.hpp"
#include <cassert>
//example from Boost web-site
int main () {
  // Create a 3D array that is 3 x 4 x 2
  typedef boost::multi_array<double, 3>
array_type;
  typedef array_type::index index;
  array_type A(boost::extents[3][4][2]);

  // Assign values to the elements
  int values = 0;
  for(index i = 0; i != 3; ++i)
    for(index j = 0; j != 4; ++j)
      for(index k = 0; k != 2; ++k)
        A[i][j][k] = values++;

}
```

Drawbacks of Existing Libraries

- The array libraries are large and complex.
- Scientific code becomes dependent on a large non-standard array library.
- Usage of the library is limited to those who are willing to install the array libraries and learn their use from large manuals.

Multi-Dimensional Arrays and orca_array C++ library

Design Philosophy for orca_array

- Features like native arrays in other languages.
- Fill gap between C++ arrays and large libraries.
- Convenient to include with applications.
- Readable code so user of library can verify correctness.
- Efficiency similar to native arrays in C++.
- Easy to debug array index out of bounds.

Design choices for orca_array

- Easy to declare array, to use array, and to pass array to a function.
- Single header file orca_array.hpp file so convenient to include in scientific applications.
- Efficient accessing of array elements.
- Compile time option to turn on array bounds checking.
- Compile time option for C-order or Fortran-order of accessing array. (row-major vs column-major)

Design Choices for orca_array

- Prohibit copy constructor and assignment operator so semantics are similar to C++ native arrays
- Must pass multi-dimensional array by reference.
- Use member function at( ) for subscripting instead of overloading operator [ ].
- If array bounds checking is on, raise SEG fault on error so user can find line of code in debugger.
- No default constructor since multi-dimensional array without size specified for each dimension has no meaning.
- No virtual member functions since class is not designed for inheritance.

Max number of dimensions

- Explicit code for array1d, array2d, …. array7d instead of using template parameter dimension N.
- More than 7 dimensional arrays are very rare since memory usage increases exponentially like L^N where L is the size of one dimension and N is the number of dimensions.
- 7 dimensions is enough for most computational physics and applied mathematics applications and hence until recently it was the limit for Fortran.
- Straight-forward to modify code for array8d etc.
- One could also write a program to generate code for arrayNd for arbitrary dimension N.

```
//orca_array “user manual” page 1 of 2
//array1d to array7d usage

#include “orca_array.hpp”
using namespace orca_array;

//defining 2D array
array2d<double> b(10,8);

//using 2D array
b.at(3,4)=3.14;

//passing 2D array to function
//m, n are optional
double trace(int m, int n,
array2d<double>  & myarray2d){
. . .
}
```

//orca_array “user manual” page 2 of 2
//compile time options in orca_array.hpp:
array bounds check

#define ARRAY_BOUNDS_CHECK 1

Program will encounter segmentation fault
if array index is out of bounds.
Compile program with -g option and run in
debugger to identify line of code.

column-major vs row-major

#define FORTRAN_ORDER 1
First index changes fastest

#define FORTRAN_ORDER 0
Last index changes fastest (C order)

```
#include "orca_array.hpp"
//example for orca_array
using namespace orca_array;

int main () {
  // Create a 3D array that is 3 x 4 x 2
  array3d<double> A(3,4,2);

  // Assign values to the elements
  int values = 0;
  for(int i = 0; i != 3; ++i)
    for(int j = 0; j != 4; ++j)
      for(int k = 0; k != 2; ++k)
        A.at(i,j,k) = values++;

}
```

Performance

- Correct answer to every performance questions is “It Depends”.
- It depends on the CPU, cache size, memory, disk, compiler, compiler options, type of problem, size of problem, algorithm etc.
- Actually running the code (many times) is the only reliable way of estimating performance.

orca_array Performance

- For code accessing array elements on LHS and RHS of assignment operator, orca_array performance is within few percent of native array performance.
- g++/gcc -O3, clang/clang++ -O3, icpc/icc -O3
- Number of elements in array must be large enough and/or number of array computations must be large enough.
- For small arrays with small number of computations, orca_array is slower than native arrays possibly due to overhead for dynamic allocation of memory and compiler optimization for native arrays. However, performance is very good in absolute terms.

```
template <class array_element_type>
class array2d{

private:
//number of rows
  int size1;
//number of columns
  int size2;

  array_element_type *  internal_array;

private:
//prohibit copy constructor
  array2d( array2d &);

//prohibit assignment operator
  array2d & operator=(array2d &);
```

```
//constructor
public:
  array2d(int  dim1, int dim2){

      if( dim1 <= 0 ){
      . . . .
      }
      else if( dim2 <=0){
      . . . .
      }
      else{
       size1=dim1;
       size2=dim2;
       internal_array =
      new array_element_type[size1*size2];
      }

}//end of constructor
```

```
//destructor
public:

  ~array2d(){

    delete [] internal_array;

  }
```

```
  public:

inline int length1(void){
    return size1;
}

inline int length2(void){
    return size2;
}
```

```
public:
    inline  array_element_type & at(int
x1, int x2){

#if ARRAY_BOUNDS_CHECK == 1

   if( (x1<0) || (x1>= size1) ){
     . . . .
   }

   if( (x2<0) || (x2>= size2) ){
    . . . .
   }

#endif
```

```
// at() (continued)

//x1 is row number and x2 is column number

#if FORTRAN_ORDER == 1
//fortran convention
//first index changes fastest
   return  internal_array[x2*size1 + x1];
#else
//C convention
//last index changes fastest
    return internal_array[x1*size2 + x2];
#endif

  }
```

```
//For array7d the at() function
//contains below code. F1.. F7 and
// C1..C7 are related to sizes of
// array dimensions size1, ...size7

#if FORTRAN_ORDER == 1
//fortran convention
//first index changes fastest
     return internal_array[x7*F7 + x6*F6 +
x5*F5 + x4*F4 + x3*F3 + x2*F2 +x1*F1];

#else
//C convention
//last index changes fastest
    return internal_array[x1*C1 + x2*C2 +
x3*C3 + x4*C4 + x5*C5 + x6*C6 + x7*C7];
#endif
```

Conclusions

- orca_array library for multi-dimensional arrays is convenient to use and convenient to include with scientific applications.
- User “manual” is a few lines only.
- It has compile times option for array bounds checking and for C-order or Fortran-order access.
- It has good performance.
- orca_array repository on github

Questions?