II. Introduction to Data Structures 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:

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

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

Attempt 2:

```c
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!

→ Quite often there's a tradeoff:

Example 2: Searching an online phone directory

Linear search?  OK for Calvin College, but too slow for Grand Rapids or New York

→ ________________ 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

Example 3: 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

→ ________________ is an important factor.

Example 4: Text processing

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

→ ________________ is an important factor
Definitions
1. An abstract data type (ADT) is:

   together with

Why “abstract?” Data, operations, and relations studied ________________________________

________ not __________

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

   and

Examples: Attempts 1 and 2 for TFA

3. Data abstraction: Separating the __________ of a data type from its ________________.
   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.

B. Simple Data Types (§2.2)

Memory:

2-state devices ↔ ________________

Organized into ________________ and __________ (machine dependent — e.g., 4 bytes).

Each byte (or word) has an __________ making it possible to store and retrieve contents of any given memory location.

Therefore:
- The most basic form of data: ________________________________
- 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)

   Basic operations:
   - and: \&\&
   - or: ||
   - not: !

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 =

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

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

   - **Two's complement representation**
     
     For \( n \geq 0 \): 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
        _________________
     3. Add 1
        _________________

     → Good for arithmetic computations (see p. 38)
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 \(n \geq 0\), 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: 011111110101000

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

d. **Problems:**

→ ____________________: 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:
   \[
   b_1.b_2b_3\ldots \times 2^k \quad \text{with each } b_i \text{ a bit and } b_1 = 1 \text{ (unless number is 0)}
   \]
   \[\uparrow\]
   
   mantissa   exponent
   or fractional part

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 (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.)

   — 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 #1

→ 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 ________________, where the basic operation is _____________________________________________.

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 ________________

Example:

   or better, use a named constant to specify the array capacity:
   
   const int CAPACITY = 100;

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

   const int CAPACITY = 100;

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

As an ADT In C++

ordered ←→ indices are numbered 0, 1, 2, . . . , CAPACITY – 1
fixed size ←→ CAPACITY specifies the capacity of the array
same type elements ←→ element_type is the type of elements
direct access ←→ 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 \__________\ and an \__________\ (or subscript) and is written

\[ array\_name[i] \]

Here \( i \) is an integer expression with \( 0 \leq i \leq CAPACITY - 1 \). This subscript operator returns a \__________\ \__________\ since it is a \__________\ called an \__________\ (or \__________\) \__________\ whose type is the specified \__________\ of the array.

This means that an 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:

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

// Read values into the first numScores elements of score

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

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:

```
rate = [0, 1, 2, 3, 4];
```

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

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

initializes rate as follows:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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:

```c
const int NAME_LENGTH = 10;
char collegeName[NAME_LENGTH] = {'C', 'a', 'l', 'v', 'i', 'n'};
```

initializes collegeName as follows:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a</td>
<td>l</td>
<td>v</td>
<td>i</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```c
char collegeName[NAME_LENGTH] = ________________
```
II. Data Structures and ADTs

Note 3: The null character '\0' (ASCII code is 0) is used as ______________.

Thus, character arrays used to store strings should be declared large enough to _______________

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______________________________, provided there is room for it.

Example:

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 ________________________________.

Example:

Note: This explains the brackets in constant declarations such as:

```cpp
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 __________________ of the array. Each array reference is then translated into an ___________ 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

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

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

```cpp
score[3] →
```

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

For an array variable `array_name`, its value is actually ________________________________ and
is the address of \texttt{array\_name[index]}. An array reference 
\texttt{array\_name[index]}
is equivalent to ______________

Here, * is the ______________ operator;
\texttt{*ref} returns:

For example, the statement
\begin{verbatim}
cout << score[3] << endl;
\end{verbatim}
could also be written
\begin{verbatim}
cout << ______________ << endl;
\end{verbatim}

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

\section*{D. C-Style Multidimensional Arrays}

\subsection*{1. Introduction}

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

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
         & Test 1 & Test 2 & Test 3 & Test 4 \\
\hline
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   \\
\hline
\end{tabular}
\end{center}

Use a two-dimensional array.

\subsection*{2. Declaring two-dimensional arrays}

\textbf{a. Usual form of declaration:}

\begin{verbatim}
element\_type array\_name[NUM\_ROWS][NUM\_COLUMNS];
\end{verbatim}

Example:
\begin{verbatim}
const int NUM\_ROWS = 30,
             NUM\_COLUMNS = 5;
\end{verbatim}

or using a typedef:
\begin{verbatim}
const int NUM\_ROWS = 30,
             NUM\_COLUMNS = 5;
\end{verbatim}
II. Data Structures and ADTs

b. Initializing

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

Example:

```c
double rates[2][3] =
```

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)

```c
row index  column index
```

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

Example:

```c
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++)
        __________________________;
}
```

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:

```c
const int RANKS = 10, ROWS = 30, COLUMNS = 5;
typedef __________________________ 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, i35, i36, i37, i38, i39, i40, i41, i42, i43, i44, i45, i46, i47, i48, NUM_INSEAM_SIZES};

typedef 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

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

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

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

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 ______________________________________.
```

Then, scoresTable[i][j] should be thought of as (scoresTable[i])[j], that is, as finding the j-th element of scoresTable[i].
II. Data Structures and ADTs

II-13

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

\[
scoresTable[10][3]
\]

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__________, so ________________________________________________________________________
__________________________________________________________________________
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:

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,

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

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 ________________________________.

Examples:

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

<table>
<thead>
<tr>
<th>32</th>
<th>°F</th>
</tr>
</thead>
<tbody>
<tr>
<td>degrees</td>
<td>scale</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>September</th>
<th>23</th>
<th>1998</th>
</tr>
</thead>
<tbody>
<tr>
<td>month</td>
<td>day</td>
<td>year</td>
</tr>
</tbody>
</table>

C++ provides _________ and ____________ 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 **struct** (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** so that items can be stored in / retrieved from them; but it differs from an array in that its elements may be of ____________________________.

3. Declaration (C-style):
   ```
   struct TypeName
   {
      declarations of members //of any types
   };
   ```
4. Examples:
   a. Temperature:
      \[ \text{32} \quad ^\circ F \]
      \text{degrees scale}
   
   b. Date:
      \text{September 23 1998}
      \text{month day year}
   
   c. Phone Listing:
      \begin{tabular}{|c|c|c|c|c|}
        \hline
        name & street & city & state & phone # \\
        \hline
        John Q. Doe & 12345 Calvin Rd. & Grand Rapids, MI & 9571234 \\
        \hline
      \end{tabular}
      \begin{verbatim}
      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
      \end{verbatim}
   
   d. Coordinates of a point: (Members need not have different types.)
      \[3.73 \quad -2.51\]
      \begin{verbatim}
      struct Point
      {  
        double xCoord, 
        yCoord;
      };
      Point p, q;
      \end{verbatim}
   
   d. Test scores: (Members may be structured types — e.g., arrays.)
      \begin{tabular}{|c|c|c|c|c|}
        \hline
        id-number & 83 & 79 & 92 & 85 \\
        \hline
      \end{tabular}
      \begin{verbatim}
      struct TestRecord
      {  
        unsigned idNumber,
        \textbf{score}[4];
      };
      
      TestRecord
      studentRecord, gradeBook[30];
      \end{verbatim}

5. Heirarchical (or nested) structs
Since the type of a member may be any type, it may be another struct. For example:

<table>
<thead>
<tr>
<th>John Q. Doe</th>
<th>12345 Calvin Rd</th>
<th>Grand Rapids, MI 9571234</th>
<th>June 17 1975</th>
<th>3.95</th>
<th>92.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>street</td>
<td>city &amp; state</td>
<td>phone #</td>
<td>month</td>
<td>day</td>
</tr>
</tbody>
</table>

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 ____________:
one of these is the ____________

    struct_var.member_name

Examples:

Input a value into the month member of birthday:

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++)

Output the name stored in student:

    cout <<
F. A Quick Look at Unions

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

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

2. A union differs from a struct in that the members ____________________. 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;
   }
II. Data Structures and ADTs

**Execution:**

Structure: 123456789 and 6.95336e-310  false  
Union: 123456789 and 3.21193e-273  true  
Structure: 123456789 and 0.123  true  
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Marital Status</th>
<th>Date</th>
<th>Spouse</th>
<th>Dependents</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>40</td>
<td>M</td>
<td>January 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mary Smith</td>
<td>8</td>
<td></td>
<td>January 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fred Jones</td>
<td>17</td>
<td>S</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jane</td>
<td>24</td>
<td>D</td>
<td>February 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peter</td>
<td>25</td>
<td>W</td>
<td>February 22</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```c
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 — __________ — but other fields that are different can be combined into a single structure by using a __________ — to add a ________________.

```c
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; // __________ 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.,

```c
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;

Addresses:

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, ..., f_n, requiring w_1, ..., w_n cells of storage, respectively:

Address of s.f_k = base address of s + offset

= __________________________

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 _____________________
• Use ___________________ to build _____________________________ for the variants
  (Derived classes inherit all of the non-private members of the base class.)

G. A commercial for OOP

Two programming paradigms:

_______________: 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

_______________: 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.
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 ________________ for the data members — use a ________________
2. Need ________________ for the operations.
3. "Package" declarations of these together in a ________________

See Fogire 2.2
II. Data Structures and ADTs

7. Problems with C-Style Arrays

a. 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 ___________ which do this automatically.

b. There are virtually no

   Basic reason for this disparity:

   There is no numeric equivalent of __________ that can be used to

   ____________________________________________________________________

   ![Diagram of name and intArray arrays with arrows indicating start and quit processing]
Solution 1 (non-OOP): In addition to the array, pass its _____ (and perhaps its _______) to functions.

Example: Function to output an array of doubles:

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

Function call: Print(cout, dubArray, ______________);

Example: Function to input an array of doubles:

```c
void Read(istream & in, double theArray[],
          ______________);
{
    ______________;
    for (;;) {
        in >> theArray[______________];
        if (in.eof()) break;
        ______________;
        if (______________________) // 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 ________________,
which means that it should ________________

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): ______________ all three pieces of information — the array, its capacity and its size —
_______________. This is the approach used by the vector class template.