Scientific Computing: Arrays

Erik Spence

SciNet HPC Consortium

21 January 2014

Today's class

Today we will discuss the following topics:

- Arrays: general use.
- Arrays: are actually pointers.
- Arrays: multidimensional.
- Arrays: existing library functionality.

SciNet compute • calcul CANADA

Arrays must be dealt with carefully

Most scientific programming depends on arrays in one form or another. They show up everywhere:

- Fields.
- Grid information.
- Discretization.
- Linear algebra.

However, C++ was not designed with arrays in mind. You need to understand how they work to avoid the various pitfalls that can show up.

SciNet compute • calcul CANADA

Initializing static arrays

You seen static arrays already. They are only useful if you know the size of your array ahead of time.

A few points about initialization:

```
int cards[4] = {3, 6, 8, 10}; // Okay.
int hand[4];                  // Okay.
hand[4] = {3, 6, 8, 10};      // Not allowed.  Needs a variable type.
hand = cards;                 // Not allowed.
int hand[500] = {0};          // Okay.  Sets all values to 0.
int cards[] = {3, 6, 8, 10};  // Okay, and encouraged (no counting errors).
```

SciNet compute • calcul CANADA

Beware the end of the array

```
#include <iostream>          // MyArray.cpp
int main() {
  int a[] = {0, 1, 2, 3, 4};

  int *b = new int[5];
  for(int i = 0; i < 5; i++) b[i] = i;

  // Print out a and b.
  for(int i = 0; i < 7; i++) std::cout << a[i] << " ";
  std::cout << std::endl;

  for(int i = 0; i < 7; i++) std::cout << b[i] << " ";
  std::cout << std::endl;
  delete [] b;   return 0;
}
```

Beware the end of the array

```
#include <iostream>          // MyArray.cpp
int main() {
  int a[] = {0, 1, 2, 3, 4};

  int *b = new int[5];
  for(int i = 0; i < 5; i++) b[i] = i;

  // Print out a and b.
  for(int i = 0; i < 7; i++) std::cout << a[i] << " ";
  std::cout << std::endl;

  for(int i = 0; i < 7; i++) std::cout << b[i] << " ";
  std::cout << std::endl;
  delete [] b;   return 0;
}
```

```
ejspence@mycomp ~> g++ MyArray.cpp -o MyArray
ejspence@mycomp ~> ./MyArray
0 1 2 3 4 0 9420816
0 1 2 3 4 0 135137
```

Passing arrays to functions

```
#include <iostream>          // MyArray2.cpp
const int ArSize = 8;

int sum_arr(int arr[], int n) {
  int total = 0;
  for(int i = 0; i < n; i++) total += arr[i];
  return total;
};

int main() {
  int cookies[ArSize] = {1, 2, 4, 8, 16, 32, 64, 128};
  int sum = sum_arr(cookies, ArSize);
  std::cout << "Total cookies eaten:  " << sum << std::endl;
  return 0;
}
```

Passing arrays to functions

```
#include <iostream>          // MyArray2.cpp
const int ArSize = 8;

int sum_arr(int arr[], int n) {
  int total = 0;
  for(int i = 0; i < n; i++) total += arr[i];
  return total;
};

int main() {
  int cookies[ArSize] = {1, 2, 4, 8, 16, 32, 64, 128};
  int sum = sum_arr(cookies, ArSize);
  std::cout << "Total cookies eaten:  " << sum << std::endl;
  return 0;
}
```

```
ejspence@mycomp ~> g++ MyArray2.cpp -o MyArray2
ejspence@mycomp ~> ./MyArray2
Total cookies eaten:  255
```

Passing arrays to functions, cont.

```
#include <iostream>          // MyArray3.cpp
const int ArSize = 8;

int sum_arr(int arr[], int n) {
  int total = 0;
  for(int i = 0; i < n; i++) total += arr[i];
  arr[0] += 10;      return total;
};
int main() {
  int cookies[ArSize] = {1, 2, 4, 8, 16, 32, 64, 128};
  int sum = sum_arr(cookies, ArSize);
  int sum2 = sum_arr(cookies, ArSize);
  std::cout << "Total cookies eaten:  " << sum << std::endl;
  std::cout << "Total cookies eaten:  " << sum2 << std::endl;     return 0;
}
```

Passing arrays to functions, cont.

```
#include <iostream>          // MyArray3.cpp
const int ArSize = 8;

int sum_arr(int arr[], int n) {
  int total = 0;
  for(int i = 0; i < n; i++) total += arr[i];
  arr[0] += 10;      return total;
};
int main() {
  int cookies[ArSize] = {1, 2, 4, 8, 16, 32, 64, 128};
  int sum = sum_arr(cookies, ArSize);
  int sum2 = sum_arr(cookies, ArSize);
  std::cout << "Total cookies eaten:  " << sum << std::endl;
  std::cout << "Total cookies eaten:  " << sum2 << std::endl;     return 0;
}
```

```
ejspence@mycomp ~> g++ MyArray2.cpp -o MyArray2
ejspence@mycomp ~> ./MyArray2
Total cookies eaten:  255
Total cookies eaten:  265
```

Arrays are actually pointers

The function works because

- as we all remember, C++ functions pass arguments by value, not by reference (as in Fortran);
- but an array variable is actually a *pointer* to the first element of the array, not the whole array itself;
- thus the function takes a copy of the pointer to the array, and is able to manipulate the original array in memory, without making a copy of it.
- This saves on memory, and is faster, since the array isn't being copied.
- If you want your array to be protected from being modified by a function, include the const flag in the function prototype.

SciNet compute • calcul CANADA

Arrays are actually pointers, cont.

```
#include <iostream>          // MyArray4.cpp
int main() {
  float a[] = {10.0, 20.0, 30.0};   float *b = new float[3];   float *p;
  for(int i = 0; i < 3; i++) *(b + i) = a[i];

  p = a;    std::cout << "p = " << p << " *p = " << *p << std::endl;

  p = p + 1;
  std::cout << "p = " << p << ", *p = " << *p << std::endl;
  std::cout << "b = " << b << ", *b = " << *b << std::endl;
  std::cout << "b[1] = " << b[1] << ", b[2] = " << *(b + 2) << std::endl;
  return 0;
}
```

Arrays are actually pointers, cont.

```
#include <iostream>          // MyArray4.cpp
int main() {
  float a[] = {10.0, 20.0, 30.0};   float *b = new float[3];   float *p;
  for(int i = 0; i < 3; i++) *(b + i) = a[i];

  p = a;    std::cout << "p = " << p << " *p = " << *p << std::endl;

  p = p + 1;
  std::cout << "p = " << p << ", *p = " << *p << std::endl;
  std::cout << "b = " << b << ", *b = " << *b << std::endl;
  std::cout << "b[1] = " << b[1] << ", b[2] = " << *(b + 2) << std::endl;
  return 0;
}
```

```
ejspence@mycomp ~> g++ MyArray4.cpp -o MyArray4
ejspence@mycomp ~> ./MyArray4
p = 0x7fff67e23b30, *p = 10
p = 0x7fff67e23b34, *p = 20
b = 0x999010, *b = 10
b[1] = 20, b[2] = 30
```

Arrays of objects

Arrays can be of any variable type, even objects.

```
// ArrayObj.cpp
#include "StoneWt.h"

int main() {
  StoneWt weights[4] = {
    StoneWt(4, 6.7), StoneWt(9, 4.5), StoneWt(), StoneWt(1, 3.4)};

  for(int i = 0; i < 4; i++) weights[i].show_stn();
  return 0;
}
```

Arrays of objects

Arrays can be of any variable type, even objects.

```
// ArrayObj.cpp
#include "StoneWt.h"

int main() {
  StoneWt weights[4] = {
    StoneWt(4, 6.7), StoneWt(9, 4.5), StoneWt(), StoneWt(1, 3.4)};

  for(int i = 0; i < 4; i++) weights[i].show_stn();
  return 0;
}
```

```
ejspence@mycomp ~> g++ ArrayObj.cpp -c -o ArrayObj.o
ejspence@mycomp ~> g++ StoneWt.o ArrayObj.o -o ArrayObj
ejspence@mycomp ~> ./ArrayObj
The weight is 4 stone and 6.7 pounds.
The weight is 9 stone and 4.5 pounds.
The weight is 0 stone and 0 pounds.
The weight is 1 stone and 3.4 pounds.
```

2D arrays

Arrays are actually pointers; 2D arrays are pointers to arrays of pointers.

```
int data[3][4] = {{1, 2, 3, 4}, {9, 8, 7, 6}, {2, 4, 6, 8}};
```

represents an array of 3 pointers, each of which points to an array of 4 integers. Note that C++ is row major, versus column major (as in Fortran), meaning that this array is stored in memory as written above, and is normally written

$$\begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \\ \mathbf{9} & \mathbf{8} & \mathbf{7} & \mathbf{6} \\ \mathbf{2} & \mathbf{4} & \mathbf{6} & \mathbf{8} \end{bmatrix}$$

When you pass such an array to a function, you must specify the size of the arrays being pointed to, so that C++ knows how to index things properly:

```
int sum_arr(int data[][4], int size);
```

Use pointers for passing dynamic arrays

```
#include <iostream>          // MyArray7.cpp
int sum_arr(int **p, const int numrows, const int numcols);
void deallocate_mem(float **p, const int numrows);

int main() {
  int numrows = 3, numcols = 4;
  int **p = new int *[numrows];

  for(int i = 0; i < numrows; i++) {
    p[i] = new int[numcols];
    for(int j = 0; j < numcols; j++) {
      p[i][j] = i + j;
      std::cout << p[i][j] << " ";
    }
    std::cout << std::endl;
  }
  std::cout << "Total = " << sum_arr(p, numrows, numcols) << std::endl;
  deallocate_mem(p, numrows);
  return 0;
}
```

Use pointers for passing dynamic arrays

```
// MyArray7.cpp, continued
int sum_arr(int **p, const int numrows, const int numcols) {
  int total = 0;
  for(int i = 0; i < numrows; i++)
    for(int j = 0; j < numcols; j++) total += p[i][j];
  return total;
};

void deallocate_mem(float **p, const int numrows) {
  for(int i = 0; i < numrows; i++) delete [] p[i];
  delete [] p;
};
```

```
ejspence@mycomp ~> g++ MyArray7.cpp -o MyArray7
ejspence@mycomp ~> ./MyArray7
p is
0 1 2 3
1 2 3 4
2 3 4 5
Total = 30
```

Allocating 2D arrays

Do you understand the difference between these two functions?

```
float **allocate_matrix1(int n, int m) {

  float **a = new float *[n]; // First array is an array of pointers.

  for(int i = 0; i < n; i++) a[i] = new float[m];
  return a;
}
```

```
float **allocate_matrix2(int n, int m) {

  float **a = new float *[n]; // First array is an array of pointers.

  a[0] = new float[n * m];
  for(int i = 1; i < n; i++) a[i] = &a[0][i * m]; // & is the memory address.
  return a;
}
```

What is the advantage of the second over the first?

SciNet compute • calcul CANADA

Deallocating 2D arrays

The second code allocates continuous memory, while the first does not. The discontinuous block can be deleted like this:

```
void deallocate_matrix1(float **a, int numrows) {
  for(int i = 0; i < numrows; i++) delete [] a[i];
  delete [] a;
}
```

The continuous block is deleted like this:

```
void deallocate_matrix2(float **a) {
  delete [] a[0];
  delete [] a;
}
```

Note that delete must be called as many times as new was called during the allocation.

Accessing memory quickly

Which is faster?

```
int sum_arr1(int **p, const int numrows, const int numcols) {
  int total = 0;
  for(int i = 0; i < numrows; i++)
    for(int j = 0; j < numcols; j++) total += p[i][j];
  return total;
}

int sum_arr2(int **p, const int numrows, const int numcols) {
  int total = 0;
  for(int j = 0; j < numcols; j++)
    for(int i = 0; i < numrows; i++) total += p[i][j];
  return total;
}
```

Why?

SciNet compute • calcul CANADA

C++ is *row major*

Arrays are stored in memory in blocks of rows:

```
int data[3][4] = {{1, 2, 3, 4}, {9, 8, 7, 6}, {2, 4, 6, 8}};
```

But those blocks aren't necessarily continuous to each other if you declare your blocks like this:

```
float **allocate_matrix(int n, int m) {

  float **a = new float *[n]; // First array is an array of pointers.

  for(int i = 0; i < n; i++) a[i] = new float[m];
  return a;
}
```

When using blocks ('slices') of arrays for calculations, if possible, arrange your arrays so that you are looping over the *last* index. The last index will always be the most-continuous block of memory.

Existing C++ matrix packages

There are several C++ packages available to allow you to do matrix algebra (Armadillo, Eigen, Blitz++, boost). These packages come with built-in functionality that you may need:

- All the usual matrix multiplication operations.
- Matrix inversion, solving systems of equations.
- Decompositions, and factorizations.
- Eigenvalue calculations.
- Many operations have already been parallelized.

These are useful, and should be used first before trying to speed things up by building your own.

There are also the BLAS and LAPACK libraries, which are the classic libraries for doing linear algebra.

One such example: Eigen

```
// MyEigen.cpp
#include <iostream>
#include <Eigen/Dense>

int main() {
  // 2 x 2 matrix, of type float.
  Eigen::Matrix<float, 2, 2> A, B;

  A << 2, -1, -1, 3;
  B << 1, 2, 3, 1;

  std::cout << "A:" << A << std::endl;
  std::cout << "B:" << B << std::endl;
  std::cout << "A + B:" << A + B <<
    std::endl;
  std::cout << "A - B:" << A - B <<
    std::endl;
  std::cout << "1.6 * A:" << 1.6 * A <<
    std::endl;    return 0;
}
```

```
ejspence@mycomp ~> g++
  -I/home/ejspence/include/eigen
  MyEigen.cpp -o MyEigen
ejspence@mycomp ~> ./MyEigen
A:
2 -1
-1 3
B:
1 2
3 1
A + B:
3 1
2 4
A - B:
1 -3
-4 2
1.6 * A:
3.2 -1.6
-1.6 4.8
ejspence@mycomp ~>
```

Eigen: matrix manipulation

```
// MyEigen2.cpp
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::Matrix<float, 2, 2> mat;
  mat << 1, 2, 3, 4;
  Eigen::Matrix<float, 2, 1> u(-1, 1),
    v(2, 0);

  std::cout << "mat * mat:" <<
    mat * mat << std::endl;
  std::cout << "mat * u:" <<
    mat * u << std::endl;
  std::cout << "u^T * mat:" <<
    u.transpose() * mat << std::endl;
  std::cout << "u^T * v:" <<
    u.transpose() * v << std::endl;
  return 0;
}
```

```
ejspence@mycomp ~> g++
  -I/home/ejspence/include/eigen
  MyEigen2.cpp -o MyEigen2
ejspence@mycomp ~> ./MyEigen2
mat * mat:
7 10
15 22
mat * u:
1
1
u^T * mat:
2 2
u^T * v:
-2
ejspence@mycomp ~>
```

Eigen: systems of equations

```
// MyEigen3.cpp
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::Matrix<float, 2, 2> A, b, x;
  A << 2, -1, -1, 3;
  b << 1, 2, 3, 1;

  std::cout << "A:" << A << std::endl;
  std::cout << "b:" << b << std::endl;
  // Solve with LU decomposition.
  x = A.lu().solve(b);
  std::cout << "The solution is:" <<
    x << std::endl;
  std::cout << "Eigenvalues of A::" <<
    A.eigenvalues() << std::endl;
  return 0;
}
```

```
ejspence@mycomp ~> g++
  -I/home/ejspence/include/eigen
  MyEigen3.cpp -o MyEigen3
ejspence@mycomp ~> ./MyEigen3
A:
2 -1
-1 3
b:
1 2
3 1
The solution is:
1.2 1.4
1.4 0.8
Eigenvalues of A:
(1.38917,0)
(3.61803,0)
ejspence@mycomp ~>
```

SciNet compute • calcul CANADA