II. Introduction to Data Structure and Abstract Data Types — C-Style Types

A. Introduction (§2.1)

One important aspect of the design phase is the selection and design of appropriate data types to organize the data to be processed; indeed, this is the real essence of OOP (object-oriented programming).

Example 1: Trans-Fryslan Airlines (pp. 30-31)

Attempt 1:

```
enum SeatStatus {OCCUPIED, UNOCCUPIED};
SeatStatus seat1, seat2, . . . , seat10;
```

Simple data organization, but horrible algorithms for the basic operations!

ALGORITHM TO LIST UNOCCUPIED SEATS

```
1.  If seat1 is UNOCCUPIED
        Display 1.
2.  If seat2 is UNOCCUPIED
        Display 2.
3.  If seat3 is UNOCCUPIED
        Display 3.
          ⋮
10.  If seat10 is UNOCCUPIED
        Display 10.
```

ALGORITHM TO RESERVE A SEAT

```
1. Set done to false.
2. If seat1 is UNOCCUPIED do the following:
     a. Display "Do you wish to assign Seat #1?".
     b. Get response from user.
     c.  If response is 'y'  then to the following:
           i. Set seat1 to OCCUPIED.
          ii. Set done to true.
3. If not done and seat2 is UNOCCUPIED then do the following:
     a. Display "Do you wish to assign Seat #2?".
     b. Get response from user.
     c.  If response is 'y'  then to the following:
           i. Set seat2 to OCCUPIED.
          ii. Set done to true.
                 ⋮
```

Attempt 2:

```
const int MAX_SEATS = 10; // upper limit on the number of seats

enum SeatStatus {OCCUPIED, UNOCCUPIED};
typedef SeatStatus SeatList[MAX_SEATS];

SeatList seat;
```

More complex data organization, but much nicer algorithms for the basic operations!

ALGORITHM TO LIST UNOCCUPIED SEATS

```
1. For number ranging from 0 to MAX_SEATS - 1 do the following:
      If seat[number] is UNOCCUPIED
         Display number
```

ALGORITHM TO RESERVE A SEAT

```
1. Read number of seat to be reserved.
2. If seat[number] is UNOCCUPIED
      Set seat[number] to OCCUPIED.
   Else
      Display a message that the seat having this number has already been assigned.
```

→ Quite often there's a tradeoff:
simplicity of data organization ↔ simplicity/elegance of algorithms

Example 2: Searching an online phone directory
Linear search?
OK for Calvin College, but too slow for Grand Rapids or New York

→ Amount of data is an important factor. May have to restructure the data set for efficient processing — e.g., keep it ordered and use binary search or an indexed sequential search

Example3: Compiler lookup of an identifier's memory address, type, . . . in a symbol table
Linear search? No, too slow
Binary search? No, too much work to keep sorted
Use hash tables

→ Number of accesses & speed required is an important factor.

Example 4: Text processing
Store in an array / vector?
OK for text analysis — word counts, average word length, etc.
Not for word-processing — Too inefficient if many insertions & deletions

→ Static vs. dynamic nature of the data is an important factor

Definitions

1. An **abstract data type** (**ADT**) is:
   a collection of related data items
   together with
   basic relations between them and operations to be performed on them.

   Why "abstract?" Data, operations, and relations are studied independently of how they are going to be implemented.

   What not *how*

   Example:
   Data items: seats for TFA
   Basic operations: find unoccupied sets, reserve a set, cancel a seat assignment.

2. An **implementation** of an ADT consists of
   storage structures (commonly called **data structures**) to store the data items
   and
   algorithms for the basic operations and relations.

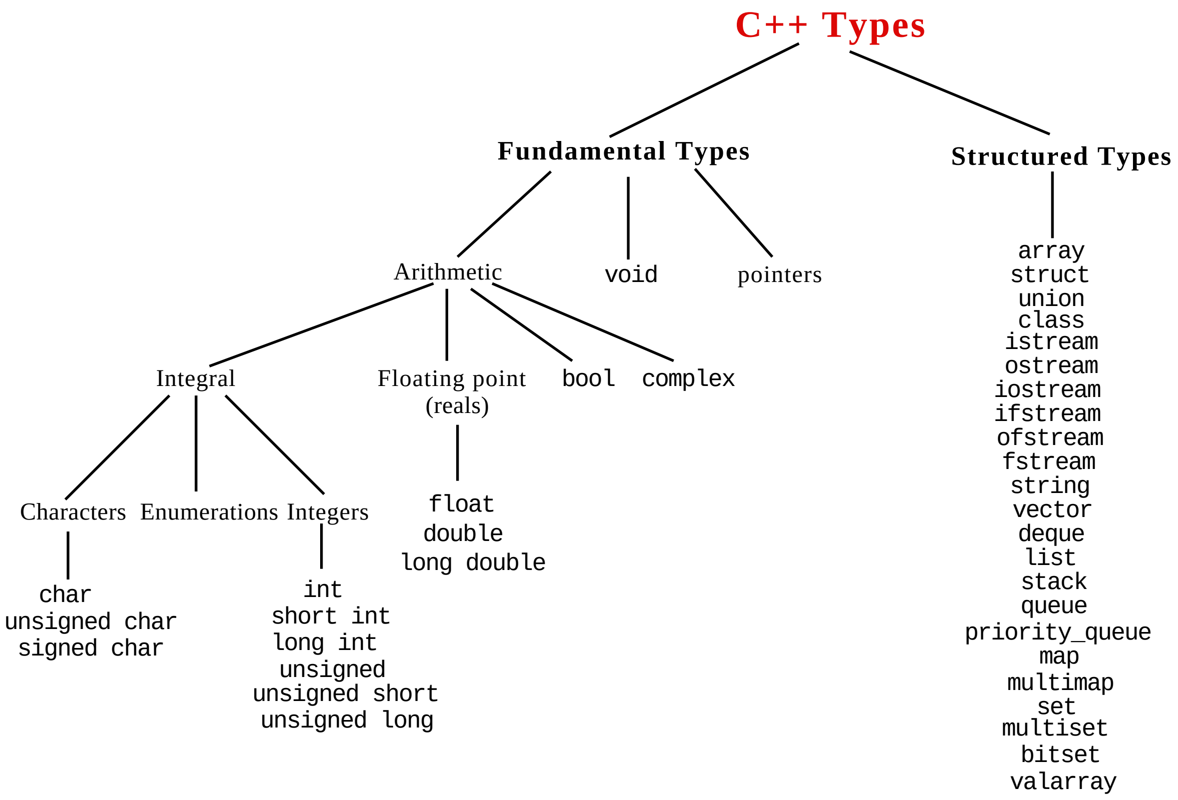
   Examples: Attempts 1 and 2 for TFA

3. **Data abstraction**: Separating the definition of a data type from its implementation.

   An important concept in software design.

Usually the storage structures / data structures used in implementation are those provided in a language or built from them. So we look first at those provided in C++. We begin by reviewing the simple types — `int`, `double`, etc. — and then the structured ones.

Diagram from p. 33



B. Simple Data Types (§2.2)

Memory:

2-state devices ↔ **bits 0 and 1**
Organized into **bytes** (8 bits) and **words** (machine dependent — e.g., 4 bytes).
Each byte (or word) has an **address** making it possible to store and retrieve contents of any given memory location.

Therefore:

- The most basic form of data: **sequences of bits**
- We can view *simple data types* (values are atomic — can't be subdivided) as ADTs.
- Implementations have:
  Storage structures: memory words
  Algorithms: system hardware/software to do basic operations.

1. Boolean data

Data values: {`false`, `true`}

In C/C++: `false` = 0, `true` = 1 (or nonzero)

Could store 1 value per bit, but usually use a byte (or word)

Operations: and: && (See bit tables on p. 34)
or: ||
not: !

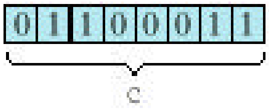
&&	0	1
0	0	0
1	0	1

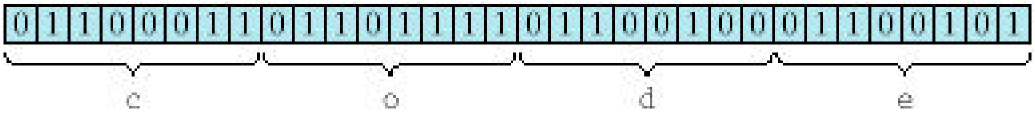
||	0	1
0	0	1
1	1	1

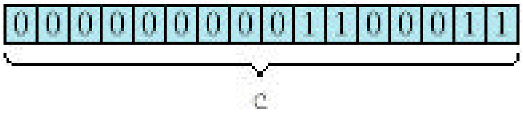
x	!*x*
0	1
1	0

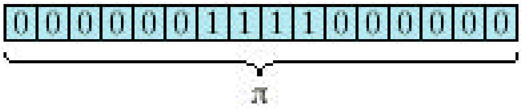
2. Character Data

Store numeric codes (ASCII, EBCDIC, Unicode) in 1 byte for ASCII and EBCDIC, 2 bytes for Unicode (see examples on p. 35).









Basic operation: comparison to determine if =, <, >, etc. — use their numeric codes

3. Integer Data

Nonegative (unsigned) integer: type `unsigned` (and variations) in C++

Store its base-two representation in a fixed number w of bits (e.g., w = 16 or w = 32)

$$88 = 0000000001011000_2$$

0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0

Signed integer: type `int` (and variations) in C++

Store in a fixed number w of bits using one of the following:

a. **Sign-magnitude representation**

Save one bit for sign (0 = +, 1 = –) and use base-two representation in the other bits.

88 → 0000000001011000 –88 → 1000000001011000
↑ ↑
sign bit sign bit

→ Not good for arithmetic computations

b. **Two's complement representation**

For n ≥ 0 : Use ordinary base-two representation with leading (sign) bit 0

For –n:
(1) Find w-bit base-2 representation of n
(2) Complement each bit.
(3) Add 1
(From right, change all 1's up to first 0; change this 0 to a 1.)

Example: –88

1. 88 as a 16-bit base-two number 0000000001011000
2. Complement this bit string 1111111110100111
3. Add 1 1111111110101000

1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0

→ Good for arithmetic computations (see p. 38)

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

These work for both + and – integers

5 + 7:

```
                111←¾¾ carry bits
   0000000000000101
+  0000000000000111
   0000000000001100
```

5 + –6:

```
   0000000000000101
+  1111111111111010
   1111111111111111
```

c. **Biased representation**

Add a constant *bias* to the number (typically, 2^{w-1});
then find its base-two representation.

Examples:

88 using w = 16 bits and bias of 2^{15} = 32768

1. Add the bias to 88, giving 32856
2. Represent the result in base-two notation: 1000000001011000

Note: For n ≥ 0, just change leftmost bit of binary representation of n to 1

–88:

1. Add the bias to -88, giving 32680
2. Represent the result in base-two notation: 0111111110101000

→ Good for comparisons; so, it is commonly used for exponents in floating-point representation of reals.

d. Problems:

→ **Overflow**: Too many bits to store.

→ Not a perfect representation of (mathematical) integers; can only store a finite (sub)range of them.

4. Real Data

Types `float` and `double` (and variations) in C++

IEEE Floating-Point Format

Single precision:

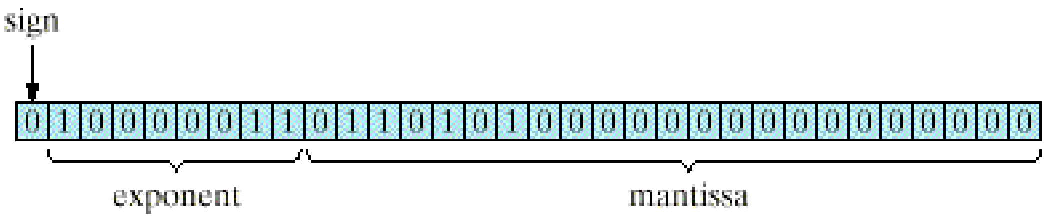
1. Write binary representation in floating-point form:

$$\underbrace{b_1.b_2b_3\ldots}_{\text{mantissa or fractional part}} \times \underset{\substack{\uparrow\\ \text{exponent}}}{2^k}$$ with each b_i a bit and $b_1 = 1$ (unless number is 0)

2. Store:
   - — sign of mantissa in leftmost bit (0 = +, 1 = –)
   - — biased binary rep. of exponent in next 8 bits (bias = 127)
   - — bits $b_2b_3\ldots$ in rightmost 23 bits. (Need not store b_1 — know it's 1)

Example: $22.625 = 10110.101_2$ (see p.41)

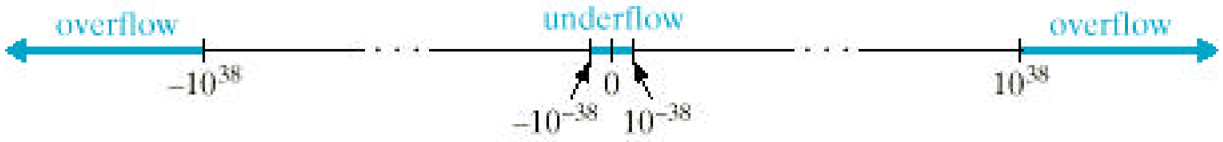
Floating point form: $1.0110101_2 \times 2^4$



Problems:

→ Exponent overflow/underflow (p. 41)

Only a finite range of reals can be stored exactly.



→ Roundoff error (pp. 41-42)

— Only a finite subset of this range of reals can be stored exactly.
(Most reals do *not* have terminating binary representations.)

Example: $0.7 = (0.101100110011001100110011001100110\ldots)_2$

— Roundoff error may be compounded in a sequence of operations.
(Some of the usual laws of arithmetic do not hold — associative, distributive)

— Be careful in comparing reals with `==` and `!=`.

Assignment #2

→ Be able to answer the questions in Quick Quiz 2.2.

→ Write out the following to hand in next Wednesday, Feb. 10:

Exercises 2.2	1
	10, 12 (Exers 2, 4 in sign-magnitude)
	16, 18 (Exers 2, 4 in two's complement)
	22, 24 (Exers 2, 4 in biased notation)
	27, 32, 37, 38, 40, 43

We've been looking at simple types. Now we look at *structured data types* (also called *data structures*) that store collections of data. We will first review/ introduce arrays and structs from a "traditional" point of view (i.e., as used in C and many other languages). Classes will be considered in detail very soon. A large part of this course will focus on how these (and other) data types are used to construct other useful data types.

C. C-Style One-Dimensional Arrays (§2.3)

1. Def of an array as an ADT:

A fixed-size sequence (ordered set) of elements, all of the same type, where the basic operation is **direct access to each element in the array so values can be retrieved from or stored in this element**.

Properties:

- Fixed number of elements
- Must be ordered so there is a first element, a second one, etc.
- Elements must be the same type (and size);
  use arrays only for homogeneous data sets.
- Direct access: Access an element, just by giving its location — the time to access each element is the same for all elements, regardless of position.

  [In contrast to sequential access: To access an element, must first access all those that precede it.]

2. Declaring arrays in C++

```
element_type array_name[CAPACITY];
```

where

`element_type` is any type,

`array_name` is the name of the array — any valid identifier.

`CAPACITY` (a positive integer constant) is the number of elements in the array

→ The compiler reserves a block of consecutive memory locations, enough to hold *`CAPACITY`* values of type *`element_type`*. (These are consecutive memory locations, except possibly if *`CAPACITY`* or the size of *`element_type`* objects is very large).

The elements (or positions) of the array, are indexed **`0`**, **`1`**, **`2`**, . . ., ***`CAPACITY`*** **`- 1`**.

Example:

```
double score[100];
```

or better, use a named constant to specify the array capacity:

```
const int CAPACITY = 100;
double score[CAPACITY];
```

Note: Can use **`typedef`** with array declarations; for example,

```
const int CAPACITY = 100;
typedef double ScoresArray[CAPACITY];
ScoresArray score;
```

→ How well does this implement the general definition of an array:

As an ADT		In C++
ordered	←¾¾¾¾¾→	indices are numbered 0, 1, 2, . . ., *`CAPACITY`* `- 1`
fixed size	←¾¾¾¾¾→	*`CAPACITY`* specifies the capacity of the array
same type elements	←¾→	*`element_type`* is the type of elements
direct access	←¾¾¾¾→	Subscript operator `[]`

3. Subscript operator

The **subscript operator []** is an actual operator and not simply a notation/punctuation as in some other languages. Its two operands are an **array variable** and an **integer index** (or subscript) and is written

array_name[*i*]

Here *i* is an integer expression with $0 \le i \le \mathit{CAPACITY} - 1$. This subscript operator returns **a reference to (alias of) the element in location *i* in *array_name***; so it is a **variable**, called an **indexed** (or **subscripted**) **variable**, whose type is the specified *element_type* of the array.

This means that this array reference can be used on the left side of an assignment, in input statements, etc. to store a value in a specified location in the array.

Examples:

```
// Zero out all the elements of score

for (int i = 0; i < CAPACITY; i++)
  score[i] = 0.0;

// Read values into the first numScores elements of score

for (int i = 0; i < numScores; i++)
  cin >> score[i];

// Display the values stored in the first numScores
// elements of score

for (int i = 0; i < numScores; i++)
  cout << score[i] << endl;
```

4. Array Initialization

In C++, arrays can be initialized when they are declared.

a. Numeric arrays:

element_type num_array[*CAPACITY*] = {*list_of_initial_values*};

Example:

```
double rate[5] = {0.11, 0.13, 0.16, 0.18, 0.21};
```

declares `rate` to be an array of 5 real values and intializes `rate` as follows:

	0	1	2	3	4
rate	0.11	0.13	0.16	0.18	0.21

Note 1: If fewer values are supplied than the declared capacity of the array, the remaining elements are assigned 0.

```
double rate[5] = {0.11, 0.13, 0.16};
```

intializes `rate` as follows:

	0	1	2	3	4
rate	0.11	0.13	0.16	0	0

Note 2: It is an error if more values are supplied than the declared size of the array, How this error is handled, however, will vary from one compiler to another. In Gnu C++ ???

Note 3: If no values are supplied, array elements are undefined (i.e., garbage values).

b. Character arrays:

They may be initialized in the same manner as numeric arrays.

Example:

```
char vowel[5] = {'A', 'E', 'I', 'O', 'U'};
```

declares `vowel` to be an array of 5 characters and initializes it as follows:

	0	1	2	3	4
vowel	A	E	I	O	U

Note 1: If fewer values are supplied than the declared size of the array, the zeros used to fill unitialized elements are interpreted as the null character `'\0'` whose ASCII code is 0.

Example:

```
const int NAME_LENGTH = 10;

char collegeName[NAME_LENGTH] = {'C', 'a', 'l', 'v', 'i', 'n'};
```

initializes `collegeName` as follows:

	0	1	2	3	4	5	6	7	8	9
collegeName	C	a	l	v	i	n	\0	\0	\0	\0

Note 2: Character arrays may be initialized using **string constants**. For example, the following declaration is equivalent to the preceding:

```
char collegeName[NAME_LENGTH] = "Calvin";
```

Note 3:The null character **'\0'** (ASCII code is 0) is used as an **end-of-string mark.**

Thus, character arrays used to store strings should be declared large enough to store the null character. If it is not, one cannot expect some of the string functions and operations to work correctly. If a character array is initialized with a , string constant, the end-of-string mark is added automatically, provided there is room for it.

Example:

```
char collegeName[7] = {'C', 'a', 'l', 'v', 'i', 'n', '\0'};
char collegeName[7] = "Calvin";
```

c. Initializations with no array size specified

The array capacity may be omitted in an array declaration with an initializer list. In this case, the number of elements in the array will be **the number of values in the initializer list**.

Example:

```
double rate[] = {0.11, 0.13, 0.16};
```

Note: This explains the brackets in constant declarations such as

```
const char IN_FILE[] = "employee.dat";
```

5. Addresses

When an array is declared, the address of the first byte (or word) in the block of memory associated with the array is called the **base address** of the array. Each array reference is then translated into an **offset** from this base address.

For example, suppose each element of array `score` will be stored in 8 bytes and the base address of `score` is `0x1396`. A statement such as

```
cout << score[3] << endl;
```

requires that the array reference `score[3]` first be translated into a memory address:

```
score[3] →   0x1396 + 3 * sizeof(double)
           = 0x1396 + 3 * 8
           = 0x13ae
```

The contents of the memory word with this address 0x13ae can then be retrieved and displayed. An **address translation** like this is carried out each time an array element is accessed.

For an array variable *`array_name`*, its value is actually the **base address of *`array_name`*** and ***`array_name`* + *`index`*** is the address of *`array_name`*`[`*`index`*`]`. An array reference

```
array_name[index]
```

is equivalent to

```
*(array_name + index)
```

Here, `*` is the **dereferencing** operator;

`*`*`ref`* returns **the contents of the memory location with address *`ref`***.

z

For example, the statement

```
cout << score[3] << endl;
```

could also be written

```
cout << *(score + 3) << endl;
```

Note: No bounds checking of indices is done! (See pp. 50-51)

D. C-Style Multidimensional Arrays

1. Introduction

Example: Suppose we wish to store and process a table of test scores for several different students on several different tests:

	Test 1	Test 2	Test 3	Test 4
Student 1	99.0	93.5	89.0	91.0
Student 2	66.0	68.0	84.5	82.0
Student 3	88.5	78.5	70.0	65.0
:	:	:	:	:
:	:	:	:	:
Student-n	100.0	99.5	100.0	99.0

Use a two-dimensional array**.**

2. Declaring two-dimensional arrays

a. Usual form of declaration:

```
element_type array_name[NUM_ROWS][NUM_COLUMNS];
```

Example:

```
const int NUM_ROWS = 30,
          NUM_COLUMNS = 5;

double scoresTable[NUM_ROWS][NUM_COLUMNS];
```

or using a `typedef`:

```
const int NUM_ROWS = 30,
          NUM_COLUMNS = 5;

typedef double TwoDimArray[NUM_ROWS][NUM_COLUMNS];

TwoDimArray scoresTable;
```

b. Initializing

List the initial values in braces, row by row; may use internal braces for each row to improve readability.

Example:

```
double
  rates[2][3] = {{0.50, 0.55, 0.53},    // first row
                 {0.63, 0.58, 0.55} };  // second row
```

3. Processing two-dimensional arrays

Use doubly-indexed variables**:**

Example: `scoresTable[2][3]` is the entry in row 2 (numbered from 0) and column 3 (numbered from 0)

↑ ↑

row index column index

Typically use nested loops to vary the two indices, most often in a rowwise manner.

Example:

```
int numStudents, numTests,
    i, j;                   // indices;

cout >> "# students and # of tests? ";
cin >> numStudents >> numTests;

cout << "Enter " << numTests << " test scores for student\n";
for (i = 0; i < numStudents; i++)
{
  cout << '#' << i + 1 << ':';
  for (j = 0; j < numTests; j++)
    cin >> scoresTable[i][j];
}
```

4. Higher-Dimensional Arrays

The methods for two-dimensional arrays extend in the obvious way.

a. Example: To store and process a table of test scores for several different students on several different tests for several different semesters:

```
const int RANKS = 10, ROWS = 30, COLUMNS = 5;

typedef double ThreeDimArray[RANKS}[ROWS][COLUMNS;

ThreeDimArray gradeBook;
```

`gradeBook[4][2][3]` is the score on page 4 (numbered from 0)
for student 2 (numbered from 0)
on test 3 (numbered from 0)

b. Still higher dimensions
Example like the automobile-inventory example on pp. 54-5

```
enum BrandType {Levi, Wrangler, CalvinKlein, Lee, BigYank, NUM_BRANDS};
enum StyleType {baggy, tapered, straightleg, designer, NUM_STYLES};
enum WaistType {w28, w29, w30, w31, w32, w33, w34, w35, w36,
                w37, w38, w39, w40, w41, w42, w43, w44, w45,
                w46, w47, w48, NUM_WAIST_SIZES};
enum InseamType {i26, i27, i28, i29, i30, i31, i32, i33, i34, i34, i36,
                 NUM_INSEAM_SIZES};

typdef int
   JeansArray[NUM_BRANDS][NUM_STYLES][NUM_WAIST_SIZES][NUM_INSEAM_SIZES];

   JeansArray jeansInStock;
```

The value of

```
jeansInStock[Levi][Designer][w32][i31]
```

is the number of Levi's designer 32 × 31 jeans that are in stock. The statement

```
jeansInStock[brand][style][waist][inseam]--;
```

might be used to record the sale (i.e., decrement the inventory) of one pair of jeans of brand `brand`, style `style`, waist size `waist`, and inseam length `inseam`.

5. Arrays of Arrays

Consider again the declaration

```
double scoresTable[30][4];
```

This is really a declaration of a one-dimensional array having 30 elements, each of which is a one-dimensional array of 4 real numbers; that is, `scoresTable` is a one-dimensional array of rows, each of which has 4 real values. This declaration is thus equivalent to a declaration like

```
typedef double RowOfTable[4];

RowOfTable scoresTable[30];
```

or, since `typedef` is used once, why not use it twice:

```
typedef double RowOfTable[4];
typedef RowOfTable TwoDimArray[30];

TwoDimArray scoresTable;
```

With any of the declarations, we can always view a two-dimensional array like `scoresTable` as an array of rows of a table. In fact,

`scoresTable[i]` is **the `i`-th row of the table**.

Then, `scoresTable[i][j]` should be thought of as `(scoresTable[i])[j]`, that is, as finding the `j`-th element of `scoresTable[i]`.

Address Translation:
This array-of-arrays structure of multidimensional arrays also explains how address translation is carried out. Suppose the base address of `scoresTable` is 0x12345:

```
scoresTable[10][3]
        →  0x12345 + 10 * sizeof(RowOfTable) + 3 * sizeof(double)
         = 0x12345 + 10 * 4 * sizeof(double)  + 3 * sizeof(double)
         = 0x12345 + (10 * 4 + 3) * 8
         = 0x1249d
```

What about higher-dimensional arrays?

An n-dimensional array should be viewed (recursively) as a one-dimensional array whose elements are (n - 1)-dimensional arrays.

6. Arrays as Parameters

Passing an array to a function actually passes the base address of the array. Thus the parameter has the same address as the argument, so modifying the parameter will modify the corresponding array argument.

This also means that the array capacity is not available to the function unless passed as a separate parameter.

Example: In `void Print(theArray[100], int theSize);`

can just as well use:

```
void Print(theArray[], int theSize);
```

Now, what about multidimensional arrays?

```
void Print(double table[][], int rows, int cols)
```

doesn't work. Best to use a typedef to declare a global type identifier and use it to declare the types of the parameters.:

For example,

```
typedef double TwoDimArray[30][4];

TwoDimArray scoresTable;

void Print(TwoDimArray table, int rows, int cols)
  . . .
```

Assignment #2: Due: Friday., Feb. 19

Be able to answer questions in Quick Quiz 2.3

P. 61: 1, 3, 5, 6, 8, 10, 11, 13, 15, 17, 19

Problems with C-Style Arrays

a. *Capacity cannot change*.

b. *Vrtually no predefined operations or functions on non-*`char` *arrays.*

→ Must pass size (and/or capacity) to array-processing functions.

c. Deeper Problem: In OOP, *objects should be self-contained* .

→ C-style arrays aren't.

Solution (OOP): Encapsulate array, capacity, size, and operations in a class.

→ `vector`

E. Intro. to Structs

1. When is a structure needed?

Up to now, our approach to designing a program (and software in general) has been:

1. Identify the **objects** in the problem.
   1a. . . .
2. Identify the **operations** in the problem.
   1a. If the operation is not predefined, write a **function** to perform it.
   1b. If the function is useful for other problems, store it in a **library**.
3. Organize the objects and operations into an algorithm.
4. Code the algorithm as a program.
5. Test, execute, and debug the program.
6. Maintain the program

Since predefined types may not be adequate, we add:

1a. If the predefined types are not adequate to model the object,
create a new data type to model it (e.g., enumerations).

Now, suppose the object being modeled has **multiple attributes**.

Examples :

A temperature has:
— a *degrees* attribute
— a *scale* attribute (Fahrenheit, Celsius, Kelvin)

32	F
degrees	*scale*

A date has:
— a *month* attribute
— a *day* attribute
— a *year* attribute

September	23	1998
month	*day*	*year*

C++ provides **structs** and **classes** to create new types with multiple attributes. So we might add to our design methodology:

1. Identify the objects in the problem.
   1a. If the predefined types are not adequate to model the object,
   create a new type to model it.
   1b. If the object has multiple attributes, **create a struct or class to represent objects of that type**.

2. As an ADT, a **structure** (usually abbreviated to **struct** and sometimes called a **record**) is like an array in that it is has a *fixed size*, it is *ordered*, and the basic operation is *direct access* to its members so that items can be stored in / retrieved from them; but it differs from an array in that its elements may be of **different types**.

3. Declaration (C-style):

```
struct TypeName
{
  declarations of members  //of any types
};
```

4. Examples:

a. Temperature:

32	F
degrees	*scale*

```
struct Temperature
{
  double degree; // number of degrees
  char scale;   // temp. scale (F, C, K, ...)
};

Temperature temp;
```

b. Date:

September	23	1998
month	*day*	*year*

```
struct Date
{
  string month;   // name of month
  int day,        // day number
      year;       // year number
};

Date birthDay,
     currentDate;
```

c. Phone Listing:

John Q. Doe	12345 Calvin Rd.	Grand Rapids, MI	9571234
name	*street*	*city & state*	*phone #*

```
struct DirectoryListing
{
  string name,             // name of person
         street,           // street address
         cityAndState;     // city, state (no zip)

  unsigned phoneNumber;  // 7-digit phone number
};

DirectoryListing entry,     // entry in phone book
                 group[20]; // array of directory listings
```

d. Coordinates of a point: (Members need not have different types.)

3.73	–2.51
x coord.	*y coord.*

```
struct Point
{
  double xCoord,
         yCoord;
};

Point p, q;
```

e. Test scores: (Members may be structured types — e.g., arrays.)

012345	83	79	92	85
id-number	*list of scores*			

```
struct TestRecord
{
  unsigned idNumber,
           score[4];
};

TestRecord studentRecord, gradeBook[30];
```

Heirarchical (or nested) structs

Since the type of a member may be any type, it may be another struct. For example,

`John Q. Doe`	`12345 Calvin Rd.`	`Grand Rapids, MI`	`9571234`	`June`	`17`	`1975`	`3.95`	`92.5`
name	*street*	*city & state*	*phone #*	*month*	*day*	*year*	*gpa*	*credits*
___ DirectoryListing ___/				___ Date ___/			real	real

```
struct PersonalInfo
{
   DirectoryListing ident;
   Date birth;
   double cumGPA,
          credits;
 };

PersonalInfo student;
```

6. The scope of a member identifier is the struct in which it is defined.

   Consequences:
   — A member identifier may be used outside the struct for some other purpose.
   — A member cannot be accessed outside the struct just by giving its name.

7. Direct access to members of a struct (or class) is implemented using **member access operators**: one of these is the **dot operator (.)**

   struct_var.member_name

   Examples:

   Input a value into the `month` member of `birthday`:
   ```
   cin >> birthDay.month;
   ```

   Calculate y coordinate of a point on y = 1/x:
   ```
   if (p.xCoord != 0.0)
     p.yCoord = 1.0 / p.xCoord;
   ```

   Sum the scores in `studentRecord`:
   ```
   double sum = 0;
   for (int i = 0; i < 4; i++)
     sum += studentRecord.score[i];
   ```

   Output the name stored in student:
   ```
   cout << student.ident.name << endl;
   ```

F. A Quick Look at Unions (p. 68)

1. A union has a definition like that of a struct, with "struct" replaced by "union":

```
union TypeName  ←¾¾¾¾¾¾    TypeName is optional
{
  declarations of members  //of any types
};
```

2. A union differs from a struct in that the members **share memory**. Memory is (typically) allocated for the largest member, and all the other members share this memory.

Example:

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

struct Struct
{
  int i;
  double d;
  bool b;
};

union Union
{
  int i;
  double d;
  bool b;
};

int main()
{
  Struct s;
  Union u;
  s.i = 123456789;
  u.i = 123456789;
  cout << "Structure: " << s.i << " and " << s.d << "  "
       << (s.b ? "true" : "false") << endl;
  cout << "Union:     " << u.i << " and " << u.d << "  "
       << (u.b ? "true" : "false") << endl;

  s.d = 0.123;
  u.d = 0.123;
  cout << "Structure: " << s.i << " and " << s.d << "  "
       << (s.b ? "true" : "false") << endl;
  cout << "Union:     " << u.i << " and " << u.d << "  "
       << (u.b ? "true" : "false") << endl;

  s.b = true;
  u.b = true;
  cout << "Structure: " << s.i << " and " << s.d << "  "
       << (s.b ? "true" : "false") << endl;
  cout << "Union:     " << u.i << " and " << u.d << "  "
       << (u.b ? "true" : "false") << endl;
}
```

Execution:

```
Structure: 123456789 and 6.95336e-310  false
Union:     123456789 and 3.21193e-273  true
Structure: 123456789 and 0.123  false
Union:     1069513965 and 0.123  true
Structure: 123456789 and 0.123  true
Union:     97517 and 2.06932e-309  true
```

Note: If data is stored in a union using one member and accessed using another member of a different type, the results are implementation dependent.

3. Example: Suppose a file contains:

```
|John Doe 40 M            |<———  name, age, marital status (married)
|January 30 1980          |<——— wedding date
|Mary Smith Doe  8        |<——— spouse, # dependents
|Fred Jones  17  S        |<——— name, age, marital status (single)
|T                        |<——— available
|Jane VanderVan  24 D     |<——— name, age, marital status (divorced)
|February 21 1998   N     |<——— divorce date,  remarried (No)]
|Peter VanderVan 25 W     |<——— name, age, marital status (widower)
|February 22 1998   Y     |<——— date became a widower,  remarried (Yes)
|    :                    |
|    :                    |
```

Since there are three types of records, we would need three types of structs:

```
struct MarriedPerson
{
   string name;
   short age;
   char marStatus;    // S = single, M = married, W = was married
   Date wedding;      // date s/he was married
   string spouse;     // name of spouse
   short dependents;  // number of dependents
};
struct SinglePerson
{
   string name;
   short age;
   char marStatus;
   bool available;    // true if person is available, else false
};
struct WasMarriedPerson
{
   string name;
   short age;
   char marStatus;
   Date divorceOrDeath;// date s/he was divorced/widow(er)ed
   char remarried;     // Y or N
};
```

4. Structs like these with some common members — **fixed part** — but other fields that are different can be combined into a single structure by using a **union** — to add a **variant part**.

```
struct Date
{
  string month;
  short day, year;
};

struct MarriedInfo
{
  Date wedding;
  string spouse
  short dependents;
};

struct SingleInfo
{
  bool available;
};

struct WasMarriedInfo
{
  Date divorceOrDeath;
  char remarried;
};

struct PersonalInfo
{
  string name;
  short age;
  char marStatus; // Tag: S = single, M = married, W = was married
  union
  {
    MarriedInfo married;
    SingleInfo single;
    WasMarriedInfo wasMarried;
  };
};

PersonalInfo person;
```

Typically process such a structure using a `switch` for the variant part: e.g.,

```
cin >> person.name >> person.age >> person.marStatus;
switch(Person.marStatus)
{
   case 'M':  cin >> person.married.wedding.month
                  >> person.married.wedding.day
                  >> person.married.wedding.year
                  >> person.married.spouse
                  >> person.married.dependents;
              break;
   case 'S':  cin >> available;
              break;
   case 'W':  cout << "Enter . . . ";
              cin >> person.wasMarried.divorceOrDeath.month
                  >> person.wasMarried.divorceOrDeath.day
                  >> person.wasMarried.divorceOrDeath.year
                  >> person.wasMarried.remarried;
}
```

5. Address translation for structs and unions: (p. 70)

```
enum YearInSchool {fresh, soph, jun, sen, spec};
struct StudentRecord
{
  int number;
  char name[21];
  double score[3];
  YearInSchool year;
}

//PersonalInfo as before
StudentRecord s;
PersonalInfo p;
```

<u>Addresses:</u>

```
s = 0x33a18
p = 0x339d8
Struct S:
0x33a18 number
0x33a1c name
0x33a38 score
0x33a50 year
Struct P:
0x339d8 name
0x339ee age
0x339f0 marStatus
0x339f2 married.wedding.month
0x339fc married.wedding.day
0x339fe married.wedding.year
0x33a00 married.spouse
0x33a16 married.dependents
0x339f2 wasMarried.divorceOrDeath.month
0x339fc wasMarried.divorceOrDeath.Day
0x339fe wasMarried.divorceOrDeath.year
0x33a00 wasMarried.remarried
```

If a struct `s` has fields $f_1, \ldots, f_n$, requiring $w_1, \ldots, w_n$ cells of storage, respectively:

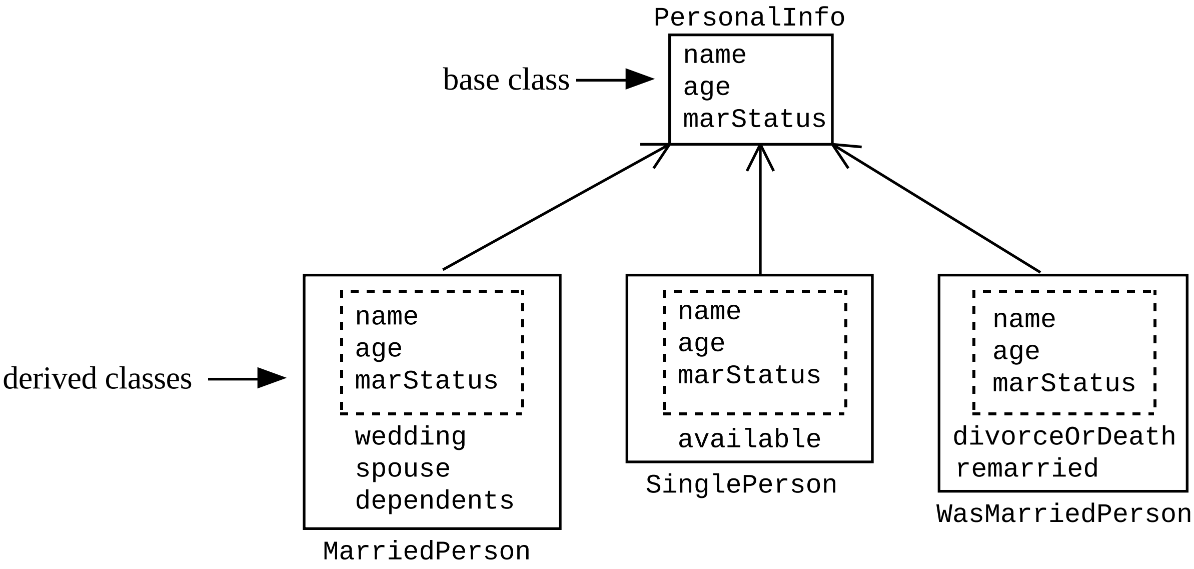
$$\begin{aligned}\text{Address of } s.f_k &= \text{base address of } s + \text{offset} \\ &= \text{base address of } s + \sum_{i=1}^{k-1} w_i\end{aligned}$$

For structs like `p` that contain unions: Allocate space for the *largest variant*, and then *overlay* variants in this space.

6. These kinds of variant structures aren't used much anymore. (p. 69)

Instead, in OOP languages:

- Encapsulate the common information in a base class
- Use inheritance to build derived classes for the variants
  (Derived classes inherit all of the non-private members of the base class.)



G. A commercial for OOP

Two programming paradigms:

Procedural: commonly used with*procedural* languages such as C, FORTRAN, and Pascal
Action-oriented — concentrates on the *verbs* of a problem's specification

Programmers:

- Identify basic tasks to be performed to solve problem
- Implement the actions required to do these tasks as subprograms (procedures/functions/subroutines)
- Group these subprograms into programs/modules/libraries, which together make up a complete system for solving the problem

O*bject-oriented:* Uses in *OOP* languages like C++, Java, and Smalltalk
Focuses on the *nouns* of a problem's specification

Programmer:

- Determine what objects are needed for a problem and how they should work together to solve the problem.
- Create types called *classes* made up of*data members* and *function members* to operate on the data. Instances of a type (class) are called *objects*.

An Example — Creating a Data Type in a procedural (C-type) language (pp. 74-78)

Problem: Create a type `Time` for processing times in standard hh:mm AM/PM form and in military-time form.

Data Members:

Hours (0, 1, ..., 12)
Minutes (0, 1, 2, ..., 59)
AM or PM indicator ('A' or 'P')
MilTime (military time equivalent)

Some Operations :

1. Set the time
2. Display the time
3. Advance the time
4. Determine if one time is less than another time.

Implementation:

1. Need **storage** for the data members — use a **struct**
2. Need **functions** for the operations.
3. "Package" declarations of these together in a header file.

```
/** Time.h ----------------------------------------------------------
   This header file defines the data type Time for processing time.
   Basic operations are:
     Set:      To set the time
     Display:  To display the time
     Advance:  To advance the time by a certain amount
     LessThan: To determine if one time is less than another
--------------------------------------------------------------------*/

#include <iostream>
using namespace std;

struct Time
{
  unsigned hour,
           minute;
  char AMorPM;        // 'A' or 'P'
  unsigned milTime;   // military time equivalent
};
```

```
/* Set sets the time to a specified values.
 *
 *  Receive:   Time object t
 *             hours, the number of hours in standard time
 *             minutes, the number of minutes in standard time
 *             AMPM ('A' if AM, 'P' if PM
 *  Pass back: The modified Time t with data members set to
 *               the specified values
 ******************************************************************/

void Set(Time & t, unsigned hours, unsigned minutes, char AMPM);


/* Display displays time t in standard and military format using
 * output stream out.

 *  Receive:   Time t and ostream out
 *  Output:    The time T to out
 *  Pass back: The modified ostream out
 ******************************************************************/

void Display(const Time & t, ostream & out);


/* Advance increments a time by a specified value.
 *
 *  Receive:   Time object t
 *             hours, the number of hours to add
 *             minutes, the number of minutes to add
 *  Pass back: The modified Time t with data members incremented
 *               by the specified values
 ******************************************************************/

void Advance(Time & t, unsigned hours, unsigned minutes);


/* Determine if one time is less than another time.
 *
 *  Receive:  Times t1 and t2
 *  Return:   True if t1 < t2, false otherwise.
 ******************************************************************/

bool LessThan(const Time & t1, const Time & t2);
```

```
//========= Time.cpp -- implements the functions in Time.h =========

#include "Time.h"

/*** Utility functions -- might be added as basic operations later ***/

int ToMilitary(unsigned hours, unsigned minutes, char AMPM);

void ToStandard(unsigned military,
                unsigned & hours, unsigned & minutes, char& AMPM);

void Set(Time & t, unsigned hours, unsigned minutes, char AMPM)
{
  if (hours >= 1 && hours <= 12 &&
      minutes >= 0 && minutes <= 59 &&
      (am_pm == 'A' || am_pm == 'P'))
  {
    t.hour = hours;
    t.minute = minutes;
    t.AMorPM = AMPM;
    t.milTime = ToMilitary(hours, minutes, AMPM);
  }
  else
    cout << "*** Can't set time with these values ***\n";
    // t remains unchanged
}

void Display(const Time & t, ostream & out)
{
  out << t.hour << ':'
      << (t.minute < 10 ? "0" : "") << t.minute
      << ' ' << t.AMorPM << ".M.  ("
      << t.milTime << " mil. time)";
}

void Advance(Time & t, unsigned hours, unsigned minutes)
{
  // Advance using military time
   t.milTime += 100 * hours + minutes;
   unsigned milHours = t.milTime / 100,
            milMins = t.milTime % 100;

  // Adjust to proper format
   milHours +=  milMins / 60;
   milMins %= 60;
   milHours %= 24;
   t.milTime = 100 * milHours + milMins;


  // Now set standard time
   ToStandard(t.milTime, t.hour, t.minute, t.AMorPM);
}

bool LessThan(const Time & t1, const Time & t2)
{
  return (t1.milTime < t2.milTime);
}
```

```
/***** UTILITY FUNCTIONS *****/

/* ToMilitary converts standard time to military time.
 *
 *  Receive: hours, minutes, AMPM
 *  Return:  The military time equivalent
 ******************************************************************/

int ToMilitary (unsigned hours, unsigned minutes, char AMPM)
{
  if (hours == 12)
    hours = 0;

  return hours * 100 + minutes + (AMPM == 'P' ? 1200 : 0);
}


/* ToStandard converts military time to standard time.
 *
 *  Receive: military, a time in military format
 *  Return:  hours, minutes, AMPM -- equivalent standard time
 *******************************************************************/

void ToStandard(unsigned military,
                unsigned & hours, unsigned & minutes, char & AMPM)
{
  hours = (military / 100) % 12;
  if (hours == 0)
     hours = 12;
  minutes = military % 100;
  AMPM = (military / 100) < 12 ? 'A' : 'P';
}
```

```
//========= Test driver =========

#include <iostream>
#include "Time.h"
uses namespace std;

int main()
{
  Time mealTime,
       goToWorkTime;

  Set(mealTime, 5, 30, 'P');

  cout << "We'll be eating at ";
  Display(mealTime, cout);
  cout << endl;

  Set(goToWorkTime, 5, 30, 'P');    // Try other values also: 'A' -> 'P'
  cout << "You leave for work at ";
  Display(goToWorkTime, cout);
  cout << endl;
  if (LessThan(MealTime. goToWorkTime))
    cout << "If you hurry, you can eat first.\n";
  else
    cout << "Sorry you can't eat first.\n";

  Advance(goToWorkTime, 0, 30);    // Try other values also: 0 -> 12)
  cout << "You go into work later at ";
  Display(goToWorkTime, cout);
  cout << endl;
  if (LessThan(MealTime. goToWorkTime))
    cout << "If you hurry, you can eat first.\n";
  else
    cout << "Sorry you can't eat with us.\n";
  cout << endl;

}
```

Execution:

```
We'll be eating at 5:30 P.M.  (1730 mil. time)
You leave for work at 5:30 P.M.  (1730 mil. time)
Sorry you can't eat first.
You go into work later at 6:00 P.M.  (1800 mil. time)
If you hurry, you can eat first.
```

7. Problems with C-Style Arrays

a. *The capacity of a C-style array cannot change*.

Solution 1 (non-OOP): Use run-time arrays.

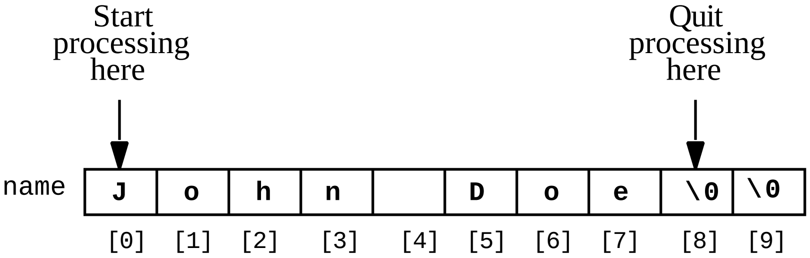
— Construct B to have required capacity
— Copy elements of A into first part of B
— Deallocate A

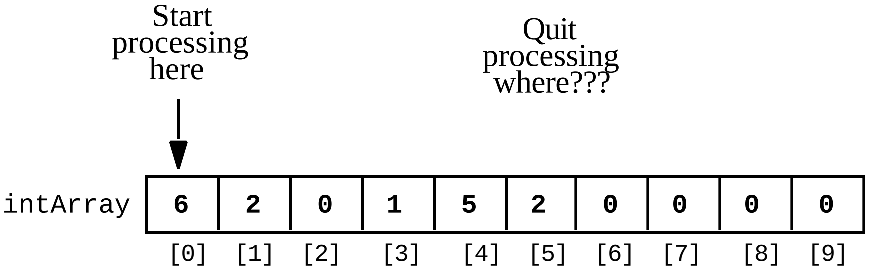
Solution 2 (OOP): Use `vector`s which do this automatically.

b. *There are virtually no predefined operations or functions on non-*`char` *arrays*.

Basic reason for this disparity:

There is no numeric equivalent of the NUL character that can be used to mark the end of a sequence of numbers.





Solution 1 (non-OOP): In addition to the array, pass its size (and perhaps its capacity) to functions.

Example: Function to output an array of `doubles`:

```
void Print(ostream & out, double theArray[], int itsSize)
{
   for (int i = 0; i < itsSize; i++)
      out << theArray[i] << endl;
}
```

Function call: `Print(cout, dubArray, dubArraySize);`

Example: Function to input an array of `doubles`:

```
void Read(istream & in, double theArray[],
          const int ITS_CAPACITY,  int & itsSize)
{
   itsSize = 0;

   for (;;)
   {
      in >> theArray[itsSize];

      if (in.eof()) break;

      itsSize++;
      if (itsSize >= ITS_CAPACITY) // prevent out-of-range error
      {
         cerr << "\nRead warning: array is full!\n";
         return;
      }
   }
}
```

Function call:

```
int mySize;
Read(cin, dubArray, CAPACITY, mySize);
```

- **The Deeper Problem**.
  One of the principles of object-oriented programming is that *an object should be autonomous (self-contained)*, which means that it should carry within itself all of the information necessary to describe and operate upon itself.

  C-style arrays violate this principle. In particular, they carry neither their size nor their capacity within them, and so *C-style arrays are not self-contained objects*.

Solution 2 (OOP): Encapsulate all three pieces of information — the array, its capacity and its size — within a class. This is the approach used by the `vector<T>` class template.