C++ Programming for Engineers

Petar Jervic
Francisco Luro
Andrew Moss (course-resp)

May 2, 2012

Introduction to pointers

- Memory allocation and access
- Motivation for dynamic memory structures
- Connection between arrays and pointers
- Building data-structures
  - 2D arrays

Memory organisation

Two kinds of memory

- Stack — smaller, automatic, function scopes / local variables
- Heap — larger, manual, dynamic allocation

In both cases memory is an array of bytes

- Each 8-bit value has a unique address
- Each address is an unsigned integer label
- The actual numbers change depending on machine, O/S, compiler . . .
- The *address-space* determines how many unique labels
  - e.g. a 32-bit address space has 2^{32} labels, or 4GB
  - e.g. a 64-bit address space has 2^{64} labels, about 16 exabytes

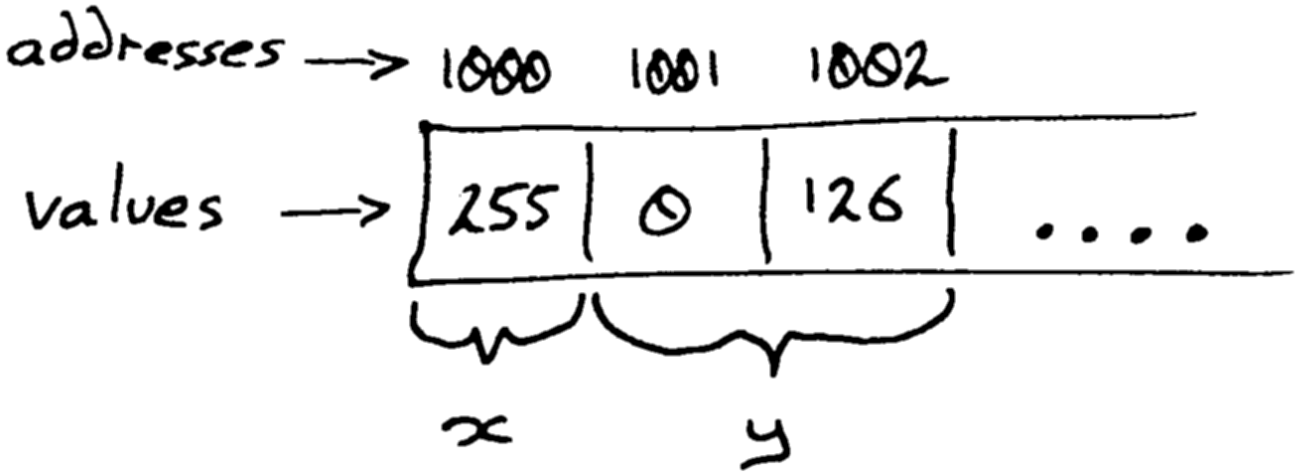
Names and labels

Each byte in memory has a numeric label

- We normally refer to these labels as *addresses*
- Every variable in the program occupies some of these addresses

When you declare a variable it creates a name

- The compiler decides how to map names onto address(es)
- Programmer normally ignores the layout, exact addresses etc




```
{
char x;
int y;
}
```

Addresses are numbers

This is useful

- We know how to perform operations with numbers
- We can find patterns that we can use to solve problems
- They are a basic type that we can perform lots of operations upon

If we have simple problem with named values then...

- we do not need addresses — variable names are enough
- there are a fixed number of values — not flexible

More complex problems require more flexible programs

- Change the size of storage to adapt to the problem
- Which part of the data to access becomes dynamic
- Cannot be written statically as a name

Example

Problem	Calculate the median of three numbers	Calculate the median of some numbers
Data	Four numbers	$n + 1$ numbers
Could loop	Yes	Yes
Must loop	No	Yes
Could use array	Yes	Yes
Must use array	No	Yes

Static and Dynamic Data

When something is static...

- we know the value when we write the program
- it is a constant with the same value every time the program runs

When something is dynamic...

- we only know what type of value it is
- the actual value can be different each time the program runs

The first place that we saw this distinction was in basic syntax

- Variables hold dynamic values, e.g. `int x; string y;`
- Constants are static value, e.g. `17, "bob", 'x'`

Static and Dynamic Behaviour

The first programs that we saw always did the same thing

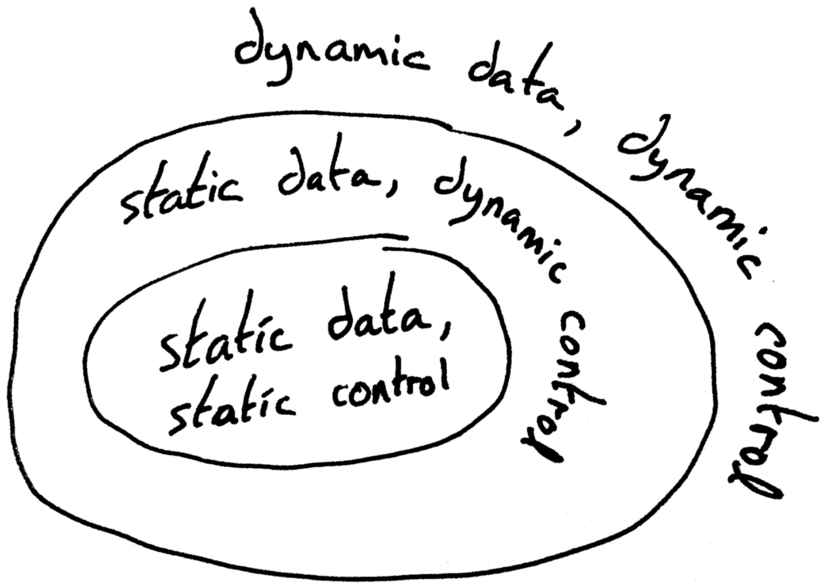
- Simple sequences of statements
- This is static behaviour, the same for all data

Then we introduced `if` and `while` statements

- Allows behaviour to change in response to the data
- This is a dynamic form of behaviour

Power of the programming language

- How many problems can it solve?
- Started with smallest and simplest case
- Now we look at more complex cases



Arrays vs Collections of variables

So far we have focused on how to use arrays

- Syntax for declaration, e.g. `int x[5], y[5];`
- Syntax for access, e.g. `x[3] = y[2];`

In the simple cases above could have used multiple variables

- Five variables in the first case (`x0, x1, ...`)
- The accesses become normal assignment, e.g. `x3 = y2;`

But in more complex cases this breaks down

- Dynamic access can be expressions, e.g. `x[i] = 3;`
- But with just variable names we cannot do this...

```
if(i==0) x0 = 3;
else if(i==1) x1 = 3;
...
```

Arrays vs Memory

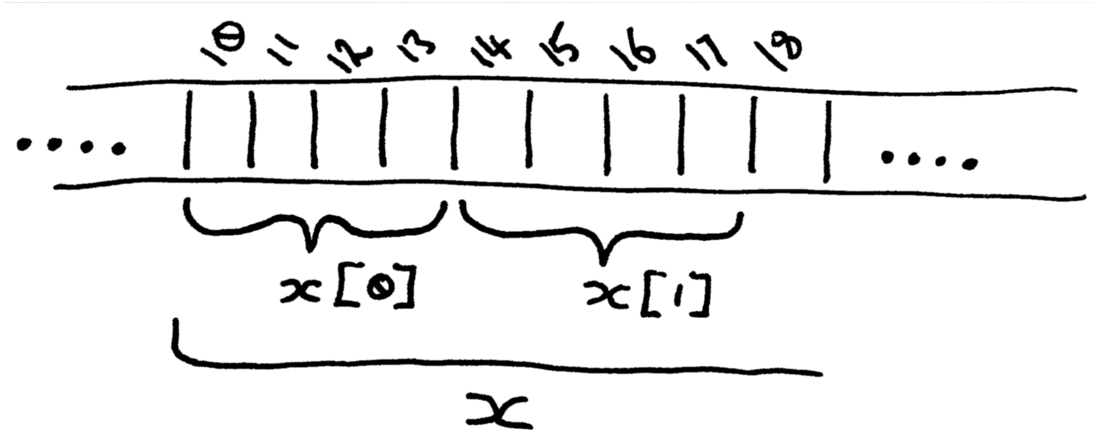
An array is a collection of variables

- Elements have a name that is part label in the program
- But part is now a number that we can do arithmetic upon
- It has become dynamic data

Both are a series of values, with numeric labels

- For memory each address is a single byte
- For an array each index is one element of the base type

Consider an array of ints on a 32-bit machine



Pointer Syntax

Array syntax is a special case of a pointer

- Each address is a number
- We can store numbers and perform arithmetic on them
- So we can manipulate addresses within the program

Pointer types are indicated with a star

- `int x;` name addresses that hold an int
- `int *y;` name addresses that hold the address of an int

Find the address of something with an ampersand

- `y = &x;` sets y to the address of x

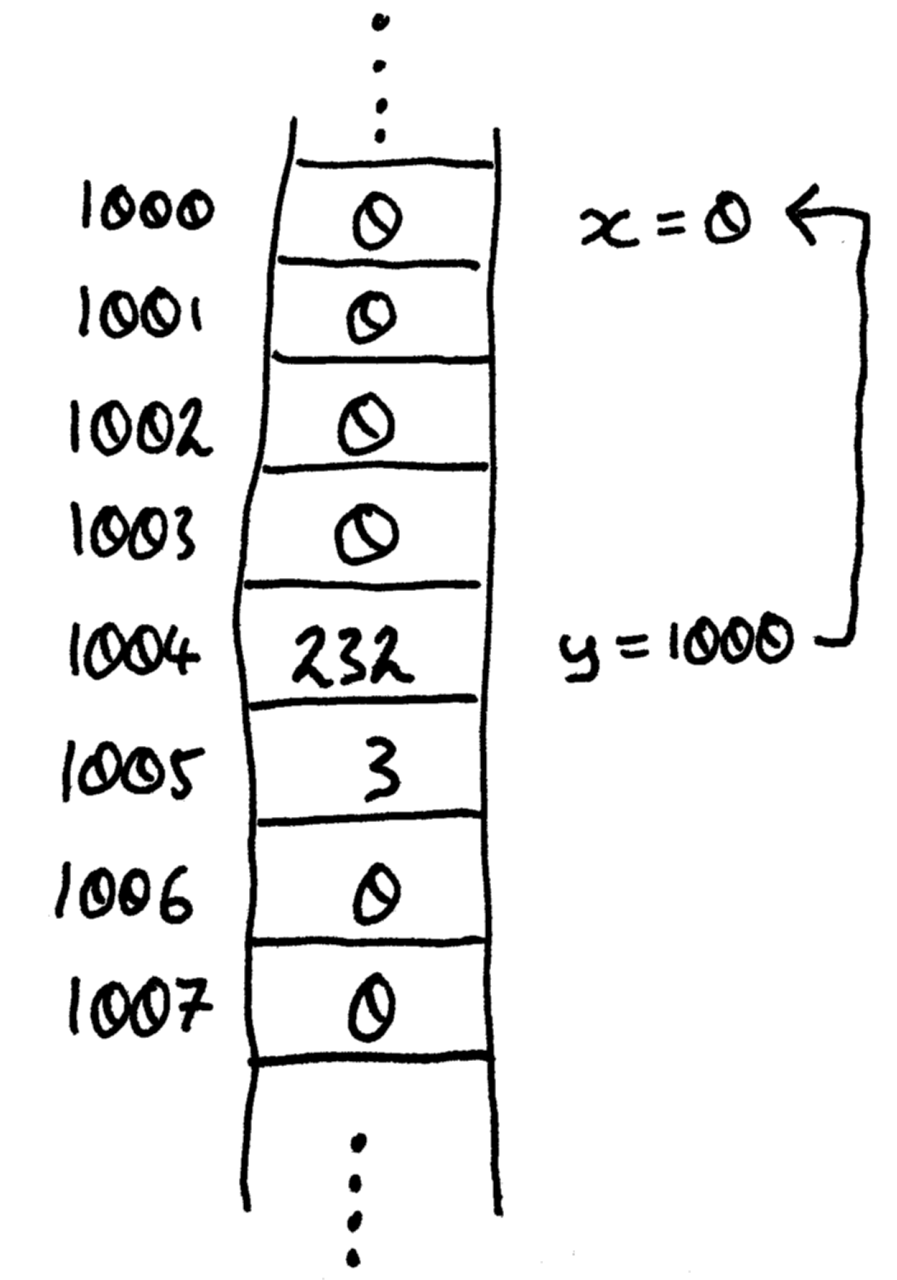
Pointer Syntax II

Consider these declarations

- `int x, y=&x;`
- One int variable
- One int pointer
- The pointer is set to the address of the variable

We say “y points to x”

- We can access x “through the pointer y”
- This also uses star syntax
- `*y = 5;`
- Or we could say `y[0] = 5;`



Break

Pointers and Arrays

Arithmetic on pointers uses the size of the type

- Consider a pointer to an integer `int *x=1000;`
- The value of the expression `x+1` may not be `1001...`
- It depends on how many bytes are in an int
  - Remember, on a 32-bit architecture this will be 4
  - On a 64-bit architecture this will be 8
- So on a 32-bit machine `x+1` would equal `1004`

Arithmetic on pointers is similar to arithmetic on array indices

- In an array `x[0]` would be the first value, `x[1]` the second...
- The addresses probably jump by more than one each time
- Using a pointer `*x` would the first value, `*(x+1)` the second...
- In general addition on a pointer accesses the target memory like an array

Equivalence

`x[i] = *(x+i)`

Dynamic Allocation

So far we have seen how to perform dynamic allocation, e.g.

Dynamic Allocation Example

```
int *x=new int[10];
```

- The type in the declaration (`int *`) is a pointer
- We can access it later using array syntax, e.g. `x[3]`
- The pointer variable is being initialised to a value
- The value is an address on the heap, with space reserved
- The `new` operator takes a type, and an array size

The variable `x` is a local variable, but the target array is not

- The pointer will be deallocated when the function exits
- But the array will remain allocated

Separating the pointer from the storage

Copying pointers vs copying arrays
Returning pointers from functions as values
Multiple pointers to same memory

2D Arrays

Arrays with more than one dimension are not built into the language

- NOTE: all static sizes is a special case we ignore
- Given two sizes w and h, we need wh elements
- Given indices x and y we need to identify the chosen element

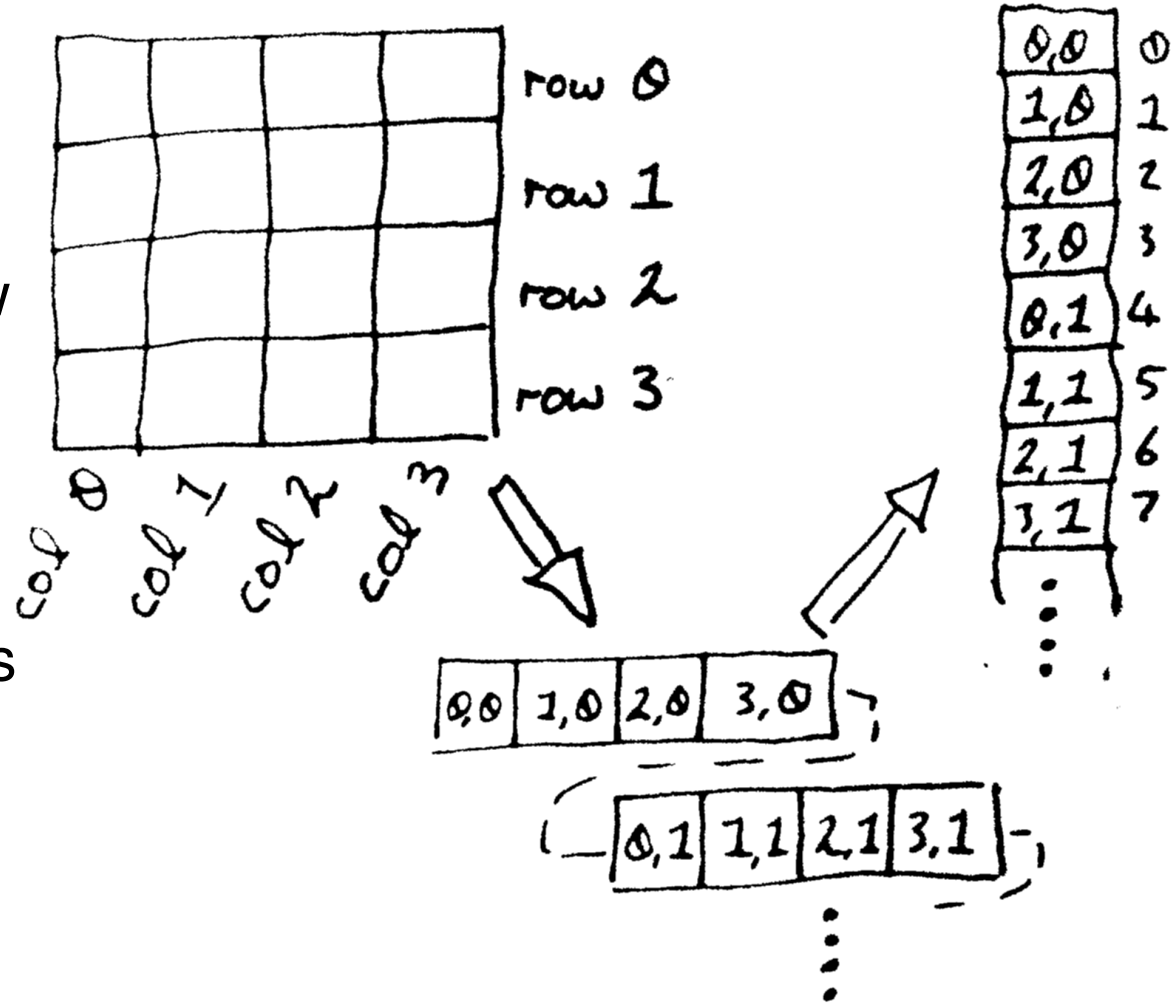
We need to build this out of something the language provides

- Using 1D arrays and/or pointers
- Two basic approaches
  - Manually striding the indices into a 1D array
  - Using an array of arrays

Approach I : Striding a 2D array into 1D

Overview

- Split the 2D array by row
- Lots of 1D strips
- Concatenate them
- Result is 1D array
- Map 2D to 1D for access
- $s(x, y) = yw + x$
- Useful to map 1D to 2D
- $s^{-1}(i) = i \bmod w, \lfloor \frac{i}{w} \rfloor$



This way is more complicated, but just uses what you have seen so far

Approach II : Arrays of arrays using pointers

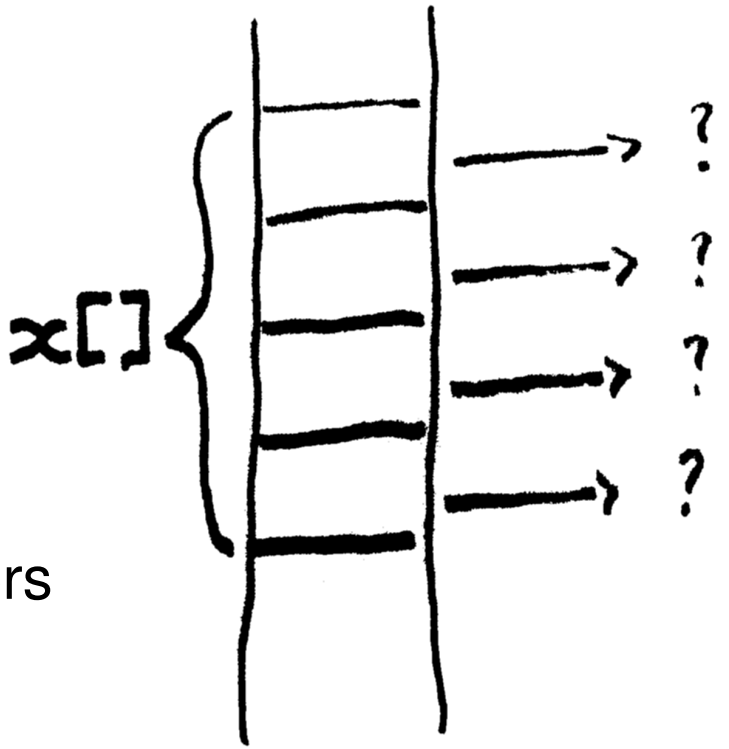
Life would be simpler if we could use an array of 1D arrays

- Because pointers and arrays are (roughly) the same, we can...
- The idea is that `int[][]` ≈ `int*[]` ≈ `int**`
- We read the final type as “an int pointer pointer”
- Or, a pointer to a pointer to an integer
- The syntax looks quite natural

```
int **x = ...
...
x[2][3] = x[0][0];
```

- We just need to fill in the dots

Approach II : The outer array



The outer array is a collection of int pointers

- `int **x = new int*[4];`
- This allocates the memory for the pointers
- Access each pointer using array notation
  - e.g. `x[0]` is the first pointer
- Not yet allocated the memory for each int array

Approach II : The inner arrays

```
int **makeArray(int w, int h)
{
int **x = new int*[h];
for( int i=0; i<h; i++)
  x[i] = new int[w];
return x;
}
...
int **array = makeArray(4,4);
array[0][0] = 3;
```