Structures

What is a structure?
- One or more values, called members, with possibly dissimilar types that are stored together.
- Used to group together different types of variables under the same name.
- Aggregates a fixed set of labeled objects, possibly of different types, into a single object (like a record)

What is a structure NOT?
- Since members are NOT the same type/size, they are not as easy to access as array elements that are the same size.
- Structure variable names are NOT replaced with a pointer in an expression (like arrays)
- A structure is NOT an array of its members so can NOT use subscripts.
struct tag {member_list} variable_list;

---

**struct S {**
  int a;
  float b;
} x;

Declarers x to be a structure having two members, a and b. In addition, the structure tag S is created for use in future declarations.

**struct {**
  int a;
  float b;
} z;

Omitting the tag field; cannot create any more variables with the same type as z.

**struct S {**
  int a;
  float b;
};

Omitting the variable list declares another structure variable y with the same type as x.

**struct S y;**

**struct S;**

Incomplete declaration which informs the compiler that S is a structure tag to be defined later.
This declaration introduces the type struct fraction (both words are required) as a new type.

- C uses the period (.) to access the fields in a record.
- You can copy two records of the same type using a single assignment statement, however == does not work on structs (see note link).

```c
struct fraction {  
    int numerator;  
    int denominator;  
} // can’t initialize

struct fraction f1, f2; // declare two fractions
f1.numerator = 25;
f1.denominator = 10;
f2 = f1; // this copies over the whole struct
```
Structure Declarations (cont)

- So tag, member_list and variable_list are all optional, but cannot all be omitted; at least two must appear for a complete declaration.

```
struct {
    int a;
    char b;
    float c;
} x;
```

- Single variable x contains 3 members

```
struct {
    int a;
    char b;
    float c;
} y[20], *z;
```

- Treated different by the compiler
  DIFFERENT TYPES
  i.e. z = &x is ILLEGAL

- So all structures of a given type must be created in a single declaration? NO.

- An array of 20 structures (y); and
- A pointer to a structure of this type (z)
More Structure Declarations

- The TAG field
  - Allows a name to be given to the member list so that it can be referenced in subsequent declarations
  - Allows many declarations to use the same member list and thus create structures of the same type

```c
struct SIMPLE {
    int a;
    char b;
    float c;
};
```

So ➞ struct SIMPLE x;
struct SIMPLE y[20], *z;

Now x, y, and z are all the same kind of structure

Associates tag with member list; does not create any variables
MEMBERS
- Any kind of variable that can be declared outside a structure may also be used as a structure member.
- Structure members can be scalars, arrays, pointers and even other structures.

ACCESS using dot operator
- Two operands
  - Left = name of structure variable
  - Right = name of the desired member
  - Result = the designated member

OPERATOR PRECEDENCE
- The subscript and dot operators have the same precedence and all associate left to right.
- The dot operator has higher precedence then the indirection

Pointer2Structure
- \( \rightarrow \) operator
- Left = *must* be a pointer to a structure
- Right = member

Example
- \((*sp).a == sp\rightarrow a\)
- Indirection built into arrow/infix operator
- Follow the address to the structure

```c
struct COMPLEX {
    float f;
    int a[20];
    long *lp;
    struct SIMPLE s;
    struct SIMPLE sa[10];
    struct SIMPLE *sp;
} cmplx, cmp[10];
```
typedefs \( \rightarrow \) \texttt{typedef <type> <name>;} \\

\begin{itemize}
  \item Ex1: \#define true 1 \#define false 0
typedef \texttt{int} bool;
bool flag = false;
  \end{itemize}

\begin{itemize}
  \item Ex2: char *\texttt{ptr_to_char}; // new variable
typedef char* \texttt{ptr_to_char}; // new type
ptr_to_char a; // new variable
\end{itemize}
Using typedefs with Structures

A typedef statement introduces a shorthand name for a type. The syntax is...

```c
typedef <type> <name>;
```
- shorter to write
- can simplify more complex type definitions

```c
typedef struct {
    int a;
    char b;
    float c;
} Simple;
```

So ➔ Simple x;
Simple y[20], *z;

Now x, y, and z are all the same TYPE.

Similar to ➔ int x;
int y[20], *z;