

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:

| 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 |

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...

- structure - a heterogeneous collection of data values called members or fields
- each member has a type and a unique name
  - individual members are accessed by name

The definition of a structured type specifies the type name as well as the types and names of each of the members:

```
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  
};
```

Again, definitions of data types are almost always global since they are required throughout a program. The statement above does not declare a variable... it defines a type. As long as the type definition is in scope we may declare variables of that type in the usual way:

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

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;
```

In memory:

```
Blacksburg, VA  
Knoxville, TN  
244  
205  
71.2
```

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;
cin >> R.Side[0] >> R.Side[1] >> R.Side[2] >> R.Side[3];
R.Hue = NOCOLOR;

int Area = R.Side[0] * R.Side[1];
```

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

Assume the following declarations:

```
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.**

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

For example:

```
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:

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

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

```
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.**

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

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