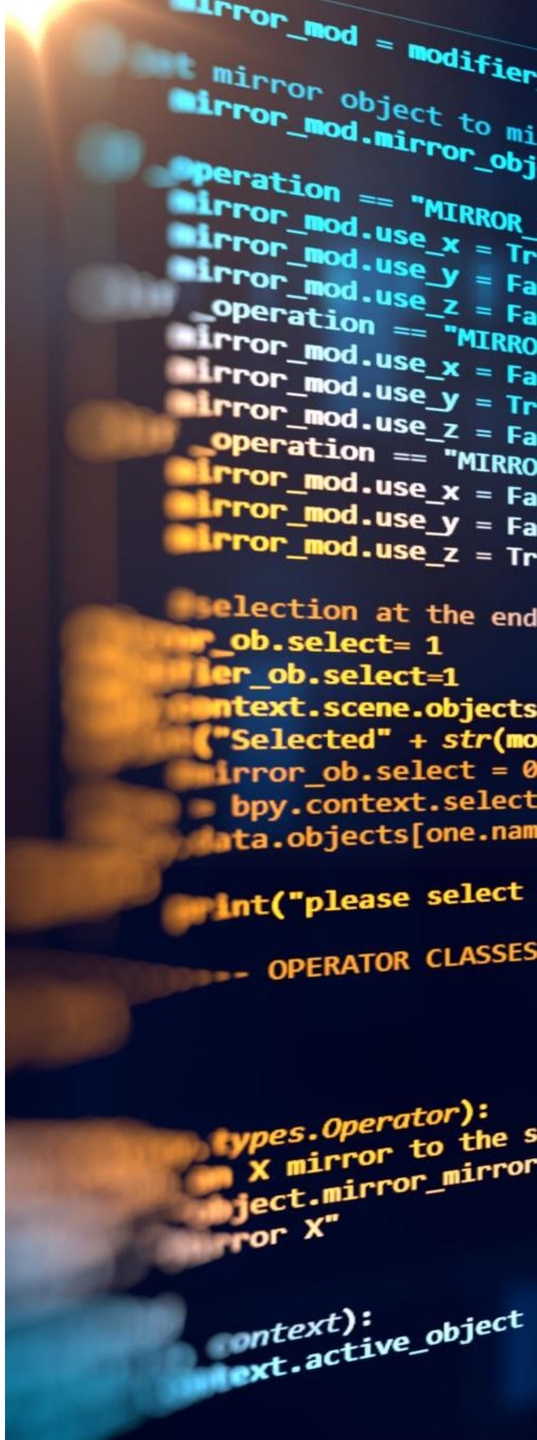
CS 1037
Fundamentals of Computer Science II

C Advances (cont.)

Ahmed Ibrahim



Please use the following QR code to check in and record your attendance.

User-defined (UD) data types

Recap: 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.

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.

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

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

Two steps

```c
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:

```c
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
```

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.

```c
#include <stdio.h>

typedef struct {
    int x;
    int y;
} POINT;

void display(POINT p) {
    printf("%d, %d\n", p.x, p.y);
}

int main() {
    POINT p1 = {2, 3};
    display(p1);   // Output: 2, 3
    return 0;
}
```

Question!

Consider the following program using structures and pointers:

```c
#include <stdio.h>

typedef struct {
    int id;
    char name[20];
    float score;
} RECORD;

void updateRecord(RECORD *r, int new_id, float new_score) {
    r->id = new_id;
    r->score = new_score;
}

int main() {
    RECORD s1 = {1, "Brian", 78};
    RECORD s2 = {2, "William", 80};
    RECORD *ptr = &s1;

    updateRecord(ptr, 3, 90);
    s2 = s1;

    printf("s2.id = %d, s2.name = %s, s2.score = %.1f",
    s2.id, s2.name, s2.score);
    return 0;
}
```

What will be printed by the program?

A) s2.id = 2, s2.name = William, s2.score = 80.0

B) s2.id = 3, s2.name = William, s2.score = 90.0

C) s2.id = 3, s2.name = Brian, s2.score = 90.0

D) s2.id = 1, s2.name = Brian, s2.score = 78.0

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.

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

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

```c
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.

```c
#include <stdio.h>

typedef struct {
    int x;
    int y;
} POINT;

int main() {
    POINT p1;       // Declare structure variable
    POINT *ptr;     // Declare pointer to structure

    ptr = &p1;      // Assign address of p1 to ptr

    // Assign values using the pointer
    ptr->x = 10;
    ptr->y = 20;

    // Access values using the structure variable
    printf("Using structure variable: x = %d, y = %d\n"
    , p1.x, p1.y);

    // Access values using the pointer
    printf("Using pointer: x = %d, y = %d\n", ptr->x, ptr->y);

    return 0;
}
```

Real-World Scenario Question:

Imagine you are designing a program to track the coordinates of drones flying in a 2D space. Each drone's position is stored as a POINT structure with x and y coordinates. Now, you need to update the positions of two drones after a movement command is issued.

```c
#include <stdio.h>

typedef struct {
    int x;
    int y;
} POINT;

void moveDrone(POINT *drone, int deltaX, int deltaY) {
    drone->x += deltaX;
    drone->y += deltaY;
}

int main() {
    POINT drone1 = {100, 200};
    POINT drone2 = {300, 400};

    moveDrone(&drone1, 50, 100);
    moveDrone(&drone2, -50, -100);

    printf("Drone 1 Position: x = %d, y = %d\n", drone1.x, drone1.y);
    printf("Drone 2 Position: x = %d, y = %d\n", drone2.x, drone2.y);

    return 0;
}
```

Real-World Scenario Question!

Imagine you are designing a program to track the coordinates of drones flying in a 2D space. Each drone's position is stored as a POINT structure with x and y coordinates. Now, you need to update the positions of two drones after a movement command is issued.

After executing the movement commands (lines 17 and 18), what will be the new positions of drone1 and drone2?

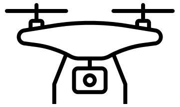
A) Drone 1 Position: x = 150, y = 300 and Drone 2 Position: x = 250, y = 300

B) Drone 1 Position: x = 150, y = 100 and Drone 2 Position: x = 350, y = 500

C) Drone 1 Position: x = 50, y = 100 and Drone 2 Position: x = 300, y = 400

D) Drone 1 Position: x = 200, y = 400 and Drone 2 Position: x = 100, y = 200

```c
1   #include <stdio.h>
2
3   typedef struct {
4       int x;
5       int y;
6   } POINT;
7
8   void moveDrone(POINT *drone, int deltaX, int deltaY) {
9       drone->x += deltaX;
10      drone->y += deltaY;
11  }
12
13  int main() {
14      POINT drone1 = {100, 200};
15      POINT drone2 = {300, 400};
16
17      moveDrone(&drone1, 50, 100);
18      moveDrone(&drone2, -50, -100);
19
20      printf("Drone 1 Position: x = %d, y = %d\n", drone1.x, drone1.y);
21      printf("Drone 2 Position: x = %d, y = %d\n", drone2.x, drone2.y);
22
23      return 0;
24  }
```

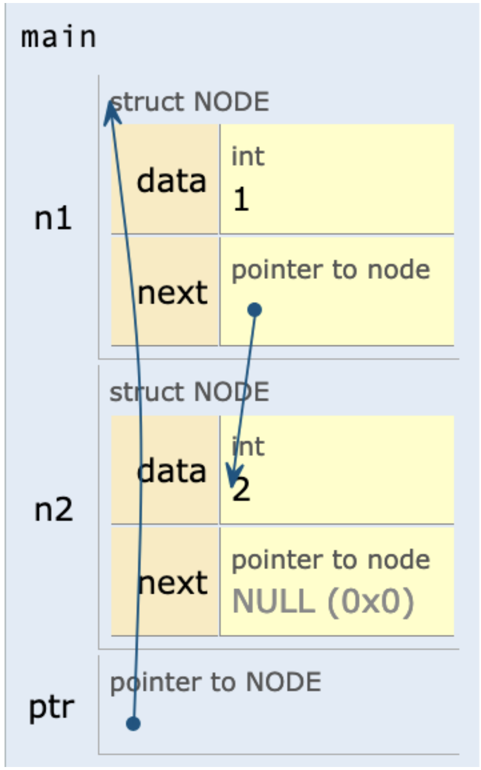


Self-referential Structures

- Self-referential structures include a reference to data of their <u>own type</u>.
- 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**.

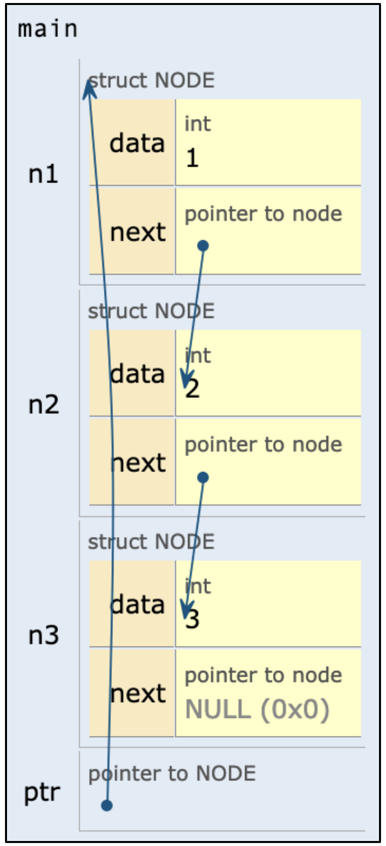
```c
#include <stdio.h>

int main() {

    typedef struct node {
    int data;
    struct node *next;
    } NODE;

     NODE n1, n2, *ptr = &n1;
     n1.data = 1;
     n1.next = &n2;
     n2.data = 2;
     n2.next = NULL;

     // output: 1
     printf("%d\n", ptr->data);

     // output: 2
     printf("%d\n", ptr->next->data);

     return 0;
}
```

Memory:



The concept of the Linked List

```c
#include <stdio.h>

int main() {

    typedef struct node {
      int data;
      struct node *next;
    } NODE;
    NODE n1, n2, n3, *ptr = &n1;
    n1.data = 1;
    n1.next = &n2;
    n2.data = 2;
    n2.next = &n3;
    n3.data = 3;
    n3.next = NULL;

    return 0;
}
```



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.

```c
#include <stdio.h>

// Define the structure POINT
typedef struct {
    int x;
    int y;
} POINT;

// Function to display all points using an array
void display_all(POINT p[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d, %d\n", p[i].x, p[i].y);
}

int main() {
    // Define an array of 2 POINT type elements
    POINT pa[2] = {{2, 3}, {4, 5}};

    // Pass array name as reference
    display_all(pa, 2);

    return 0;
}
```

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.

```c
#include <stdio.h>

// Define the structure POINT
typedef struct {
    int x;
    int y;
} POINT;

// Function to display all points using an array
void display_all(POINT p[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d, %d\n", p[i].x, p[i].y);
}

int main() {
    // Define an array of 2 POINT type elements
    POINT pa[2] = {{2, 3}, {4, 5}};

    // Pass array name as reference
    display_all(pa, 2);

    return 0;
}
```

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.
- sizeof(a) = sizeof(union record) = sizeof(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.

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

typedef struct record {
  union {
    char name[20];
    int roll_no;
  };
  int marks;
} ST;

int main() {
  ST st;
  st.roll_no = 1;
  printf("%d",  st.roll_no);
  strcpy(st.name, "John");
  printf("\n%s", st.name);
  printf("\n%d", st.roll_no);
  return 0;
}
```

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

```c
#include <stdio.h>

typedef enum boolean {
  false,    // false is associated to 0
  true      // true is associated to 1
} BOOLEAN;

int main(){
    printf("%d\n", sizeof(BOOLEAN));      // output: 4
    printf("%d\n", false);                // output: 0
    printf("%d\n", true);                 // output: 1

    BOOLEAN flag;

    flag = true;
    printf("%d\n", flag);                   // output: 1

    BOOLEAN flag1 = false;
    flag = flag1;
    printf("%d\n",  flag);                 // output: 0

}
```

Practical Exercise!

- 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

```c
#include <stdio.h>

typedef enum week {Sunday, Monday, Tuesday,
Wednesday, Thursday, Friday, Saturday} weekdays;

int main()
{
    // creating today variable of enum week type
    weekdays today;
    today = Wednesday;
    printf("Day %d\n",today+1);
    return 0;
}
```

Thank you

References

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