Arrays allow a programmer to organize lists of values that are all of the same type (homogeneous).

But we are often faced with data values of differing types (heterogeneous) that are logically related, as we have seen earlier with trip data:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Destination</th>
<th>Miles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blacksburg, VA</td>
<td>Knoxville, TN</td>
<td>244</td>
<td>3:25</td>
</tr>
<tr>
<td>Knoxville, TN</td>
<td>Nashville, TN</td>
<td>178</td>
<td>2:35</td>
</tr>
<tr>
<td>Nashville, TN</td>
<td>Memphis, TN</td>
<td>210</td>
<td>3:17</td>
</tr>
<tr>
<td>Memphis, TN</td>
<td>Little Rock, AR</td>
<td>137</td>
<td>2:05</td>
</tr>
<tr>
<td>Little Rock, AR</td>
<td>Texarkana, TX</td>
<td>141</td>
<td>2:10</td>
</tr>
</tbody>
</table>

While we can organize this data using parallel arrays, that is unsatisfying because we have then distributed the logically related values for a single trip across several different data structures.

A better design would be to have a new data type `Trip` such that a variable of type `Trip` would contain all the related values for a single trip.

The `struct` mechanism allows C++ programmers to do just that…

```
struct Trip {
    string Origin;       // starting point of trip
    string Destination;  // ending point of trip
    int    Miles;        // distance traveled
    int    Minutes;      // time required, in min
    double MPH;          // average speed
};
```

```
Trip firstLeg;
const int MAXLEGS = 100;
Trip Itinerary[MAXLEGS];  // array of Trips
```


**Structures 2**

**10. Structures**

A variable of a struct type will contain the values for all of the specified members for that type. Of course, the values of those members will just be random garbage until they have been properly initialized. Assuming we've initialized the members of the variable `firstLeg`, we would have a memory layout something like this:

```
Trip firstLeg;
```

To reference a particular member of a struct variable, state the variable name followed by a period followed by the member name:

```
firstLeg.Origin      = "Blacksburg, VA";
firstLeg.Destination = "Knoxville, TN";
firstLeg.Miles       = 244;
firstLeg.Minutes     = 205;
firstLeg.MPH         = 71.2;
```

The period is called the member selector operator.

**Member Access Example**

The members of a struct variable may be used just as if they were simple variables of the specified type.

```
enum Color {RED, GREEN, BLUE, NOCOLOR};
struct Rectangle {
  Color Hue;
  int xNW,
       yNW;
  int Side[4];
};
Rectangle R;
cin >> R.xNW >> R.yNW;
R.Hue = NOCOLOR;
int Area = R.Side[0] * R.Side[1];
```
Aggregate Operations

An aggregate operation is an operation that is performed on a data structure, such as a structured variable, as a whole rather than performed on an individual member.

Assume the following declarations:

```cpp
struct Trip { int MPH; char Origin[25]; char Destination[25]; int Miles, Minutes; }; Trip X, Y;
```

Assuming that both X and Y have been initialized, consider the following statements and expressions involving aggregate operations:

- `X = Y;`       // _____ assigning one struct to another
- `X == Y;`      // _____ comparing two structs with a relational operator
- `cout << X;`   // _____ inserting an struct to a stream
- `X + Y`        // _____ performing arithmetic with structs
- `return X;`    // _____ using a struct as the return value from a function
- `Foo(X);`      // _____ passing an entire struct as a parameter

Of course, the operations that are not supported by default may still be implemented via user-defined functions.

struct Variables as Parameters

Entire struct variables can be passed to functions as parameters. By default, struct is passed by value.

For example:

```cpp
void setMPH(Trip& aTrip) {
    aTrip.MPH = MINPERHOUR * double(aTrip.Miles) / aTrip.Minutes;
}
```

Since struct variables tend to be large, it is generally better to pass them by constant reference than by value if possible:

```cpp
void printTrip(ofstream& Out, const Trip& aTrip) {
    Out << left << setw(MAXNAMELENGTH + 1) << aTrip.Origin;
    Out << setw(MAXNAMELENGTH + 1) << aTrip.Destination;
    Out << right << setw(10) << aTrip.Minutes / MINSPERHOUR << ':';
    Out << setfill('0') << setw(2) << (aTrip.Minutes % MINSPERHOUR);
    Out << setfill(' ') << setw(10) << setprecision(1) << aTrip.MPH;
}
```
### Implementing `struct` Operations

A programmer can implement any needed operations that are not supported automatically:

```cpp
bool areSame(const Trip& T1, const Trip& T2) {
    if ( (T1.Origin == T2.Origin) &&
         (T1.Destination == T2.Destination) ) {
        return true;
    }
    return false;
}

Note that the meaning of equality for trips must be defined. Here we require the trips to have the same starting and ending points. Other definitions could also make sense; it depends on the context.
```

```cpp
bool isFaster(const Trip& T1, const Trip& T2) {
    return (T1.MPH < T2.MPH);
}
```

```cpp
Trip reverseTrip(const Trip& T) {
    Trip Reverse = T;
    Reverse.Origin      = T.Destination;
    Reverse.Destination = T.Origin;
    return (Reverse);
}
```

### Array of Structures

In most applications of `struct` types, an array (or some other data structure) is used to organize a collection of individual `struct` variables.

An array of structures provides an alternative to using a collection of parallel arrays:

```cpp
const int MAXTRIPS = 500;
Trip Itinerary[MAXTRIPS];
```

Of course, to refer to a member of one of the `struct` variables in the array we must combine array syntax with `struct` syntax.

To set the first array element would require the syntax:

```cpp
Itinerary[0].MPG = 52.88;
```

To set the `MPG` member of the fifth `struct` variable would require the syntax:

```cpp
Itinerary[4].MPG = 52.88;
```
Designing with Structures

Structured types allow the creation of specialized data types which better model the logical relationships among the data values being stored and manipulated.

This allows the designer to consider the problem from a higher, more natural, level.

For example, we may design the step "print a trip" and defer the details of that until a later refinement of the design. Of course, we may do that without structured types, but the use of structured types encourages designers to think at that level.

Using a structured type may also promote code reuse if the type is suitable for inclusion in independent applications. The Trip type presented in these notes could be relevant to a number of different programs.

Properly speaking, a data type is a collection of values and the operations that may be performed on them. For a struct type, most of these operations will be implemented in separate user-defined functions, and those may also be recyclable into other applications.

Application: the Trip Program Revisited

Consider implementing the simple trip program using an array of structures to organize the trip data.

Clearly we may use an array of Trip variables, as defined earlier, to store the data:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Destination</th>
<th>Miles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blacksburg, VA</td>
<td>Knoxville, TN</td>
<td>244</td>
<td>3:25</td>
</tr>
<tr>
<td>Knoxville, TN</td>
<td>Nashville, TN</td>
<td>178</td>
<td>2:35</td>
</tr>
<tr>
<td>Nashville, TN</td>
<td>Memphis, TN</td>
<td>210</td>
<td>3:17</td>
</tr>
<tr>
<td>Memphis, TN</td>
<td>Little Rock, AR</td>
<td>137</td>
<td>2:05</td>
</tr>
<tr>
<td>Little Rock, AR</td>
<td>Texarkana, TX</td>
<td>141</td>
<td>2:10</td>
</tr>
</tbody>
</table>

By using an array of structures we will be able to implement the same operations as with a collection of parallel arrays, but the data organization will be much simpler. Almost all of the functions will have much simpler parameter lists.

We will consider portions of such an implementation here.
The Overall Organization

Since the data is all encapsulated in a single array of structures we achieve a very simple overall design:

```cpp
int main() {  
    Trip Itinerary[MAXTRIPS]; // list of trip data
    int numTrips = 0;  // number of trips reported.

    numTrips = readTripData(Itinerary);   // Read the given trip data.

    // Calculate the average MPH for each of the trips:
    calcAllMPH(Itinerary, numTrips);

    // Calculate the statistics for the summary report:
    int totalMiles, totalMinutes;
    double overallMPH;
    calcStats(Itinerary, numTrips, totalMiles, totalMinutes, overallMPH);

    // Write output file:
    writeReport(Itinerary, numTrips, totalMiles, totalMinutes, overallMPH);
    return 0; // Terminate a successful execution.
}
```

Managing Structured Input

Logically, reading and storing the trip data is relatively straightforward, given our earlier discussions. However, the syntax for using an array of `struct` variables bears examination:

```cpp
int readTripData(Trip List[]) {  
    int tripsRead = 0;  // counter for trips

    ifstream In(dataFileName.c_str()); // Open the input file.

    // If the input file does not exist, this will detect that.
    // We handle that by printing an error message and stopping the program.
    if ( In.fail() ) {  
        cout << "Data file not found: " // Write an error message...
        << dataFileName // including the file name.
        << endl // Bang "return"
        << "Exiting now..." << endl; // Finish the message.
        exit(1); // Terminate a failed execution.
    }

    // . . . continues on next slide . . .
```
Managing Structured Input

Here is the input management code. Note the use of a local `Trip` variable to "buffer" the input and the use of a helper function:

```c++
// . . . continued from preceding slide . . .
In.ignore(INT_MAX, '\n');  // Skip over the two header lines in the
In.ignore(INT_MAX, '\n');  // trip data input file.

Trip nextTrip;
nextTrip = readOneTrip(In);  // Try to read data for a trip.

while ( In && (tripsRead < MAXTRIPS) ) {
    List[tripsRead] = nextTrip;  // Store the trip that was just read
    tripsRead++;                  // . . . and count it.
    nextTrip = readOneTrip(In);   // Try to read data for another trip.
}
In.close();
return tripsRead;  // Tell the caller how many trips were read.
```
Managing Updates to the Array

The input code does not update the Trip records to include the average speed. This function manages that. Note the use of another helper function.

```c
void calcAllMPH(Trip Itinerary[], int numTrips) {
    int Idx;
    for (Idx = 0; Idx < numTrips; Idx++) {
        Itinerary[Idx].MPH = calcMPH(Itinerary[Idx].Miles,
                                      Itinerary[Idx].Minutes);
    }
}
```

The caller "extracts" the appropriate members of the current Trip variable and passes them to the helper function.

```c
double calcMPH(int Miles, int Minutes) {
    return (MINPERHOUR * double(Miles) / double(Minutes));
}
```

Alternative Implementation

The preceding implementation is correct but it requires both functions to incorporate information about the internal structure of a Trip variable. Here is a revision that makes the top-level function relatively unaware of those internals.

```c
void calcAllMPH(Trip Itinerary[], int numTrips) {
    int Idx;
    for (Idx = 0; Idx < numTrips; Idx++) {
        setMPH(Itinerary[Idx]);
    }
}
```

Now the caller simply passes an entire data element of the array to its helper function. The caller has no need to "know" anything about the internal organization of a Trip variable. That's all encapsulated within the helper function now.

```c
void setMPH(Trip& T) {
    T.MPH = MINPERHOUR * double(T.Miles) / double(T.Minutes);
}
```
### Writing the List to File

Compare the top-level function here to `calcAllMPH()`. See a "pattern"?

```cpp
void writeTripData(ofstream& Out, const Trip Itinerary[], int numTrips) {
    int Idx;
    for (Idx = 0; Idx < numTrips; Idx++) {
        writeOneTrip(Out, Itinerary[Idx]);
    }
}
```

Note the two uses of pass by constant reference to prevent unnecessary modification of data.

```cpp
void writeOneTrip(ofstream& Out, const Trip& toPrint) {
    Out << left << setw(MAXNAMELENGTH + 1) << toPrint.Origin;
    Out << setw(MAXNAMELENGTH + 1) << toPrint.Destination;
    Out << right << setw( 7) << toPrint.Miles
        << setw(10) << toPrint.Minutes
        << setw( 8) << setprecision(1) << toPrint.MPH
        << endl;
}
```

### Hierarchical Organization

In some cases a designer will plan a structured type with members that are also reasonably considered to be structured themselves.

For example, we might reconsider the design of the `Trip` type to allow a more useful description of the origin and destination. Logically each of these is just a "place" which we might consider to be represented by a city name and a state abbreviation:

```cpp
struct Place {
    string City;
    string State;
};
```

This approach provides two custom data types, instead of one. Variables of type `Place` might well be useful in other applications.

Hierarchical organization is often designed as a natural way to take advantage of already-existing custom types.

Of course a hierarchical organization also requires slightly more complex syntax when accessing the members of a structured member.
The usual syntax rules apply. It's just a matter of keeping straight exactly what the syntax means. Given the type definitions from the previous slide:

```c++
Trip toSmithsonian;
toSmithsonian.Origin.City = "Blacksburg";
toSmithsonian.Origin.State = "VA";
...
toSmithsonian.Miles = xxx;
toSmithsonian.Minutes = xxx;
...
```

Alternatively we could use an aggregate assignment to set the Place members of our Trip variable:

```c++
Place WashDC;
WashDC.City = "Washington";
WashDC.State = "DC";
Trip toSmithsonian;
toSmithsonian.Destination = WashDC;
...
```