

CS 1037


Fundamentals of Computer
Science II

C Programming Features

Ahmed Ibrahim



User-defined (UD) data types



UD data types

- There are three constructs for user-defined data types: **structures**, **unions**, and **enumerations**.
- A **structure** type (**struct**) is used to store a collection of data objects of various types, such as:
 - Primitive data types
 - Pointers
 - Arrays
 - Other user-defined data types that are already defined
- A **union** type stores one data object from a given set of data types at the same memory location.
- An **enumeration** type is a simple and straightforward tool for representing symbolic constants by integer values, making your data representation tasks easier.
- Extended data types built using these constructs form a **hierarchy of data types**.
- Applying these three construction methods to existing data types can create new data types.

Structures

- A structure (**struct**) is a user-defined data type. It consists of a **list of variables** of **various** data types (also called **members** or **fields**).
- The size of a structure is the sum of the sizes of all member's data types.
- A structure type variable is stored in a memory block, in which values of its members are stored in contiguous memory locations.
- The following is the **struct** definition syntax:

```
struct struct_name {  
    data_type_1 member_name_1;  
    data_type_2 member_name_2;  
    .....  
    data_type_k member_name_k;  
};
```

Structures (cont.)

- The structure definition does not allocate memory. It only defines a template that tells the compiler the memory layout of the structure and details of its members.
- The structure type (**struct**) variable declaration tells the compiler to allocate a memory block for the variable.
- Each member of a structure type variable will have its memory space.

```
struct record {  
    int id;  
    float score;  
} s1; // define struct record type and  
      // use it to declare variable s1  
  
// declare struct record type variable s2  
struct record s2;
```

- This defines a structure type named **struct record**.
- `sizeof(struct record)` is equal to `sizeof(int) + sizeof(float) = 8 bytes`.

Example

```
1 #include <stdio.h>
2 int main(){
3     struct record {
4         int id;
5         float score;
6     };
7     printf("sizeof(struct record):%d\n", sizeof(struct record));
8     struct record s1;
9     s1.id = 1;
10    s1.score = 78;
11    printf("s1.id:%d\n", s1.id);
12    printf("s1.score:%f\n", s1.score);
13    printf("sizeof(s1):%d\n", sizeof(s1));
14    printf("s1 address:%p\n", &s1);
15    printf("s1.id address:%p\n", &s1.id);
16    printf("s1.score address:%p\n", &s1.score);
17    return 0;
```

Initialization

Output:

```
sizeof(struct record):8
s1.id:1
s1.score:78.000000
sizeof(s1):8
s1 address:0xffff000bd8
s1.id address:0xffff000bd8
s1.score address:0xffff000bdc
```

Structure Variable Initialization

```
1 #include <stdio.h>
2 #include <string.h>
3
4 struct Book {
5     char title[50];
6     char author[50];
7     int pages;
8     float price;
9 };
10
11 int main() {
12     // Initializing a structure variable
13     struct Book book1 = {"C Programming",
14         "Dennis Ritchie", 300, 29.99};
15
16     // Accessing and printing structure members
17     printf("Title: %s\n", book1.title);
18     printf("Author: %s\n", book1.author);
19     printf("Pages: %d\n", book1.pages);
20     printf("Price: $%.2f\n", book1.price);
21
22     return 0;
23 }
```

- Initializing a structure-type variable means assigning some constant values to the members of a structure variable.

General Syntax

The general syntax to initialize a structure variable is as follows:

```
struct struct_name {  
    data_type_1 member_name_1;  
    data_type_2 member_name_2;  
    ... } struct_var = {constant1, constant2, ... };
```

```
| struct struct_name struct_var = {constant1, constant2, ... };
```

Initialization should match their corresponding member types in the structure definition.

typedef

- The **typedef** keyword in C allows you to create an alias or a new name for an existing data type, making your code easier to read and manage.
- It's particularly useful for simplifying complex declarations or when working with **structures**, **enums**, or **pointers**.
- Example: **typedef** int **INT**;
- **Typedef** creates an alias **INT** that can be used interchangeably with int. After this definition, both **INT** a = 10 and int a = 10 are equivalent.

```
struct record {  
    int id;  
    float score;  
};
```

// Declaring a variable of the structure **type**
struct record s1;

Two steps

```
struct record {  
    int id;  
    float score; };  
  
typedef struct record RECORD;  
  
// Now you can use RECORD instead of struct record  
RECORD s1;  
s1.id = 1;  
s1.score = 95.5;
```

Simplified:

```
typedef struct {  
    int id;  
    float score;  
} RECORD;  
  
// Now, RECORD is directly the type name  
RECORD s1;
```

Why Use `typedef` with `struct`?

- **Code Simplification:** It makes your code cleaner and easier to read, especially when dealing with complex or frequently used data structures.
- **Readability:** Using `typedef` gives you the flexibility to create descriptive names that better convey the purpose or function of the data type.
- **Portability:** It allows for easier modification or replacement of data types if needed. For example, if you want to change `RECORD` to a different structure definition, you only need to update the typedef line.

Structure Variable Assignment

- A structure variable can be assigned to another structure variable of the same type and can be used in initialization.

```
typedef struct {  
    int id;  
    name[20]; // an array can be added in a structure  
    float score;  
} RECORD;  
  
RECORD s1 = {1, "Brian", 78};  
RECORD s2 = {2, "William", 80};  
RECORD s3 = s1; // declare and initialize s3 by s1  
RECORD s4;      // declare and initialize s4  
s4 = s2;        // assign s2 to s4
```

Nested Structures

- Structure members can be of any data type already defined.
- So, a structure-type member can be placed within another structure.

Such a structure that contains another structure as its member is called a nested structure.

```
typedef struct {  
    char first_name[20];  
    char mid_name[20];  
    char last_name[20];  
} NAME;
```

```
typedef struct {  
    int dd;  
    int mm;  
    int yy;  
} DATE;
```



```
typedef struct {  
    int id;  
    NAME name;  
    DATE dob;  
} STUDENT;
```

Pointers on Structures

- A structure-type pointer can be declared to hold the address of a structure-type variable.
- Then, use the arrow (**point-to**) operator **->** to access members of the structure variable.

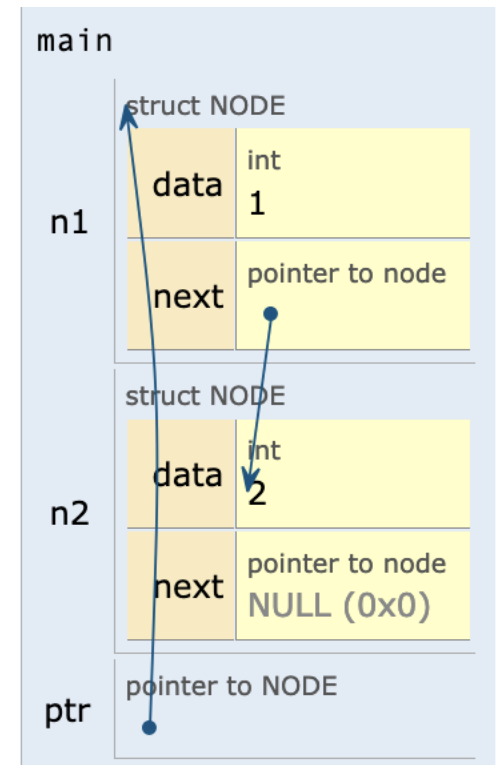
```
1  #include <stdio.h>
2
3  typedef struct {
4      int x;
5      int y;
6  } POINT;
7
8  int main() {
9      POINT p1;          // Declare structure variable
10     POINT *ptr;         // Declare pointer to structure
11
12     ptr = &p1;          // Assign address of p1 to ptr
13
14     // Assign values using the pointer
15     ptr->x = 10;
16     ptr->y = 20;
17
18     // Access values using the structure variable
19     printf("Using structure variable: x = %d, y = %d\n",
20           p1.x, p1.y);
21
22     // Access values using the pointer
23     printf("Using pointer: x = %d, y = %d\n", ptr->x, ptr->y);
24
25     return 0;
26 }
```

Self-referential Structures

- Self-referential structures include a reference to data of their own type.
- In the **struct** node definition, the **next** variable acts as a pointer to a **struct** node type.
- The method of self-referential structures is the foundation for building linked data structures such as **linked lists** and **trees**.

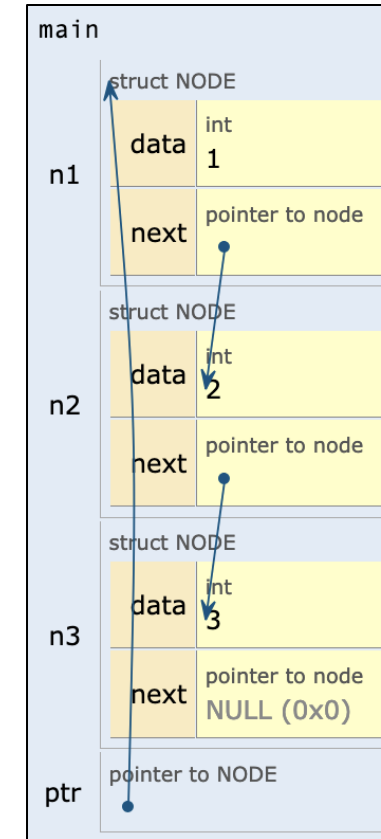
```
1  #include <stdio.h>
2
3  int main() {
4
5      typedef struct node {
6          int data;
7          struct node *next;
8      } NODE;
9
10     NODE n1, n2, *ptr = &n1;
11     n1.data = 1;
12     n1.next = &n2;
13     n2.data = 2;
14     n2.next = NULL;
15
16     // output: 1
17     printf("%d\n", ptr->data);
18
19     // output: 2
20     printf("%d\n", ptr->next->data);
21
22     return 0;
23 }
```

Memory:



The concept of the Linked List

```
1  #include <stdio.h>
2
3  int main() {
4
5      typedef struct node {
6          int data;
7          struct node *next;
8      } NODE;
9      NODE n1, n2, n3, *ptr = &n1;
10     n1.data = 1;
11     n1.next = &n2;
12     n2.data = 2;
13     n2.next = &n3;
14     n3.data = 3;
15     n3.next = NULL;
16
17     return 0;
18 }
```



Structures and Functions

For function input and output, structure types behave like primitive data types.

- Structure variables can be passed to a function by value or by reference.
- A structure can also be used as a function's return type, and the returned value can be assigned to a variable of the same type or passed to another function.

```
1  #include <stdio.h>
2
3  typedef struct {
4      int x;
5      int y;
6  } POINT;
7
8  void display(POINT p) {
9      printf("%d, %d\n", p.x, p.y);
10 }
11
12 int main() {
13     POINT p1 = {2, 3};
14     display(p1); // Output: 2, 3
15     return 0;
16 }
```

Question!

- Write a C program that defines a structure POINT with two integer members `x` and `y`. Then, create an array of `POINT` with two elements, initialize it, and implement a function to display all the points in the array:
 - `display_all(POINT p[], int n)`: A function that takes the array of `POINT` structures and the number of elements as arguments and prints all points using array indexing.
 - In the main function, demonstrate calling the function to display the array's contents.

```

1  #include <stdio.h>
2
3  // Define the structure POINT
4  typedef struct {
5      int x;
6      int y;
7  } POINT;
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24  }

```

Write a C program that defines a structure POINT with two integer members x and y. Then, create an array of POINT with two elements, initialize it, and implement a function to display all the points in the array:

display_all(POINT p[], int n): A function that takes the array of POINT structures and the number of elements as arguments and prints all points using array indexing. In the main function, demonstrate calling the function to display the array's contents.

Pass an array of structures to functions

- Passing an array of structures to a function is the same as passing an array to a function.

```
1  #include <stdio.h>
2
3  // Define the structure POINT
4  typedef struct {
5      int x;
6      int y;
7  } POINT;
8
9  // Function to display all points using an array
10 void display_all(POINT p[], int n) {
11     int i;
12     for (i = 0; i < n; i++)
13         printf("%d, %d\n", p[i].x, p[i].y);
14 }
15
16 int main() {
17     // Define an array of 2 POINT type elements
18     POINT pa[2] = {{2, 3}, {4, 5}};
19
20     // Pass array name as reference
21     display_all(pa, 2);
22
23     return 0;
24 }
```



Stop and think

Compare structures and arrays regarding memory storage, member types, assignment, and passing to functions.

- **Memory Storage** –Structures store members based on their data types, while arrays store elements in contiguous memory.
- **Member Types** –Arrays have elements of the same type accessed by **index**, whereas structures can have members of different types accessed by name.
- **Passing to Functions** –Passing a structure to a function copies the entire structure, while passing an array copies only its address.

Unions



- A **union** is a user-defined data type that can hold any variable of a given set of member variables at the same memory location.
- A **union-type** variable can hold different types of values but can only hold one type of value at a time.
- The size of a **union** is determined by the maximum size of its member types. This means a union will be as large as its largest member type.
- $\text{sizeof}(a) = \text{sizeof}(\text{union record}) = \text{sizeof}(\text{RECORD}) = \max\{4, 8\} = 8$
- Access the union's variable using the dot operator.

Syntax of defining a union data type:

```
union union-name {  
    data_type_1 var-name_1;  
    data_type_2 var-name_2;  
    ...  
    data_type_k var-name_k;  
};
```

Example:

```
union record {  
    int id;  
    double score;  
};  
union record a;  
typedef union record RECORD;  
RECORD b;
```

8 Bytes on
modern systems

Union Within a Structure

- A union allows multiple variables to share the same memory location.
- In this example, **name** and **roll_no** share the same memory space.
- Only one union member can hold a value at any given time.
- While using a union can save memory, it's critical to exercise caution. Updating one member will affect the others sharing the memory, so be mindful of this potential pitfall.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  typedef struct record {
5      union {
6          char name[20];
7          int roll_no;
8      };
9      int marks;
10 } ST;
11
12 int main() {
13     ST st;
14     st.roll_no = 1;
15     printf("%d", st.roll_no);
16     strcpy(st.name, "John");
17     printf("\n%s", st.name);
18     printf("\n%d", st.roll_no);
19     return 0;
20 }
```




Stop and think

What are the differences between unions and structures?

Stop and think

1. Each member variable of a structure has its own memory location, so it can hold all member variable data simultaneously.
2. Union uses the same memory location for all member variables. Since all members share the same memory location, it can only hold one member's data simultaneously.
3. Like structures, union types support assignment operations, pointer operations, passing union to functions by values or references, and return types.
4. The syntax of using unions is the same as using structures. Access the union's variable using the dot operator.

Enumerations

- An enumeration (type) is a user-defined data type to represent integral constants by symbols/names.
- An enumeration type variable is stored memory as int type, so the sizeof enumeration data type is equals to sizeof(int), int operations can be applied to the enumeration type.
- C uses the keyword `enum` to define an enumeration type.
- The syntax for defining an enumeration type is as follows:

```
enum enum_name {  
    const_1,      // or const_1 = value_1,  
    const_2,      // or const_2 = value_2,  
    .....  
    const_k       // or const_k = value_k  
};
```

Enumerations

- The `BOOLEAN` type is created using `typedef` and used to declare variables like `flag` and `flag1`.
- The `sizeof(BOOLEAN)` output shows that the enumeration size is 4 bytes on this system.
- The following example demonstrates how enumeration values are treated as integers:
 - `false` prints 0
 - `true` prints 1

```
1  #include <stdio.h>
2
3  typedef enum boolean {
4      false,    // false is associated to 0
5      true      // true is associated to 1
6  } BOOLEAN;
7
8  int main(){
9      printf("%d\n", sizeof(BOOLEAN));    // output: 4
10     printf("%d\n", false);              // output: 0
11     printf("%d\n", true);               // output: 1
12
13     BOOLEAN flag;
14
15     flag = true;
16     printf("%d\n", flag);                // output: 1
17
18     BOOLEAN flag1 = false;
19     flag = flag1;
20     printf("%d\n", flag);                // output: 0
21
22 }
```

Question!

- Write a C program that defines an enumeration week with the values Sunday through Saturday.
 - Use `typedef` to create an alias on **weekdays** for this enum.
 - Create a variable of the weekdays type and assign it the value Wednesday.
 - Print the integer representation of today + 1 using the `printf` function.
- The program should output the integer corresponding to the next day after "Wednesday" in the week.

Enum week

```
1  #include <stdio.h>
2
3  typedef enum week {Sunday, Monday, Tuesday,
4  Wednesday, Thursday, Friday, Saturday} weekdays;
5
6  int main()
7  {
8      // creating today variable of enum week type
9      weekdays today;
10     today = Wednesday;
11     printf("Day %d\n",today+1);
12     return 0;
13 }
```



Thank
you

References

- Data Structures Using C, second edition, by Reema Thareja, Oxford University Press, 2014.