C structures are collections of related variables
- May contain variables of many different types

Compared to arrays...
- Similar in that one variable holds several values together
- Different in that arrays can only contain elements of the same data type

Commonly used to define
- Records to be stored in files
- Structures are derived data types
- They are constructed using objects of other types

Structure Definition
- Example: An employee record:
  ```
  struct empRec {
      char *lastName;
      char *firstName;
      int age;
      float salary;
  };
  ```
  Allows all of the information about an employee to be aggregated under one variable

More Detailed Explanation
- Keyword struct introduces the structure definition
- The structure tag is the identifier empRec
- Structure tag names the structure definition
- Used with the keyword struct to declare variables of the structure type
- The structure type is: struct empRec
- Structure definition must end with a semicolon
- Note that the preceding example does not declare any variables.
  - It just defines the type of a structure named empRec

Structure Members
- The structure members are the fields declared within the braces of the structure definition
  - In this case: lastName, firstName, age, salary
- Members must have unique names
- But two different structure definitions may include members with the same name

More About Structure Members
- Structure members can be variables of:
  - Basic data types such as int, float, char
  - Arrays
  - Other structures (other than itself)
- Structure member can not be
  - Variables of the same structure type
  - But could be pointers to the same type
- Important: A structure definition does not declare any variables or reserve any space in memory
  - It creates a new data type that can, in turn, be used to used to declare variables
Placement of Structure Definitions

- Structures may be defined outside of `main()`
  ```c
  struct card {
    char *face;
    char *suit;
  };
  int main() {
    /* main code goes here */
  }
  
  This definition can then be used anywhere in the file.
  
- Alternatively a structure definition can be placed inside of the body of `main()` or another function.
  - In this case, the definition is local to the function in which it occurs.

Declaring Structure Variables

- Declared just like variables of other data types
  ```c
  struct card a, deck[52], *cPtr;
  
  This declaration declares:
  - `a` to be a variable of type `struct card`
  - `deck` to be an array of 52 elements of type `struct card`
  - `cPtr` to be a pointer to a variable of type `struct card`
  ```

Structure Operations

- Structure operations include...
  - Structure assignments
  - Address (&) operator
  - Accessing members
  - Using `sizeof` operator
  - NOT comparing structures
  - Why not?

Initializing Structure Variables

- Structure variables can be initialized like arrays
  ```c
  struct card a = {"Three", "Hearts"};
  ```
  Creates a variable `a` of type `struct card`
  - Initializes the member `face` to point to a character string "Three"
  - Initializes `suit` to "Hearts"
Structure Variable Initialization—Continued

- If there are fewer elements in the initializer list than members
  - Numerics members are initialized to 0
  - Pointers are initialized to NULL
- A structure variable may also be initialized by:
  - Assigning a structure variable of the same type:
    ```
    struct card b = a; /* copies all members of a to b*/
    ```
  - Assigning values to the individual members of the structure

Accessing Members

- Two operators used to access members of structure variables:
  - Structure member operator, or dot (.)
  - Structure pointer operator, or arrow (->)

```c
struct card a, *aPtr;
    ... 
printf("%s ", a.suit);
printf("%s ", aPtr->suit);
printf("%s ",(*aPtr).suit);
```

- What's the difference?

Explanation of Syntax

- `a.suit` evaluates to the value stored in `suit`
- `aPtr->suit` evaluates to the value stored in `suit`
- `(*aPtr).suit`:
  - `aPtr` points to the entire structure `a`
  - When `aPtr` is dereferenced, it contains the value of `a` and hence can access its member `suit`
- Hence all three are equivalent

Passing Structure Variables as Parameters to Functions

- Can pass structure variables to functions by passing:
  - Individual structure members
  - an entire structure
  - a pointer to a structure
- Default is pass by value
- In order to pass a structure variable by reference, must pass a pointer to the structure variable (just like pass-by-reference for any other variable)
- Note: Arrays of structures, like all other arrays, are automatically passed by reference

Function Example

- Prints structure variable two ways
  - Passing the entire structure variable to a function (by value)
  - Passing a pointer to the structure variable to a function
- One is call by value and the other is call by reference
- Note: Passing a structure as a parameter is different than passing an array
  - default is pass-by-value for a structure variable

Function Example

```c
#include <stdio.h>

struct student {
    char *name;
    int number;
    char grade;
};

void printStudent1(struct student);
void printStudent2(struct student *);

main() {
    struct student stud1 = {"Bob", 52329, 'B'};
    struct student stud2 = {"Jim", 02134, 'A'};
    printStudent1(stud1); /* Pass in structure */
    printStudent2(&stud2); /* Pass in pointer to it*/
}
```

Continued on next slide.
void printStudent1(struct student st) {
    /* Note: st is NOT a pointer but actual structure */
    printf("student's name is %s\n", st.name);
    printf("student's number is %d\n", st.number);
    printf("student's grade is %c\n", st.grade);
}

void printStudent2(struct student *st) {
    printf("student's name is %s\n", st->name);
    printf("student's number is %d\n", st->number);
    printf("student's grade is %c\n", st->grade);
}

Another Function Example

- Illustrates the difference between:
  - Passing structure variables by value
  - Passing structure variables by reference
- Prints structure variable before and after calls to various modifyStudent() functions

Second Example

```c
#include <stdio.h>

struct student {
    char *name;
    int number;
    char grade;
};

void printStudent1(struct student);
void modifyStudent1(struct student);
void modifyStudent2(struct student *);

main() {
    struct student stud1 = {"Bob", 52329, 'B'};
    struct student stud2 = {"Jill", 02134, 'A'};
    printf("Before modifyStudent1 function call\n");
    printStudent1(stud1);
    modifyStudent1(stud1);
    printf("After modifyStudent1 function call\n");
    printStudent1(stud1);
    printf("Before modifyStudent2 function call\n");
    printStudent1(stud1);
    modifyStudent2(&stud1);
    //end of main()
}
```

void modifyStudent1(struct student st) {
    st.name = "Bill";
    st.number = 94305;
    st.grade = 'F';
}

void modifyStudent2(struct student *st) {
    st->name = "Bill";
    st->number = 94305;
    st->grade = 'F';
}

void printStudent1(struct student st) {
    /* Note: st is NOT a pointer but actual structure */
    printf("student's name is %s\n", st.name);
    printf("student's number is %d\n", st.number);
    printf("student's grade is %c\n", st.grade);
}

Typedefs

- Provides a way for creating “synonyms” or “aliases” for previously defined data types
- Names of structure types are often defined with typedef to create shorter type names
- Example:
  typedef struct card Card;
  Makes the new type name Card that can be used in place of the name struct card
- Note: typedef does not create a new type but rather a new name for an existing type
Another way of using typedef:

- Can create a structure type so a structure tag is not required
- Example
  typedef struct {
    char *face;
    char *suit;
  } Card;
- Creates the structure type Card without need for a separate typedef statement

Using a typedef to Declare Variables

- Now we can use the typedef Card to declare variables of type struct card
- Example:
  Card deck[52]; /* Creates an array of 52 card structures*/

Benefits of using typedef

- Meaningful names help make programs self-documenting
- Often typedef is used to create synonyms for the basic data types, too
- Example
  typedef *char charPointer;
  // Creates new name for type *char