Packaging an Array Data Structure
  Why Package?
  The vector class

Array Algorithms
  Maximum and Minimum
  Selection Sort
  Shuffle

Arrays of Arrays
  Static Arrays of Arrays
  Dynamic 2-D Arrays
  Matrices
Packaging an Array Data Structure

Why Package?
The DataPack Structure
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. (fill level)
- 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.

```cpp
typedef double BT;
vector<BT> vec;
for(;;) {
    fin >> data);
    if( fin.eof()) break;
    vec.push_back( data );
}
```
- 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.
Array Algorithms using vectors

Max and Min

Selection Sort

Shuffle
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 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 \texttt{min()} using vector.

Return the subscript of the minimum value in the vector. Note that the loop stops at \texttt{vec.size()}, because vector positions beyond that do not contain data.

\begin{verbatim}
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.
}
\end{verbatim}
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 right end 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

```c
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 \( r = \text{rand}() \% k \)
- 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 last item in the array, decrement \( k \), and repeat until \( k=1 \).
Code for Shuffle

```c
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.
        data[randy] = data[k-1];
        data[k-1] = swapper;
    }
}
```
Arrays of Arrays

Two-Dimensional Arrays
Static Arrays of Arrays
Dynamic 2-D Arrays
Dynamic Matrices
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 subscripts and initialized by a series of array initializers:

\[
\text{short multTable}[4][4] = \{ \{1, 2, 3, 4\}, \\
\{2, 4, 6, 8\}, \\
\{3, 6, 9, 12\}, \\
\{4, 8, 12, 16\} \};
\]

To refer to an array slot, the row subscript is given first:

\[
\text{multTable}[2][3] \text{ is 12.}
\]
Using an array of arrays.

We typically process an array of arrays with nested for-loops. This example creates an addition table for the numbers 1..10

```c
const int nrows = 10, ncols = 10;
int addTable[nrows][ncols];
int row, col;

for( row=0; row<nrows; ++row )
    for( col=0; row<ncols; ++col )
        addTable[row][col] = row + col;
```
Memory Layout

char letters[3][4];

conceptual view:

```
0,0  0,1   0,2   0,3  1,0 1,1   1,2   1,3  2,0  2,1   2,2   2,3
0,0  0,1  0,2  0,3
   a    b    c   ... is translated as the address of its first slot.
```

actual layout of array cells in memory:

```
 a   b   c   d  e   f   g   h   j   k   m   n
0,0  0,1  0,2  0,3  1,0 1,1   1,2   1,3  2,0  2,1   2,2   2,3
```

The address of 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.
Array Type Definitions

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

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

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

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

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.

```c
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:

```c
    cout << backbone[1][0];
```
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:

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

The dynamic array of array pointers is a 2-D structure that fits many applications. However, 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 a flat array of BT values.
  
  ```cpp
  BT* matrix = new BT[ nrows * ncols ] ;
  ```

- 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:

  ```cpp
  subscript = row * ncols + col;
  ```
Arrays of Arrays

Matrices

After the allocation

```
<table>
<thead>
<tr>
<th>nrows</th>
<th>matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>〇</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ncols</th>
<th>matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
```

Conceptual layout

Physical layout
Using a Dynamic Matrix as One Piece

A dynamic matrix is the right data structure for images. Mass input can be done efficiently with read().

typedef unsigned char pixel;
ifstream pic;
pic.open("mypic.pgm", ios_base::in|ios_base::binary);
int n = nrows * ncols;
pixel* matrix = new pixel[n];
pic.read( matrix, n );
if (!pic.good()) cerr<<"mypic.pgm is damaged" <<endl;

After input, the image can be processed one pixel at a time.
**Subscripting a Dynamic Matrix**

A dynamic matrix can also be processed one number at a time using a loop and a single subscript. (Assume that \texttt{nrows} and \texttt{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.

```cpp
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 Dynamic 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.

```c++
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.