Lecture#5

Structs

CENG 102- Algorithms and Programming II,

2024-2025, Spring

Contains materials from:
P. Deitel, H. Deitel, ‘’C How to Program with an Introduction to C++’’, 8th edition, Pearson

10.1 Introduction

- Structures—sometimes referred to as aggregates—are collections of related variables under one name.
- Structures **may contain variables of many different data types**—in contrast to arrays, which contain *only* elements of the same data type.
- *Pointers* and *structures* facilitate the formation of more complex **data structures** such as linked lists, queues, stacks and trees.

10.1 Introduction

We have to declare structure in C before using it in our program. In structure declaration, we specify its member variables along with their datatype. We can use the struct keyword to declare the structure in C using the syntax given.

Syntax

```
struct structure_name {
    data_type member_name1;
    data_type member_name1;
    ....
    ....
};
```

10.2 Structure Definitions

- Structures are derived data types—they're constructed using objects of other types.
- Consider the following structure definition:
  - ```
    struct card {
        char *face;
        char *suit;
    };
    ```
- Keyword `struct` introduces the structure definition.
- The identifier `card` is the structure name (structure tag).
    ```

- Variables declared within the braces of the structure definition are the structure's members (fields).
- Each structure definition *must* end with a semicolon.

Struct keyword

Tag or structure tag

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

Members or
Fields of structure

10.2 Structure Definitions (Cont.)

- The definition of `struct card` contains members `face` and `suit`, each of type `char *`.
- Structure members can be variables of the primitive data types (e.g., `int`, `float`, etc.), or aggregates, such as arrays and other structures.

10.2 Structure Definitions (Cont.)

- For example, the following `struct` contains `char` array members for an employee's first and last names, an `unsigned int` member for the employee's age, a `char` member that would contain `'M'` or `'F'` for the employee's gender and a `double` member for the employee's hourly salary:
  - ```
    struct employee {
        char firstName[20];
        char lastName[20];
        unsigned int age;
        char gender;
        double hourlySalary;
    };
    ```
    ```

10.2.1 Self-Referential Structures

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

10.2 Structure Definitions (Cont.)

- Because `ePtr` is a pointer (to type `struct employee2`), it's permitted in the definition.
- A structure containing a member that's a pointer to the *same* structure type is referred to as a self-referential structure.
- Self-referential structures are used to build linked data structures.

Common Programming Error 10.2

A structure cannot contain an instance of itself.

10.2.2 Defining Variables of Structure Types

- Structure definitions do *not* reserve any space in memory; rather, each definition creates *a new data type* that's used to define variables.
- Structure variables are defined like variables of other types.

  - `struct card aCard, deck[52], *cardPtr;`
    - declares `aCard` to be a variable of type `struct card`
    - declares `deck` to be an array with 52 elements of type `struct card`
    - declares `cardPtr` to be a pointer to `struct card`.

10.2 Structure Definitions (Cont.)

- Variables of a given structure type may also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition.

10.2 Structure Definitions (Cont.)

- Structure Variable Declaration with Structure Template

```
struct structure_name {
    data_type member_name1;
    data_type member_name1;
    ....
    ....
}variable1, varaible2, ...;
```

10.2 Structure Definitions (Cont.)

- For example, the preceding definition could have been incorporated into the `struct card` definition as follows:

    - ```
      struct card {
          char *face;
          char *suit;
      } aCard, deck[52], *cardPtr;
      ```
      ```

10.2 Structure Definitions (Cont.)

```
struct Person
{
  // code for members
} prsn1, prsn2, p[20];
```

```
struct Person
{
  // code for members
};

void main()
{
  struct Person prsn1, prsn2, p[20];
}
```

10.2.3 Structure Tag Names

- 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.

Good Programming Practice 10.1

Always provide a structure tag name when creating a structure type. The structure tag name is required for declaring new variables of the structure type later in the program.

10.2.4 Operations That Can Be Performed on Structures

- The only valid operations that may be performed on structures are:
  - assigning structure variables to structure variables of the *same* type,
  - taking the address (&) of a structure variable,
  - accessing the members of a structure variable,
  - using the `sizeof` operator to determine the size of a structure variable.

Common Programming Error 10.3

Assigning a structure of one type to a structure of a different type is a compilation error.

10.3 Initializing Structures

- Structures may not be compared using operators "==" and "!=".

- Structures can be initialized using **initializer lists** as with arrays.
- To initialize a structure, follow the variable name in the definition with an equal sign and a brace-enclosed, comma-separated list of initializers.

10.3 Initializing Structures

- Initialization using Initializer List

```
struct structure_name str = { value1, value2, value3 };
```

10.3 Initializing Structures

- For example, the declaration
  - `struct card aCard = {"Three", "Hearts"};`

  creates variable `aCard` to be of type `struct card` (as defined in Section 10.2) and initializes member `face` to `"Three"` and member `suit` to `"Hearts"`.

10.3 Initializing Structures (Cont.)

- **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)**.

- Structure variables may also be initialized in assignment statements by assigning a structure variable of the *same* type, or by assigning values to the *individual* members of the structure.

10.3 Initializing Structures

```
struct Point
{
   int x = 0;  // COMPILER ERROR:  cannot initialize members here
   int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

10.4 Accessing Structure Members

- Two operators are used to access members of structures:
  - the structure member operator (`.`)—a.k.a. the dot operator
  - and the structure pointer operator (`->`)—a.k.a. the arrow operator
- The structure member operator accesses a structure member via the structure variable name.
- For example, to print member `suit` of structure variable `aCard` defined in Section 10.3, use the statement
  - `printf("%s", aCard.suit); // displays Hearts`

10.4 Accessing Structure Members (Cont.)

- The structure pointer operator accesses a structure member via a pointer to the structure.
- Assume that the pointer `cardPtr` has been declared to point to `struct card` and that the address of structure `aCard` has been assigned to `cardPtr`.
  - `struct card *cardPtr = &aCard;`
- To print member `suit` of structure `aCard` with pointer `cardPtr`, use the statement
  - `printf("%s", cardPtr->suit); // displays Hearts`

10.4 Accessing Structure Members (Cont.)

- The expression `cardPtr->suit` is equivalent to `(*cardPtr).suit`, which dereferences the pointer and accesses the member `suit` using the structure member operator.

- **The parentheses are needed** here because the structure member operator **(`.`) has a higher precedence** than the pointer dereferencing operator (*).

Common Programming Error 10.4

Inserting space between the - and > components of the structure pointer operator is a syntax error.

Common Programming Error 10.5

Attempting to refer to a structure member by using only the member's name is a syntax error.

Common Programming Error 10.6

Not using parentheses when referring to a structure member that uses a pointer and the structure member operator (e.g., `*cardPtr.suit`*) is a syntax error. To prevent this problem use the arrow (->) operator instead.*

10.4 Accessing Structure Members (Cont.)

- The program of Fig. 10.2 demonstrates the use of the structure member and structure pointer operators.
- Using the structure member operator, the members of structure `aCard` are assigned the values `"Ace"` and `"Spades"`, respectively
- Pointer `cardPtr` is assigned the address of structure `aCard`
- Function `printf` prints the members of structure variable `aCard` using the structure member operator with variable name `aCard`, the structure pointer operator with pointer `cardPtr` and the structure member operator with dereferenced pointer `cardPtr`

```c
// Fig. 10.2: fig10_02.c
// Structure member operator and
// structure pointer operator
#include <stdio.h>

// card structure definition
struct card {
   char *face; // define pointer face
   char *suit; // define pointer suit
};

int main(void)
{
   struct card aCard; // define one struct card variable

   // place strings into aCard
   aCard.face = "Ace";
   aCard.suit = "Spades";

   struct card *cardPtr = &aCard; // assign address of aCard to cardPtr

```

Fig. 10.2 | Structure member operator and structure pointer operator. (Part 1 of 2.)

```
    printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
       cardPtr->face, " of ", cardPtr->suit,
       (*cardPtr).face, " of ", (*cardPtr).suit);
}
```

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

Fig. 10.2 | Structure member operator and structure pointer operator. (Part 2 of 2.)

10.5 Using Structures with Functions

- **Structures may be passed to functions**
  - by passing individual structure members,
  - by passing an entire structure,
  - by passing a pointer to a structure.
- When individual structure members or an entire structure is passed to a function, they're passed by value.
- Therefore, the members of a caller's structure cannot be modified by the called function.
- **To pass a structure by reference, pass the address** of the structure variable.

10.5 Using Structures with Functions (Cont.)

- **Arrays of structures—like all other arrays—are automatically passed by reference.**
- To pass an array by value, create a structure with the array as a member.
- Structures are passed by value, so the array is passed by value.

Common Programming Error 10.7

Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller's structure values in the called function is a logic error.

Performance Tip 10.1

Passing structures by reference is more efficient than passing structures by value (which requires the entire structure to be copied).

```
#include <stdio.h>

//struct definition
struct POINT{
   int x;
   int y;
};

int main()
{
   struct POINT p; //creating a struct with the type of POINT
   //Values are assigned to the variables of p
   p.x=234;
   p.y=987;
   printf("x=%d\ny=%d", p.x, p.y);
   return 0;
}
```

```
x=234
y=987
```

```c
#include <stdio.h>
#include <string.h>
struct book{
  char title[10];
  double price;
  int pages;
} b1; //creating a struct with the type of book

int main (){
  strcpy(b1.title, "Learn C"); //or sprintf(b1.title, "Learn C");
  b1.price = 675.50;
  b1.pages = 325;
  printf("Title: %s\n", b1.title);
  printf("Price: %lf\n", b1.price);
  printf("No of Pages: %d\n", b1.pages);
  return 0;
}
```

Output

```
Title: Learn C
Price: 675.500000
No of Pages: 325
```

```c
#include <stdio.h>

struct Person {
  char name[50];
  int citNo;
  float salary;
} person1;

int main() {
  // assign value to name of person1
  sprintf(person1.name, "George Orwell");
  // assign values to other person1 variables
  person1.citNo = 1984;
  person1. salary = 2500;
  // print struct variables
  printf("Name: %s\n", person1.name);
  printf("Citizenship No.: %d\n", person1.citNo);
  printf("Salary: %.2f", person1.salary);
  return 0;
}
```

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

```
#include <stdio.h>
struct str1 {
        int i;
        char c;
        float f;
        char s[30];
};
struct str2 {
        int ii;
        char cc;
        float ff;
};

int main()
{
        struct str1 var1 = { 1, 'A', 1.00, "GeeksforGeeks" }, var2;
        struct str2 var3 = { .ff = 5.00, .ii = 5, .cc = 'a' };
        var2 = var1;
        printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
                var1.i, var1.c, var1.f, var1.s);
        printf("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",
                var2.i, var2.c, var2.f, var2.s);
        printf("Struct 3\n\ti = %d, c = %c, f = %f\n", var3.ii,
                var3.cc, var3.ff);
        return 0;
}
```

Output

```
Struct 1:
    i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 2:
    i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 3
    i = 5, c = a, f = 5.000000
```

10.6 `typedef`

- The keyword `typedef` provides a mechanism for creating **synonyms (or aliases)** for previously defined data types.
- Names for structure types are often defined with `typedef` to create shorter type names.

10.6 `typedef`

- The following statement
  - `typedef struct card Card;`

  defines the new type name **`Card`** as a synonym for type **`struct card`**.

- The following definition
  - ```
    typedef struct card{
        char *face;
        char *suit;
    } Card;
    ```

  creates the structure type `Card` without the need for a separate `typedef` statement.
    ```

10.6 typedef (Cont.)

- `Card` can now be used to declare variables of type `struct card`.
- The declaration
  - `Card deck[52]; //equivalent to "struct card deck[52]"`

  declares an array of 52 `Card` structures (i.e., variables of type `struct card`).
- Creating a new name with `typedef` does *not* create a new type; `typedef` simply creates a new type name, which may be used as an alias for an existing type name.

Good Programming Practice 10.3

Capitalize the first letter of `typedef` *names to emphasize that they're synonyms for other type names.*

Good Programming Practice 10.4

Using `typedef`*s can help make a program more readable and maintainable.*

```c
#include <stdio.h>
#include <string.h>

typedef struct Person {
  char name[50];
  int citNo;
  float salary;
} person;

int main() {
  person p1;
  strcpy(p1.name, "George Orwell");
  p1.citNo = 1984;
  p1. salary = 2500;
  printf("Name: %s\n", p1.name);
  printf("Citizenship No.: %d\n", p1.citNo);
  printf("Salary: %.2f", p1.salary);
  return 0;
}
```

Output:

Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00

```c
#include <stdio.h>
// defining structure
struct str1 {
   int a;
};
// defining new name for str1
typedef struct str1 strname;

// another way of using typedef with structures
typedef struct str2 {
   int x;
} str2;

int main()
{
   // creating structure variables using new names
   strname var1 = { 20 };
   str2 var2 = { 314 };
   printf("var1.a = %d\n", var1.a);
   printf("var2.x = %d", var2.x);
   return 0;
}
```

Output

```
var1.a = 20
var2.x = 314
```

10.7 Example: High-Performance Card Shuffling and Dealing Simulation

- The program in Fig. 10.3 represents the deck of cards as an array of structures and uses high-performance shuffling and dealing algorithms.
- The program output is shown in Fig. 10.4.

```c
// Fig. 10.3: fig10_03.c
// Card shuffling and dealing program using structures
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CARDS 52
#define FACES 13

// card structure definition
struct card {
   const char *face; // define pointer face
   const char *suit; // define pointer suit
};

typedef struct card Card; // new type name for struct card

// prototypes
void fillDeck(Card * const wDeck, const char * wFace[],
   const char * wSuit[]);
void shuffle(Card * const wDeck);
void deal(const Card * const wDeck);

```

Fig. 10.3 | Card shuffling and dealing program using structures. (Part 1 of 4.)

```
int main(void)
{
   Card deck[CARDS]; // define array of Cards

   // initialize array of pointers
   const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
      "Six", "Seven", "Eight", "Nine", "Ten",
      "Jack", "Queen", "King"};

   // initialize array of pointers
   const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};

   srand(time(NULL)); // randomize

   fillDeck(deck, face, suit); // load the deck with Cards
   shuffle(deck); // put Cards in random order
   deal(deck); // deal all 52 Cards
}

```

Fig. 10.3 | Card shuffling and dealing program using structures. (Part 2 of 4.)

```
// place strings into Card structures
void fillDeck(Card * const wDeck, const char * wFace[],
   const char * wSuit[])
{
   // loop through wDeck
   for (size_t i = 0; i < CARDS; ++i) {
      wDeck[i].face = wFace[i % FACES];
      wDeck[i].suit = wSuit[i / FACES];
   }
}

// shuffle cards
void shuffle(Card * const wDeck)
{
   // loop through wDeck randomly swapping Cards
   for (size_t i = 0; i < CARDS; ++i) {
      size_t j = rand() % CARDS;
      Card temp = wDeck[i];
      wDeck[i] = wDeck[j];
      wDeck[j] = temp;
   }
}
```

Fig. 10.3 | Card shuffling and dealing program using structures. (Part 3 of 4.)

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

Fig. 10.3 | Card shuffling and dealing program using structures. (Part 4 of 4.)

```
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
```

Fig. 10.4 | Output for the high-performance card shuffling and dealing simulation.

10.7 Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- In the program, function `fillDeck` initializes the `Card` array in order with "Ace" through "King" of each suit.
- The `Card` array is passed to function `shuffle`, where the high-performance shuffling algorithm is implemented.
- Function `shuffle` takes an array of `52` `Cards` as an argument.
- The function loops through the `52` `Cards`.

10.7 Example: High-Performance Card Shuffling and Dealing Simulation (Cont.)

- For each card, a number between 0 and 51 is picked randomly.
- Next, the current `Card` and the randomly selected `Card` are swapped in the array
- A total of 52 swaps are made in a single pass of the entire array, and the array of `Card`s is shuffled!