`struct` Properties

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

```
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.Y = 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

`struct` Limitations

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

```
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;
}
```

```
Location A;

// 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

```
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) {
   . . .
}
. . .
```

More Complex `struct` Types

```
// 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.

Designing Data Representation

One fact is clear enough: a rational value consists of two integer values.

The obvious C approach would be:

```
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

```
/**
 *   Compute the sum of Left and Right.
 *   Pre:
 *         *Left and *Right have been properly initialized.
 *   Returns:
 *         A pointer to a Rational object equal to *Left + *Right.
 */
Rational* Rational_Add(const Rational* Left, const Rational* Right)
{

   Rational *Sum = malloc(sizeof(Rational));

   Sum->Top    = Left->Top * Right->Bottom +
                 Left->Bottom * Right->Top;
   Sum->Bottom = Left->Bottom * Right->Bottom;
   Rational_Normalize(Sum);

   return Sum;
}
```

```
Rational First, Second;
...   // initialize First and Second
Rational *Sum = Rational_Add(&First, &Second);
```

A Different Take on That

```
/**
 *   Compute the sum of Left and Right.
 *   Pre:
 *        *pSum is a Rational object
 *        Left and Right have been properly initialized.
 *   Post:
 *        *pSum is a normalized representation of Left + Right.
 */
void Rational_Add(Rational* const pSum, const Rational Left,
                                        const Rational Right) {

   pSum->Top    = Left.Top * Right.Bottom +
                  Left.Bottom * Right.Top;

   pSum->Bottom = Left.Bottom * Right.Bottom;

   Rational_Normalize(pSum);
}
```

```
Rational First, Second, Sum;
...   // initialize First and Second
Rational_Add(&Sum, First, Second);
```

An Array Type

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

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

```
bool iArray_Init(iArray* const pA, uint32_t Size) {

   if ( pA == NULL ) return false;
   pA->Data = calloc(Size * sizeof(int32_t));
   if ( pA->Data == NULL ) {
      pA->Dimension = pA->Usage = 0;
      return false;
   }
   pA->Dimension = Size;
   pA->Usage = 0;
   return true;
}
```

Safe Array Insertion

Mutator operations could now be implemented safely:

```
bool iArray_Append(iArray* const pA, int32_t Elem) {

   if ( pA == NULL ||
        pA->Dimension == pA->Usage) {
      return false;
   }
   pA->Data[Usage] = Elem;
   pA->Usage++;
   return true;
}
```

Reject insertion if array is full

`Data[usage]` is the first unused cell in the array

Or…

```
bool iArray_Append(iArray* const pA, int32_t Elem) {

   if ( pA == NULL ) return false;

   if ( pA->Dimension == pA->Usage) {

      int32_t *temp = realloc(pA->Data, 2 * pA->Dimension);
      if ( pA->Data == NULL ) {
         return false;
      }

      pA->Data = temp;
      pA->Dimension = 2 * pA->Dimension;
   }

   pA->Data[Usage] = Elem;
   pA->Usage++;
   return true;
}
```

Check whether array is full

realloc() will:

- Allocate new array or simply "grow" the old one
- Copy data from old array to new array, if necessary
- Deallocate old array, if necessary
- Return NULL if fails