MOD 1

Features of C Language

1. Machine Independent or Portable
2. Mid-level Programming Language
3. Structured Programming Language
4. Rich Library
5. Memory Management
6. Fast Speed
7. Pointers
8. Recursion
9. Extensible

First Program of C Language

```
#include <stdio.h>
void main() {
   printf("VIT-SCOPE");
}
```

Description of the C Program

- **#include <stdio.h>**: Includes the standard input/output library functions. The printf() function is defined here.
- **void main()**: The main function is the entry point of every C program. The void keyword indicates it returns no value.
- **printf()**: Used to print data on the console.

Output of the Program

VIT-SCOPE

Input/Output Functions in C

1. printf()

- **Purpose**: Used for output; prints a statement to the console.
- **Syntax**:

```
printf("format string", arguments_list);
```

- **Format Specifiers**: %d (integer), %c (character), %s (string), %f (float), etc.

2. scanf()

- **Purpose**: Used for input; reads data from the console.

- **Syntax**:

scanf("format string", argument_list);

- Example:

```
void main() {
   int x;
   scanf("%d", &x);
   printf("X=%d", x);
}
```

Data Types in C

1. **Basic Data Types**: int, char, float, double
2. **Derived Data Types**: array, pointer, structure, union
3. **Enumeration Data Type**: enum
4. **Void Data Type**: void

Keywords in C

- Reserved words that cannot be used as variable or constant names.
- **Total Keywords**: 32
  List: auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

---

Variables in C

- **Definition**: Containers for storing data values.
- **Syntax**:

type variableName = value;

- Example:

```
int myNum;
myNum = 15;
```

Variable Naming Rules

1. Names can contain letters, digits, and underscores.
2. Names must start with a letter or an underscore.
3. Names are case-sensitive.

4. Names cannot contain whitespaces or special characters.
5. Reserved words cannot be used.

Format Specifiers in C

- **Purpose**: Used with printf() to specify the type of data being output.
- **Syntax**: Begins with % followed by a specific character.

Common Format Specifiers

Specifier	Description
%c	Character
%d	Signed integer
%e/%E	Scientific notation (float)
%f	Float
%g/%G	Float (current precision)
%i	Signed integer
%ld/%li	Long integer
%lf	Double
%Lf	Long double
%lu	Unsigned integer/long
%lli/%lld	Long long integer
%llu	Unsigned long long
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned integer
%x/%X	Hexadecimal representation
%n	Prints nothing
%%	Prints % character

Operators in C

- **Definition**: Symbols used to perform operations like arithmetic, logical, and bitwise.
- **Precedence & Associativity**: Determines evaluation order and direction of operators (left-to-right or right-to-left).

Example:

int value = 10 + 20 * 10; // Evaluated as 10 + (20 * 10)

Types of Operators

1. **Based on Operands**:
    - **Unary**: Operates on a single operand.
    - **Binary**: Operates on two operands.
    - **Ternary**: Operates on three operands.
2. **Based on Operations**:
    - Arithmetic, Relational, Logical, Bitwise, Assignment, Ternary, Miscellaneous.

Arithmetic Operators

Operator	Description	Example
+	Addition	A + B = 30
-	Subtraction	A - B = -10
*	Multiplication	A * B = 200
/	Division	B / A = 2
%	Modulus	B % A = 0
++	Increment (by 1)	A++ = 11
--	Decrement (by 1)	A-- = 9

Relational Operators

Operator	Description	Example
==	Equal to	(A == B) false
!=	Not equal	(A != B) true
>	Greater than	(A > B) false
<	Less than	(A < B) true

Operator	Description	Example
>=	Greater than or equal to	(A >= B) false
<=	Less than or equal to	(A <= B) true

Logical Operators

Operator	Description	Example
&&	Logical AND (both true)	(A && B) false
`		`
!	Logical NOT (negates condition)	!(A && B) true

Bitwise Operators

Operator	Description	Example
&	AND	(A & B) = 12
`	`	OR
^	XOR	(A ^ B) = 49
~	Complement	~A = -61
<<	Left Shift	A << 2 = 240
>>	Right Shift	A >> 2 = 15

Assignment Operators

Operator	Description	Example
=	Assign value	C = A + B
+=	Add and assign	C += A
-=	Subtract and assign	C -= A
*=	Multiply and assign	C *= A
/=	Divide and assign	C /= A
%=	Modulus and assign	C %= A
<<=	Left Shift and assign	C <<= 2

Operator	Description	Example
>>=	Right Shift and assign	C >>= 2
&=	Bitwise AND and assign	C &= 2
`	=`	Bitwise OR and assign
^=	Bitwise XOR and assign	C ^= 2

Ternary Operator

- **Syntax**: variable = Expression1 ? Expression2 : Expression3
- **Example**:

```
int m = 5, n = 4;
(m > n) ? printf("m is greater") : printf("n is greater");
```

Miscellaneous Operators

Operator	Description	Example
sizeof()	Returns size of variable	sizeof(a) (int: 4)
&	Address of variable	&a
*	Pointer to a variable	*a

Control Statements

1. **Selection**: if, if-else, switch
2. **Loops**: for, while, do-while
3. **Jump**: break, continue, goto

Examples

1. **If-Else**:

```
int age;
scanf("%d", &age);
if (age >= 18)
   printf("Eligible to vote.");
else
   printf("Not eligible to vote.");
```

2. **Switch**:

```
switch (x > y && x + y > 0) {
   case 1: printf("hi"); break;
   case 0: printf("bye"); break;
   default: printf("Hello bye");
}
```

C SWITCH STATEMENT

```
#include <stdio.h>
int main() {
   int x = 10, y = 5;
   switch(x > y && x + y > 0) {
      case 1: printf("hi"); break;
      case 0: printf("bye"); break;
      default: printf("Hello bye");
   }
}
```

LOOPS IN C

Loops execute a block of code multiple times.

Types of Loops:

1. **Do-While Loop**: Executes code at least once.
   Syntax:

```
do {
   // code
} while (condition);
```

Example:

```
int i = 1;
do {
   printf("%d \n", i);
   i++;
} while (i <= 10);
```

2. **While Loop**: Used when the number of iterations is unknown.
   Syntax:

```
while (condition) {
   // code
}
```

Example:

```
int i = 1;
while (i <= 10) {
   printf("%d \n", i);
   i++;
}
```

3. **For Loop**: Ideal when the number of iterations is known.
   Syntax:

```
for (initialization; condition; incr/decr) {
   // code
}
```

Example:

```
int i, number;
printf("Enter a number: ");
scanf("%d", &number);
for (i = 1; i <= 10; i++) {
   printf("%d \n", number * i);
}
```

BREAK STATEMENT

Used to exit a loop or switch case.
Syntax: break;
Example:

```
int i = 0;
while (1) {
   printf("%d ", i++);
   if (i == 10) break;
}
```

CONTINUE STATEMENT

Skips the current iteration and continues the loop.
Syntax: continue;
Example:

```
int i = 0;

while (i != 10) {

   printf("%d", i);

   continue;

   i++;

}
```

GOTO STATEMENT

Unconditionally jumps to a labeled statement.
Syntax:

```
goto label;

// code

label:

// code
```

Example:

```
void checkEvenOrNot(int num) {

   if (num % 2 == 0) goto even;

   else goto odd;

   even: printf("%d is even", num); return;

   odd: printf("%d is odd", num);

}

int main() {

   checkEvenOrNot(26);

   return 0;

}
```

COMMENTS IN C

1. **Single Line:** // comment
2. **Multiline:**

```
/* comment
   spanning
   multiple lines */
```

PRECISION CONTROL

1. **Default Precision:** 6 decimal places (%f).
   Examples:

```
printf("%.1f", 3.14159); // Output: 3.1
printf("%.4f", 3.14159); // Output: 3.1416
```

2. **Width Specification:**

```
printf("%6.2f", 3.14);   // Output: "  3.14"
printf("%06.2f", 3.14);  // Output: "003.14"
```

3. **Sign Formatting:**

```
printf("%+6.2f", 3.14);  // Output: " +3.14"
printf("%+6.2f", -3.14); // Output: " -3.14"
```

4. **Scientific Notation:**

```
printf("%.2e", 31415.926); // Output: 3.14e+04
```

MATH.H FUNCTIONS

Function	Description	Example	Output
pow(x, y)	Returns xyx^yxy	pow(2, 3)	8.0
sqrt(x)	Returns the square root of xxx	sqrt(16)	4.0

C Development Environment, which involves the following phases:

1. **Editor**:
   - The programmer writes and saves the source code in a text editor, which is stored on the disk.
2. **Preprocessor**:
   - The preprocessor processes the source code, handling directives like #include and #define, and prepares the code for the compiler.
3. **Compiler**:
   - Converts the preprocessed source code into machine-level object code, which is stored on the disk.

4. **Linker**:
    - Links the object code with necessary libraries to generate an executable file, stored on the disk.
5. **Loader**:
    - Loads the executable program into the primary memory for execution.

MOD 2

ARRAYS IN C

- **Definition**: A collection of homogeneous elements stored in contiguous memory.
- **Declaration**:
  data_type array_name[array_size];
  Example: int marks[7];
- **Types of Arrays**:
    1. **1D Array**: Linear collection of elements.
        - Example Code:

```
int values[5];

for(int i = 0; i < 5; ++i) scanf("%d", &values[i]);

for(int i = 0; i < 5; ++i) printf("%d\n", values[i]);
```

        - **Output**: Displays input integers.
    2. **2D Array**: Represented as rows and columns (matrix).
        - **Declaration**: data_type array_name[size1][size2];
          Example: int arr[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
        - Example Code:

```
float a[2][2], b[2][2], result[2][2];

for (int i = 0; i < 2; ++i)

   for (int j = 0; j < 2; ++j) result[i][j] = a[i][j] + b[i][j];
```

        - **Output**: Displays sum of two matrices.
- **Advantages**:
    1. Code optimization.
    2. Easy data traversal.
    3. Data sorting.
    4. Random access.

STRINGS IN C

- **Definition**: Array of characters ending with a NULL character (\0).
- **Declaration**:
    - Using **char array**: char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    - Using **string literal**: char greeting[] = "Hello";
- **String Functions**:
    - strcpy(s1, s2): Copies s2 into s1.
    - strcat(s1, s2): Concatenates s2 onto s1.
    - strlen(s1): Returns length of s1.
    - strcmp(s1, s2): Compares s1 and s2.
    - strchr(s1, ch): Finds first occurrence of ch in s1.
    - strrev(s1): Reverses s1.
    - strlwr(s1): Converts s1 to lowercase.
    - strupr(s1): Converts s1 to uppercase.
- **Example**:

```
char greeting[] = "Hello";
printf("Greeting message: %s\n", greeting);
```

FUNCTIONS IN C

- **Definition**: A reusable block of code executed when called.
- **Components**:
    - **Declaration**: return_type function_name(data_type parameter);
    - **Definition**: Contains header and body.
    - **Invocation**: variable = function_name(arguments);
- **Advantages**:

1. Code reusability.
2. Optimization.

- **Example**:

```
float cent2fahr(float c) {
   return c * 9 / 5 + 32;
}
```

Invocation:

```
float fahr = cent2fahr(cent);
```

- **Parameter Passing**:
  Actual parameters' values are copied to formal parameters.
  Example:

```
double area(double r) { return 3.14 * r * r; }
```

- **Return Value**:
  - Functions return values using the return statement.
  - Example:

```
int gcd(int a, int b) {
   while (b % a != 0) {
      int temp = b % a;
      b = a;
      a = temp;
   }
   return a;
}
```

- **Output Example**:

```
printf("GCD of %d and %d is %d\n", x, y, gcd(x, y));
```

Local Variables

- Variables like radius and area inside a function are local.
- Formal parameters (e.g., d in circle_area) are local to the function.

Function Notes

- **Formal Parameters**: Local to the function; not recognized outside.
- **Call and Return**:
  - Value-returning functions (return type ≠ void) can be included in expressions.
  - Functions cannot be defined inside other functions but can call each other in nested or recursive calls.

Example: Nested Calls

```
int ncr(int n, int r) {
   return fact(n) / (fact(r) * fact(n - r));
}
int fact(int n) {
   int temp = 1;
   for (int i = 1; i <= n; i++) temp *= i;
```

```
    return temp;
}
```

Types of Functions

- **User-defined**: Custom functions created by users.
- **Predefined**: Built-in functions like scanf() and getch().

Call by Value

- The function works on a copy of the value.
- Changes inside the function do not affect the original variable.

Example:

```
void change(int num) {
    num += 100;
}
```

Call by Reference

- The function modifies the actual value by accessing its address.

Example:

```
void change(int *num) {
    *num += 100;
}
```

Recursion

- **Definition**: A function calling itself.
- **Example**: Factorial Calculation

```
long int multiplyNumbers(int n) {
    return (n >= 1) ? n * multiplyNumbers(n - 1) : 1;
}
```

Type Modifiers

- **Modifiers**: signed, unsigned, long, short.
- **Qualifiers**:
    - **const**: Prevents variable modification.

- **volatile**: Indicates the variable may change unexpectedly.

Storage Classes

- **auto**: Local, garbage value by default, block-specific scope.
- **extern**: Global, retains value across files.
- **static**: Retains value between function calls.
- **register**: Stored in CPU registers for faster access.

Example: auto Storage Class

```
int main() {
   auto int j = 1;
   {
      auto int j = 2;
      {
         auto int j = 3;
         printf("%d", j); // Output: 3
      }
      printf("%d", j); // Output: 2
   }
   printf("%d", j); // Output: 1
}
```

Storage Classes in C

EXTERN

- **Definition**: Used for global variables and functions shared across multiple files.
- **Purpose**: To reference variables or functions defined in other files.
- **Key Points**:
  - Variables declared using extern are global.
  - They can be accessed throughout the program.
  - Cannot be initialized; must be defined in another file.

Example:

```
// File1.c
```

```
int x = 10;  // Definition


// File2.c
extern int x;  // Reference to the variable defined in File1
```

STATIC

- **Definition**: A local variable that retains its value between function calls and is only accessible within the function or block.
- **Key Points**:
    - Default initial value is zero.
    - Initialized only once during its lifetime.
    - Accessible only to the specific function/block where it's defined.
    - Lifespan: Entire program runtime.

Example:

```
#include <stdio.h>
void counter() {
   static int count = 0; // Static variable
   count++;
   printf("Count: %d\n", count);
}

int main() {
   counter();
   counter();
   return 0;
}
```

Output:

Count: 1

Count: 2

REGISTER

- **Definition**: Stores local variables within CPU registers instead of RAM for faster access.
- **Key Points**:

- Similar to auto storage class.
- Local variables are stored in CPU registers instead of memory.
- Limited to the specific block.
- Faster access than variables in memory.

Example:

```
#include <stdio.h>
void function() {
    register int i;  // Register variable
    for (i = 0; i < 5; i++) {
        printf("%d ", i);
    }
}

int main() {
    function();
    return 0;
}
```

Output:

0 1 2 3 4

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
auto	Inside a function/block	Memory	Unpredictable	Within the function/block	Within the function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within the function/block
extern	Outside all functions	Memory	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Memory	Zero	Within the function/block	program runtime
Static (global)	Outside all functions	Memory	Zero	Global	program runtime

MOD 3

POINTER IN C

- **Definition**: A pointer is a variable that stores the memory address of another variable.
- **Declaration**: Must declare a pointer before using it.
    - Syntax: data type *var-name;
    - Example: int *ip;, double *dp;, float *fp;

ADVANTAGES OF POINTERS IN C

- Reduces code and improves performance.
- Enables returning multiple values from a function.
- Allows accessing any memory location.

SYMBOLS IN POINTERS

- **&** (Address-of operator): Finds the address of a variable.
- ***** (Indirection operator): Accesses the value at a specific address.

POINTER DECLARATION

- Syntax examples:
    - int *ptr;
    - int (*ptr)();
    - int (*ptr)[2];

Example:

int a = 5;

int *ptr;

ptr = &a; // ptr stores the address of 'a'

EXAMPLES

- **Basic Pointer Example**:

```
#include <stdio.h>

int main() {
    int var = 20;
    int *ip = &var;
```

```
    printf("Address of var: %u\n", &var);
    printf("Address stored in ip: %u\n", ip);
    printf("Value at ip: %d\n", *ip);
    return 0;
}
```

Output:

Address of var: bffd8b3c

Address stored in ip: bffd8b3c

Value at ip: 20

- **Swapping with Pointers**:

```
#include <stdio.h>
int main() {
    int a = 10, b = 20, *p1 = &a, *p2 = &b;
    printf("Before swap: *p1=%d *p2=%d", *p1, *p2);
    *p1 = *p1 + *p2;
    *p2 = *p1 - *p2;
    *p1 = *p1 - *p2;
    printf("\nAfter swap: *p1=%d *p2=%d", *p1, *p2);
    return 0;
}
```

NULL POINTER

- A pointer assigned with NULL has no valid address.
- NULL pointer value is 0.
- **Example**:

```
int *ptr = NULL;
printf("The value of ptr is: %u\n", ptr);
```

Output: The value of ptr is 0

POINTER ARITHMETIC

- Pointers support addition and subtraction operations.
- Incrementing a pointer (p1++) moves it by the size of the type it points to.

Example:

- If p1 holds 2000 and int is 4 bytes:
    - p1++ makes it 2004.
    - p1-- makes it 1996.

POINTER COMPARISON

- You can compare two pointers.
    - Example: if(p < q) printf("p points to lower memory than q");
- Useful when pointers point to elements of a common object like an array.

POINTERS AND ARRAYS

- p1 = str; assigns pointer p1 to the first element of the array str.
- Access array elements using pointer arithmetic:
    - *(p1 + 4) accesses the 5th element.

POINTER ARITHMETIC ON ARRAYS

```
int arr[] = {1, 2, 3, 4, 5};
int* ptr = arr;
for (int i = 0; i < 5; i++) {
   printf("%d ", ptr[i]);  // Access elements using pointer arithmetic
   ptr++;
}
```

POINTER TO POINTER

- **Types of Pointers**:
    - **Null Pointer**: Points to no valid address (NULL).
    - **Void Pointer**: Generic pointer type, can point to any data type.
    - **Wild Pointer**: Uninitialized pointer, can lead to undefined behavior.
    - **Dangling Pointer**: Pointer pointing to a memory location that has been freed.

POINTER TYPES AND USAGE IN C

1. **NULL Pointer**:
    - A pointer assigned NULL has no valid address.

- Example: int *ptr = NULL;

2. **Void Pointer**:
    - A pointer declared with void can point to any data type.
    - To print or dereference, it needs to be typecasted.
    - Example: void *ptr;
3. **Wild Pointer**:
    - A wild pointer is declared but not assigned a valid memory address.
    - It can cause segmentation faults if used incorrectly.
4. **Dangling Pointer**:
    - A pointer that points to a memory location that has been deallocated.
    - Example: If p points to a variable at memory location 1004 and the memory is deallocated, p becomes a dangling pointer.

INDEXING POINTERS

- An array name without an index is treated as a pointer to the first element of the array.
    - Example:

```
char p[10];
p == &p[0]; // True
```

POINTERS AS ARRAYS

- An array name is a pointer to its first element, and a pointer can be indexed like an array.
    - Example:

```
int *p, i[10];
p = i;
p[5] = 100; // Using index
*(p+5) = 100; // Using pointer arithmetic
```

POINTERS IN MULTIDIMENSIONAL ARRAYS

- A multidimensional array can be accessed using pointer arithmetic.
    - Example:

```
int a[10][10];
*(a + 0 * 10 + 4); // Equivalent to a[0][4]
```

- A 2D array can be treated as a pointer to an array of 1D arrays:

- Example:

```
int num[10][10];
void pr_row(int j) {
   int *p = (int *) &num[j][0]; // Pointer to first element in row
   for (int t = 0; t < 10; ++t) printf("%d ", *(p+t));
}
```

ARRAYS OF POINTERS

- Pointers can be stored in arrays. Example:

```
int *x[10];
x[2] = &var; // Assign address of var to the 3rd element
```

- To pass an array of pointers to a function:

```
void display_array(int *q[]) {
   for (int t = 0; t < 10; t++) printf("%d ", *q[t]);
}
```

RETURNING MULTIPLE VALUES FROM A FUNCTION

- **Using Pointers**: Modify variables via pointers.
  - Example:

```
void func(int *var1, int *var2, char *var3) {
   *var1 = 40;
   *var2 = 50;
   *var3 = 'X';
}
```

- **Using Arrays**: Return multiple values by modifying an array.
  - Example:

```
int * func(int *tempVar) {
   *tempVar = 40;
   *(tempVar + 1) = 50;
   *(tempVar + 2) = 60;
   return tempVar;
}
```