The C `struct` mechanism is vaguely similar to the Java/C++ `class` mechanisms:

- supports the creation of user-defined data types
- `struct` types encapsulate data members

```
struct Location {
  int X, Y;
};
```

But there are vital differences:

- `struct` data members are "public", in fact there is no notion of access control
- `struct` types cannot have function members
- there is no concept of inheritance or of polymorphism
A struct Example

```c
struct Location { // declare type globally
    int X, Y;
};

int main() {
    struct Location A; // declare variable of type Location
    A.X = 5; // set its data members
    A.Y = 6;

    struct Location B; // declare another Location variable
    B = A; // copy members of A into B
    return 0;
}
```

Note:

- assignment is supported for `struct` types
- type declaration syntax used here requires specific use of `struct` in instance declarations
Another `struct` Example

```c
struct _Location {  // declare type globally
  int X, Y;
};

typedef struct _Location Location;  // alias a type name

int main() {
  Location A;  // declare variable of type Location
  A.X = 5;    // set its data members
  A.X = 6;

  Location B;  // declare another Location variable
  B = A;      // copy members of A into B
  return 0;
}
```

Note:
- use of `typedef` creates an alias for the `struct` type
- simplifies declaration of instances
What else is supported naturally for `struct` types? Not much…

- no automatic support for equality comparisons (or other relational comparisons)
- no automatic support for I/O of `struct` variables
- no automatic support for deep copy
- no automatic support for arithmetic operations, even if they make sense…
- can pass `struct` variables as parameters (default is pass-by-copy of course)
- can `return` a `struct` variable from a function
- can implement other operations via user-defined (non-member) functions
A struct Function Example

```c
struct _Location {    // declare type globally
    int X, Y;
};

typedef struct _Location Location;    // alias a type name

void initLocation(Location* L, int x, int y) {
    (*L).X = x;    // alternative:  L->X = x;
    (*L).Y = y;
}
// call:
initLocation(&A, 5, 6);
```

Note:

- must pass Location object by pointer so function can modify original copy
- given a pointer to a struct variable, we access its members by dereferencing the pointer (to get its target) and then using the member selector operator ' . '
- the parentheses around the *L are necessary because * has lower precedence than .
- however, we can write L->X instead of (*L).X.
- use of address-of ' & ' operator in call to create pointer to A
Another `struct` Function Example

```c
struct _Location {  // declare type globally
    int X, Y;
};

typedef struct _Location Location;  // alias a type name

Location updateLocation(Location Old, Location Move) {
    Location Updated;  // make a local Location object
    Updated.X = Old.X + Move.X;  // compute its members
    Updated.Y = Old.Y + Move.Y;
    return Updated;  // return copy of local object;
}
```

Note:

- we do not allocate `Updated` dynamically (via `malloc`); there is no need since we know at compile time how many we need (1) and we can just return a copy and avoid the cost of a dynamic allocation at runtime
- in C, dynamic allocation should only be used when logically necessary
Typical `struct` Code Organization

// header file Location.h contains declaration of type and // supporting functions
#ifndef LOCATION_H
#define LOCATION_H

struct _Location { // declare type globally
    int X, Y;
};

typedef struct _Location Location; // alias a clean type name

Location updateLocation(Location Old, Location Move);
...
#endif

// Source file Location.c contains implementations of supporting // functions
#include "Location.h"
Location updateLocation(Location Old, Location Move) {
    ...
}
...
// A struct type may contain array members, members of other
// struct types, anything in fact:
#ifndef QUADRILATERAL_H
#define QUADRILATERAL_H
#include "Location.h"
#define NUMCORNERS 4

struct _Quadrilateral {
    Location Corners[NUMCORNERS];
};
typedef struct _Quadrilateral Quadrilateral;
...
#endif

Note:

- even though you cannot assign one array to another and you cannot return an array from a function, you can do both of those things with a struct variable that contains an array member

- Why?
Example: Rational Numbers

One shortcoming in C is the lack of a type to represent *rational numbers*.

A *rational number* is the ratio of two integers, where the denominator is not allowed to be zero.

Rational numbers are important because we cannot represent many such fractions exactly in decimal form (e.g., 1/3).

The `struct` mechanism in C allows us to implement a type that accurately represents rational numbers (within the restrictions imposed by the limited range of integer types).

The following slides are a case study based on a course project.
One fact is clear enough: a rational value consists of two integer values.

The obvious C approach would be:

```c
struct _Rational {
  int32_t Top;
  int32_t Bottom;
};
typedef struct _Rational Rational;
```

A forward-looking approach might use `int64_t` instead, buying increased range and doubling the storage cost.

Another thought would be to normalize the representation by using a `uint32_t` for the denominator, so that a negative rational would always use a negative numerator.

For this example, we'll stick with the C code shown above.
Designing Operations

When implementing a data type, we must consider what operations would be expected or useful to potential users.

In this case, we have mathematics as a guide:

- creating a Rational object with any valid value
- adding two Rational objects to yield a third Rational object
- subtracting two Rational objects to yield a third Rational object
- multiplying two Rational objects to yield a third Rational object
- dividing two Rational objects to yield a third Rational object
- taking the absolute value of a Rational object, yielding a second Rational object
- negating a Rational object, yielding a second Rational object
- comparing two Rational objects, with equals, less-than, etc.
- taking the floor/ceiling of a Rational object, yielding an integer
A Specific Operation

```c
/**
 * Compute the sum of Left and Right.
 * Pre:
 * Left and Right have been properly initialized.
 * Returns:
 * A Rational object X equal to Left + Right.
 */
Rational Rational_Add(const Rational Left, const Rational Right) {
    Rational Sum;
    Rational_Normalize(&Sum);
    return Sum;
}
```
An Array Type

One way to address (some of) the shortcomings in C arrays would be to implement:

```c
struct _iArray {
    int32_t* Data;
    uint32_t Dimension;
    uint32_t Usage;
};
typedef struct _iArray iArray;

bool iArray_Init(iArray* A, uint32_t Size) {
    A->Data = calloc(Size * sizeof(int32_t));
    if ( A->Data == NULL ) {
        A->Dimension = A->Usage = 0;
        return false;
    }
    A->Dimension = Size;
    A->Usage = 0;
    return true;
}
```
An Array Type

Access operations could now be implemented safely:

```c
bool iArray_Append(iArray* const A, int32_t Elem) {
    if (A == NULL ||
        A->Dimension == A->Usage) {
        return false;
    }
    A->Data[Usage] = Elem;
    A->Usage++;
    return true;
}
```

It's easy to think of a number of other, similar functions.