c_programming_session_4

اللهمَّ علِّمنا ما ينفعنا، وانفعنا بما علمتنا، وزدنا علمًا. وافتح علينا فتحًا عظيمً.

Tags: c programming
Status: #Adult

C Programming Session 4 - Data Types, Structures, and Storage

Data Type Modifiers

Sign Modifiers

Data Type	Signed Range	Unsigned Range
`char`	-128 to 127	0 to 255
`short int`	-32,768 to 32,767	0 to 65,535
`int`	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
`long int`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615

Size Modifiers

Type	Size
`char`	1 byte
`short int`	2 bytes
`int`	4 bytes
`long int`	4/8 bytes (system dependent)
`long long int`	8 bytes
`double`	8 bytes

Type	Size
`long double`	16 bytes

Custom Type Definitions

```c
#include <stdio.h>

int main() {
    // Define custom types for clarity
    typedef signed char sint8;
    typedef unsigned char uint8;
    typedef short signed int sint16;
    typedef short unsigned int uint16;
    typedef long signed int sint32;
    typedef long unsigned int uint32;

    printf("sint8 size: %d bytes\n", sizeof(sint8));      // 1
    printf("uint8 size: %d bytes\n", sizeof(uint8));      // 1
    printf("sint16 size: %d bytes\n", sizeof(sint16));    // 2
    printf("uint16 size: %d bytes\n", sizeof(uint16));    // 2
    printf("sint32 size: %d bytes\n", sizeof(sint32));    // 4
    printf("uint32 size: %d bytes\n", sizeof(uint32));    // 4

    return 0;
}
```

Signed Number Representation

Three Methods of Representing Signed Numbers

1. Sign-Magnitude

- MSB represents sign (0 = positive, 1 = negative)
- Remaining bits represent magnitude

```
1000 1100 = -12
1000 0001 = -1
1000 1101 = -13
```

2. One's Complement

- Negative numbers: invert all bits of positive representation

```
0000 0001 = 1
1111 1110 = -1
1111 1111 = -0  // Problem: two representations of zero
```

3. Two's Complement (Most Common)

- Negative numbers: invert all bits and add 1

```
0000 0001 = 1
1111 1111 = -1
```

Storage Class Modifiers

Auto Storage Class

```c
void function() {
    auto int x = 10;  // 'auto' keyword rarely used (default for local vars)
}
```

Property	Description
Scope	Local to function/block
Lifetime	Destroyed at function execution duration
Linkage	No linkage
Default Value	Garbage value
Memory Location	RAM (Stack)
Usage	This is the default storage class for local variables, so it is rarely explicitly used.

Register Storage Class

```c
void function() {
    register int counter;  // Suggest storing in CPU register
```

```
    // Note: Cannot use & operator with register variables
}
```

Property	Description
Scope	Local to function/block
Lifetime	Destroyed at function execution duration
Linkage	No linkage
Default Value	Garbage value
Memory Location	CPU register or RAM
Usage	Suggests to the compiler that the variable should be stored in a CPU register for faster access. The compiler may or may not honor this request based on availability and optimization strategies.

Static Storage Class

Local Static Variables

```c
#include <stdio.h>

void fibonacci() {
    static int a = 0, b = 1;  // Initialized only once
    int next = a + b;
    printf("%d ", next);
    a = b;
    b = next;
}

int main() {
    for (int i = 0; i < 8; i++) {
        fibonacci();  // Prints: 1 2 3 5 8 13 21 34
    }
    return 0;
}
```

Initialization Check Example

```c
#include <stdio.h>

void initializeOnce() {
    static int initialized = 0;

    if (!initialized) {
        printf("Running setup...\n");
        initialized = 1;
    } else {
        printf("Already initialized.\n");
    }
}

int main() {
    initializeOnce();  // "Running setup..."
    initializeOnce();  // "Already initialized."
    initializeOnce();  // "Already initialized."
    return 0;
}
```

Global Static Variables

```c
// file1.c
static int private_var = 100;  // Only accessible within this file

// file2.c
extern int private_var;  // ERROR: Cannot access static variable from file1.c
```

Property	Description
Scope	Local static: function scope, Global static: file scope
Lifetime	Entire program duration
Linkage	Internal linkage (global static), No linkage (local static)
Default Value	Zero
Memory Location	RAM (Data segment)
Usage	Useful for maintaining state within a function or for creating file-private global entities.

Extern Storage Class

```c
// file1.c
int global_var = 42;

// file2.c
extern int global_var;  // Declaration (not definition)

int main() {
    printf("%d", global_var);  // Accesses variable from file1.c
    return 0;
}
```

Property	Description
Scope	Global
Lifetime	Entire program duration
Linkage	External linkage
Default Value	Zero
Memory Location	RAM (Data segment)
Usage	Declares that a variable or function is defined in another file, allowing it to be accessed across multiple source files in a project. It does not allocate memory but rather refers to an existing definition.

Build Process

Compilation Stages

```
1. Source Files        2. Preprocessor       3. Compiler
   file1.c       →        file1.i      →       file1.s
   file2.c       →        file2.i      →       file2.s
   file3.c       →        file3.i      →       file3.s

4. Assembler          5. Linker
   file1.o      →       executable.exe
```

```
    file2.o      →          ↑
    file3.o      →          ↑
```

Stage	Input	Output	Purpose
Preprocessor	`.c` files	`.i` files	Handle `#include`, `#define`, comments
Compiler	`.i` files	`.s` files	Convert to assembly language
Assembler	`.s` files	`.o` files	Convert to object code
Linker	`.o` files	`.exe` file	Combine objects, resolve symbols

Preprocessor Directives

Macro Definitions

```c
#define PI 3.14159
#define MAX_SIZE 100

// Function-like macros
#define AREA(l, w) ((l) * (w))
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    int radius = 5;
    float area = PI * SQUARE(radius);  // Expands to: 3.14159 * ((5) * (5))
    return 0;
}
```

Predefined Macros

Macro	Description
`__DATE__`	Current compilation date
`__FILE__`	Current source file name
`__LINE__`	Current line number
`__STDC__`	1 if compiler follows ANSI C standard
`__TIME__`	Current compilation time

```c
#include <stdio.h>

int main() {
    printf("File: %s\n", __FILE__);
    printf("Line: %d\n", __LINE__);
    printf("Date: %s\n", __DATE__);
    printf("Time: %s\n", __TIME__);
    return 0;
}
```

Conditional Compilation

```c
#define VERSION 2

#if VERSION > 1
    printf("Advanced version");
#elif VERSION == 1
    printf("Basic version");
#else
    printf("Legacy version");
#endif
```

Header Guards

```c
// math_utils.h
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

// Header content goes here
int add(int a, int b);
float multiply(float a, float b);

#endif  // MATH_UTILS_H
```

Multi-line Macros

```c
#define PRINT_INFO() \
    printf("Name: Fares\n"); \
    printf("Course: NTI\n"); \
    printf("Language: C\n");
```

```c
// Alternative with do-while
#define PRINT_INFO() do { \
    printf("Name: Fares\n"); \
    printf("Course: NTI\n"); \
    printf("Language: C\n"); \
} while(0)
```

Constant and Volatile Qualifiers

Const Qualifier

```c
const int MAX_USERS = 100;  // Must be initialized, cannot be changed
const int *ptr1;            // Pointer to constant data
int * const ptr2 = &var;    // Constant pointer to data
const int * const ptr3 = &var; // Constant pointer to constant data
```

Volatile Qualifier

```c
volatile uint8_t sensor_data = 0;  // Value can change unexpectedly
```

Why use volatile?

- Prevents compiler optimization
- Forces reads from memory (not cached registers)
- Essential for:
  - Hardware registers
  - Interrupt service routines
  - Multi-threaded applications

Structures

Basic Structure Definition

```c
struct person {
    uint8_t id;
    uint16_t salary;
    uint32_t address;
};

// Usage
```

```
struct person employee;
employee.id = 107;
employee.salary = 5000;
printf("Employee ID: %d\n", employee.id);
```

Structure Initialization

```
// Method 1: During declaration
struct person emp1 = {10, 5000, 0x1234};

// Method 2: After declaration
struct person emp2;
emp2.id = 20;
emp2.salary = 6000;
emp2.address = 0x5678;
```

Multiple Declarations

```
struct person {
    uint8_t id;
    uint16_t salary;
    uint32_t address;
} emp1, emp2, emp3;  // Declare multiple variables
```

Typedef with Structures

```
typedef struct {
    uint8_t id;
    uint16_t salary;
    uint32_t address;
} Person;

Person emp1, emp2;  // No need for 'struct' keyword
```

Structure Memory Layout (Padding)

```
struct example {
    uint8_t  a;    // 1 byte
    uint16_t b;    // 2 bytes
    uint8_t  c;    // 1 byte
};
```

Memory Layout with Padding:

```
| a | pad | b   b | c | pad |
|---|-----|-------|---|-----|
  1    1    2       1       1
Total: 6 bytes (not 4) due to alignment requirements
```

Optimized Layout:

```
struct optimized {
    uint8_t  a;    // 1 byte
    uint8_t  c;    // 1 byte
    uint16_t b;    // 2 bytes
};
```

Memory Layout without Padding:

```
| a | c | b   b |
|---|---|-------|
  1   1    2
Total: 4 bytes
```

Bit Fields

Basic Bit Field Usage

```
struct flags {
    uint8_t flag1 : 1;    // Uses 1 bit
    uint8_t flag2 : 2;    // Uses 2 bits
    uint8_t flag3 : 1;    // Uses 1 bit
    // Total: 4 bits used, stored in 1 byte
};

struct flags status;
status.flag1 = 1;
status.flag2 = 3;  // Binary: 11
status.flag3 = 0;
```

Practical Bit Field Example

```
struct control_register {
    uint8_t enable    : 1;
    uint8_t mode      : 2;
    uint8_t speed     : 3;
    uint8_t reserved  : 2;
} __attribute__((packed));  // Ensure no padding

struct control_register ctrl;
ctrl.enable = 1;
ctrl.mode = 2;
ctrl.speed = 5;
```

Unique and Repeated Number Algorithms

Find Unique Numbers

```
#include <stdio.h>

void find_unique(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        int count = 0;

        // Count occurrences
        for (int j = 0; j < size; j++) {
            if (arr[i] == arr[j]) {
                count++;
            }
        }

        if (count == 1) {
            printf("%d at index %d is unique\n", arr[i], i);
        }
    }
}

int main() {
    int array[] = {8, 9, 4, 5, 7, 9, 2, 5};
    find_unique(array, 8);
    return 0;
}
```

Find Repeated Numbers

```c
#include <stdio.h>

void find_repeated(int arr[], int size) {
    int processed[size];

    // Initialize processed array
    for (int i = 0; i < size; i++) {
        processed[i] = 0;
    }

    for (int i = 0; i < size; i++) {
        if (processed[i]) continue;  // Skip if already processed

        int count = 0;

        // Count and mark all occurrences
        for (int j = 0; j < size; j++) {
            if (arr[i] == arr[j]) {
                count++;
                processed[j] = 1;
            }
        }

        if (count > 1) {
            printf("%d appears %d times\n", arr[i], count);
        }
    }
}

int main() {
    int array[] = {8, 9, 4, 5, 8, 9, 2, 5};
    find_repeated(array, 8);
    return 0;
}
```

Frequency-Based Approach

```c
#include <stdio.h>

void find_repeated_frequency(int arr[], int n, int max_val) {
    int freq[max_val + 1] = {0};  // Initialize frequency array

    printf("Repeated elements: ");
```

```
    for (int i = 0; i < n; i++) {
        freq[arr[i]]++;
        if (freq[arr[i]] == 2) {  // Print only on second occurrence
            printf("%d ", arr[i]);
        }
    }
    printf("\n");
}

int main() {
    int arr[] = {1, 2, 3, 4, 2, 5, 3, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    find_repeated_frequency(arr, n, 6);
    return 0;
}
```

Practice Tasks

Task 1: Time Structure

Create a time structure to store hours, minutes, and seconds. Allow user input and display using the structure.

```
/*1- Time Struct to save Hours, Minutes, Seconds
user enter hour and minutes and secounds
print using struct
%hhu to print unsigned char correctrly
*/

#include <stdio.h>
#include "(9_jul)standard.h"
struct Time{
        uint8 hours;
        uint8 minutes;
        uint8 seconds;
        // uint16 Salary;
    };

int main(){
    struct Time savetime;
    do {
        // Hours
        do {
```

```
            printf("Enter Hours [0-23]: ");
            scanf("%d", &savetime.hours);
            if (savetime.hours >= 24)
                printf("Invalid hours, please enter [0, 23].\n");
        } while (savetime.hours >= 24);
        // Minutes
        do {
            printf("Enter Minutes [0-59]: ");
            scanf("%d", &savetime.minutes);
            if (savetime.minutes >= 60)
                printf("Invalid minutes, please enter [0, 59].\n");
        } while (savetime.minutes >= 60);
        // Seconds
        do {
            printf("Enter Seconds [0-59]: ");
            scanf("%d", &savetime.seconds);
            if (savetime.seconds >= 60)
                printf("Invalid seconds, please enter [0, 59].\n");
        } while (savetime.seconds >= 60);

        printf("Time entered is: %02d : %02d : %02d\n",savetime.hours,
savetime.minutes, savetime.seconds);
    }while (1);
}
```

```
Enter Hours [0-23]: 24
Invalid hours, please enter [0, 23].
Enter Hours [0-23]: 8
Enter Minutes [0-59]: 60
Invalid minutes, please enter [0, 59].
Enter Minutes [0-59]: 9
Enter Seconds [0-59]: 60
Invalid seconds, please enter [0, 59].
Enter Seconds [0-59]: 5
Time entered is: 08 : 09 : 05
```

Task 2: Pointer Swapping

Implement a swap function using pointers.

```
/*2- Swap using Pointer
    remimber
    int*ptr;
```

```
    ptr = &x;  //read
    *ptr =5;   //write x = 5
    */
#include <stdio.h>
void swap_ptr_way(int * ptr1,int * ptr2);

int main(){
    int x,y;
    printf("enter x value : ");
    scanf("%d",&x);

    printf("enter y value : ");
    scanf("%d",&y);

    printf("Before : x = %d, y = %d\n",x ,y);
    swap_ptr_way(&x,&y);
    printf("After : x = %d, y = %d\n",x ,y);
}

void swap_ptr_way(int * ptr1,int * ptr2){
    int temp = *ptr1;   //temp location of first variable
    *ptr1 = *ptr2;  //first pointer now location of second variable
    *ptr2 = temp;   //second pointer now location of first variable
}
```

```
enter x value : 8
enter y value : 12
Before : x = 8, y = 12
After : x = 12, y = 8
```

Task 3: Pointer Factorial

Calculate factorial using pointer parameters.

```
// 3- Factorial using Pointer

#include <stdio.h>
void fact_point(int z,int *ptr1);

int main(){
   int x,fact;
    do {
        printf("\nenter x value : ");
```

```
        scanf("%d",&x);
        fact_point(x,&fact);
        printf("%d factorial is : %d\n",x,fact);
    } while (1);
}

void fact_point(int z,int*ptr1){
    *ptr1 =1;
    for(int i=1 ; i<=z ; i++){
        *ptr1 = *ptr1 * i;
    }
}
```

```
enter x value : 5
5 factorial is : 120

enter x value : 6
6 factorial is : 720
```

Task 4: Sum Function with Pointer Return

Create a function that returns a pointer to the sum of two arguments.

```
/*function return pointer for the summation of two arguments
hint:
func(int x, int y, int * ptr){}
*/

#include <stdio.h>

void sum_point(int x,int y,int*sum);

int main(){
    int x,y,temp;
    int*ptr;
    ptr = &temp;
    do{
        printf("enter x value : ");
        scanf("%d",&x);
        printf("enter y value : ");
        scanf("%d",&y);
        sum_point(x,y,ptr);
        printf("sum is %d and the location of pointer is
```

```
%p\n",*ptr,ptr);
    } while (1);
}

void sum_point(int x,int y,int*sum){
    *sum = x + y;
}
```

```
enter x value : 5
enter y value : 4
sum is 9 and the location of pointer is 00000000005ffe8c
```

Task 5: Employee Management

Create an employee structure with ID, name, and salary for 3 employees. Find and display information of the employee with maximum salary.

```
/*Employee Struct: ID Name Salary (3 employee )
print   info of the employee with Max Salary
*/

#include <stdio.h>
#include "(9_jul)standard.h"

typedef struct Emp{
        uint8 Name;
        uint16 ID;
        uint16 Salary;
    }Emp;

Emp calc_Max_salary( Emp x,  Emp y,  Emp z);

int main(){
    do {
        Emp emp[3];             // insted of using 3 struct could use
struct array
         for (int i = 0; i < 3; i++){
            printf("Enter employee %d name (one letter): ", i + 1);
            scanf(" %c", &emp[i].Name);

            printf("Enter employee %d ID: ", i + 1);
            scanf("%d", &emp[i].ID);
```

```
            printf("Enter employee %d Salary: ", i + 1);
            scanf("%d", &emp[i].Salary);
            printf(" \n");
         }
        Emp Max = calc_Max_salary(emp[0],emp[1],emp[2]);
        printf("\nEmployee with Max Salary:\n");
        printf("name: %c\nID: %d\nSalary:
%d\n",Max.Name,Max.ID,Max.Salary);
    } while (1);
}

 Emp calc_Max_salary( Emp x,  Emp y,  Emp z){
    Emp max = x;
        if (y.Salary > max.Salary)
         max = y;
        if (z.Salary > max.Salary)
         max = z;
        return max;
}
```

```
Enter employee 1 name (one letter): A
Enter employee 1 ID: 101
Enter employee 1 Salary: 5750

Enter employee 2 name (one letter): B
Enter employee 2 ID: 102
Enter employee 2 Salary: 6400

Enter employee 3 name (one letter): C
Enter employee 3 ID: 103
Enter employee 3 Salary: 8000


Employee with Max Salary:
name: C
ID: 103
Salary: 8000
```

Next Session Preview

- Enumerations (enum)

- Unions
- Advanced preprocessor directives
- File organization and project structure