MAA OMWATI DEGREE COLLEGE HASSANPUR

(PALWAL)

NOTES

BSC 1st Sem

Computer Fundamentals and Problem Solving Using C

Unit-1

Problem Definition

Definition: Problem definition involves identifying and clearly stating the problem that needs to be solved by a program.
Key steps include:

1. Understanding the requirements (input, processing, output).
2. Breaking the problem into smaller, manageable parts.
3. Identifying constraints and edge cases.

Program Design

Program design is the process of planning a solution before coding. It involves:

1. **Defining Inputs and Outputs**:
    - What data will the program accept?
    - What results should it produce?
2. **Developing a Logical Flow**:
    - Identify steps to achieve the solution.
    - Structure these steps in a clear and logical order.
3. **Choosing a Programming Approach**:
    - Procedural or object-oriented, based on the problem.
4. **Representing the Plan**:
    - Use tools like algorithms, pseudocode, and flowcharts.

Debugging

Definition: Debugging is the process of identifying, analyzing, and removing errors in a program.

Steps in Debugging:

1. **Reproduce the Error**: Understand when and how the error occurs.

2. **Locate the Error**: Use debugging tools, logging, or step-by-step code analysis.
3. **Fix the Error**: Correct the faulty logic or syntax.
4. **Test the Fix**: Ensure that the error is resolved without introducing new ones.

Types of Errors in Programming

1. **Syntax Errors**:
    - Occur due to violations of language rules.
    - Example: Missing a semicolon or unmatched parentheses.
2. **Logical Errors**:
    - The program runs but produces incorrect results.
    - Example: Incorrect formula implementation.
3. **Runtime Errors**:
    - Occur during program execution.
    - Example: Division by zero or accessing an out-of-bounds array.
4. **Compilation Errors**:
    - Errors that prevent the program from being compiled.
    - Example: Incorrect function signatures.

Techniques of Problem Solving

1. Flowcharting

A **flowchart** is a graphical representation of a program's logic or process flow.
Symbols:

- **Oval**: Start/End.
- **Rectangle**: Process or operation.
- **Diamond**: Decision point.
- **Arrow**: Flow of control.

Advantages:

- Visual clarity of processes.
- Useful for debugging and understanding logic.

2. Algorithms

An **algorithm** is a step-by-step procedure to solve a problem.
Characteristics:

- Finiteness: Must terminate after a finite number of steps.
- Definiteness: Steps should be clear and unambiguous.
- Input/Output: Accepts inputs and produces outputs.
- Effectiveness: Each step must be basic enough to be executed.

Example:
Algorithm for Adding Two Numbers:

1. Start.
2. Input two numbers: A and B.
3. Calculate the sum: SUM = A + B.
4. Output the result: SUM.
5. End.

Overview of C Programming Language

History of C

- **Developed by**: Dennis Ritchie in 1972 at Bell Labs.
- **Evolution**:
    - Evolved from languages like **B** and **BCPL**.
    - Initially designed for system programming, particularly for developing the UNIX operating system.
- **Standardization**:

- Standardized by ANSI (American National Standards Institute) in 1989 (ANSI C).
- Further refined by ISO standards (ISO C).

Importance of C

1. **Foundation Language**: Serves as a base for many modern programming languages like C++, Java, Python, and more.
2. **Portability**: Programs written in C are highly portable across platforms.
3. **Performance**: Known for its efficiency and close-to-hardware functionality.
4. **Versatility**: Suitable for developing system software (e.g., operating systems) and application software (e.g., games).
5. **Rich Library Support**: Provides a wide range of built-in functions.

Elements of C

1. Character Set

The basic building blocks of C programs:

- **Letters**: A-Z, a-z.
- **Digits**: 0-9.
- **Special Characters**: +, -, *, /, %, @, #, etc.
- **White Spaces**: Space, tab, newline.
-

2. Identifiers and Keywords

- **Identifiers**:
  Names used for variables, functions, arrays, etc.
  - Must begin with a letter or an underscore (_).
  - Cannot use reserved keywords.

- **Keywords**:
  Predefined reserved words with specific meanings in C. Examples:
    - int, float, if, while, return, etc.

3. Data Types

C provides various data types to define variables:

1. **Basic Types**:
    - int, float, char, double.
2. **Derived Types**:
    - Arrays, pointers, functions.
3. **Enumeration Types**:
    - enum.
4. **Void**:
    - Represents "no type". Used for functions that do not return a value.

4. Constants and Variables

- **Constants**: Fixed values that do not change during program execution. Examples:
    - const int x = 5;
    - #define PI 3.14 (Symbolic Constant).
- **Variables**: Memory locations that store values and can change during execution.
    - Declared using a data type, e.g., int num;.

5. Assignment Statement

Used to assign values to variables. Example:

int a;

a = 10;

6. Symbolic Constants

Symbolic constants are defined using #define directive. Example:

#define PI 3.14159

Structure of a C Program

A C program typically has the following structure:

```
#include <stdio.h>  // Preprocessor directive

int main() {        // Main function
   // Variable declaration
   int a = 10, b = 20, sum;

   // Process
   sum = a + b;

   // Output
   printf("Sum: %d", sum);

   return 0;        // Return statement
}
```

I/O Functions: printf() and scanf()

- **printf()**: Outputs data to the screen. Example:

printf("Hello, World!");

printf("Sum = %d", sum);

- **scanf()**: Takes input from the user. Example:

int x;

scanf("%d", &x); // Reads an integer from the user

Operators and Expressions

Operators in C:

1. **Arithmetic Operators**: +, -, *, /, %.
2. **Relational Operators**: ==, !=, <, >, <=, >=.
3. **Logical Operators**: &&, ||, !.
4. **Bitwise Operators**: &, |, ^, ~, <<, >>.
5. **Assignment Operators**: =, +=, -=, *=, /=, %=.
6. **Unary Operators**: ++, --, sizeof, & (address-of).
7. **Ternary Operator**: ?:.

Expressions:

Combinations of operators and operands. Example:

int result = (a + b) * c;

Type Casting and Conversion

- **Type Casting**: Explicit conversion of one data type to another. Example:

float result = (float) a / b;

- **Type Conversion**: Implicit conversion performed by the compiler. Example:

float x = 5; // Automatically converted to 5.0

Operator Hierarchy and Associativity

- **Operator Precedence**: Determines the order in which operators are evaluated.
  Example:
  * and / have higher precedence than + and -.
- **Associativity**: Determines the direction of evaluation when operators have the same precedence.
  Example:
  - Left to Right: *, /, %.
  - Right to Left: Assignment operators (=).

Example:

```
int result = 10 + 5 * 2;  // Multiplication is evaluated first
```

Unit-2

Decision-Making and Looping in C Programming

C provides various control structures for decision-making, enabling programs to choose between different actions based on conditions.

1. if Statement

Executes a block of code if the condition is true.
Syntax:

```
if (condition) {
   // Code to execute if the condition is true
}
```

Example:

```
int x = 10;
if (x > 0) {
   printf("x is positive.");
}
```

2. if-else Statement

Adds an alternative block to execute if the condition is false.
Syntax:

```
if (condition) {
   // Code to execute if the condition is true
} else {
   // Code to execute if the condition is false
}
```

Example:

```
int x = -5;
if (x > 0) {
   printf("x is positive.");
} else {
   printf("x is not positive.");
}
```

3. Nested if Statement

Allows multiple levels of decision-making by placing one if inside another.

Syntax:

c

Copy code

```
if (condition1) {
   if (condition2) {
      // Code to execute if both conditions are true
   }
}
```

Example:

```
int x = 5, y = 10;
if (x > 0) {
   if (y > 0) {
      printf("Both x and y are positive.");
   }
}
```

4. else-if Ladder

Used when there are multiple conditions to check.

Syntax:

c

Copy code

```
if (condition1) {
   // Code for condition1
} else if (condition2) {
   // Code for condition2
} else {
   // Code if none of the conditions are true
}
```

Example:

```
int x = 0;
if (x > 0) {
   printf("x is positive.");
} else if (x < 0) {
   printf("x is negative.");
} else {
   printf("x is zero.");
}
```

5. switch Statement

Checks a variable against multiple constant values and executes a corresponding block.

Syntax:

```
switch (expression) {
   case value1:
      // Code for value1
      break;
   case value2:
      // Code for value2
      break;
   default:
      // Code if no case matches
}
```

Example:

```
int day = 3;
switch (day) {
   case 1: printf("Monday"); break;
   case 2: printf("Tuesday"); break;
   case 3: printf("Wednesday"); break;
   default: printf("Invalid day");
}
```

6. goto Statement

Transfers control to a labeled statement.

Syntax:

```
goto label;
...
label:
// Code here
```

Example:

```
int x = 10;
if (x > 0) {
   goto positive;
}
printf("x is not positive.");
positive:
printf("x is positive.");
```

Looping in C

Loops execute a block of code multiple times based on a condition.

1. while Loop

Executes as long as the condition is true.
Syntax:

```
while (condition) {
   // Code to execute
}
```

Example:

c

Copy code

```
int i = 1;
while (i <= 5) {
   printf("%d ", i);
   i++;
}
```

2. do-while Loop

Executes the code at least once before checking the condition.
Syntax:

```
do {
   // Code to execute
} while (condition);
```

Example:

```
int i = 1;
do {
   printf("%d ", i);
   i++;
} while (i <= 5);
```

3. for Loop

Used for a known number of iterations.
Syntax:

```
for (initialization; condition; increment/decrement) {
   // Code to execute
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
   printf("%d ", i);
}
```

Jumps in Loops

1. break Statement

Exits a loop immediately when executed.
Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        break;
    }
    printf("%d ", i);  // Output: 1 2
}
```

2. continue Statement

Skips the current iteration and proceeds to the next iteration. **Example**:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;
    }
    printf("%d ", i);  // Output: 1 2 4 5
}
```

Nested Loops

A loop inside another loop. **Example**:

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        printf("i = %d, j = %d\n", i, j);
    }
}
```

Output:

i = 1, j = 1

i = 1, j = 2

i = 2, j = 1

i = 2, j = 2

i = 3, j = 1

i = 3, j = 2

Unit-3

Functions, Arrays, Strings, and Pointers in C Programming

Functions in C

Functions are reusable blocks of code designed to perform a specific task. They help organize and modularize programs.

1. Standard Mathematical Functions

C includes several built-in mathematical functions available in the <math.h> library:

- **sqrt(x)**: Returns the square root of x.
- **pow(x, y)**: Returns x raised to the power y.
- **abs(x)**: Returns the absolute value of x.
- **sin(x), cos(x), tan(x)**: Return trigonometric values.
- **log(x)**: Returns the natural logarithm of x.
- **Example**:

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

int main() {
   double x = 9.0;
   printf("Square root: %.2f\n", sqrt(x));
   return 0;
}
```

2. Input/Output in C

Unformatted I/O Functions

- **getchar()**: Reads a single character from standard input.

- **putchar()**: Writes a single character to standard output.
- **Example**:

```
char c = getchar();
putchar(c);
```

Formatted I/O Functions

- **scanf()**: Reads formatted input.
- **printf()**: Writes formatted output.
- **Example**:

```
int num;
scanf("%d", &num);  // Input an integer
printf("Number: %d", num);
```

3. String Manipulation Functions

Available in <string.h>:

- **strcpy(dest, src)**: Copies string src to dest.
- **strlen(str)**: Returns the length of str.
- **strcmp(str1, str2)**: Compares two strings.
- **strcat(dest, src)**: Concatenates src to the end of dest.
- **Example**:

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

int main() {
   char str1[20] = "Hello";
   char str2[] = " World";
   strcat(str1, str2);
```

```
    printf("%s", str1);  // Output: Hello World
    return 0;
}
```

4. User-Defined Functions

Definition

Functions created by the programmer to perform specific tasks.

Function Prototype

Declares a function's name, return type, and parameters. **Example**:

```
int add(int, int);  // Function prototype
```

Local and Global Variables

- **Local Variables**: Declared inside a function; scope is limited to that function.
- **Global Variables**: Declared outside all functions; accessible throughout the program.

Passing Parameters

1. **Pass by Value**: Copies the value of the argument.
2. **Pass by Reference**: Uses pointers to allow modification of the actual variable.

Recursion

A function calls itself directly or indirectly. **Example**:

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Arrays in C

Arrays store multiple values of the same type.

1. Definition and Types

- **1D Array**: Stores a list of elements.
- **2D Array**: Stores a matrix of elements.
- **Multidimensional Arrays**: Arrays with more than two dimensions.

2. Initialization and Processing

- **Declaration and Initialization**:

```
int arr[5] = {1, 2, 3, 4, 5};
```

- **Processing**:

```
for (int i = 0; i < 5; i++) {
   printf("%d ", arr[i]);
}
```

3. Passing Arrays to Functions

Arrays are passed by reference.
Example:

```
void display(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
}
```

Strings in C

Strings are arrays of characters terminated by a null character (\0).

1. Declaration and Initialization

- **Declaration**:

```
char str[10];
```

- **Initializatio**

n: char str[] =

"Hello";

2. Input/Output of String Data

- **Input**:

scanf("%s",

str);

- **Output**:

printf("%s",

str);

Pointers in C

1. Definition

Pointers store the address of another variable.

2. Declaration and Initialization

- **Declaratio**

n: int *ptr;

- **Initializatio**

n: int x = 10;

int *ptr = &x; // Pointer stores the address of x

3. Accessing Values

- **Dereferencing**:

printf("%d", *ptr); // Accesses the value of x via ptr

4. Pointers and Arrays

- The name of an array acts as a pointer to its first element.

  Example:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;
printf("%d", *(ptr + 2));  // Outputs 3
```