Computing and Statistical Data Analysis
Lecture 5

Arrays

Pointers

Strings

Glen Cowan
RHUL Physics

Arrays

An array is a fixed-length list containing variables of the same type.

Declaring an array: ***data-type variableName*[*numElements*];**

```
int score[10];
double energy[50], momentum[50];
const int MaxParticles = 100;
double ionizationRate[MaxParticles];
```

The number in brackets `[]` gives the total number of elements,e.g. the array **`score`** above has 10 elements, numbered 0 through 9. The individual elements are referred to as

`score[0], score[1], score[2], ..., score[9]`

The index of an array can be any integer expression with a value from zero up to the number of elements minus 1. If you try to access **`score[10]`** this is an error!

Arrays, continued

Array elements can be initialized with assignment statements and otherwise manipulated in expressions like normal variables:

```
const int NumYears = 50;
int year[NumYears];
for(int i=0; i<NumYears; i++){
  year[i] = i + 1960;
}
```

Note that C++ arrays always begin with zero, and the last element has an index equal to the number of elements minus one.

This makes it awkward to implement, e.g., n-dimensional vectors that are naturally numbered $\boldsymbol{x} = (x_1, ..., x_n)$.

For static arrays as shown here the size must be a constant. Its value must be known at compile time. Later we will see how to declare dynamic arrays, whose size is determined at run time.

Multidimensional arrays

An array can also have two or more indices. A two-dimensional array is often used to store the values of a matrix:

```
const int numRows = 2;
const int numColumns = 3;
double matrix[numRows][numColumns];
```

Again, notice that the array size is 2 by 3, but the row index runs from 0 to 1 and the column index from 0 to 2.

The elements are stored in memory in the order:

```
matrix[i][j], matrix[i][j+1], etc.
```

Usually we don't need to know how the data are stored internally. (Ancient history: in FORTRAN, the left-most index gave adjacent elements in memory.)

Initializing arrays

We can initialize an array together with the declaration:

```
int myArray[5] = {2, 4, 6, 8, 10};
```

Similar for multi-dimensional arrays:

```
double matrix[numRows][numColumns] =
                        { {3, 7, 2}, {2, 5, 4} };
```

In practice we will usually initialize arrays with assignment statements.

Example: multiplication of matrix and vector

```
// Initialize vector x and matrix A
const int nDim = 5;
double x[nDim];
double A[nDim][nDim];
for (int i=0; i<nDim; i++){
  x[i] = someFunction(i);
  for (int j=0; j<nDim; j++){
    A[i][j] = anotherFunction(i, j);
  }
}

// Now find y = Ax
double y[nDim];
for (int i=0; i<nDim; i++){
  y[i] = 0.0;
  for (int j=0; j<nDim; j++){
    y[i] += A[i][j] * x[j];
  }
}
```

Passing arrays to functions

Suppose we want to use an array **a** of length **len** as an argument of a function. In the function's declaration we say, e.g.,

```
double sumElements(double a[], int len);
```

We don't need to specify the number of elements in the prototype, but we often pass the length into the function as an **int** variable.

Then in the function definition we have, e.g.,

```
double sumElements(double a[], int len){
  double sum = 0.0;
  for (int i=0; i<len; i++){
    sum += a[i];
  }
  return sum;
}
```

Passing arrays to functions, cont.

Then to call the function we say, e.g.,

```
double s = sumElements(myMatrix, itsLength);
```

Note there are no brackets for **`myMatrix`** when we pass it to the function.

You could, however, pass **`myMatrix[i]`**, not as a matrix but as a **`double`**, i.e., the i^{th} element of **`myMatrix`**. For example,

```
double x = sqrt(myMatrix[i]);
```

Passing arrays to functions

When we pass an array to a function, it works as if passed by reference, even though we do not use the `&` notation as with non-array variables. (The array name is a "pointer" to the first array element. More on pointers later.)

This means that the array elements could wind up getting their values changed:

```
void changeArray (double a[], int len){
  for(int i=0; i<len; i++){
    a[i] *= 2.0;
  }
}

int main(){
  ...
  changeArray(a, len);    // elements of a doubled
```

Passing multidimensional arrays to functions

When passing a multidimensional array to a function, we need to specify in the prototype and function definition the number of elements for all but the left-most index:

```
void processImage(int image[][numColumns],
                  int numRows, int numColumns){
    ...
```

(But we still probably need to pass the number of elements for both indices since their values are needed inside the function.)

Pointers

A pointer variable contains a memory address. It 'points' to a location in memory. To declare a pointer, use a star, e.g.,

```
int* iPtr;
double * xPtr;
char *c;
float *x, *y;
```

Note some freedom in where to put the star. I prefer the first notation as it emphasizes that `iPtr` is of type "pointer to `int`".

(But in `int* iPtr, jPtr;` only `iPtr` is a pointer--need 2 stars.)

Name of pointer variable can be any valid identifier, but often useful to choose name to show it's a pointer (suffix `Ptr`, etc.).

Pointers: the `&` operator

Suppose we have a variable `i` of type `int`:

```
int i = 3;
```

We can define a pointer variable to point to the memory location that contains `i`:

```
int* iPtr = &i;
```

Here `&` means "address of". Don't confuse it with the `&` used when passing arguments by reference.

Initializing pointers

A statement like

```
int* iPtr;
```

declares a pointer variable, but does not initialize it. It will be pointing to some "random" location in memory. We need to set its value so that it points to a location we're interested in, e.g., where we have stored a variable:

```
iPtr = &i;
```

(just as ordinary variables must be initialized before use).

Dereferencing pointers: the * operator

Similarly we can use a pointer to access the value of the variable stored at that memory location. E.g. suppose `iPtr = &i;` then

```
int iCopy = *iPtr;     // now iCopy equals i
```

This is called 'dereferencing' the pointer. The * operator means "value stored in memory location being pointed to".

If we set a pointer equal to zero (or `NULL`) it points to nothing. (The address zero is reserved for null pointers.)

If we try to dereference a null pointer we get an error.

Why different kinds of pointers?

Suppose we declare

```
int* iPtr;      //  type "pointer to int"
float* fPtr;    //  type "pointer to float"
double* dPtr;   //  type "pointer to double"
```

We need different types of pointers because in general, the different data types (`int, float, double`) take up different amounts of memory. If declare another pointer and set

```
int* jPtr = iPtr + 1;
```

then the `+1` means "plus one unit of memory address for `int`", i.e., if we had `int` variables stored contiguously, `jPtr` would point to the one just after `iPtr`.

But the types `float`, `double`, etc., take up different amounts of memory, so the actual memory address increment is different.

Passing pointers as arguments

When a pointer is passed as an argument, it divulges an address to the called function, so the function can change the value stored at that address:

```
void passPointer(int* iPtr){
  *iPtr += 2;           // note *iPtr on left!
}

...
int i = 3;
int* iPtr = &i;
passPointer(iPtr);
cout << "i = " << i << endl;     // prints i = 5
passPointer(&i);              // equivalent to above
cout << "i = " << i << endl;     // prints i = 7
```

End result same as pass-by-reference, syntax different. (Usually pass by reference is the preferred technique.)

Pointers vs. reference variables

A reference variable behaves like an alias for a regular variable.
To declare, place `&` after the type:

```
int i = 3;
int& j = i;          // j is a reference variable
j = 7;
cout << "i = " << i << endl;  // prints i = 7
```

Passing a reference variable to a function is the same as passing a normal variable by reference.

```
void passReference(int& i){
  i += 2;
}

passReference(j);
cout << "i = " << i << endl;  // prints i = 9
```

What to do with pointers

You can do lots of things with pointers in C++, many of which result in confusing code and hard-to-find bugs.

One of the main differences between Java and C++: Java doesn't have pointer variables (generally seen as a Good Thing).

To learn about "pointer arithmetic" and other dangerous activities, consult most C++ books; we will not go into it here.

The main usefulness of pointers for us is that they will allow us to allocate memory (create variables) dynamically, i.e., at run time, rather than at compile time.

Dynamic arrays

An array's name is a pointer to its first element. We can create a "dynamic array" using the **new** operator:

```
double* array;
int len;
cout << "Enter array length" << endl;
cin >> len;          // array length set at run time
array = new double[len];
   ...
```

When we're done (e.g. at end of function where it's used), we need to delete the array:

```
delete [] array;
```

Strings (the old way)

A string is a sequence of characters. In C and in earlier versions of C++, this was implemented with an array of variables of type **`char`**, ending with the character **`\0`** (counts as a single 'null' character):

```
char aString[] = "hello";  // inserts \0 at end
```

The **`cstring`** library (**`#include <cstring>`**) provides functions to copy strings, concatenate them, find substrings, etc. E.g.

```
char* strcpy(char* target, const char* source);
```

takes as input a string **`source`** and sets the value of a string **`target`**, equal to it. Note **`source`** is passed as **`const`** -- it can't be changed.

You will see plenty of code with old "C-style" strings, but there is now a better way: the **`string`** class (more on this later).

Example with `strcpy`

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
  char string1[] = "hello";
  char string2[50];
  strcpy(string2, string1);
  cout << "string2:  " << string2 << endl;
  return 0;
}
```

No need to count elements when initializing string with `"  "`.

Also `\0` is automatically inserted as last character.

Program will print: `string2 = hello`

Wrapping up lecture 5

We are now almost done with the part of C++ that resembles C, i.e., the part that doesn't deal with classes or objects.

We've seen arrays (static and dynamic).

We've introduced pointers (but not yet done much with them).

We've seen briefly "C-style" strings.

Next we cover classes and objects -- the core of object oriented programming.