C PROGRAMMING
Lecture 6

1st semester 2023-2024

Exercise

```
#include <stdio.h>

int apples = 100;
int bananas = 50;
int oranges = 75;

int main() {
    int* applePtr = &apples;
    int* bananaPtr = &bananas;
    int* orangePtr = &oranges;

    printf("Shop Inventory:\n");
    printf("Apples: %d\nBananas: %d\nOranges: %d\n", *applePtr, *bananaPtr,
*orangePtr);

    // Add 25 more apples
    *applePtr += 25;

    printf("Updated Shop Inventory:\n");
    printf("Apples: %d\nBananas: %d\nOranges: %d\n", *applePtr, *bananaPtr,
*orangePtr);

    return 0;
}
```

Exercise

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

#define MAX_ITEMS 5
#define MAX_NAME_LENGTH 50


void addItem(char items[][MAX_NAME_LENGTH], double prices[], int *numItems, const char
*name, double price) {
    if (*numItems < MAX_ITEMS) {
        strcpy(items[*numItems], name);
        prices[*numItems] = price;
        (*numItems)++;
    } else {
        printf("Shop is full. Cannot add more items.\n");
    }
}

void displayItems(const char items[][MAX_NAME_LENGTH], const double prices[], int numItems)
{
    printf("Shop Items:\n");
    for (int i = 0; i < numItems; i++) {
        printf("%s - $%.2f\n", items[i], prices[i]);
    }
}
```

Exercise

```
int main() {
    char shopItems[MAX_ITEMS][MAX_NAME_LENGTH];
    double itemPrices[MAX_ITEMS];
    int itemCount = 0;

    addItem(shopItems, itemPrices, &itemCount, "Product A", 10.99);
    addItem(shopItems, itemPrices, &itemCount, "Product B", 15.49);
    addItem(shopItems, itemPrices, &itemCount, "Product C", 5.99);

    displayItems(shopItems, itemPrices, itemCount);

    return 0;
}
```

TYPEDEF

1) **Purpose of typedef**:

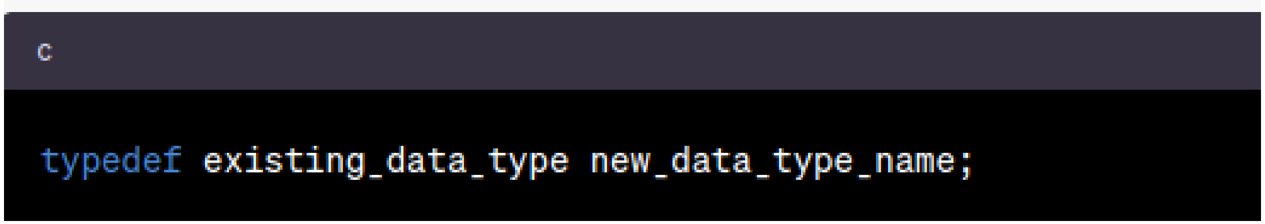
- typedef is used to create user-defined data type names (type aliases) in C. It allows you to define new names for existing data types, making the code more readable and self-explanatory.

2) **Creating Type Aliases**:

- You can use typedef to create aliases for various data types, including **primitive** data types (e.g., int, float), structures, **enumerations**, and custom data types.

TYPEDEF

3) Basic Syntax:



```c
typedef existing_data_type new_data_type_name;
```

existing_data_type: The data type for which you want to create an alias.
new_data_type_name: The new name you want to assign to the existing data type.

TYPEDEF

```
#include <stdio.h>

typedef double Balance;

int main() {
    Balance account1Balance = 1000.50;
    printf("Account 1 Balance: $%.2lf\n",
account1Balance);
    return 0;
}
```

TYPEDEF

4) Benefits:

- Improved code readability: typedef allows you to use more descriptive names for data types, making your code self-documenting.
- Portability: It can make code more portable by using abstract type names rather than specific data types.
- Easier maintenance: If you need to change the underlying data type in the future, you can do so in one place (the typedef declaration) without affecting the rest of your code.

TYPEDEF

```
#include <stdio.h>

typedef enum {
    SAVINGS,
    CHECKING,
    LOAN
} AccountType;

int main() {
    AccountType account1Type = SAVINGS;
    printf("Account 1 Type: %d\n", account1Type);
    return 0;
}
```

Structures

- Structure:
  - Collection of one or more variables
  - a tool for grouping heterogeneous elements together (different types)
- Array: a tool for grouping homogeneous elements together
- Help organize complicated data, permits group of related variables to be handled as a unit
- Example: storing calendar dates (day, month, year), students (name, address, telephone)

Structures

Benefits of Structures:

- Structures are useful for organizing and managing complex data.
- They facilitate code readability and maintainability by grouping related data together.
- Structures are commonly used in various applications, including database records, graphics, and more.

Structures

- A struct declaration defines a type:

  ```
  struct [label]{
        type member1;
        …
        type membern;
  };
  ```

- Keyword struct introduces a structure declaration (a list of declarations enclosed in {})
- Variables declared in a structure are called members

Structures

- A structure member and a regular variable can have the same name without conflict
- Struct declaration defines a type; } can be followed by variable names:

  `struct { … } var1, var2, … , varn;`
- If not followed by variable names, it reserves no storage
- If tagged (given a label), tag can be used for variables as structure type instances
- A member of a particular structure is used in an expression by `structure_name.member`

Structures

- A structure can be initialized by following it with a list of constant expressions for the members:

```
  struct point{
                int p1;
                int p2;
        } pt;
  pt = (struct point) { 100, 200 };
//struct point pt = {100, 200 };
```

or

```
pt = (struct point) { .p1 = 100, .p2 = 200 };
```

- By specifying values for each member:

```
pt.p1=100;
pt.p2=200;
```

Structures

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

int main() {
    struct Point p1 = {3, 4};
    printf("Point p1: x = %d, y = %d\n", p1.x, p1.y);
    return 0;
}
```

Structures

```
#include <stdio.h>

struct Point {
    int x;
    int y;
}pt;

int main() {
    pt.x=3;
    pt.y=4;
    printf("Point p1: x = %d, y = %d\n", pt.x, pt.y);
    return 0;
}
```

Structures

```
#include <stdio.h>

struct Point {
    int x;
    int y;
}pt;

int main() {
    pt=(struct Point) {5,6};
    //pt = (struct Point) { .x = 100, .y = 200 };
    printf("Point p1: x = %d, y = %d\n", pt.x, pt.y);
    return 0;
}
```

Structures and typedef

- With typedef can define new datatypes
- Used to shorten declaration of structure variables
- Avoids using struct keyword for every variable of that structure type:

```
typedef struct pct{
    int x;
    int y;
  }  point;
  point p1;
  p1.x = 0;
  p1.y = 0;
  point p2 = {.x=100, .y=200 };
```

Structures and typedef

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

// Define a structure for Student
struct Student {
    char name[50];
    int rollNumber;
    float marks;
};

typedef struct Professor {
    char name[50];
    int rollNumber;
    float marks;
} Professor;
```

Structures and typedef

```
// Create a typedef for the Student structure
typedef struct Student Student;

int main() {
    // Declare and initialize a student using the typedef
    Student student1;
    Professor prof1;
    strcpy(student1.name, "Alice");
    student1.rollNumber = 101;
    student1.marks = 85.5;

    strcpy(prof1.name, "Jean");
    prof1.rollNumber = 101;
    prof1.marks = 85.5;

    // Display student information
    printf("Student: %s\nRoll Number: %d\nMarks: %.2f\n", student1.name,
student1.rollNumber, student1.marks);

    // Display professor information
    printf("Professor: %s\nRoll Number: %d\nMarks: %.2f\n", prof1.name,
prof1.rollNumber, prof1.marks);


    return 0;
}
```

Operations on structures

- Possible operations on a structure :
  - initialize by a list of constant member values (expressions)
  - copy or assign to it as a unit
    - this includes passing arguments to functions and returning values from functions as well.
  - use its address (with &)
  - access its members.
  - structures can’t be compared as units !

Example

```
#include <stdio.h>
int main (void)
{
   struct date
   {
      int month;
      int day;
      int year;
   };
   struct date today, tomorrow;
   const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
   31, 31, 30, 31, 30, 31 };

   printf ("Enter today's date (mm dd yyyy): ");
   scanf ("%i%i%i", &today.month, &today.day, &today.year);
if ( today.day != daysPerMonth[today.month - 1] ) {
   tomorrow.day = today.day + 1;
   tomorrow.month = today.month;
   tomorrow.year = today.year;
   }
   else if ( today.month == 12 ) {
      tomorrow.day = 1;
   tomorrow.month = 1;
   tomorrow.year = today.year + 1;
   }
   else {
   tomorrow.day = 1;
   tomorrow.month = today.month + 1;
   tomorrow.year = today.year;
   }
   printf ("Tomorrow's date is %i/%i/%i.\n",
   tomorrow.month,
                 tomorrow.day, tomorrow.year );
   return 0;
   }
```

Structures Containing Complex DataTypes

- Structures can contain any data type
  - Can contain arrays
  - Can contain other structures
- Also called complex structures

```
typedef struct {
    int a;
    double arr[3];
  }  some_struct;
 some_struct s1;
 s1.a = 10;
 int i;
 for (i=0; i<3; i++) {
   s1.arr[i] = i;
}
```

Structures Containing Complex DataTypes

```
struct address{
          unsigned int house_number;
          char *street_name;
          int zip_code;
          char *country;
        };
struct customer{
          char *name;
          struct address billing;
          struct address shipping;
        };
struct customer c1;
c1.name="John Spencer";
c1.billing.street_name="Second Avenue";
c1.shipping.street_name=c1.billing.street_name;
```

Arrays of Structures

- Declaring an array of structures is like declaring any other kind of array.
- Each element of the array is a structure of type struct. Thus, array[0] is one structure, array[1] is a second structure, and so on.
- To identify members of an array of structures, the rules used for individual structures apply: structure name followed by the dot operator and then with the member name:

  array[0].member_name

Array of Structures

- Can declare an array of structs:

  ```
  Point points[10];
  ```

- Each array element is a struct.
- To access member of a particular element:

  ```
  points[4].x = 100;
  ```

- Because the [] and . operators are at the same precedence and associate left-to-right, this is equivalent to:

  ```
  (points[4]).x = 100;
  ```

Array of Structures

```
#include <stdio.h>

struct Employee {
    char name[50];
    int employeeID;
    double salary;
};
```

Array of Structures

```
int main() {
    struct Employee employees[4]; // Array of 4 Employee structures

    // Initialize employee records
    strcpy(employees[0].name, "John Doe");
    employees[0].employeeID = 1001;
    employees[0].salary = 55000.0;

    strcpy(employees[1].name, "Jane Smith");
    employees[1].employeeID = 1002;
    employees[1].salary = 60000.0;

    // ...

    // Print employee information
    for (int i = 0; i < 4; i++) {
        printf("Employee %d\n", i + 1);
        printf("Name: %s\nEmployee ID: %d\nSalary: %.2f\n", employees[i].name,
employees[i].employeeID, employees[i].salary);
        printf("\n");
    }

    return 0;
}
```

Array of Structures

```
#include <stdio.h>

// Define a structure for Student
typedef struct {
    char name[50];
    int rollNumber;
    float marks;
} Student;

int main() {
    // Create an array of Student using the typedef
    Student students[3];

    // Initialize the array of students
    students[0] = (Student){"Alice", 101, 85.5};
    students[1] = (Student){"Bob", 102, 92.0};
    students[2] = (Student){"Charlie", 103, 78.5};

    // Display student information from the array
    for (int i = 0; i < 3; i++) {
        printf("Student %d: Name: %s, Roll Number: %d, Marks: %.2f\n", i + 1,
students[i].name, students[i].rollNumber, students[i].marks);
    }

    return 0;
}
```

Pointers to Structures

- You can have pointers to structures.
- Just as pointers to arrays are easier to manipulate than the arrays themselves, pointers to structures are in general easier to manipulate than structures themselves.
- In some older implementations, a structure can't be passed as an argument to a function, but a pointer to a structure can.

Pointers to Structures

```
struct student *s1;
```

- The syntax is the same as for the other pointer declarations: First is the keyword struct, then the structure label and then an asterisk (*) followed by the pointer name.
- Declaration does not create a new structure, but the pointer is made to point to existing structure of that type.
- Unlike the case for arrays, the name of a structure is not the address of the structure; The & operator is needed.

Pointers to Structures

- In order to access the value of a structure member, one can use -> operator.
- A structure pointer followed by the -> operator works the same way as a structure name followed by the . (dot) operator.
- It is important to note that pointer_name is a pointer, but pointer_name->member is a member of the pointed-to structure.
- Another method to access a member value is to use (*) dereference operator:

```
struct_name.member_name ==
   (*ptr_name).member_name
```

Pointers to Structures

- We can declare and create a pointer to a struct:

  ```
  Point *pointPtr; pointPtr = &points[4];
  ```

- To access a member of the struct addressed by pointPtr:

  ```
  (*pointPtr).x = 100;
  ```

- Because the . operator has higher precedence than *,this is NOT the same as:

  ```
  *pointPtr.x = 100;
  ```

- C provides special syntax for accessing a struct memberthrough a pointer:

  ```
  pointPtr->x = 100;
  ```

Pointers to Structures

```
#include <stdio.h>

// Define a structure for a 2D point
struct Point {
    int x;
    int y;
};

int main() {
    // Create a static instance of the Point structure
    struct Point point1 = {3, 4};

    // Declare a pointer to a Point structure
    struct Point *pointPtr;

    // Assign the address of point1 to the pointer
    pointPtr = &point1;

    // Access and modify structure members using the pointer
    pointPtr->x = 7;
    pointPtr->y = 9;

    // Display the updated values using the pointer
    printf("Point (x, y) = (%d, %d)\n", pointPtr->x, pointPtr->y);

     // Access and modify structure members using the pointer
    (*pointPtr).x = 70;
    (*pointPtr).y = 90;

    // Display the updated values using the pointer
    printf("Point (x, y) = (%d, %d)\n", (*pointPtr).x, (*pointPtr).y);

    return 0;
}
```

Structures and Functions

- For passing parameters, possible approaches:
  - Pass members
  - Pass structure
  - Pass a pointer to a structure
- For returning:
  - Return a member
  - Return a structure
  - Return a pointer

Structures and Functions

- Individual fields can be passed to functions in usual way; if the member is of basic type, the value is passed, if it is an array, the address is passed
- The entire structure is passed by value
- Alternative, pass a pointer

Structures and Functions

```
struct point{
    int p1;
    int p2;
};
struct point create(int x, int y){
  struct point p;
  p.p1=x;
  p.p2=y;
  return p;
}
```

Structures and Functions

```
struct point{
    int x;
    int y;
};
void display(struct point p){
  printf("The x coordinate for the point: %d\n",p.x);
  printf("The y coordinate for the point: %d\n",p.y);
}
int main(){
    struct point pct;
    printf("Enter x value: ");
    scanf("%d",&pct.x);
    printf("Enter y value: ");
    scanf("%d",&pct.y);
    display(pct);
    return 0;
}
```

Structures and Functions

```c
#include <stdio.h>

// Define a structure for a 2D point
struct point {
    int p1;
    int p2;
};

// Function to create and initialize a point structure
struct point create(int x, int y) {
    struct point p;
    p.p1 = x;
    p.p2 = y;
    return p;
}

int main() {
    // Create a point using the create function
    struct point myPoint = create(3, 4);

    // Access and print the values of the point
    printf("Point (p1, p2) = (%d, %d)\n", myPoint.p1, myPoint.p2);

    return 0;
}
```

Structures and Functions

```c
#include <stdio.h>

// Define a structure for a point in 2D space
struct Point {
    int x;
    int y;
};

// Function that accepts a Point structure as an argument
void printPoint(struct Point point) {
    printf("Point (x, y) = (%d, %d)\n", point.x, point.y);
}

// Function that returns the distance between two points
float calculateDistance(struct Point p1, struct Point p2) {
    int dx = p1.x - p2.x;
    int dy = p1.y - p2.y;
    return sqrt(dx * dx + dy * dy);
}

int main() {
    // Create two Point structures
    struct Point point1 = {3, 4};
    struct Point point2 = {7, 9};

    // Pass a Point structure to a function
    printPoint(point1);

    // Calculate and display the distance between two points
    float distance = calculateDistance(point1, point2);
    printf("Distance between the two points: %.2f\n", distance);

    return 0;
}
```

Structures and Functions

- Unlike an array, a struct is always passed by value into a function.
- The struct members are copied to the function and changes inside the function are not reflected outside the function.
- To see the changes outside the function, solution is to pass a pointer to a struct.

```
int distance(Point *pctA, Point *pctB){
if (pctA->x == pctB->x && pctA->y == pctB->y)
{
   return 0;
}
else
   ...
}
```

Structures and Functions

```
#include <stdio.h>

// Define a structure for a 2D point
struct Point {
    int x;
    int y;
};

// Function to initialize a Point structure using pointers
void initializePoint(struct Point *point, int x, int y) {
    point->x = x;
    point->y = y;
}

int main() {
    struct Point p1, p2;

    // Initialize points using the initializePoint function
    initializePoint(&p1, 3, 4);
    initializePoint(&p2, 7, 9);

    printf("Points: %d %d\n ", p1.x, p1.y);

    return 0;
}
```

Structures and Functions

```
#include <stdio.h>

// Define a simple operation structure
struct Operation {
    int (*func)(int, int);  // Function pointer to an operation
};

// Function for addition
int add(int a, int b) {
    return a + b;
}

// Function for subtraction
int subtract(int a, int b) {
    return a - b;
}

// Function for multiplication
int multiply(int a, int b) {
    return a * b;
}
```

Structures and Functions

```
int main() {
    // Create an array of Operation structures
    struct Operation operations[] = {
        {add},
        {subtract},
        {multiply}
    };

    int num1 = 10;
    int num2 = 5;

    for (int i = 0; i < 3; i++) {
        int result = operations[i].func(num1, num2);
        printf("Operation %d result: %d\n", i + 1, result);
    }

    return 0;
}
```

Constant Pointers

- constant pointer: when the address it is pointing to can't be changed
- a constant pointer, if already pointing to an address, can't point to a new address
- Declaration:

```
<pointer_type> *const <pointer_name>
```

Constant Pointers

```
#include<stdio.h>
int main()
{
    int nr1 = 0;
    int nr2 = 1;
    int *const ptr = &nr1; //constant ptr
    ptr = &nr2; //Illegal assignement!!!!
    return 0;
}
```

Pointers to Constants

- type of pointer that can't change the value at the address pointed by it.
- Declaration:

```
const <pointer_type> *<pointer_name>;
```

Pointers to Constants

```
#include<stdio.h>
int main()
{
    int nr1 = 0;
    const int *ptr = &nr1; //pointer to
constant
    *ptr = 2; // Illegal assignement!!!
//Cannot change the value at address
// pointed by 'ptr'.
    return 0;
}
```

Pointers to Constants

```
#include <stdio.h>

// Define a structure for a 2D point
struct Point {
    int x;
    int y;
};

// Function to initialize a Point structure using
pointers
void initializePoint(struct Point *point, int x, int
y) {
    point->x = x;
    point->y = y;
}
```

Pointers to Constants

```
// Function to calculate the distance between two points
double calculateDistance(const struct Point *point1, const struct
Point *point2) {
    int dx = point1->x - point2->x;
    int dy = point1->y - point2->y;
    return sqrt(dx * dx + dy * dy);
}

int main() {
    struct Point p1, p2;

    // Initialize points using the initializePoint function
    initializePoint(&p1, 3, 4);
    initializePoint(&p2, 7, 9);

    // Calculate the distance between the points using the
calculateDistance function
    double distance = calculateDistance(&p1, &p2);

    // Display the distance
    printf("Distance between points: %.2f\n", distance);

    return 0;
}
```