\textbf{struct Properties}

The C \textbf{struct} mechanism is vaguely similar to the Java/C++ \textbf{class} mechanisms:

- supports the creation of user-defined data types
- \textbf{struct} types encapsulate data members

\begin{verbatim}
struct Location {
    int X, Y;
};
\end{verbatim}

But there are vital differences:

- \textbf{struct} data members are "public", in fact there is no notion of access control
- \textbf{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
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

```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;
    (*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

```c
// 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

```c
// 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?