SDN150S – SOFTWARE DESIGN 1

Structures, Unions, Bit Field Manipulation and Typedef

DEPARTMENT OF ELECTRICAL ELECTRONIC AND COMPUTER ENGINEERING
CAPE PENINSULA UNIVERSITY OF TECHNOLOGY

Cape Peninsula University of Technology

STRUCTURES

- The structure can be defined as a collection of dissimilar and/or related data items under one name.
- Structure in C is used for defining user-defined data types.
- Apart from the primitive data type such as integer, float, double, etc., with structure we can also define our own data type depending on our own requirements.
- Similarly to arrays, structures also allows for the combination of data items of different kinds.
- In essence, structures can be used to represent a record.

DEFINING A STRUCTURE

- To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows:

```
struct [structure tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

- The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.
- At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.

ACCESSING STRUCTURE MEMBERS (1 OF 2)

- To access any member of a structure, we use the dot (.) operator
- The dot operator is coded as a period between the structure variable name and the structure member that we wish to access.
- Here is an example of how to access a structure in a program: The program finds the area of a rectangle having two properties i.e. length and breadth.

```
#include <stdio.h>
struct Rectangle // structure definition
{
    int length, breadth; // variable declaration
};
int main()
{
    struct Rectangle r; // struct variable declaration
    r.length = 45; r.breadth = 23;
    printf ("Area of Rectangle: %d", r.length * r.breadth);
    return 0;
}
```

OUTPUT

```
Area of Rectangle: 1035
```

ACCESSING STRUCTURE MEMBERS (2 OF 2)

- The example below shows how to access a book structure in a program:

```
1. #include <stdio.h>
2. #include <string.h>
3. struct Books
4. {
5. char title[50]; char author[50];
6. char subject[100]; int book_id;
7. };
8. int main( )
9. {
10.struct Books Book1; /* Declare Book1 of type Book */
11.struct Books Book2; /* Declare Book2 of type Book */
12./* book 1 specification */
13.strcpy( Book1.title, "C Programming");
14.strcpy( Book1.author, "Nuha Ali");
15.strcpy( Book1.subject, "C Programming Tutorial");
16.Book1.book_id = 6495407;
17./* book 2 specification */
18.strcpy( Book2.title, "Telecom Billing");
19.strcpy( Book2.author, "Zara Ali");
20.strcpy( Book2.subject, "Telecom Billing Tutorial");
21.Book2.book_id = 6495700;
21.Book2.book_id = 6495700;
22./* print Book1 info */
23.printf( "Book 1 title : %s\n", Book1.title);
24.printf( "Book 1 author : %s\n", Book1.author);
25.printf( "Book 1 subject : %s\n", Book1.subject);
26.printf( "Book 1 book_id : %d\n", Book1.book_id);
27./* print Book2 info */
28.printf( "Book 2 title : %s\n", Book2.title);
29.printf( "Book 2 author : %s\n", Book2.author);
30.printf( "Book 2 subject : %s\n", Book2.subject);
31.printf( "Book 2 book_id : %d\n", Book2.book_id);
32.return 0;
33.}
```

OUTPUT

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

STRUCTURE AS FUNCTION ARGUMENTS

- You can pass a structure as a function argument in the same way you pass any other variable or pointer.

```
1. #include <stdio.h>
2. #include <string.h>
3. struct Books
4. {
5. char title[50]; char author[50];
6. char subject[100]; int book_id;
7. };
8. /* function declaration */
9. void printBook( struct Books book );
10.int main( )
11.{
12.struct Books Book1; /* Declare Book1 of type Book */
13.struct Books Book2; /* Declare Book2 of type Book */
14./* book 1 specification */
15.strcpy( Book1.title, "C Programming");
16.strcpy( Book1.author, "Nuha Ali");
17.strcpy( Book1.subject, "C Programming Tutorial");
18.Book1.book_id = 6495407;
19./* book 2 specification */
20.strcpy( Book2.title, "Telecom Billing");
21.strcpy( Book2.author, "Zara Ali");
22.strcpy( Book2.subject, "Telecom Billing Tutorial"
   );
23.Book2.book_id = 6495700;
24./* print Book1 info */
25.printBook( Book1 );
26./* Print Book2 info */
27.printBook( Book2 );
28.return 0;
29.}
30.void printBook( struct Books book )
31.{
32.printf( "Book title : %s\n", book.title);
33.printf( "Book author : %s\n", book.author);
34.printf( "Book subject : %s\n", book.subject);
35.printf( "Book book_id : %d\n", book.book_id);
36.}
```

OUTPUT

SAME AS LAST EXAMPLES

POINTER TO STRUCTURES (1 OF 2)

- You can define pointers to structures in the same way as you define pointer to any other variable:

```
struct Books *struct_pointer;
```

- Now, you can store the address of a structure variable in the above-defined pointer variable.
- However, to find the address of a structure variable, place the '&' operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

- Also, to access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

```
struct_pointer->title;
```

POINTER TO STRUCTURES (2 OF 2)

- Let consider the following example using structure pointer.

```
1. #include <stdio.h>
2. #include <string.h>
3. struct Books
4. {
5. char title[50];
6. char author[50];
7. char subject[100];
8. int book_id;
9. };
10./* function declaration */
11.void printBook( struct Books *book );
12.int main( )
13.{
14.struct Books Book1; /* Declare Book1 of type Book */
15.struct Books Book2; /* Declare Book2 of type Book */
16./* book 1 specification */
17.strcpy( Book1.title, "C Programming");
18.strcpy( Book1.author, "Nuha Ali");
19.strcpy( Book1.subject, "C Programming Tutorial");
20.Book1.book_id = 6495407;
21./* book 2 specification */
22.strcpy( Book2.title, "Telecom Billing");
23.strcpy( Book2.author, "Zara Ali");
24.strcpy( Book2.subject, "Telecom Billing Tutorial"
   );
25.Book2.book_id = 6495700;
26./* print Book1 info by passing address of Book1 *
   /
27.printBook( &Book1 );
28./* print Book2 info by passing address of Book2 *
   /
29.printBook( &Book2 );
30.return 0;
31.}
32.void printBook( struct Books *book )
33.{
34.printf( "Book title : %s\n", book->title);
35.printf( "Book author : %s\n", book->author);
36.printf( "Book subject : %s\n", book->subject);
37.printf( "Book book_id : %d\n", book->book_id);
38.}
```

OUTPUT

SAME AS LAST TWO EXAMPLES

UNION (1 OF 7)

- A union is a special data type available in C that allows to store different data types in the same memory location.
- You can define a union with many members, but only one member can contain a value at any given time.
- Unions provide an efficient way of using the same memory location for multiple purpose.
- To define a union, you must use the union statement in the same way as you did while defining a structure.
- The union statement defines a new data type with more than one member for your program.

UNION (2 OF 7)

- The syntax of the union statement is as follows:

```
union [union tag]
{
member definition;
member definition;
...
member definition;
} [one or more union variables];
```

- The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.
- At the end of the union's definition, before the final semicolon, you can specify one or more union variables, but it is optional.

UNION (3 OF 7)

- Here is the way you would define a union type named Data having three members i, f, and str:

```
union Data
{
int i;
float f;
char str[20];
} data;
```

- Now, a variable of Data type can store an integer, a floating-point number, or a string of characters.
- It means a single variable in the same memory location, can be used to store multiple types of data.
- You can use any built-in or user-defined data types inside a union based on your requirement.

UNION (4 OF 7)

- The memory occupied by a union will be large enough to hold the largest member of the union.
- In the example below, The union Data will occupy 20 bytes of memory space because that is the maximum space which can be occupied by a character string as specified in the code.
- The following example displays the total memory size occupied by the union:

```
#include <stdio.h>
#include <string.h>
union Data
     {int i;
     float f;
     char str[20]; };
int main( )
     {union Data data;
     printf( "Memory size occupied by data : %ld\n", sizeof(data));
     return 0;}
```

OUTPUT

Memory size occupied by data : 20

UNION (5 OF 7)

Accessing Union Members:

- Similar to structures, member of a union can be accessed using the dot operator (.).
- The keyword union is also used to define variables of union type, as shown in the example below:

```
#include <stdio.h>
#include <string.h>
union Data
    {int i; float f; char str[20];};
int main( )
    {union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
    return 0;}
```

OUTPUT

```
data.i : 1917853763
data.f :
4122360580327794860452759994368.000000
data.str : C Programming
```

*"Notice that the values of **i** and **f** members of union Data got corrupted due to occupying the same memory location with a str value member."*

UNION (6 OF 7)

- This is the same example, however, here we assign and print the values stored one member at a time, since all member share the same memory location:

```
#include <stdio.h>
#include <string.h>
union Data
    {int i; float f; char str[20];};
int main( )
    {union Data data;
    data.i = 10;
    printf( "data.i : %d\n", data.i);
    data.f = 220.5;
    printf( "data.f : %f\n", data.f);
    strcpy( data.str, "C Programming");
    printf( "data.str : %s\n", data.str);
    return 0;}
```

OUTPUT

data.i : 10
data.f : 220.500000
data.str : C Programming

UNION (7 OF 7)

Class Exercise

1. Write a C program that create a union with some members and then we print the size of the union and the value stored within them, one member at a time.
2. Write a C program that create a union with three members, the last of which should be a struct called ***StructJob***, whose members are the ***Name***, ***Salary***, and ***Identity number*** of a worker. Then print the values contained in all the members of the union.

Hint: Accessing a struct within a union use the dot operator (i.e., `union_tag.struct_tag.member_name`).

BIT FIELDS (1 OF 9)

- This allow the packing of data in a structure. It is especially useful when memory is limited or data storage is at a premium.
- Typical examples of bit field includes packing several objects into a machine word, e.g. 1 bit flags can be compacted or reading external non-standard file formats in, e.g., 9-bit integers.
- C allows us to use bit field in a structure definition by putting : the bit length after the variable.
- In the example below, the packed_struct contains 6 members: Four 1 bit flags (f1, f2, f3, & f4), a 4-bit type, and a 9-bit my_int variable.

```
struct packed_struct
{
unsigned int f1:1;
unsigned int f2:1;
unsigned int f3:1;
unsigned int f4:1;
unsigned int type:4;
unsigned int my_int:9;
} pack;
```

BIT FIELDS (2 OF 9)

- C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer.
- If this is not the case, then some compilers may allow memory overlap for the fields, while others would store the next field in the next word.

BIT FIELDS (3 OF 9)

- Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure as follows:

```
struct
{
unsigned int widthValidated;
unsigned int heightValidated;
} status;
```

- Although this structure requires 8 bytes of memory space, in actuality we are going to store either 0 or 1 in each of the variables (1 bit per variable).
- The C programming language offers a better way to utilize the memory space in such situations.

BIT FIELDS (4 OF 9)

- If you are using such variables inside a structure, then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes.
- For example, the structure can be rewritten as follows:

```
struct
{
unsigned int widthValidated : 1;
unsigned int heightValidated : 1;
} status;
```

- The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.
- If you will use up to 32 variables, each one with a width of 1 bit, then also the status structure will use only 4 bytes (8 bits = 1 byte).

BIT FIELDS (5 OF 9)

- However, as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes.
- Let us check the following example to understand the concept:

```
#include <stdio.h>
#include <string.h>
/* define simple structure */
struct
    {unsigned int widthValidated;
    unsigned int heightValidated;
    } status1;
/* define a structure with bit fields */
struct
    {unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
    } status2;
int main( )
    {printf( "Memory size occupied by status1 : %ld\n", sizeof(status1));
    printf( "Memory size occupied by status2 : %ld\n", sizeof(status2));
    return 0;}
```

OUTPUT

Memory size occupied by status1 : 8
Memory size occupied by status2 : 4

BIT FIELDS (6 OF 9)

- The declaration of a bit field has the following form inside a structure:

```
struct
{
type [member_name] : width ;
};
```

- Each element describing the bit field is given below:

Elements	Description
Type	An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
Member_name	The name of the bit-field.
Width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

BIT FIELDS (7 OF 9)

- The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit-field with a width of 3 bits as follows:

```
struct
{
unsigned int age : 3;
} Age;
```

- The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value.
- If you try to use more than 3 bits, then it will not allow you to do so.

BIT FIELDS (8 OF 9)

- Let us try the following example:

```
#include <stdio.h>
#include <string.h>
struct
{unsigned int age : 3;} Age;
int main( )
    {
        Age.age = 4;
        printf( "Sizeof( Age ) : %ld\n", sizeof(Age) );
        printf( "Age.age : %d\n", Age.age );
        Age.age = 7;
        printf( "Age.age : %d\n", Age.age );
        Age.age = 8;
        printf( "Age.age : %d\n", Age.age );
        return 0;
    }
```

OUTPUT

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Notice the warning when the input value exceeds the width of the bit-field –
"warning: unsigned conversion from 'int' to 'unsigned char:3' changes value from '8' to '0'"

BIT FIELDS (9 OF 9)

Class Exercise

1. By choosing the appropriate bit-field width in a C program, you can efficiently allocate the memory used by the program. Thus, using bit-field, what will be the size of a structure called employee, with members ***id***, ***sex***, & ***age*** assigned values 203, M, & 23, respectively.
2. Write a structure called Date in C program, whose members are ***day***, ***month***, & ***year***. Ensure the memory allocated to your program is at most 1 byte. Print the size of the struct and the date stored within its members.

TYPEDEF (1 OF 5)

- This is a C keyword, used to define alias/synonyms for an existing type in C language.
- At most cases we use typedef's to simplify the existing type declaration syntax, or to provide specific descriptive names to a type.
- The syntax of a typedef is given below:

```
typedef <existing-type> <new-type-identifier>;
```

- For example the statement below declare an alias for int with name Integer. Which means the following two statements are equivalent in meaning.

```
typedef int Integer;
```

```
int num1 = 0; // Declare a variable of int type
Integer num2 = 0; // Declare a variable of int type
```

TYPEDEF (2 OF 5)

- In this example we define a term BYTE for one-byte numbers:

```
typedef unsigned char BYTE;
```

- After this type definition, the identifier BYTE can be used as an abbreviation for the type unsigned char, for example:

```
BYTE b1, b2;
```

- By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows:

```
typedef unsigned char byte;
```

- You can use typedef to give a name to your user-defined data types as well.

TYPEDEF (3 OF 5)

- For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows:

```
#include <stdio.h>
#include <string.h>
typedef struct Books
    {char title[50]; char author[50];
    char subject[100]; int book_id;} Book;
int main( )
    {Book book;
    strcpy( book.title, "C Programming");
    strcpy( book.author, "Nuha Ali");
    strcpy( book.subject, "C Programming Tutorial"
);
    book.book_id = 6495407;
    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id);
    return 0;}
```

OUTPUT

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming
Tutorial
Book book_id : 6495407
```

TYPEDEF (4 OF 5)

TYPEDEF vs #DEFINE:

- #define is a C-directive which is also used to define the aliases for various data types similar to typedef but with the following differences:
- typedef is limited to giving symbolic names to types only, whereas #define can be used to define alias for values as well, e.g., you can define 1 as ONE, etc.
- typedef interpretation is performed by the compiler whereas #define statements are processed by the preprocessor.

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
int main( )
    {printf( "Value of TRUE : %d\n", TRUE);
    printf( "Value of FALSE : %d\n", FALSE);
    return 0;}
```

OUTPUT
Value of TRUE : 1
Value of FALSE : 0

TYPEDEF (5 OF 5)

Class Exercise

1. Write a C program to demonstrate typedef, by creating different alias for both predefined and user-defined data types.
2. Using an alias for a user defined data type write a C program to receive and display on the console, the name, department, and staff number of an employee in CPUT.