

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:

```
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:

```
Itinerary[0] = firstLeg;
```

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

```
Itinerary[4].MPG = 52.88;
```

0	Blacksburg, VA Knoxville, TN 244 205 ??
1	Knoxville, TN Nashville, TN 178 155 ??
	. . .
4	Little Rock, AR Texarkana, AR 141 160 ??
	. . .

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.

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:

Origin	Destination	Miles	Time
Blacksburg, VA	Knoxville, TN	244	3:25
Knoxville, TN	Nashville, TN	178	2:35
Nashville, TN	Memphis, TN	210	3:17
Memphis, TN	Little Rock, AR	137	2:05
Little Rock, AR	Texarkana, TX	141	2:10
. . .			

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.

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

```
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.
}
```

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:

```
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 . . .
}
```

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:

```
// . . . 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.
}
```

Here we read and return data for a single trip:

```
Trip readOneTrip(ifstream& In) {

    Trip newTrip;                // Trip variable to "bundle" the data
    int tripHours, tripMinutes;

    getline(In, newTrip.Origin, '\t'); // Read: the name of the trip origin.
    getline(In, newTrip.Destination, '\t'); // and the trip destination.
    In >> newTrip.Miles;           // the length of trip in miles
    In >> tripHours;               // hours field for trip time
    In.ignore(1, ':');             // colon separator
    In >> tripMinutes;            // minutes field for trip time
    newTrip.Minutes = convertHHMMtoMin(tripHours, tripMinutes);
    In.ignore(INT_MAX, '\n');     // Skip to beginning of next input line.

    return newTrip; // Return a copy of the Trip variable to the caller.
}
```

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.

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

As usual, we have a for loop to traverse the data array so we can process its cells one-by-one.

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

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

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.

```
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.

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

Writing the List to File

Using struct Types 10

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

```
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.

```
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;  
}
```

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:

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

```
struct Trip {
    Place Origin;
    Place Destination;
    int Miles;
    int Minutes;
    double MPH;
};
```

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:

```
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:

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