# Programming and Data Analysis for Scientists

**C++ Workshop 3**

Arrays and Vectors in C++

University of BRISTOL

**Prof Stephen Clark**

SCIF20002

# Arrays and Vectors in C++

The purpose of this workshop is to explore the concept of arrays and vectors in the C++ programming language. The *learning objectives* are:

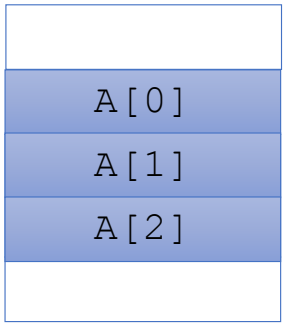
- Know how to define and use arrays in C++
- Understand their limitations (compare to Python lists)

```
x = [3, 'a', 3.2, 'bananas', [3, 2]]  # A Python list declaration!
```

- Learn the about the C++ **vector** container class
- See how they compare to arrays and allow dynamically sized data

# Declaring arrays in C++

In most programs we need collections of numbers stored in arrays. In C++ arrays are **fixed** in *size* and *type* declared explicitly in your source code:

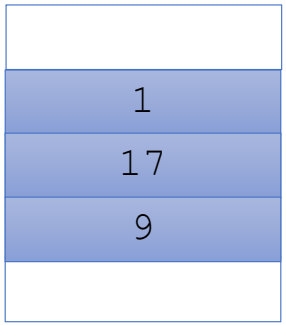



```
int A[3]; // Define an array of 3 integers
```

An array corresponds to a contiguous block of memory storing each element.

Access and assign elements as:

```
// Set the values in A
A[0] = 1;
A[1] = 17;
A[2] = 9;
```




Can initialize when declaring:

```
int A[3] = {1, 17, 9}; // Define and initialise array of 3 integers
int B[]  = {8, 23, 256, 76, 10}; // Define and initialise array with size implied
```

# Declaring arrays in C++

We define multi-dimensional array by specifying multiple sizes:

```
int A[3][3]; // Define a 2D array of 3 x 3 integers
```

Elements accessed via multiple indices `A[i][j]`.

Still corresponds to a contiguous block of memory (row-major ordering in 2D case).


```
float M[2][2] = {{2.1, 1.9}, {-3.0, 4.5}};
```

$$\begin{pmatrix} \texttt{M[0][0]} & \texttt{M[0][1]} \\ \texttt{M[1][0]} & \texttt{M[1][1]} \end{pmatrix} = \begin{pmatrix} 2.1 & 1.9 \\ -3.0 & 4.5 \end{pmatrix}$$

Generalises straightforwardly to higher dimensions …

```
int A[3][3][10]; // Define a 3D array of 3 x 3 x 10 integers
```

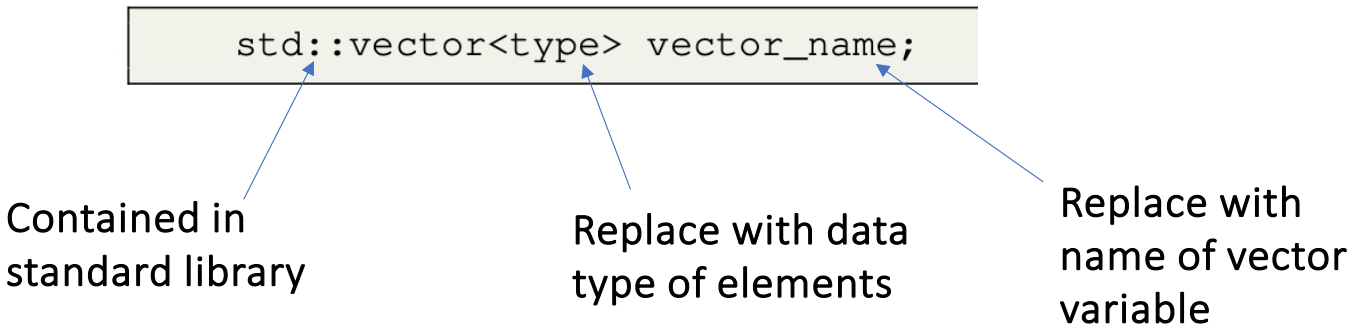
# Vectors in C++

Arrays defined so far are a relic of C. They have issues:

(a) they have fixed compile-time size
(b) they don't carry with them information about their size

C++ introduces more modern container classes to make handling data easier and safer. We will focus on one very versatile class `vector`.

We declare a vector using the (seemingly weird) notation:



# Vectors in C++

Examples best illustrate vector declarations:

```
std::vector<double> data;
```

Empty vector that will contain `doubles`

```
std::vector<int> zeros(10);
```

Vector of 10 `ints` which by default are zero

```
std::vector<int> ones(10,1);
```

Vector of 10 `ints` which are all set to one.

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13, 17};
```

Vector of `ints` explicitly specified by `{ … }` notation with size determined by the list

# Vectors in C++

Can use vectors just like arrays:

Initialised to compile-time fixed size here

Access elements using [ . ] index notation

```cpp
#include <iostream>
#include <vector> // Add this header to use vectors

int main()
{
  static const int num = 10; // Define number of elements
  std::vector<int> squares(num);  // Creates a vector with num elements.

  for (int i = 0; i < num; i++)
  {
    squares[i] = i * i;  // Vector element indexing.
    std::cout << "The square of " << i << " is " << squares[i] << std::endl;
  }

  return EXIT_SUCCESS;
}
```

# Vectors in C++

But we can also leave the vector size to be specified at run-time:

Let user specify size

Initialised to run-time size here

Can add elements to the end of the vector using `push_back()`

```
1  #include <iostream>
2  #include <vector> // Add this header to use vectors
3
4  int main()
5  {
6    int num = 0; // Will store number of elements
7    std::cout << "Number of elements num = ";
8    std::cin >> num;
9    std::cout >> std::endl;
10
11   std::vector<int> squares;  // Creates empty vector
12
13   for (int i = 0; i < num; i++)
14   {
15     squares.push_back(i*i);  // Append to the end
16     std::cout << "The square of " << i << " is " << squares[i] << std::endl;
17   }
18
19   return EXIT_SUCCESS;
20 }
```

# Multidimensional vectors

We can define multidimensional vectors as vectors of vectors:

```
std::vector<std::vector<float>> A = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
std::vector<std::vector<float>> C(Arows, std::vector<float> (Bcols, 0));
```

Can specify elements explicitly in declaration as with arrays

The type of element of the first vector is a vector itself!

Access elements using [ . ] [ . ] indexing

```
for(int i=0; i<Arows; i++)
{
  for(int j=0; j<Bcols; j++)
  {
    for(int k=0; k<Acols; k++)
    {
      C[i][j] += A[i][k]*B[k][j];
    }
    std::cout << C[i][j] << " "; // Display result
  }
  std::cout << std::endl;
}
```

# *Optional* content on **Pointers**

The following slides give some overview of the optional background on pointers available on the course website.

None of this material is required !!! – it is included for completeness and interest only …

# Declaring pointers in C/C++

Variables store data. Pointers store memory addresses to variables. We declare them just like variables, but with a * :

```
int *pointer1; // A style preferred by C programmers emphasising expressions
int* pointer2; // A style preferred by C++ programmers emphasising types
```

Three basic usages of pointers:

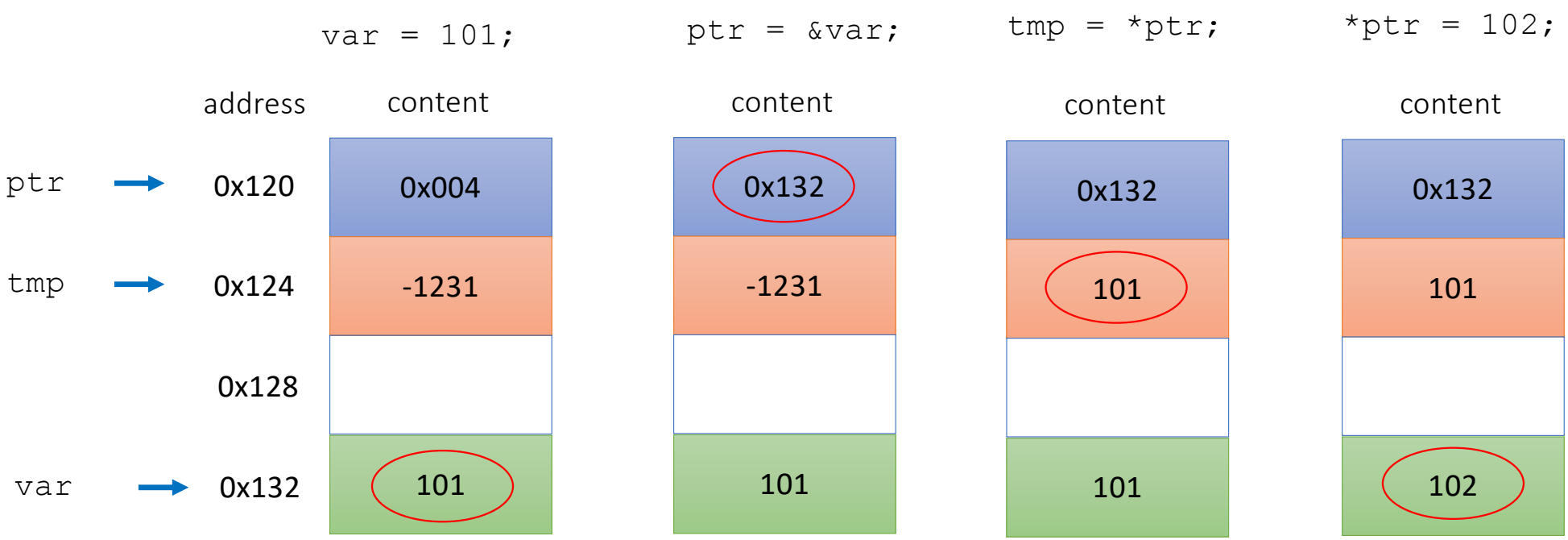
- *Referencing*
- *De-referencing*
- *Assignment*

```
int var, tmp; // Declare int variables
int* ptr;     // Declare a pointer to int
var = 101;    // Assign var

ptr = &var; // Referencing
tmp = *ptr; // De-referencing
*ptr = 102; // Assignment
```

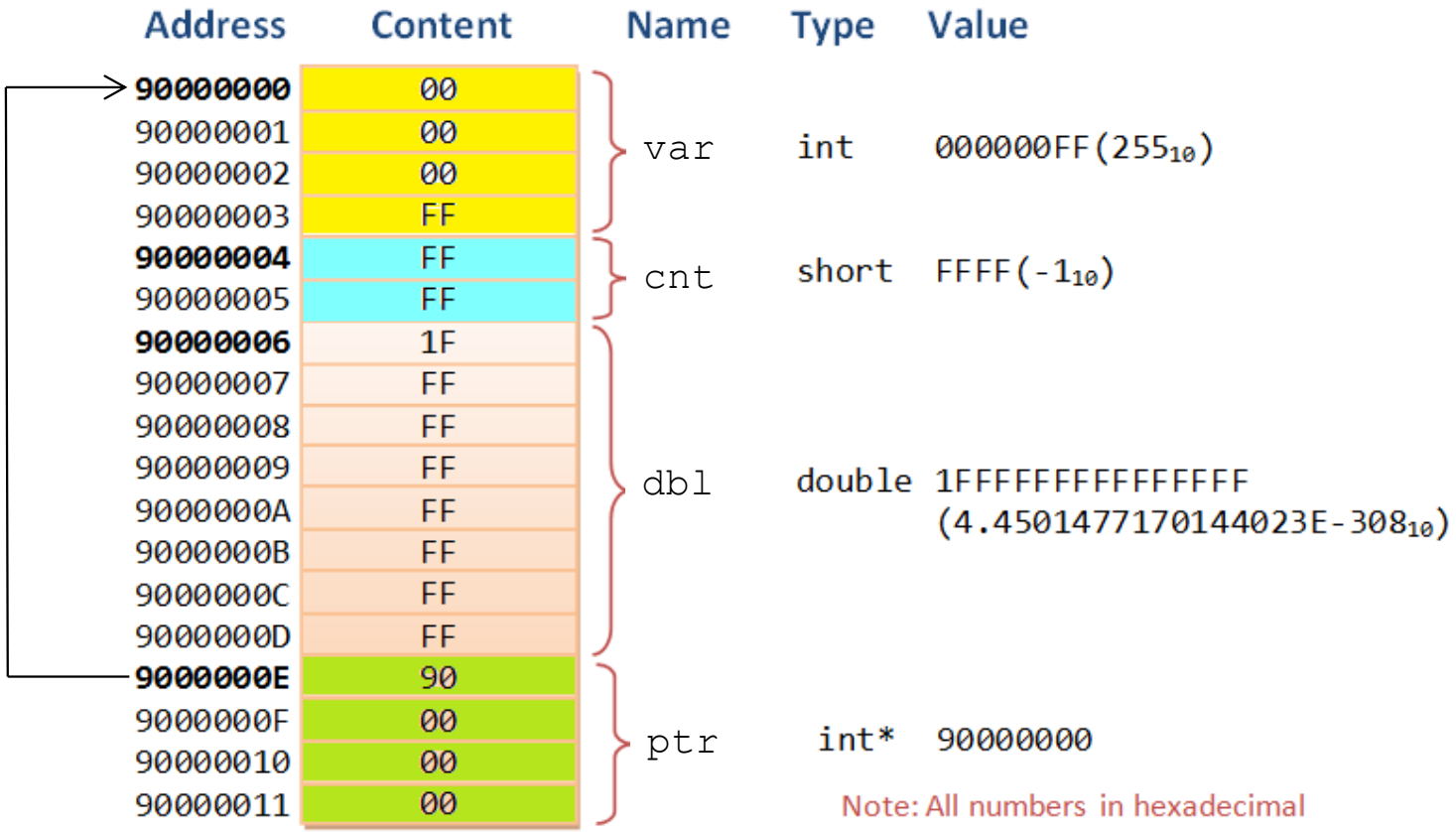
# Pointers in C/C++

These lines of code might do the following in the memory …



# Pointers in C/C++

Most processors address memory in bytes and different types occupy different sized blocks. So, a more realistic picture with other variables present is …

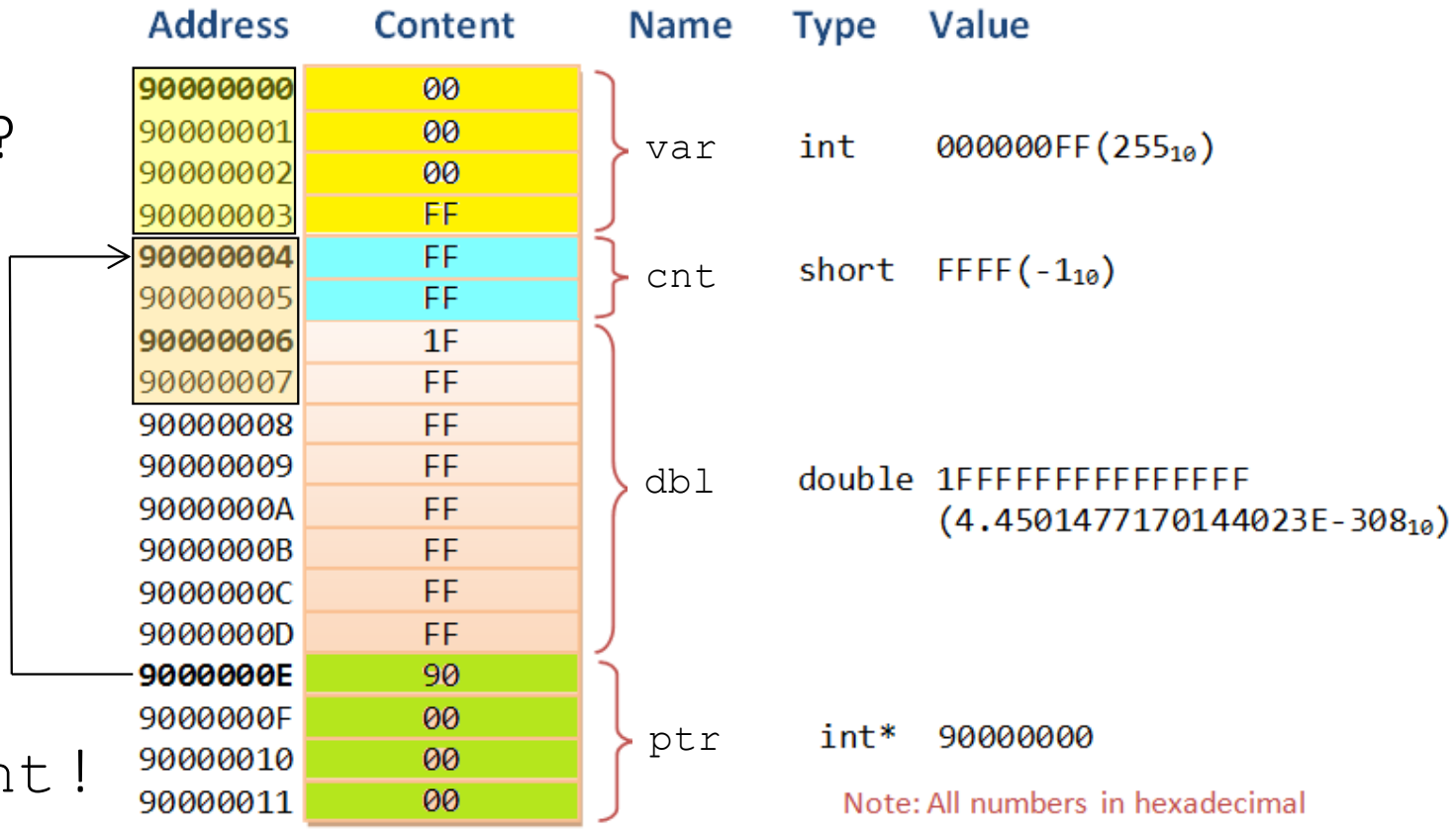


# Pointer arithmetic

What is `ptr + 1` ?

`int` pointer `ptr` is incremented by 4 bytes

Address not `int` !

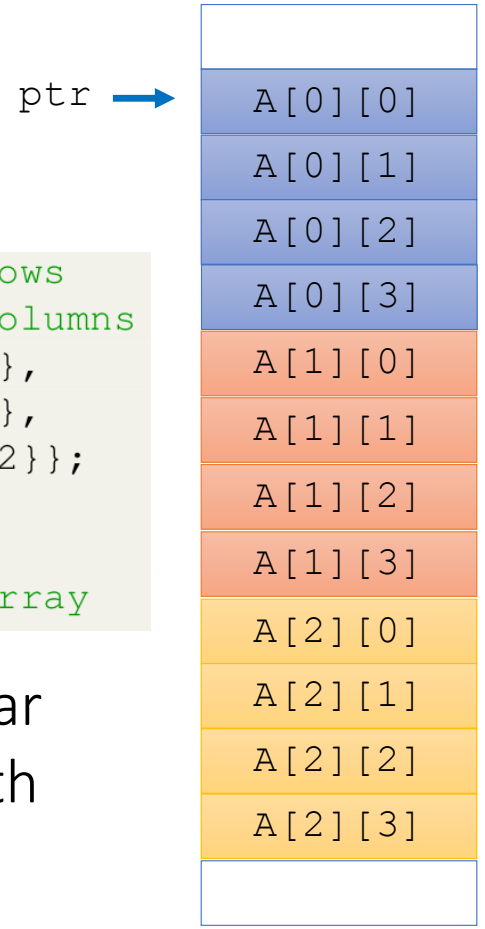


# Arrays and pointers

Take `float arr[5];`, then `arr` is a pointer equivalent to `&arr[0]`. Index notation interchangeable with pointer arithmetic: `arr[3]` ↔ `*(arr + 3)`

Not quite the same for 2D arrays:

```
5   static const int n = 3; // Number of rows
6   static const int m = 4; // Number of columns
7   float A[n][m] = {{3.14,2.71,6.28,65.43},
8                    {4.78,9.12,42.32,0.98},
9                    {8.74,3.21,90.81,24.92}};
10  float *ptr;
11
12  ptr = &A[0][0]; // Point to start of array
```

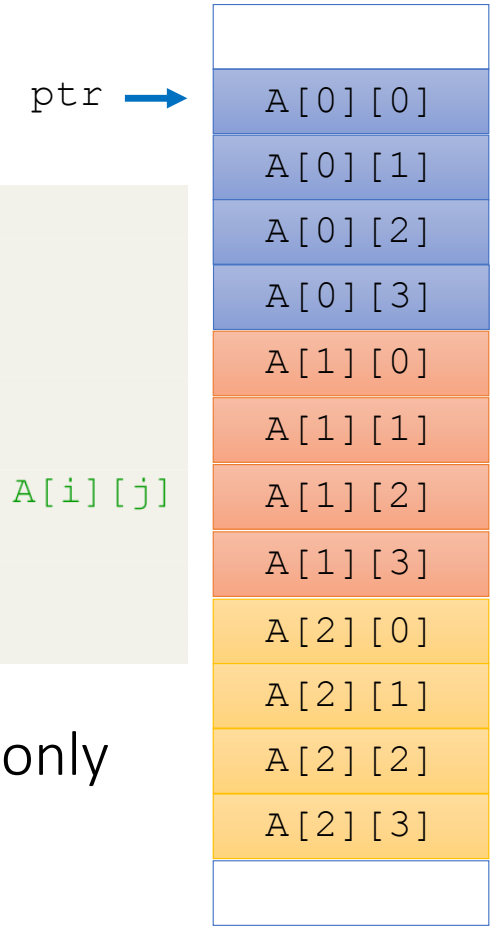



But, irrespective of their dimension, arrays are always linear <u>contiguous</u> blocks of memory which we can index over with one counter: `k = 0,1,2,..., n*m`

# Arrays and pointers

Row and column 2D array indexing can be replicated for a flattened 1D array:

```
18  // Access elements as 2D through pointer:
19  for(unsigned int i=0;i<n;i++)
20  {
21    for(unsigned int j=0;j<m;j++)
22    {
23       unsigned int k = j + m*i; // Row-major indexing
24       std::cout << ptr[k] << " "; // Here ptr[k] is equivalent to A[i][j]
25    }
26    std::cout << "\n"; // Display as a matrix
27  }
```




The advantage of this is when we need arrays whose size is only known at run-time …

# Dynamic memory allocation

We can define dynamically sized arrays by requesting a block of memory:

```
5   int *arr; // Pointer which will point at the memory we will be allocated
6   int num; // Size of our array to be specified at run-time
7
8   std::cout << "How many integers? ";
9   std::cin >> num; // Request a size
10
11  size_t arraysize = num*sizeof(int); // Number of bytes required
12
13  arr = (int*) malloc(arraysize); // Request memory and cast output as int pointer
```

We can construct multi-dimensional arrays by just requesting a large enough block and indexing over it one-dimensionally.