C Programming for Engineers

Structures, Unions

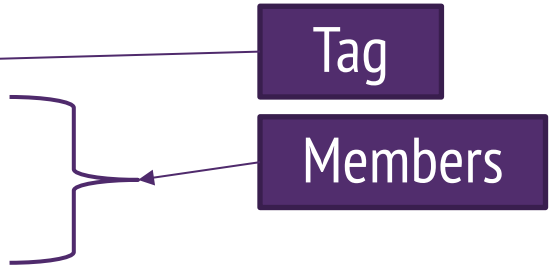
ICEN 360– Spring 2017

Prof. Dola Saha

Structure

- Collections of related variables under one name.
- Variables of may be of different data types.
-

```
struct card {
    char *face;
    char *suit;
};
```



- Keyword struct introduces the structure definition.
- Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict.

Structure Declaration

```
struct employee {
    char firstName[20];
    char lastName[20];
    unsigned int age;
    char gender;
    double hourlySalary;
};
```

```
struct employee employee1, employee2;
```

```
struct employee employees[100];
```

```
struct employee {
    char firstName[20];
    char lastName[20];
    unsigned int age;
    char gender;
    double hourlySalary;
} employee1, employee2, *employeePtr;
```

Structure Tag

- The structure tag name is optional.
- If a structure definition does not contain a structure tag name, variables of the structure type may be declared *only* in the structure definition—*not* in a separate declaration.

Self Reference

- *A structure cannot contain an instance of itself.*
- A variable of type `struct employee` cannot be declared in the definition for `struct employee`.
- A pointer to `struct employee`, may be included.
- For example,

```
struct employee2 {
    char firstName[20];
    char lastName[20];
    unsigned int age;
    char gender;
    double hourlySalary;
    struct employee2 person; // ERROR
    struct employee2 *ePtr; // pointer
};
```

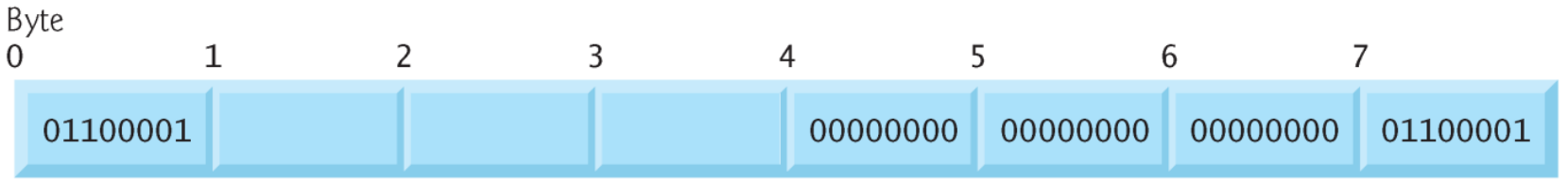
- `struct employee2` contains an instance of itself (person), which is an error.

Storage in Memory

- Structures may *not* be compared using operators == and !=, because
  - structure members are not necessarily stored in consecutive bytes of memory.
- Computers may store specific data types only on certain memory boundaries such as half-word, word or double-word boundaries.
- A word is a standard memory unit used to store data in a computer—usually 2 bytes or 4 bytes.

Storage in Memory

```
struct example {
    char c;
    int i;
} sample1, sample2;
```



Possible storage, but machine dependant

Initialization

```
struct card {
    char *face;
    char *suit;
};
```

```
struct card aCard = {"Three", "Hearts"};
```

- If there are fewer initializers in the list than members in the structure,
  - the remaining members are automatically initialized to 0
  - or NULL if the member is a pointer.
- Assignment Statement of same struct type
  - `struct card aCard1 = aCard2;`

Accessing Structure Members

- the structure member operator (.)—also called the dot operator
  - `printf("%s", aCard.suit); // displays Hearts`
- the structure pointer operator (->)—also called the arrow operator.
  - `cardPtr = &aCard;`
  - `printf("%s", cardPtr->suit); // displays Hearts`
  - Following are equivalent
    - `cardPtr->suit`
    - `(*cardPtr).suit`

Example

```
 4  #include <stdio.h>
 5
 6  // card structure definition
 7  struct card {
 8     char *face; // define pointer face
 9     char *suit; // define pointer suit
10  };
11
12  int main(void)
13  {
14     struct card aCard; // define one struct card variable
15
16     // place strings into aCard
17     aCard.face = "Ace";
18     aCard.suit = "Spades";
19
20     struct card *cardPtr = &aCard; // assign address of aCard to cardPtr
21
22     printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
23        cardPtr->face, " of ", cardPtr->suit,
24        (*cardPtr).face, " of ", (*cardPtr).suit);
25  }
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Structure with Function

- Structures may be passed to functions by
  - passing individual structure members
  - by passing an entire structure
  - by passing a pointer to a structure.
- Functions can return
  - individual structure members
  - an entire structure
  - a pointer to a structure

typedef

- The keyword `typedef` is a way to create synonyms (or aliases) for previously defined data types.
- Names for structure types are often defined with `typedef` to create shorter type names.
- Example:
  - `typedef struct card Card;`

    `Card` is a synonym for type `struct card`.
- Example:
  - ```
    typedef struct {
       char *face;
       char *suit;
    } Card;
    ```
  - `Card myCard, *myCardPtr, deck[52];`
    ```

Card Shuffling Example (1)

```
1  // Fig. 10.3: fig10_03.c
2  // Card shuffling and dealing program using structures
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  #define CARDS 52
8  #define FACES 13
9
10 // card structure definition
11 struct card {
12    const char *face; // define pointer face
13    const char *suit; // define pointer suit
14 };
15
16 typedef struct card Card; // new type name for struct card
17
18 // prototypes
19 void fillDeck(Card * const wDeck, const char * wFace[],
20    const char * wSuit[]);
21 void shuffle(Card * const wDeck);
22 void deal(const Card * const wDeck);
23
```

Card Shuffling Example (2)

```
24  int main(void)
25  {
26     Card deck[CARDS]; // define array of Cards
27
28     // initialize array of pointers
29     const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
30        "Six", "Seven", "Eight", "Nine", "Ten",
31        "Jack", "Queen", "King"};
32
33     // initialize array of pointers
34     const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
35
36     srand(time(NULL)); // randomize
37
38     fillDeck(deck, face, suit); // load the deck with Cards
39     shuffle(deck); // put Cards in random order
40     deal(deck); // deal all 52 Cards
41  }
42
```

Card Shuffling Example (3)

```
43  // place strings into Card structures
44  void fillDeck(Card * const wDeck, const char * wFace[],
45     const char * wSuit[])
46  {
47     // loop through wDeck
48     for (size_t i = 0; i < CARDS; ++i) {
49        wDeck[i].face = wFace[i % FACES];
50        wDeck[i].suit = wSuit[i / FACES];
51     }
52  }
53
54  // shuffle cards
55  void shuffle(Card * const wDeck)
56  {
57     // loop through wDeck randomly swapping Cards
58     for (size_t i = 0; i < CARDS; ++i) {
59        size_t j = rand() % CARDS;
60        Card temp = wDeck[i];
61        wDeck[i] = wDeck[j];
62        wDeck[j] = temp;
63     }
64  }
65
```

Card Shuffling Example (4)

```
66  // deal cards
67  void deal(const Card * const wDeck)
68  {
69     // loop through wDeck
70     for (size_t i = 0; i < CARDS; ++i) {
71        printf("%5s of %-8s%s", wDeck[i].face , wDeck[i].suit ,
72           (i + 1) % 4 ? "  " : "\n");
73     }
74  }
```

Card Shuffling Example (5)

```
Three of Hearts     Jack of Clubs     Three of Spades      Six of Diamonds
 Five of Hearts    Eight of Spades    Three of Clubs     Deuce of Spades
 Jack of Spades     Four of Hearts    Deuce of Hearts      Six of Clubs
Queen of Clubs    Three of Diamonds  Eight of Diamonds     King of Clubs
 King of Hearts    Eight of Hearts    Queen of Hearts    Seven of Clubs
Seven of Diamonds   Nine of Spades     Five of Clubs     Eight of Clubs
  Six of Hearts   Deuce of Diamonds    Five of Spades     Four of Clubs
Deuce of Clubs     Nine of Hearts    Seven of Hearts     Four of Spades
  Ten of Spades    King of Diamonds     Ten of Hearts     Jack of Diamonds
 Four of Diamonds   Six of Spades     Five of Diamonds     Ace of Diamonds
  Ace of Clubs     Jack of Hearts      Ten of Clubs     Queen of Diamonds
  Ace of Hearts     Ten of Diamonds    Nine of Clubs      King of Spades
  Ace of Spades    Nine of Diamonds   Seven of Spades    Queen of Spades
```

Classwork Assignment

- Write a program to generate data for N students. Use structure to create numeric ID and points (max 100) as 2 separate members. Randomly generate data for N students. Display both the ID and the points of the student who has received highest point.

Union

- A union is a *derived data type*—like a structure—with members that *share the same storage space*.
- For different situations in a program, some variables may not be relevant, but other variables are—so a union shares the space instead of wasting storage on variables that are not being used.
- The members of a union can be of any data type.
- The number of bytes used to store a union must be at least enough to hold the *largest* member.

Definition

```
union number {
    int x;
    double y;
};
```

- In a declaration, *a union may be initialized with a value of the same type as the first union member.*

```
union number value = {10};
union number value = {1.43}; // ERROR
```

Permitted Operations

- The operations that can be performed on a union are:
  - assigning a union to another union of the same type,
  - taking the address (&) of a union variable,
  - and accessing union members using the structure member operator and the structure pointer operator.
- Unions may not be compared using operators == and != for the same reasons that structures cannot be compared.

Union Example (1)

```
1  // Fig. 10.5: fig10_05.c
2  // Displaying the value of a union in both member data types
3  #include <stdio.h>
4
5  // number union definition
6  union number {
7     int x;
8     double y;
9  };
10
11 int main(void)
12 {
13    union number value; // define union variable
14
15    value.x = 100; // put an integer into the union
16    printf("%s\n%s\n%s\n  %d\n\n%s\n  %f\n\n\n",
17       "Put 100 in the integer member",
18       "and print both members.",
19       "int:", value.x,
20       "double:", value.y);
```

Union Example (2)

```
21
22      value.y = 100.0; // put a double into the same union
23      printf("%s\n%s\n%s\n  %d\n\n%s\n  %f\n",
24         "Put 100.0 in the floating member",
25         "and print both members.",
26         "int:", value.x,
27         "double:", value.y);
28   }
```

```
Put 100 in the integer member
and print both members.
int:
  100

double:
  -92559592117433136000000000000000000000000000000000000000000000.000000


Put 100.0 in the floating member
and print both members.
int:
  0

double:
  100.000000
```

Enumeration

- Keyword enum, is a set of integer enumeration constants represented by identifiers.
- Values in an enum start with 0, unless specified otherwise, and are incremented by 1.
- For example, the enumeration
  - ```
    enum months {
        JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
        OCT, NOV, DEC};
    ```

    creates a new type, `enum months`, identifiers are set to the integers 0 to 11, respectively.
- Example:
  - ```
    enum months {
        JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
        SEP, OCT, NOV, DEC};
    ```

identifiers are set to integers 1 to 12, respectively.

Enumeration Example

```
1  // Fig. 10.18: fig10_18.c
2  // Using an enumeration
3  #include <stdio.h>
4
5  // enumeration constants represent months of the year
6  enum months {
7     JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
8  };
9
10 int main(void)
11 {
12    // initialize array of pointers
13    const char *monthName[] = { "", "January", "February", "March",
14       "April", "May", "June", "July", "August", "September", "October",
15       "November", "December" };
16
17    // loop through months
18    for (enum months month = JAN; month <= DEC; ++month) {
19       printf("%2d%11s\n", month, monthName[month]);
20    }
21 }
```

Enumeration Example Output

```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
 7       July
 8     August
 9  September
10    October
11   November
12   December
```