Ch. 16 Dynamic Arrays
Ch 17 Vector Class

Objectives

- Allocating arrays dynamically
- Using Destructors
- Flex Array
- Vector Class

Dynamic memory allocation functions in C++

```
new  TypeName;
```

- Allocates a block of memory large enough to store this type and some bookkeeping information.
- Initialize the memory using the TypeName constructor.
- Returns a pointer to the beginning of the initialized object.

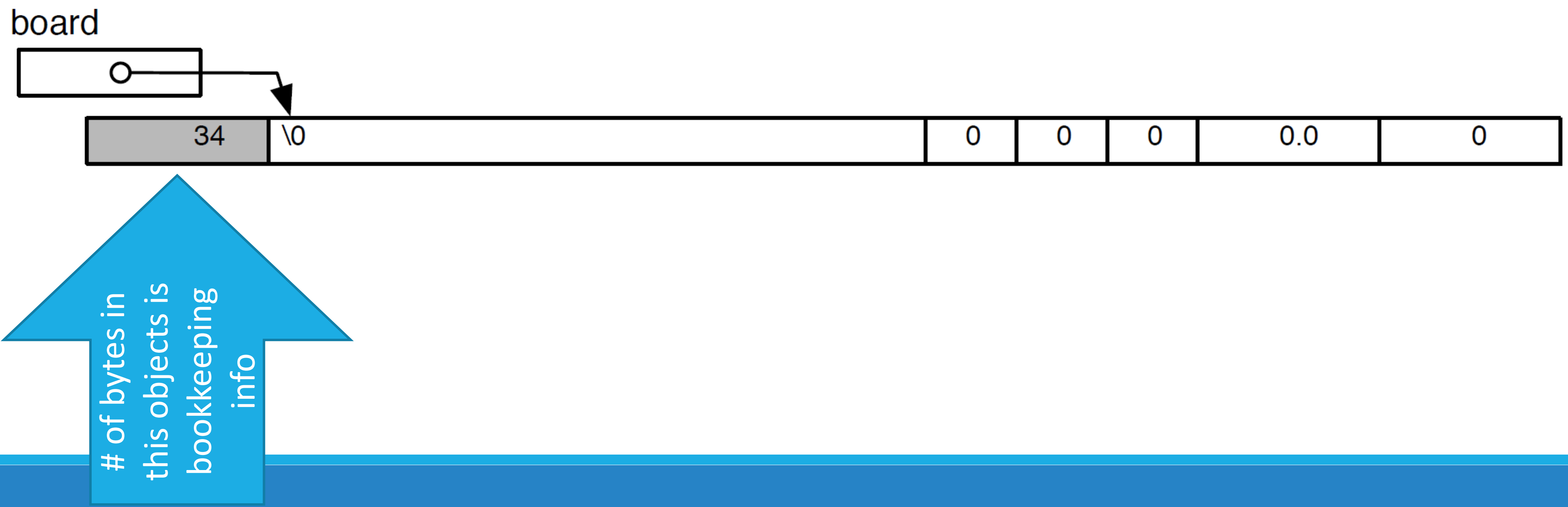
Dynamic memory allocation functions in C++

```
new TypeName[n];
```

- Allocate a block of memory large enough to **store an array of n objects** of this type, plus some bookkeeping information.
- Initialize the memory by **using the TypeName default constructor n times.**
- Return a **pointer to the beginning** of the array.
- **The array length, n must be stored as part of this allocation** because it is needed later to free the memory.

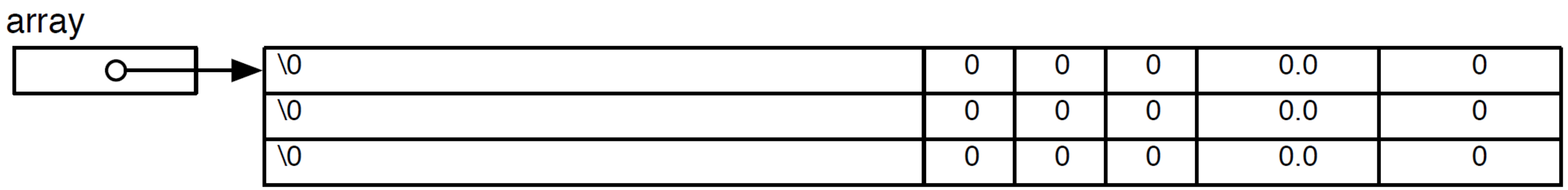
Allocating an object with a pointer

LumberT* board = new LumberT; // Allocate one object.



Allocating an Array of objects with a pointer

LumberT* array = new LumberT [3]; // Allocate three objects.



Dynamic memory De-allocation functions in C++

In order to make memory allocation dynamic, you must be able **to release memory no longer in use**.

This is the function of the **delete**. It is the opposite of allocation.

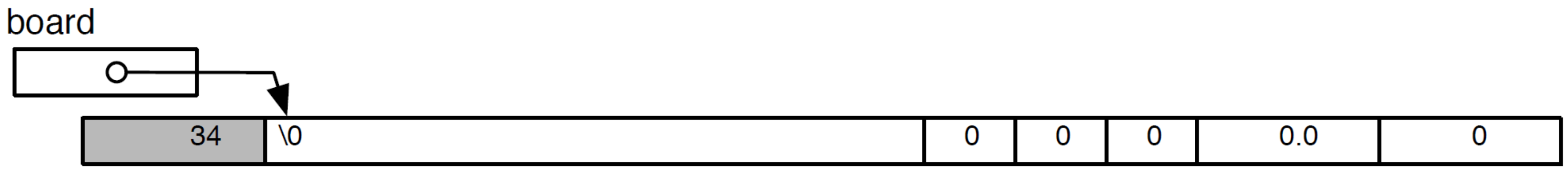
This is done by the **destructor** for objects. Same name as the constructor with ~ in front.

Dynamic memory De-allocation functions in C++

The allocation is somewhat more complex.

Some additional storage is allocated to allow the system to manage the dynamic memory. The area is, at least, the size of a long integer, and the location is normally immediately preceding the first byte of the new object.

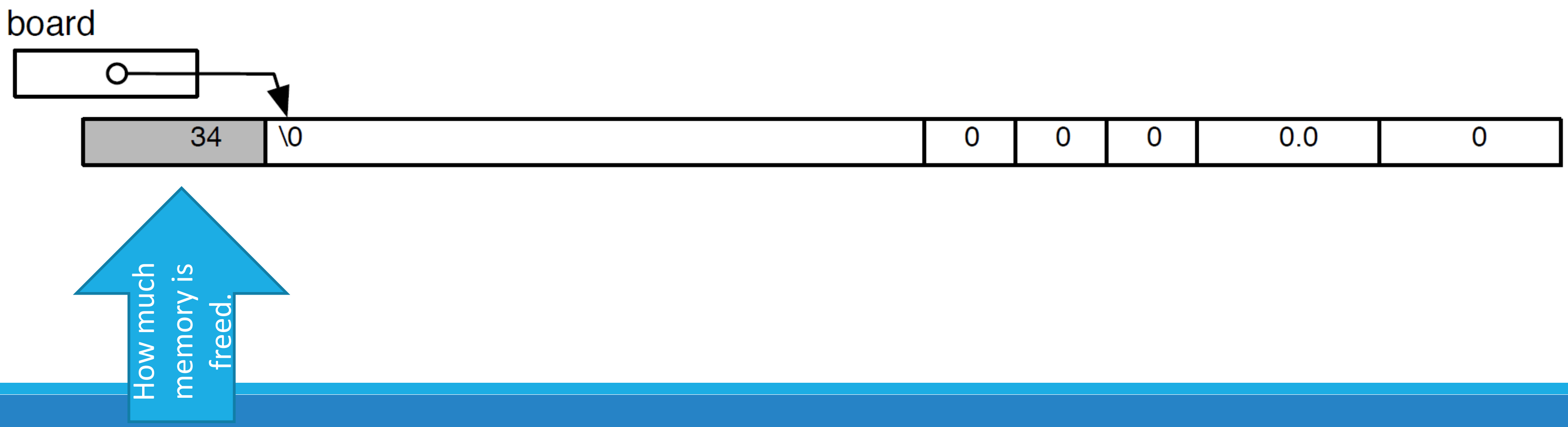
The **gray area** in the diagram represents these additional bytes that the C++ system sets aside; it **stores the total size of the allocated block** (the size of a **size_t value plus the size of the white area**). The importance of these bytes becomes clear when we discuss delete.



Dynamic memory De-allocation functions in C++

Dynamically allocated memory can be recycled by calling delete, which frees a block of memory by returning it to the system's memory manager.

The number of bytes in the gray area at the beginning of each allocated block determines how much memory is freed.



Dynamic memory De-allocation functions in C++

```
delete pointerName;
```

Recycle the memory block pointed at by the pointerName. Return it to the operating system for possible future use. A block that was **created using new should be deleted** when it is no longer needed by the program.

Objects that were created by declarations **must never be manually deleted**. (Deletion for local variables **happens automatically when the variable goes out of scope**.)

```
delete[] pointerName;
```

Recycle the entire memory block pointed at by the pointerName. **First, however, run the destructor for the base type of the array n times to free all memory attached to the objects stored in the array.**

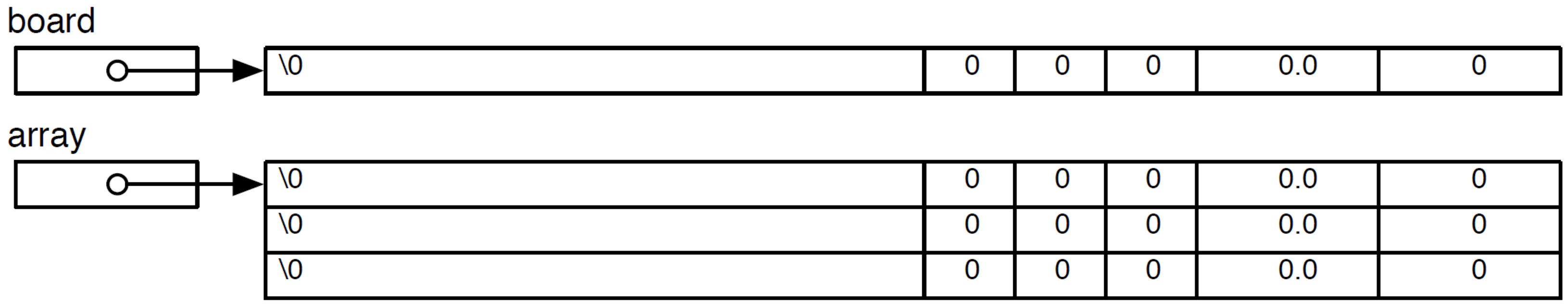
Dynamic memory De-allocation functions in C++

Simple Memory Allocation

When a programmer cannot predict the amount of memory that will be needed by a program, the **new can be used at run time to allocate the desired number of bytes.** For example:

```
LumberT* board = new LumberT;              // Allocate one object.
LumberT* array = new LumberT [3];          // Allocate three objects.
```

Before returning, the default constructor for LumberT will be called to initialize the objects. Conceptually, this code allocates memory areas like the ones in Figure 16.10



Using dynamic allocation with a class

When a class **uses new to allocate a dynamic array** in the constructor, it **must have a companion destructor that deallocates the data using `delete`.**

In the destructor, the **~**is used before the constructor name to align the two.

Flex Class – Flex Arrays

We want to replace the existing (too small) memory block by a larger one that contains the same data. To do this we define a class that contains the dynamic array, the two integers needed to manage it, and functions to handle resizing and related needs. The Flex class needs three data members:

- The dynamic array.
- The current capacity of the array, max (the number of slots in it).
- The number of data items currently stored in the array, n

At least two function members are needed.

The first `is push_back()`, which puts data into the first unoccupied slot in array.

The second is `grow()`, which actually does the resizing.

Flex Class

The push_back() method has two steps:

- Check whether there is currently space in the array for another data item. If not, call grow().
- Whether or not growth happened, put the additional data item into the first empty slot in the array, and increment n.

Flex Class `grow()`

The **`grow()`** method has several steps:

- Use a **temporary variable to point at the existing array**.
- **Double max**.
  - By doubling the array capacity each time, we guarantee that the time required to reallocate and copy data will always be acceptable, and that the total number of data items copied in the lifetime of the array will not exceed the current length of the array. This is the most efficient strategy.
- Allocate a **new memory area with the new max capacity**.
- **Copy n data objects from the old memory into the new area.**
- **Delete the old memory area**.

Finally, C++ allows a program to define the subscript operator for that class. Doing so allows the programmer to treat the Flex array as if it were an ordinary array.

#Pragma is Good Housekeeping

The **`#pragma once`** statement tells the compiler to include this file once, but if it has already been included in a module, do not include it again.

It is good style to include `#pragma once` whether or not it is needed.

In Flex, it is needed in the file trans.hpp but not in the files charges.hpp and flex.hpp.

```
#pragma once
#include <iostream>
using namespace std;

#include "trans.hpp"
typedef Transaction BT;
```

BT can be any base type

```
#define START 4                       // Default length for initial array.

class Flex {
private:  // --------------------------------------------------------------
    int max = START;                  // Current allocation size.
    int n = 0;                        // Number of array slots that contain data.
    BT* data = new BT[max];           // Allocate dynamic array of base type BT.

    void grow();                      // The Flex is full, so double the allocation.

public:  // --------------------------------------------------------------
    Flex() {}                         // An array of Transactions.
    ~Flex() { delete[] data; }        // Free dynamically allocated data.

    void push_back( BT data );        // Store data and return the subscript.
    int size() { return n; }          // Provide read-only access.
    BT& operator[]( int k );
    void print(ostream& out) { for (int k=0; k<n; ++k) out <<data[k] <<endl; }
};
```

Destructor

```
#include "flex.hpp"
// ---------------------------- Store object in array.  Grow first, if needed.
void Flex ::
push_back( BT obj ) {
    if ( n == max ) grow();        // Create more space if necessary.
    data[n] = obj;
    ++n;
}

// ---------------------------------------------- Double the allocation length.
void Flex ::
grow() {
    BT* temp = data;                               // Hang onto old data array.
    max *= 2;
    data = new BT[max];                            // Allocate a bigger one.
    for (int k=0; k<n; ++k) data[k] = temp[k];     // Copy info into new array.
    delete temp;                                   // Recycle (free) old array.
}

//------------------------------------------ Access the kth char in the array.
BT& Flex ::
operator[]( int k ) {
    if ( k >= n || k < 0 ) fatal("Flex bounds error.");
    return data[k];                 // Return reference to desired array slot.
};
```

Memory Pitfalls

Dangling Pointer, a pointer whose referent has been deleted (deallocated).

Memory Leaks – memory is not released after dynamic array grows.

Packaging an Array Data Structure
Why Package?
The DataPack Structure

VECTOR CLASS

Make your Life Easier

A basic organizational rule that applies throughout life: If two things are always used together, keep them together in one place.

- In C, an array has only one part (a series of variables of the same base type).
- But when you use a C array you also need two numbers:
  - the number of slots that are in the array (allocation length)
  - the number of slots that have data in them (number filled)
- So it makes sense to bundle the array and the two numbers needed to manage it into a single structure that can be passed with one parameter to a function.

In modern OO languages, a bundle of parts that belong together is called a class. A class includes a set of related data parts and the functions that operate on them.

C++ template classes.

A template in C++ is a way to define a generic container class that can store any type of data.

- To use a template class, write its name followed by angle brackets.
- The type of data stored is given in the angle brackets.
- The type name in the angle brackets can be any primitive type or previously-defined class type.
- Sample declarations: vector<int> or vector<BT>
- Writing this declaration instantiates the template class and produces a normal class.
- The name of the resulting class is vector<BT>

The vector template class.

vector is the name of an STL template class in C++.

- It is an array-like class that can "grow" to hold as much data as needed.
- To implement this capability, a vector needs at least these three data members:
  - A pointer to a dynamically-allocated array of objects of the base type (BT).
  - max, the number of array slots that are currently allocated.
  - size, the number of BT values currently stored in the array.
- When n equals max, the array is full.
- The next insertion will cause the array to be reallocated at double the length.
- The data will be copied into the new array and the old one will be freed.

Vector functions

vector provides these functions and many others:

- **size()**: the number of data values currently stored in the vector.
- subscript: **Use []** as you would with an ordinary array
- **push_back()**: Store a new value at the end of the vector. This is the normal way to store data in a vector.
- **data()**: a pointer to the first slot of the data array that is inside the vector.
- **begin() and end():** These return iterators (like pointers) to the first data item in the vector and the address just past the end of the vector. Use them with sort.
- **sort(iterator start, iterator end):** Sorts the data in the vector between start and end.

Using a vector in C++

- Assume the istream named fin is properly open.
- Read numbers from a file into the vector named vec, starting with the first unfilled slot.

```
typedef double BT;
vector<BT> vec;
for(;;) {
 fin >> data;
 vec.push_back( data ); //C++ 11 required
 if( fin.eof()) break;
}
```

- The vector class keeps track of the number of items that have been pushed into it.
- At the end of the operation, the value of n in the vector will correspond to the number of items in the array.

Max and Min

ARRAY ALGORITHM USING A VECTOR

Review: Max and Min functions

- Many array algorithms depend on finding the maximum or minimum value in an array.
- It is simple to write a max or min function, but we'll review the algorithm here to be sure we have a common understanding.
- We show the code for min, assuming that you can convert it to max yourself.
- This algorithm is written in terms of an **abstract type BT**. To use the code, **write a typedef at the top** to say **what type of data** you are using.
- We use this **typedef trick to make the code easy to reuse for any type** that defines operator <. Just change the typedef for BT.
- C++ allows a programmer to define operator < for any class. It is predefined for all the primitive types.

Code for min() using vector.

Return the subscript of the minimum value in the vector.

Note that the loop stops at vec.size() , because vector positions beyond that do not contain data.

```
typedef double BT;
int min( vector<BT> vec ); //function prototype
int main(){ //code to create vector and call function goes here
}//end of main()

int min( vector<BT> vec ) {
  int minPosition = 0;
  BT currentMin = vec[0];
  for (int k=1; k<vec.size(); ++k) {
    if ( vec[k] < currentMin ) {
     currentMin = vec[k]; // A new minimum.
     minPosition = k; // Remember where it is.
    }
  }
  return minPosition; // Index of min element.
}
```

In Class Exercise

Modify the program on slide 28 (previous slide) to find the max value in the array. Do your own work on this.

Show me when you’re done.

Two-Dimensional Arrays
Static Arrays of Arrays
Dynamic 2-D Arrays
Dynamic Matrices

ARRAYS OF ARRAYS

Two-Dimensional Arrays

C/C++ gives **half-way support** for

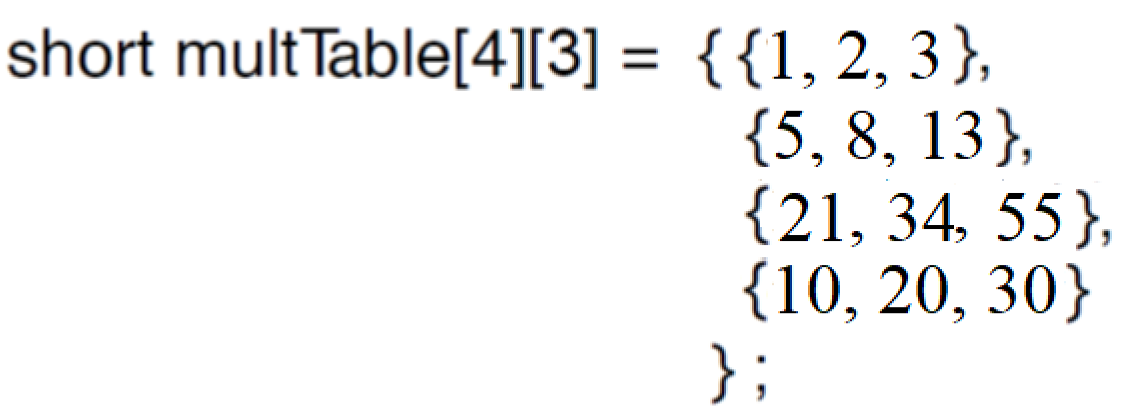
2-dimensional arrays:

- You can declare **an array of arrays**,
- Or **an array of pointers to dynamically allocated arrays**.
- With both of the above data structures, you can use **double subscripts**.
- You can also declare a **dynamically allocated matrix**, but you can't use double subscripts with it.

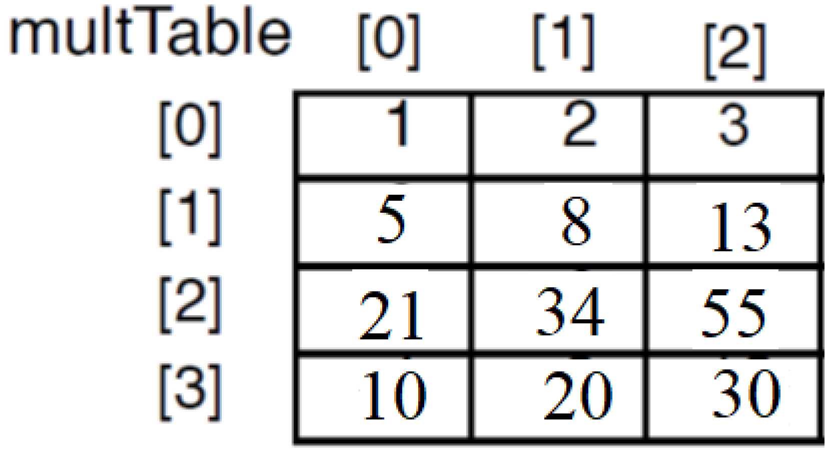
We will look at arrays of arrays first.

Declaration and Initialization

An **array of arrays is declared with two subscrip**ts and initialized by a series of array initializers:



```
short multTable[4][3] = {{1, 2, 3},
                          {5, 8, 13},
                          {21, 34, 55},
                          {10, 20, 30}
                         };
```



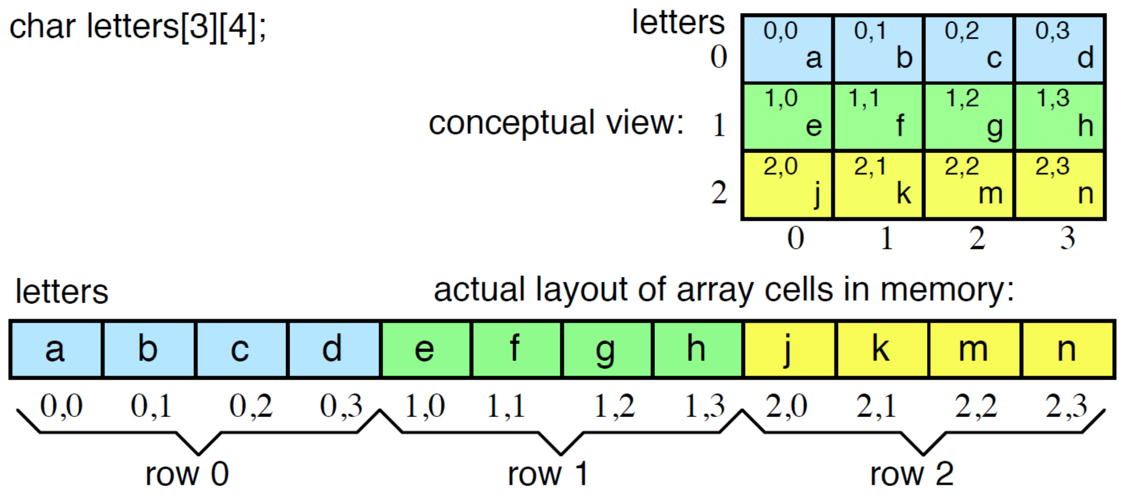
multTable	[0]	[1]	[2]
[0]	1	2	3
[1]	5	8	13
[2]	21	34	55
[3]	10	20	30

To **refer to an array slot**, the **row subscript is given first**: multTable[1][2] is 13.

Using an array of arrays.

```
const int rows = 10, cols = 10;
int addTable[rows][cols];
int row, col;
for( row=0; row<rows; ++row )
 for( col=0; col<cols; ++col )
  addTable[row][col] = row + col;
```

Memory Layout



The address translates as

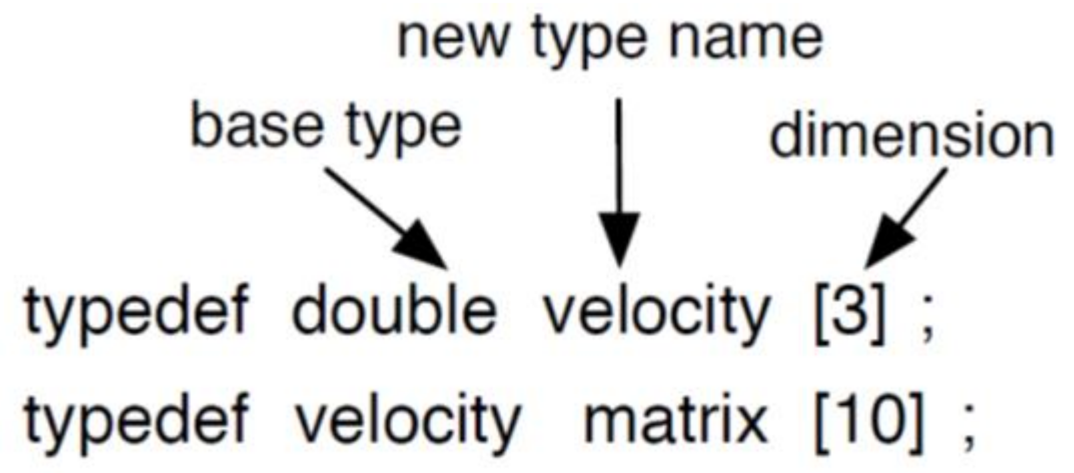
letters[r][c] = letters + r*ncols + c

Notes:

1. The name of an array is translated as the address of its first slot.

2. Address arithmetic in C is done in terms of array slots, not bytes.

3. Process arrays with **row in outer loop and column in inner loop** so it **follows order of memory locations.**



```
typedef  double  velocity  [3] ;
typedef  velocity   matrix  [10] ;
```

Array Type Definitions

Option 1:

Use two typedefs to name both the 1-dimensional and 2-dimensional parts of your data structure.

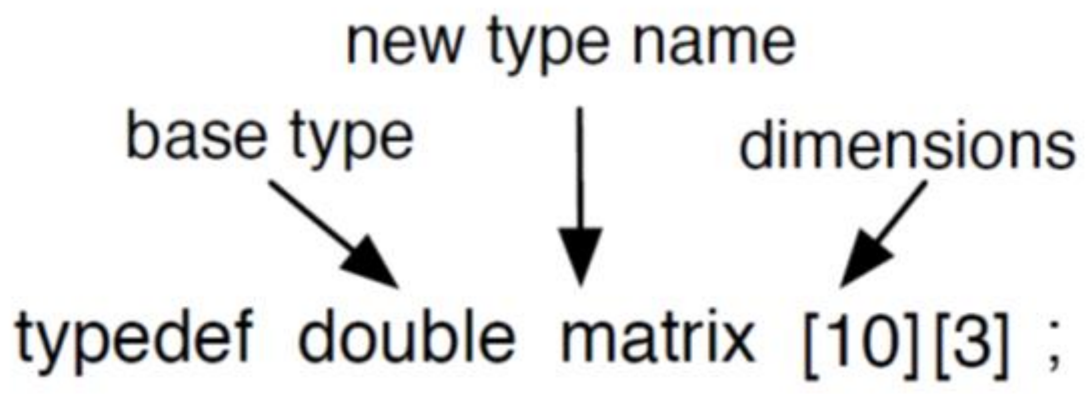
You can use double subscripts with this kind of typedef.

Array Type Definitions

Option 2:

Use this kind of typedef if the rows do not have an independent meaning.

You can use double subscripts with both kinds of typedef.

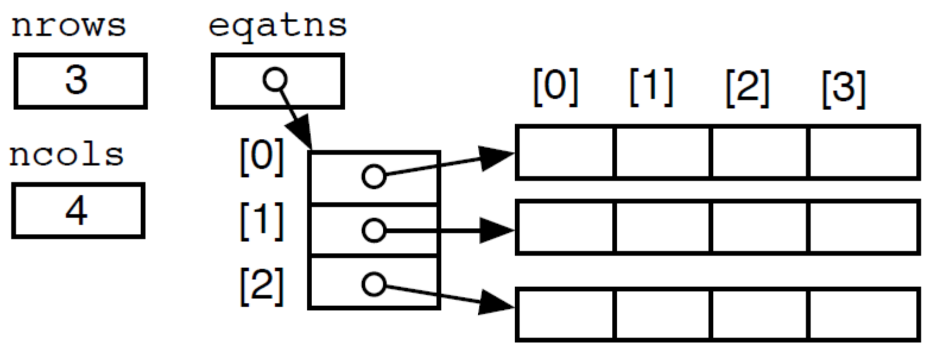


An Array of Array Pointers

If the length of a row or the number of rows is **not known at compile time**, it is possible to use dynamic allocation to **create an array of arrays dynamically.**

```
int nrows, ncols; // values to be read at run time.
BT** backbone = new BT*[nrows] ;
for (int row=0; row<nrows; ++row)
 backbone[row] = new BT[ncols];
```

This data structure is used with two subscripts, in exactly the same way as the simple static structure: `cout << backbone[1][0];`



Array Pointers

I would create this data structure to solve a system of 3 equations in 3 unknowns. The next step would be loop to read the coefficients from a file into the arrays:

```
int row, col;
for( row=0; row<nrows; ++row )
 for( col=0; col<ncols; ++col )
  instream >> eqatns[row][col] ;
```

Single-Block Matrix

RELATED TO A 2D ARRAY, A SINGLE BLOCK MATRIX ONLY HAS ONE SUBSCRIPT IN REALITY – YOU MUST CALCULATE THE SUBSCRIPT

Single-Block Matrix

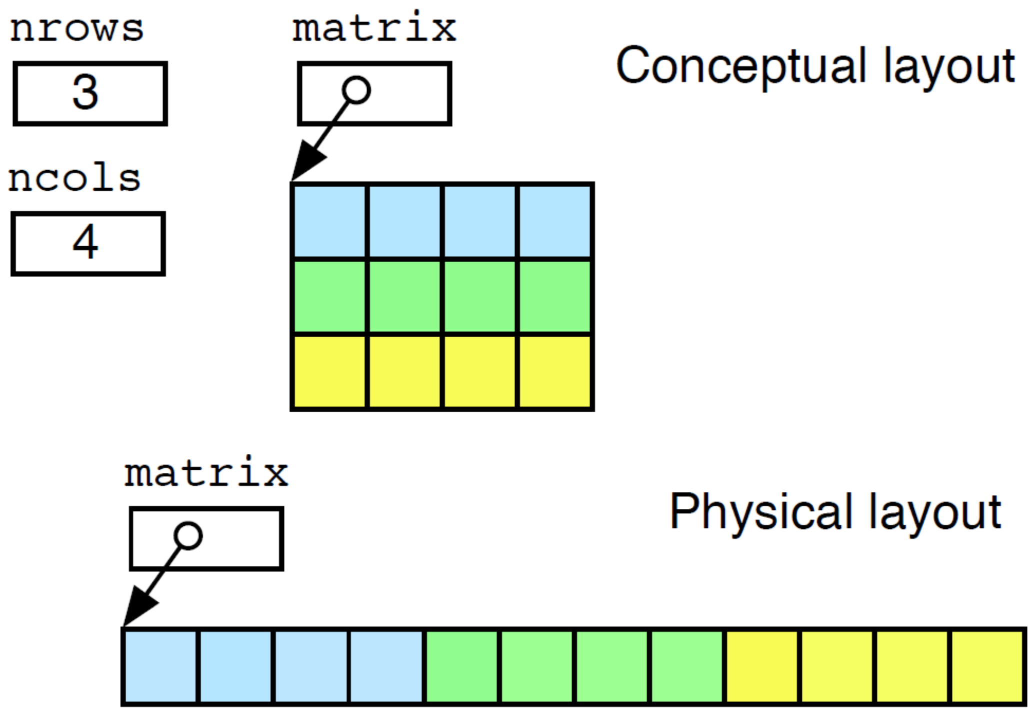
In some applications (image processing) it is important to have a **single block of memory for the whole matrix**.

- A dynamic **single-block matrix** must be created as an array of BT values.

```
BT* matrix = new BT[ numRows * numCols ] ;
```

- It **cannot be accessed using two normal subscripts**.
- **You must write and use your own subscript function** that converts the 2D subscripts into a single linear subscript:

```
subscript = row * numCols + col;
```



After the allocation

Subscripting a Single-Block Matrix

- A single-block matrix can also be processed one number at a time using a loop and a single subscript. (Assume that nrows and ncols are defined in this context.)
- Here is a less efficient way to read data into a matrix. It is appropriate for numeric data that needs input conversion.

```
int n = nrows * ncols;
BT* matrix = new BT[n];
for( int k=0; k<n; ++k )
     cin >> matrix[ k ];
```

- This technique can be used when every matrix element must be processed the same way.

2-D Subscripts for a Single-Block Matrix

Finally, a dynamic matrix can also be processed with two subscripts using a nested loop and a programmer-defined subscript function.

We assume that nrows and ncols are defined in this context.

```
int sub;
BT* matrix = new BT[nrows * ncols];
...
sub = row * ncols + col;
cout << matrix[ sub ];
```

This technique is used for random access to one cell of a matrix.

Selection Sort
Shuffle

MORE ALGORITHMS USING ARRAYS

Selection Sort using an array.

Selection sort is the second worst sorting algorithm.

- But it is useful when you need only the first few values to be sorted and are happy to leave most of the data unsorted.
- There are many variations of this algorithm.
- In this version, we search the entire array using the min algorithm to select the smallest value in the array, then swap it to the start of the array, and repeat.
- This will sort the values in descending order.
- To sort in ascending order, use a max algorithm.

Selection Sort on an array

```
void selSort( BT data[], int n ) {
  int k, last, minPosition;
  BT currentMin; // Smallest value so far.
  for (last = n-1; last>0; --last) {
   currentMin = data[0];
   minPosition = 0;
   for (k=1; k<=last; ++k) {
     if ( data[k] < currentMin ) {
      currentMin = data[k]; // A new minimum.
      minPosition = k; // Remember where it is.
     }
  }
  data[minPosition] = data[last]; // swap
  data[last] = currentMin;
} }
```

The Shuffle Algorithm

This simple loop randomizes the order of the items in an array.

- Before entering the loop, C's random number generator should be “seeded", that is, we supply a random initial value as a starting place.
- If you want unrepeatable series of random numbers, use the current time as the seed.
- If you want a repeatable series of random-looking numbers, choose some constant int as the seed.
- To get a random number r between 0 and k-1 (suitable for a subscript) set r = rand()%k
- To randomize the order of the k items in an array, select a random number between 0 and the k-1. Swap the item at that subscript with the k^{th} item (last one in the array the first time), decrement k, and repeat until k=1.

Code for Shuffle

```
void shuffle( BT data[], int n ) {
  srand((unsigned)time(NULL)); // Do this only once!
  int k, randy;
  BT swapper;
  for (k=n; k>1; --k) {
   randy = rand()%k; // Randomly choose next item.
   swapper = data[randy]; // Swap with last one (kth one).
   data[randy] = data[k-1];
   data[k-1] = swapper ;
  }
}
```

That’s All!