Arrays

PHYS2G03

McMaster University

Storing Large Datasets: Arrays

- It is not practical to store large datasets by using many individual variables or even new types (structs, classes)
- Arrays were developed for this purpose

Scalars vs. Arrays

- A regular variable is also referred to as a **scalar**
  Like a mathematical scalar it has a single value
- An **array** is equivalent to a higher dimensional object like a vector, matrix or tensor

- In C/C++ an array variable is indicated using square brackets [], e.g. a[1]

Note: The [] convention is nice – some languages use () which confuses it with functions

Referencing Arrays

- Arrays contain a list of multiple values referenced using a single name
- An **index** refers to the desired value within the list

NB:

C/C++ indexing begins at 0

Fortran, Mathematica, … indexing begins at 1
C, C++, Java, Python at 0

Arrays

```
x = 10;  // scalar variable x

A[0] = 10;  // array variable A,
            // index 0 => first data element = 10

A[j] = x;  //  index j:  A j = x

A[2*j] = 3*x;    // A 2j = 3 x

x = A[j+1]*3 ;   // x = 3 A j+1
```

Just like vector indices

Double duty

The value in [ ] brackets has two meanings depending on context

- In a declaration it is the size
- In other statements it refers to an element

```
int a[10] ;  // size of array is 10 integers

a[0] = 5;  //array variable a, first data element = 5

a[9] = 1; //array variable a,last data element = 1
```

Arrays: Legal index values

```
int a[10];  //   Define a 10 element array:  a[0 to 9]

a[1] = 5;   // OK

a[-1] = 2;    // illegal memory access

a[10] = 2;   // Illegal memory access
```

Note! Just because its illegal doesn't mean the code stops. The usual assumption Is that programmer knows what they are doing and doesn't want the compiler to waste time checking the index is legal.

Aside:

arrays, pointers, smart pointers

```
int a[10];  //   Define a 10 element array:  a[0 to 9]
a[1] = 5;   // OK
a[-1] = 2;    // illegal memory access (out of bounds)
```

Arrays are dumb because the use pointers (memory locations) to find the right index. They don't know how big the thing is supposed to be.
You can make more complicated things (e.g. classes) that are aware of the total size and/or check bounds these are often called *smart pointers*

For example, std::string s; vs. char a[10];
is like an array of characters (single bytes) but it knows how long it is (e.g. s.length()) and can make itself longer if needed.
In general C/C++ objects don't bound check because it is slower
If speed doesn't matter too much it is a lot safer to check though!

Arrays: Index problems

Statement	Memory Contains
int i;	i
float a[2],b[2];	a_0
	a_1
	b_0 a_2 ?
	b_1 a_3 ?

cd ~/bug
make memoryerror
memoryerror

```
b[0] = 0.0;
a[2] = 1.0;
```

Arrays: Index problems

Statement	Memory Contains
int i;	i
float a[2],b[2];	a_0
	a_1
	b_0 a_2 ?
	b_1 a_3 ?

The program does not crash!

a_2 just quietly trashes the memory for b_0

```
cd ~/bug
make memoryerror
memoryerror
```

```
b[0] = 0.0;
a[2] = 1.0;
```

Array Initialization

- A set of values inside braces is a constant array that can be used for initializing an array

```
int a[4]  = { 1, 2, 3, 4 };  // Initialize array a,  a[0]=1, a[1]=2, a[2]=3 a[3]=4

float b[] = { 1.0, 2.0, 3.0 }; // initialize array b, size=3
```

A size need not be given – the compiler just counts the values in { }, e.g. for b[] it will make a size of 3

Old way:

Arrays: Declarations and Constants

```
const int N = 100;
int data[10];  // Declare an array of size 10
int a[N], b[N];   // Declare two arrays of size N
float x[2*N];   // Declare a float array of size 2N
```

Traditionally constant expressions are used in array declarations
Variables declared with const in front are equivalent to constants

Modern C/C++:
Arrays: Declarations and Variables

```
int n;
n=100;
int a[n], b[n];   // Declare two arrays of size 100
scanf("%d",&n); // get a new n
float x[2*n];   // Declare a float array of size 2n
```

Changeable sizes are allowed in modern C and C++. These arrays exist in stack memory and are deleted like other variables when the function ends

Arrays and Loops
cp -r /home/2G03/circle ~/

circle.cpp

```
#include <iostream>
#include <math.h>
int main()
{
  int n=100;
  float x[n],y[n],theta;

  // n points around a circle
  for (int i=0;i<n;i++) {
    theta = 2*M_PI*float(i)/n;
    x[i] = cos(theta);
    y[i] = sin(theta);
  }
}
```

Arrays are often modified and used with loops to count through the indices

M_PI is a constant defined in math.h with the value pi

Arrays and Loops

```
#include <iostream>
#include <math.h>
int main()
{
  int n;

  std::cout << "How many points in the circle?\n";
  std::cin >> n;

  float x[n],y[n],theta;

  // n points around a circle
  for (int i=0;i<n;i++) {
    theta = 2*M_PI*float(i)/n;
    x[i] = cos(theta);
    y[i] = sin(theta);
  }

  for (int i=0;i<n;i++) {
    std::cout << "( " << x[i] << ", " << y[i] << ") ";
  }
  std::cout << "\n";
}
```

/home/2G03/circle/circle.cpp

Fancier version: Ask the user how many points before declaring the arrays

Arrays and Loops

/home/2G03/circle/plotcircle.cpp

```
#include <iostream>
#include <math.h>
#include "cpgplot.h"
int main()
{
  int n;
  std::cout << "How many points in the circle?\n";
  std::cin >> n;
  float x[n],y[n],theta;

  // n points around a circle
  for (int i=0;i<n;i++) {
    theta = 2*M_PI*float(i)/n;
    x[i] = cos(theta);
    y[i] = sin(theta);
  }

  if (cpgopen("/XWINDOW") < 0) return -1;  // exit if Xwindows not working
  // Set-up plot axes;
  cpgenv(-1.5,1.5,-1.5,1.5,0,0);
  cpglab("X", "Y", "Circle plot");
  // Plot the circle as points, point type 5 is an x
  cpgpt(n,x,y,5);
  cpgend(); // hit enter to end
}
```

pgplot can
plot any 2 arrays
of float values
as points or lines

The Makefile in
/home/2G03/circle
can link in all
the pgplot
libraries

In class exercise: histogram

- Write a program to count the number of times a number appears in a list of integers
- cp –r /home/2G03/array ~/
- gedit hist.cpp &
- make
- hist

Note the basic code has infinite loops
You may need control-C to stop it
If you run it without fixing this!

Exercise: histogram
cp –r /home/2G03/array; make; hist

hist.cpp

```
#include <iostream>

int main()
{
  // histogram
  // Takes a list of numbers and counts how many times each number occurs

  int list[] = { 1, 2, 3, 0, 1, 2, 1, 1, 0, 0 };
  int n=sizeof(list)/sizeof(int); // trick to count entries in list array (sizeof counts bytes)
  int i,j,count;

  // print the list first
  std::cout << "List of numbers ";
  for (i=0;i<n;i++) {
    std::cout << list[i] << " ";
  }
  std::cout << "\n";
```

Starter code –
Prints out the 10 list numbers

Exercise: histogram
cp –r /home/2G03/array; make; hist

hist.cpp

```
#include <iostream>
int main()
{
  // histogram
  // Takes a list of numbers and counts how many times each number occurs
  int list[] = { 1, 2, 3, 0, 1, 2, 1, 1, 0, 0 };
  int n=sizeof(list)/sizeof(int); // trick to count entries in list array (sizeof counts bytes)
  int i,j,count;
  …

  // A loop over all n entries
  for ( ; ; ) {

     // A loop to count how many match
     for ( ; ; ) {

     }
     // print how many counts for that case
   }
}
```

Exercise: histogram
cp –r /home/2G03/array; make; hist

hist.cpp

```
#include <iostream>
int main()
{
  // histogram
  // Takes a list of numbers and counts how many times each number occurs
  int list[] = { 1, 2, 3, 0, 1, 2, 1, 1, 0, 0 };
  int n=sizeof(list)/sizeof(int); // trick to count entries in list array (sizeof counts bytes)
  int i,j,count;
  …

  // A loop over all n entries
  for ( ; ; ) {     e.g. index i

     // A loop to count how many match
     for ( ; ; ) {    e.g. index j

     }
     // print how many counts for that case
   }
}
```

Note: There is a way to avoid printing a case more than once by comparing index i and j

Your final program should just print:

1 appears 4 times
2 appears 2 times
3 appears 1 times
0 appears 3 times