Lecture 6:
C Structure, Union, and Enumeration

Reference: Deitel, C How to Program, 8/e

I.1 Definition of Structure

- Structures: are collections of a **fixed number** of **related variables under one name.**
  - They are commonly used to define *records*
  - Variables may be of **many different data types**, in contrast to arrays, which contain *only* elements of the **same data type.**

- Structures are derived data types, they're constructed using objects of other data types

I.2 Syntax of C-Structure Definition

❖ Common Syntax:

```
struct <struct_Tag> {
   datatype1 variableName1;
   datatype2 variableName2;
…..
};
```

- Each **structure definition *must* end with a semicolon.**
- Keyword `struct` introduces the structure definition.
- The identifier **`<struct_Tag>`** is the structure tag, which **names** the *structure definition* and is used with **`struct`** keyword to **declare variables** of the **structure type**

❖ Structure definitions **do *not* reserve any space in memory**; rather, each definition **creates a new data type** that's used to define variables.

...Continued

❖ **Example:**

```
struct card {
    char face [10] ;
    char suit [10] ;
};
```

- The definition of `struct card` contains **members** `face` and `suit`, each of type `array of chars`.

❖ The definition

- `struct card MyCard, deck[52];`

declares:

- MyCard to be a variable of type **struct card**, and
- deck to be an **array** with 52 elements of type **struct card**.

...Continued

- 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:

```
struct card {
        char face [10] ;
        char suit [10] ;
        } MyCard, deck[52];
```

- In this instance, the *structure tag* name is **optional.**
  - However, 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.

```
struct {
        char face [10] ;
        char suit [10];
      } aCard, deck[52];
```

Use of "typedef" to create a Struct Name

- The keyword `typedef` provides a mechanism for creating **synonyms** (or aliases) for **previously defined data types.**
- **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.*

- Names for structure types are often defined with `typedef` to create **shorter type names.**
- For example, the statement
  - `typedef struct card card_t;`

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

...Continued

- C programmers often use `typedef` to define a structure type, so a **structure tag** is not required.
  - For example, the following definition:

```
typedef struct {
    char face[10];
    char suit[10];
} card_t;
```

Recommended definition

creates the structure type `card_t` without the need for a separate `typedef` statement and structure definition like in the previous slide

- The declaration

```
card_t deck[52];
```

declares an array of 52 `card_t` structures (i.e., variables of type `struct card_t`).

Summary : struct definition

```
struct card {
    char face [10] ;
    char suit [10] ;
};
 struct card MyCard;
```

①

```
struct card {
   char face [10] ;
   char suit [10] ;
   } MyCard;
```

②

```
struct {
        char face [10] ;
        char suit [10];
    } MyCard;
```

③

```
typedef struct card card_t;
struct card_t MyCard;
```

④

```
typedef struct {
    char face[10];
    char suit[10];
} card_t;
card_t MyCard;
```

⑤

Recommended definition

...Continued

- **Variables declared** within the braces of the *structure definition* are the structure's members.
- **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 **members** can be variables of the *primitive data types* (e.g., `int`, `float`, etc.), or collection such as arrays and other *structures.*
  - **Example:**

```
struct employee {
        char firstName[20];
        char lastName[20];
        unsigned int age;
        char gender;   // 'M' or 'F'
        double hourlySalary;
};
```

...Continued

- **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.
    -
```
struct employee {
    char firstName[20];
    char lastName[20];
    unsigned int age;
    char gender;
    double hourlySalary;
    struct employee person; // ERROR

};
```

- However, a **pointer** to the same struct (employee in the above), may be included (more later)

Example

```
1.  /*
2.   * Displays with labels all components of a planet_t structure
3.   */
4.  void
5.  print_planet(planet_t pl) /* input - one planet structure */
6.  {
7.        printf("%s\n", pl.name);
8.        printf("  Equatorial diameter: %.0f km\n", pl.diameter);
9.        printf("  Number of moons: %d\n", pl.moons);
10.       printf("  Time to complete one orbit of the sun: %.2f years\n",
11.              pl.orbit_time);
12.       printf("  Time to complete one rotation on axis: %.4f hours\n",
13.              pl.rotation_time);
14. }
```

```
#define STRSIZ 10

typedef struct {
      char   name[STRSIZ];
      double diameter;            /* equatorial diameter in km     */
      int    moons;               /* number of moons               */
      double orbit_time,          /* years to orbit sun once       */
             rotation_time;       /* hours to complete one
                                       revolution on axis          */
} planet_t;
```

…Continued

- Name: Jupiter
- Diameter: 142,800 km
- Moons: 16
- Orbit time: 11.9 years
- Rotation time: 9.925 hours

`planet_t blank_planet`

Variable `blank_planet`, a structure of type `planet_t`

`.name`	`\0 ? ? ? ? ? ? ? ? ?`
`.diameter`	0.0
`.moons`	0
`.orbit_time`	0.0
`.rotation_time`	0.0

```
#define STRSIZ 10

typedef struct {
        char   name[STRSIZ];
        double diameter;            /* equatorial diameter in km     */
        int    moons;               /* number of moons               */
        double orbit_time,          /* years to orbit sun once       */
               rotation_time;       /* hours to complete one
                                       revolution on axis            */
} planet_t;
```

I.3 Initialization of Structures

- Structures can be initialized using **initializer lists** as with arrays.
  - To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers.
    - For example, the declaration
      - `card_t MyCard = {"Three", "Hearts"};`

      creates variable `MyCard` to be of type `struct card`, and initializes member `face` to "`Three`"' and member `suit` to "`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).

...Continued

- Structure variables may also be initialized in **assignment statements** by:
  - assigning a **structure variable of the *same* type**, or
  - **assigning values to the *individual* members of the structure:**
    - Two operators are used to access members of a structure: the structure member operator (`.`)—also called the **dot operator**—and the structure pointer operator (`->`)—also called the arrow operator.
    - The structure *member operator* (`.`) accesses a structure member via its structure variable name.
      - For example, to print member `suit` of structure variable `MyCard`, use the statement

```
» card_t MyCard;
  printf("%s", MyCard.suit); // displays Hearts
```

Examples

Example1:

```
typedef struct
{   int resistance;
    int tolerance;
} resistor_t;           /* a new structured data type */

resistor_t r1, r2;      /* variables of type resistor */

r1.resistance = 470;        /* 470 ohms */
r1.tolerance = 5;           /* 5% */
r2.resistance = r1.resistance;
```

member access through the struct variable name and dot operator

Example2:

```
typedef struct
{   int resistance;
    int tolerance;
} resistor_t;

Resistor_t r1={4.7, 5}, r2;
                        /* assign one structure variable to another */
```

Initializer or struct to struct assignment

r2 = r1; ⟺

- `r2.resistance = r1.resistance;`
- `r2.tolerance = r1.tolerance;`

I.4 Structures and Functions

- Structures may be passed to functions by:
  - individual structure members, or
  - an entire structure, or
  - a pointer to a structure (more later)

- When structures or individual structure members are 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 (more later)

...Continued

- **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.

Struct as a Function return data type

- A **struct** variable can be passed as a parameter to a function
- A function **can return a value** of the type struct

```
/* type definition */

   typedef struct
   {
      int   day;
      int   month;
      int   year;
   } date_t;
```

data type of a function argument

```
/* function prototypes */
 void setDate( date_t theDate );     /* OK */
date_t getDate ( int projNumber ); /* OK */
```

return data type of a function

I.4. Operations on Structures

- The only valid operations that may be performed on structure variables 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, and
  - using the `sizeof` operator to determine the size of a structure variable.

- Structure variables ***should not*** **be compared using relational operators**, such as == and > because structure members are not necessarily stored in consecutive bytes of memory.
  - Sometimes there are "holes" in a structure, because computers may store specific data types only on certain memory boundaries such as half-word, word or double-word boundaries.

Comparison of Structures

- `struct` variables can be compared member-wise only

```
resistor_t r1  ={4.7, 10}, r2 ={4.7, 10};
. . . . . .    . .
   if(r1 == r2); ☒ compiler error!!
```

- To compare the values of r1 and r2, you must compare them member-wise, as follows:

```
if( r1.value==r2.value &&  r1.tolerance==r2.tolerance )
{
    . . . .
}
```

- *Example:*

```
typedef struct
{
   int month;
   int day;
}timestmp;
```

```
bool compareTimeStamps(timestmp time1, timestmp time2 );
.   .   .   .   .
/* function definition */
bool compareTimeStamps(timestmp time1, timestmp time2 )
{
   if( (time1.month != time2.month )
     || ( time1.day != time2.day ) ) return (false);
   else return (true);
}
```

Arrays of Structures

- You can declare an array of structures

```
typedef struct
{
    float x;
    float y;
}coordinate_t;

coordinate_t polygon[5];
```

`polygon` is described by 5 coordinates

Each element of this array is a structure of type `coordinate`

- You can access any structure member specifying the array index and using the member access operator – the dot

```
polygon[1].y = 12.3;
```

array name

Index of the array element

Member access operator

struct member name

Array of Structures vs. Structure of Arrays

Caution: Do not confuse **arrays made of structures** with **structures made of arrays**

```
typedef struct {
  int day;
  float mark[50];
} exam_t;                    /* an array (with 50 elements) is a structure member */

exam_t examReport;
                             /* a struct variable */
examReport.mark[3] = 76.43;
```

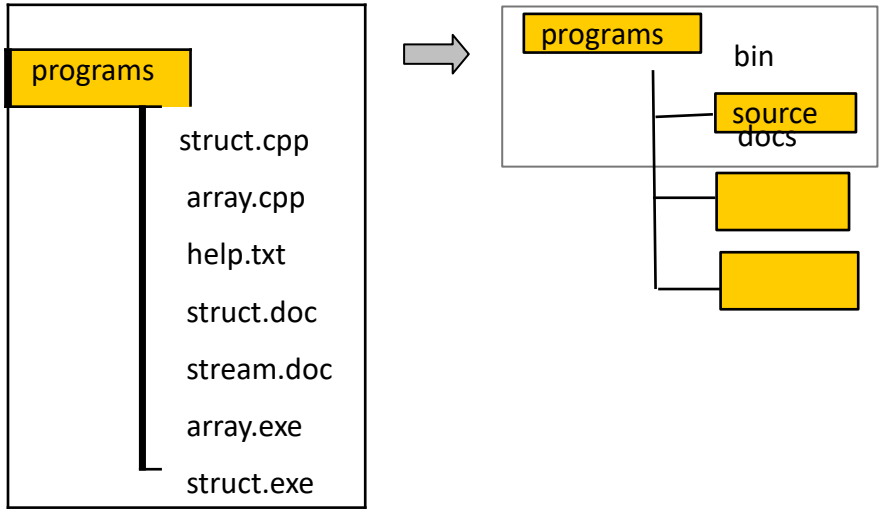
```
typedef struct{
  int day;
  double mark;
} exam_t;

exam_t examReport[50];    /* an array (with 50 elements) made of structures */

examReport[3].mark = 76.43;
```

I.5. Nested Structures

Consider the following record of an employee as an example:

```
typedef struct
{
   char first[20];
   char middle[20];
   char last[20];
   char street[40];
   char city[20];
   int  zip;
   char day;
   char month;
   char year;
   int  phone;
   int  cellphone;
   int  fax;
   int  deptID;
   float
        salary;
} employee;
```

It may be more convenient, if all structure members were divided into subcategories as it is usually done with files and folders



…Continued

```
typedef struct
{
    char first[20];
    char middle[20];
    char last[20];
    char  street[40];
    char  city[20];
    int   zip;
    char  day;
    char  month;
    char  year;
    int   phone;
    int   cellphone;
    int   fax;
    int   deptID;
    float  salary;
} employee_t;
```

```
/* members describing a name */
typedef struct
{
    char first[20];
    char middle[20];
    char last[20];
} name_t;
```

```
/*members describing an address*/
typedef struct
{
    char  street[40];
    char  city[20];
    int   zip;
} address_t;
```

```
/*members describing contact info */
typedef struct
{
    int   phone;
    int   cellphone;
    int   fax;
} contact_t;
```

…Continued

```
typedef struct    /* a top-level structure */
{
      name_t      theName;
      address_t   homeAddr;
      date_t      hireDate;
      contact_t   contactInfo;
      int         deptID;
      float       salary;
}employee_t;

  /* variable declarations */
  employee_t newEmployee;
  employee_t uowEmployees[2000];
```

- Well structured representation
- Some sub-structures can be reused to build other high level structures

Accessing Nested Structures

- You need to specify all levels of hierarchy starting from the top level

```
newEmployee.depID = 371;

newEmployee.hireDate.month = 2;
newEmployee.hireDate.year = 2004;
printf("Last name: %s", newEmployee.theName.last);

uowEmployees[3].hireDate.year = 2009;
uowEmployees[3].hireDate.month = 3;
```

```
typedef struct   /* a top-level structure */
{
    name_t      theName;
    address_t   homeAddr;
    date_t      hireDate;
    contact_t   contactInfo;
    int         deptID;
    float       salary;
}employee_t;
```

```
/* variable declarations */
employee_t newEmployee;
employee_t uowEmployees[2000];
```

```
newEmployee.homeAddr.zip = 2522;  /* OK */
newEmployee.address_t.zip = 2522;/*Error */
```

`address` is not a member, this is a data type and it must not be used here

```
typedef struct
{
char day;
char month;
char year;
} date_t;
```

II. Union

- A union is a collection of a **fixed number** of related variables **under one name** that **share the same storage space in the memory**
- A union is a ***derived data type***—like a structure—with members that ***share the same storage space*** **in the memory**
  - 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.

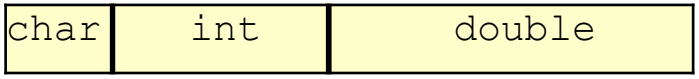
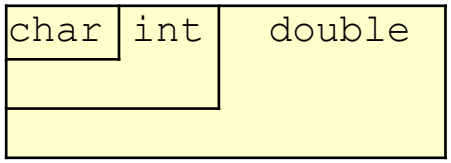
...Continued

- The **number of bytes** used to store a **union** must **be at least enough to hold the *largest* member.**
- In most cases, unions contain two or more data types.
  - **Only one member, and thus one data type, can be referenced at a time.**
  - It's your responsibility to ensure that the data in a **union** is referenced with the proper **data type.**

Union Declarations

- A **union** definition has the same format as a **structure** definition.
- Variable of type **union** can be declared as with **struct**
  - A `union` definition
    - ```
      union number {
          int x;
          double y;
      };
      ```

    indicates that `number` is a **union** type with members `int x` and `double y`.
  - Better to use **typedef:**

```
typedef union {
        int x;
     double y;
   } number_t;
```
```

Union vs. Struct

stuct Memory Allocation

```
typedef struct
{
    char   type1;
    int    type2;
    double type3;
} newType_t;
newType_t strt;
```



A block of memory allocated for a struct can store **ALL** members of the struct

union Memory Allocation

```
typedef union
{
    char   type1;
    int    type2;
    double type3;
} newType_t;
newType_t uni;
```



A block of memory allocated for a union can store **only one member at a time**

Operations on Unions

- 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,
  - accessing **union** members using the s**tructure/union member operator (.)** and the **structure/union pointer operator(->)**

- **Unions should not be compared** for the same reasons that structures cannot be compared.

Initializing Unions in Declarations

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

- For example, with the **union** number,

```
typedef union {
    int x;
 double y;
  } number_t
```

  - the statement: `number_t value = {10};`

is a valid initialization of union variable `value` because the **union** is initialized with an `int`

  - The statement: `number_t value = {1.43};`

would truncate the floating-point part of the initializer value and ideally would produce a warning from the compiler

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);
```

Fig. 10.5 | Displaying the value of a union in both member data types. (Part 1 of 2.)

...Continued

```
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:
  -92559592117433136000000000000000000000000000000000000000000000000000.000000


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

double:
  100.000000
```

Fig. 10.5 | Displaying the value of a union in both member data types. (Part 2 of 2.)

Example 2

- Function displaying a Struct with a Union type component

```
typedef struct {
      int    bald;
      hair_t h;
} hair_info_t;
```

```
typedef union {
      int   wears_wig;
      char  color[20];
} hair_t;
```

```
1.  void
2.  print_hair_info(hair_info_t hair) /* input - structure to display          */
3.  {
4.        if (hair.bald) {
5.            printf("Subject is bald");
6.            if (hair.h.wears_wig)
7.                  printf(", but wears a wig.\n");
8.            else
9.                  printf(" and does not wear a wig.\n");
10.       } else {
11.           printf("Subject's hair color is %s.\n", hair.h.color);
12.       }
13. }
```

III. Enumeration Type: Motivation

- Assume that you need to implement an *inventory management program* for your *store of electronic components.*
  - There are three *major types of capacitors*: *electrolytic*, *film* and *ceramic*.
    - What data type would you choose for variables that describe the *capacitor type* property?

1. `char` – only one character: '`e`' for electrolytic, '`f`' for film, '`c`' for ceramic
2. `int` – a whole number: `1` for electrolytic, `2` for film, `3` for ceramic
3. `float` …

- Although it is possible to use one of the basic data types for this application
  - the program may look confusing and hard for debugging and maintenance
  - wrong values ('v', 5, . . .) may be assigned by mistake **without any warnings or error messages**

Syntax of Enumeration Type

- C provides a method of defining a **data type with a limited number of values that have specific names**; it is called **enumeration**
- An **enumeration** is a data type that consists of a **set of named values** that **represent integral constants**, known as **enumeration constant**
- General syntax:

```
enum <enum_tag> {value1, value2,..., valueN};
```

- Variable of type **enumeration** can be declared as with **Struct** and **Union.**
- In the following, we use the **typedef** to create an **alias** to an **enumeration** data type

General syntax:

```
typedef enum {value1, value2,..., valueN} TypeName;
```

Example:

```
typedef enum {ELECTROLYTIC, FILM, CERAMIC } capType_t;
```

enumerators **data type**

Enumeration Type

- Once a data type is defined, you can declare its variables and process its values from the enumeration list (**enumerators**)

Example:

```
/* new data type definition */
typedef enum { RED, GREEN, BLUE, WHITE, BLACK } color_t;

int main(void)
{
   /* variable declaration */
   color_t primaryColor, secondColor;
   .  .  .  .  .  .  .  .

   /* assigning a value */
   secondColor = BLUE;

   primaryColor = YELLOW;
```

Error !

Enumerators

- Enumerators must have valid identifiers

Example of valid enumerators:

```
typedef enum  { first, second, third, fourth    }places_t;
typedef enum  {A, B, C, D, F} grades_t;
```

Example of illegal enumerators:

```
typedef enum  {1st, 2nd, 3rd,   4th} places_t;
typedef enum  {'A', 'B', 'C',   'D', 'F'} grades_t;
```

- If an identifier has been used in one enumeration type, it cannot be used **by any other enumeration type or a variable within the scope**

```
typedef enum {John, Bill, Peter, William} mname_t;
typedef enum {Susan, Ann, Lisa, Joan} fname_t;
```

Illegal

```
int Ann;
```

Enumerators Representation

- C compiler assigns an integer value to each enumerator

Example:

```
typedef enum { LOW, HIGH, ZSTATE, UNDEF } signal_t;
```

By default it starts with 0

```
LOW = 0, HIGH = 1, ..., UNDEF = 3
```

- Although it is not common, you can specify your own starting value

Example:

```
typedef enum {Sun =1,Mon,Tue,Wed,Thr,Fri,Sat} weekday_t;
```

```
Sun = 1, Mon = 2, ..., Sat = 7
```

Arithmetic Operations

- Arithmetic operations are allowed on the enum type variable

Example:

```
typedef enum { RED, GREEN, BLUE, WHITE, BLACK } colour;

colour newColor = RED, primaryColor;
primaryColor =     newColor   - 1;
primaryColor =     newColor   + RED;
primaryColor++;
primaryColor -= 2;
```

Relational Operations

- You can use the relational operators with of the same the `enum` types

Example:

```
typedef enum{ RED, GREEN, BLUE, WHITE, BLACK } colour;

colour baseColor = RED, newColor;
if( baseColour < BLUE ) newColour = RED;   /* OK */
if( newColour <= GREEN ) newColour = RED; /* OK */
while(newColour > baseColour) { . . . }    /* OK */
```

- `enum` type values are compared based upon integer values assigned to the enumerators (their positions in the list)

…Continued

- You can compare variables of different `enum` types

```
typedef enum {RED,GREEN, YELLOW} apple_t;
typedef enum{AUSTRALIAN,AFRICAN, AMERICAN} orange_t;
int main(void){
apple_t anApple = RED;
orange_t anOrange = AUSTRALIAN;
if( anApple == anOrange )
   anApple = GREEN;
printf("%d", anApple);
return 0;
}
```

The printed value is 1 which corresponds to GREEN

Enumeration and Switch

- You can use the enumeration type with the `switch` multiple selection statement

```
typedef enum { RED, GREEN, BLUE, BLACK } palette_t;

palette_t newColor;
int colorIntensity;
.   .   .   .
```

enum type selector

```
switch( newColor )
{
   case RED   : colorIntensity= 25;
                break;
   case GREEN : colorIntensity= 40;
                break;
   case BLUE  : colorIntensity= 73;
                break;
   default    : colorIntensity= 0;
}
```

Read and Write

- Input/output operations with `enum` type variables are possible, but the variables are treated as integers.

- **`printf()`** and **`scanf()`** support only fundamental data types

```
typedef enum { RED, GREEN, BLUE, WHITE } colour_t;


colour_t newColor = GREEN;
printf("%d", newColor ); /*The output will be:   1   */
scanf("%d", &newColor); /* You can input only a
                           corresponding integer value */
```

Quiz

- Would you use the enumerated type for variables that describe:
  - voltage ranging from 0 to 5.5 volts No
  - A number of electronic components used in a module Yes
  - resolution of digital video that can be only one of the following modes: QQCIF, QCIF or CIF Yes
  - transistor package type: TO1, TO3, TO202, TOP3, SOT25 or SOT30 Yes

Appendix: Anonymous Structures and Unions

- C11 supports anonymous `structs` and `unions` that can be nested in named `structs` and `unions`
- Members in a nested anonymous `struct` and `union` are considered to be members of the enclosing type and can be accessed directly through an object of the enclosing type
- Example
  - ```
    struct MyStruct {
        int member1;
        int member2;

        struct {
            int nestedMember1;
            int nestedMember2;
        };
    };
    ```
  - For a variable `myStruct` of type `struct MyStruct`, you can access the members as:
    - `myStruct.member1`
      `myStruct.member2`
      `myStruct.nestedMember1`
      `myStruct.nestedMember2`
    ```

Appendix

- Because of *boundary alignment* requirements, the size of a `struct` variable is not necessarily the sum of its members' sizes. Always use `sizeof` to determine the number of bytes in a `struct` variable.

- `struct` variables cannot be compared for equality or inequality, because they might contain bytes of undefined data. Therefore, you must compare their individual members.