Crash Course in C++

R F L Evans

www-users.york.ac.uk/~rfle500/

Course overview

- Lecture 1 - Introduction to C++
- Lecture 2 - Functions and Data
- Lecture 3 - Strings and Files
- Lecture 4 - Code Organization
- Lecture 5 - Object Oriented Programming

Lecture 2
Functions and Data

- Functions
- Arrays
- Standard library containers

Functions

- Essential concept for organized programming
- The same concept as FORTRAN subroutines

What is a function?

"A segment of code designed to perform a single task or 'function' operating on a set of variables"

Why use a function?

```
#include <iostream>

int main(){

  double degreesF = 57.0;
  double degreesC = (degreesF-32.0)*5.0/9.0;
  std::cout << degreesF << " F is " << degreesC << " C" << std::endl;

  degreesF = 86.0;
  degreesC = (degreesF-32.0)*5.0/19.0;
  std::cout << degreesF << " F is " << degreesC << " C" << std::endl;

  return 0;

}
```

Code does the same thing - but writing the same code twice can lead to bugs

Using Functions

- Write one (short) piece of code to do a single task
- Only have to debug once and less potential for errors
- Makes code cleaner and easier to read

Function syntax

```
type function_name(type argument){

  // function code

  return variable;

}
```

- Functions have a type so that they can ‘return’ a value directly
- Can also accept optional ‘arguments’ - variables which the function needs to calculate the result

Function examples

```
int three(){ // no arguments

  return 3; // always equal to three

}
```

```
int add_three(int a){

  return a+3; // return value of a+3

}
```

Using Functions

```
#include <iostream>

int main(){

  int a=5;

  a = three();
  a = add_three(a);

  return 0;

}
```

- Functions are called using their name
- Variables (without type) are passed to the function

Functions must be declared before they are used

```
#include <iostream>

// function declaration
int three(){
  return 3;
}

int main(){

  int a=5;

  a = three();

  return 0;

}
```

Functions can also be ‘forward declared’

```
#include <iostream>

// function declaration
int three();

int main(){

  int a=5;
  a = three();
  return 0;
}

// function definition
int three();{
  return 3;
}
```

Functionalized version of Fahrenheit to Celcius

```
#include <iostream>

// function to convert Fahrenheit to Celcius
double FtoC(double F){
  return (F-32.0)*5.0/9.0;
}

int main(){
  double degreesF = 57.0;
  double degreesC = FtoC(F);
  std::cout << degreesF << " F is " << degreesC << " C" << std::endl;

  degreesF = 86.0;
  degreesC = FtoC(F);
  std::cout << degreesF << " F is " << degreesC << " C" << std::endl;

  return 0;
}
```

Static Arrays

Your program needs 5 int variables

```
#include <iostream>

int main(){

  int a;
  int b;
  int c;
  int d;
  int e;

  return 0;
}
```

Your program needs 5,000 int variables

```
#include <iostream>

int main(){

  int a;
  int b;
  int c;
  ...
  int z; // err...

  return 0;
}
```

Arrays

- Array is a structure which stores many variables of the same type, eg int, double, bool etc
- Uses an ‘index’ to access each variable or ‘element’ of the array
- C++ includes static arrays, allocated arrays and standard library containers

Array declaration

```
#include <iostream>

int main(){

  int array[5]; // array with storage for five int variables
  double array_b[3]={0.0,1.0,2.0}; // array of three doubles
                                   // initialised to 0.0,1.0,2.0

  return 0;

}
```

- Arrays are declared just like normal variables
- Square brackets [ ] indicate the number of variables which can be stored in an array

Accessing array elements

```
#include <iostream>

int main(){

  int array[5]; // array with storage for five int variables

  array[0] = 5; // set the first element to 5
  array[1] = array[0]+8; // set the second element to 13

  return 0;

}
```

- Once declared, each element of the array behaves just like a normal variable
- Array 'index' starts from zero in C++ (compared to 1 in FORTRAN)
- Can be used for arithmetic, copied to other variables etc

Accessing array elements

```
array[0] = 1;
array[1] = 2;
array[2] = 0;
array[3] = 4;
array[4] = 2;
```

0 1 2 3 4

- Best way is to visualize a bunch of numbered boxes storing apples
- The index is which box you are looking at and the value is the number of apples (doubles, floats, bools etc) stored in the box

Memory errors

```
#include <iostream>

int main(){

  int array[5]; // array with storage for five int variables

  array[0] = 5; // OK
  array[5] = 78; // Oh dear
                 // This is past the last element of the array

  return 0;

}
```

- Most common cause of error
- Typically results in a ‘Segmentation fault’ and crash, but not always
- Compiler doesn’t reliably check these errors

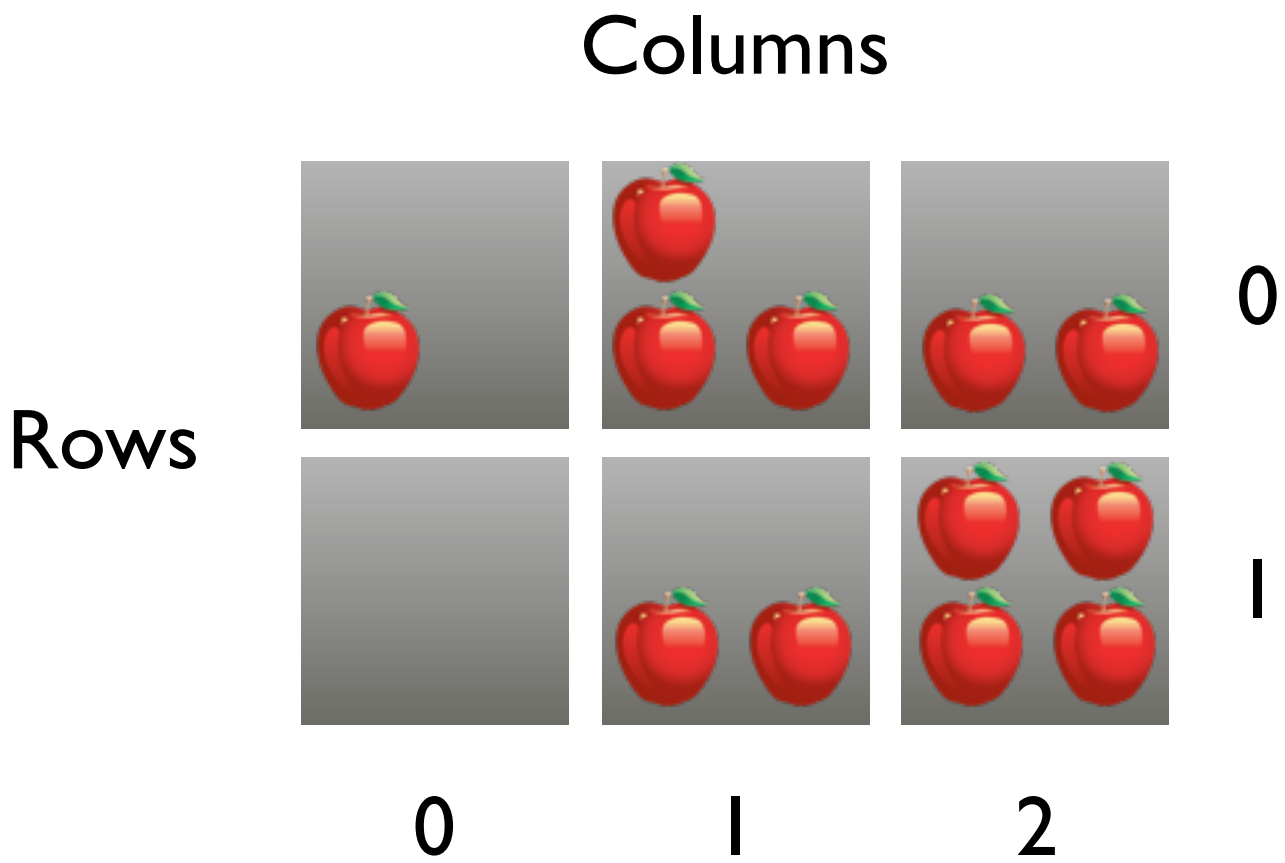
Multidimensional arrays

```
#include <iostream>

int main(){

  // Array with 15 elements [ROWS][COLUMNS]
  double array[3][5];
  // Data looks like
  // [  ] [  ]  [  ]  [  ]  [  ]
  // [  ] [  ]  [  ]  [  ]  [  ]
  // [  ] [  ]  [  ]  [  ]  [  ]

  return 0;
}
```

- Can also use higher dimension arrays
- Typically used for data relating to 2D/3D/4D spaces or similar mathematical abstractions

Multidimensional array example

```
int array[2][3]; // 2 rows and 3 columns
```

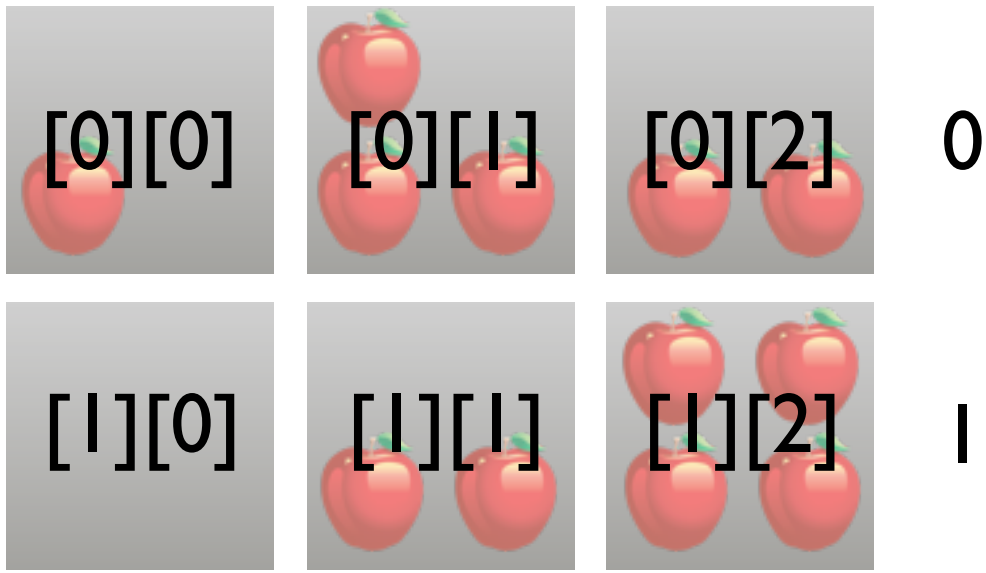


Multidimensional array example

```
array[0][0] = 1;
array[1][0] = 0;
array[1][2] = 4;
```

Columns

Rows



Multidimensional array example

```
#include <iostream>

int main(){

  // Array to store number of
  int array[3][4];

  // Used in exactly the same way as 1D arrays
  array[2][3] = 4;
  array[0][1] += array[1][2];

  return 0;
}
```

Accessing arrays with loops

```
#include <iostream>

int main(){

  int array[5]; // array with storage for five int variables

  // loop over all elements of the array and set to zero
  for(int i=0; i<5; ++i){
    array[i] = 0;
  }

  // Add i to each element of the array
  for(int i=0; i<5; ++i){
    array[i] += i;
  }

  return 0;

}
```

Accessing 2D arrays with loops

```
#include <iostream>

int main(){

  int array[2][2]; // array with storage for 2 x 2 int variables

  // nested loop over all elements of the array and set to zero
  for(int i=0; i<2; ++i){
    for(int j=0; j<2; ++j){
      array[i][j] = 0;
    }
  }

  return 0;

}
```

Static arrays are limited in size

- Static arrays are limited in size by the system (typically ~8kB)
- Large arrays must request a block of memory from the OS

Allocatable Arrays

Allocatable array declaration

```
int main(){

  int * array; // declare a pointer

  array = new int[5000000]; // allocate array with 5000000 values

  array[1234]=34567; // assign value to array

  delete[] array; // deallocate memory

  return 0;

}
```

Allocatable array declaration

```
int main(){

  int * array; // declare a pointer

  array = new int[5000000]; // allocate array with 5000000 values

  array[1234]=34567; // assign value to array

  delete[] array; // deallocate memory

  return 0;

}
```

Do not use these

Allocatable arrays are bad

- Use pointers (works with raw memory addresses)
- Need to be allocated and then deallocated before exiting the program
- Undeallocated memory is a memory leak
- No checking of array bounds
- No way to find out the size of the array
- Messy to pass to other functions
- AND there is a much better way

Storage Containers in the C++ standard library

Several container types available in the standard library

- vector, map, list, deque, valarray, array
- Automatic memory management
- Know their own size
- Bounds checking available
- Neat built-in functions such as sorting, ordering

Vectors

```
#include <vector> // include vector header

int main(){

  std::vector<int> array(5); // array for storing five int variables

  array[3]=4; // once declared behaves just like a normal array

  return 0;

}
```

Vector declarations

```
#include <vector> // include vector header

int main(){

  std::vector<double> array1(10, 5.0); // array with 10 elements
                                       // all initialized to 5.0

  std::vector<double> array2; // empty array

  array2.resize(100000); // resize array2 to contain 100000 elements

  array2.resize(100, 5.0); // resize array2 to contain 100 elements
                           // each initialized to 5.0

  array1 = array2; // make a copy of array2 and save in array1

  return 0;

}
```

Vector functions

```
#include <vector> // include vector header

int main(){

  std::vector<double> array; // empty array

  array.reserve(1000);   // reserve storage for 1000 elements
  array.resize(100);     // resize array to contain 100 elements
  array.at(50) = 25.0;   // array access with bounds checking
  array.push_back(34.0); // increase array size by 1 and
                         // save the value 34.0

  // diagnostic functions
  unsigned int array_size = array.size(); // size of array
  unsigned int array_cap  = array.capacity(); // reserved array size

  return 0;

}
```

Using vectors

```
#include <vector> // include vector header

int main(){

  std::vector<float> array(20); // array of 20 floats

  // initialize values in array
  for(int i=0; i<array.size(); ++i){
    array.at(i) = 5.0f*float(i);
  }

  return 0;

}
```

- Very safe way of accessing arrays as always check against size of array

Multidimensional vectors

```
#include <vector> // include vector header

int main(){
  // a vector of a vector of float - must have “> >”, not “>>”
  std::vector<std::vector <float> > array; // empty 2D array

  // set number of rows and columns
  int num_rows = 5;
  int num_cols = 10;
  array.resize(num_rows);
  for(int i=0; i<array.size(); ++i){
    array.at(i).resize(num_cols);
  }

  array[3][8] = 5.0f;          // fast
  array.at(2).at(9) = 10.0f; // bounds checking but slow

  return 0;
}
```

Multidimensional vectors

```
#include <vector> // include vector header

int main(){

  std::vector<std::vector <float> > array; // empty 2D array

  // set up array 5 x 10 (messy part)
  array.resize(5);
  for(int i=0; i<array.size(); ++i) array.at(i).resize(10);

  // nested loop over all elements of the array and set to 0.123
  for(int i=0; i<array.size(); ++i){
    for(int j=0; j<array[i].size(); ++j){
      array[i][j] = 0.123f;
    }
  }

  return 0;
}
```

Passing vectors to functions

```
#include <vector>
#include <iostream>

// function to sum up values
int sum(std::vector<int> array_in){
  // initialise sum
  int sum_values = 0;
  // loop over all values and add them up
  for(int i=0; i<array_in.size();++i) sum_values+=array_in[i];
  // return sum
  return sum_values;
}

int main(){
  std::vector<int> array(5,2.0);
  int sumv = sum(array); // call function and store result in sumv
  std::cout << sumv << std::endl;
  return 0;
}
```

Reference operator

```
// function to sum up values
int sum(std::vector<int>& array_in){

  // initialise sum
  int sum_values = 0;

  // loop over all values and add them up
  for(int i=0; i<array_in.size();++i) sum_values+=array_in[i];

  // return sum
  return sum_values;
}
```

- By default variables passed to functions are copied (very expensive for arrays)
- ‘Reference’ operator passes the actual variable(array) to the function (much faster)

Example functions using vectors

```
// function to zero values
void zero(std::vector<int>& array_in_out){

   // loop over all values and set to zero
  for(int i=0; i<array_in.size();++i) array_in_out[i] = 0;

  return; // note absence of variable

}

int main(){

  std::vector<int> array(5,2);

  zero(array); // zero array values

  return 0;
}
```

Example functions using vectors

```
// function returning vector<int>
std::vector<float> mul(std::vector<float> array_in, float a){

  // declare result array same size as array_in
  std::vector<float> result(array_in.size());

  // loop over all values and multiply by a
  for(int i=0; i<array_in.size();++i) result[i]=array_in[i]*a;

  return result;
}

int main(){

  std::vector<float> array(5,2.0f);

  array = mul(array, 5.0f); // multiply array by 5.0

}
```

list

```
#include <list> // include list header

int main(){

  std::list<int> mylist(5); // list of 5 int variables
  int count = 0;
// have to use iterators to access elements (set all elements to 78+c)
for(std::list<int>::iterator it=mylist.begin();it != mylist.end();++it)
{
  *it = 78+count;
  ++count;
}

  // a bit clunky but cool features
  mylist.sort(); // sort elements by number
                 // can even define custom sort function
  return 0;

}
```

list with vector

```
#include <algorithm> // cool functions for containers

int main(){
  std::vector<int> array;
  for(int i=0; i<100; ++i) array.push_back(i);  // set values in array

  std::list<int> mylist(array.size()); // list same size as array

  // copy to list
  copy(array.begin(), array.end(), mylist.begin());

  // sort elements by number
  mylist.sort();

  // copy back to vector
  copy(list.begin(), list.end(), array.begin());

  return 0;
}
```

Practical time...