

Diploma Engineering - Programming in C: Class Notes

UNIT I: Basics of C Programming

History of C Language

The C programming language has a rich history, evolving from earlier languages and becoming one of the most influential programming languages ever created.

- Early Beginnings (ALGOL, BCPL, B):
 - The roots of C can be traced back to ALGOL (Algorithmic Language), developed in the late 1950s, which introduced the concept of structured programming.
 - BCPL (Basic Combined Programming Language), developed by Martin Richards in 1967, was a typeless language intended for writing compilers and operating systems.
 - B language, developed by Ken Thompson at Bell Labs in 1969, was a simplified version of BCPL. It was used to create the first version of the UNIX operating system.
- Birth of C (1972):
 - Dennis Ritchie, also at Bell Labs, began developing C in 1972. He aimed to add data types to B and make it more powerful for system programming.
 - The primary motivation was to rewrite the UNIX operating system, which was initially written in assembly language. Writing an OS in a high-level language was revolutionary at the time.
- Evolution and Standardization:
 - K&R C (1978): Brian Kernighan and Dennis Ritchie published "The C Programming Language," which became the de-facto standard for C for many years. This version is often referred to as "K&R C."
- Legacy and Influence:
 - C became immensely popular for system programming (operating systems, embedded systems, compilers, databases) due to its efficiency and close-to-hardware access.
 - It profoundly influenced the development of many other modern programming languages, most notably C++, Java, JavaScript, C#, Python, and Go, which borrow heavily from C's syntax and concepts.

Features of C Language

C is a powerful and versatile language known for its efficiency, low-level access, and portability. Here are its key features:

1. Mid-level Language:

- C is often called a "middle-level" programming language.
- It combines the features of both high-level languages (like structured programming, readability) and low-level languages (like direct memory manipulation, bitwise operations).
- This makes it suitable for both system programming (close to hardware) and application programming.

2. Structured Programming Language:

- C supports structured programming concepts, allowing programs to be broken down into functions, loops, and conditional statements.
- This improves code readability, maintainability, and reusability, reducing the need for goto statements.

3. Rich Set of Library Functions:

- C comes with a rich collection of built-in functions (standard library) that perform common tasks such as input/output, string manipulation, mathematical operations, memory management, etc.
- These functions are organized into header files (e.g., `stdio.h`, `math.h`, `string.h`).

4. Portability:

- C programs are highly portable. Code written on one machine (with a C compiler) can often be compiled and run on different machines with minimal or no changes.
- This "write once, compile anywhere" philosophy (though not as absolute as Java) was a significant advantage.

5. Memory Management (Pointers):

- C provides direct access to memory through the use of pointers.
- This allows programmers to efficiently manage memory (allocate and deallocate dynamically) and interact directly with hardware, which is crucial for system-level programming.

6. Fast Execution Speed:

- C compilers generate highly optimized machine code.
- Its low-level capabilities and direct memory access contribute to its high execution speed, making it ideal for performance-critical applications.

7. Extensibility:

- C is an extensible language, meaning new features and user-defined functions can be easily added and integrated.
- You can create your own libraries and reuse them across different projects.

8. Case-Sensitive:

- C is a case-sensitive language.²⁵ `int` is different from `Int`, and `variable` is different from `Variable`.

9. Recursion:

- C supports recursion, where a function can call itself.²⁶ This is useful for solving problems that can be broken down into smaller, similar sub-problems (e.g., factorial, Fibonacci series).

10. Modularity:

- Programs can be divided into smaller, independent modules (functions).²⁷ These modules can be developed and tested separately and then combined to form a complete program.²⁸

11. Absence of Garbage Collection:

- Unlike some modern languages (e.g., Java, Python), C does not have automatic garbage collection.²⁹ Programmers are responsible for manually allocating and deallocating memory, which gives fine-grained control but also places a burden on the developer to prevent memory leaks.³⁰
-

1.1 Steps in Development of a Program, Flowcharts, and Algorithm Development

Steps in Development of a Program

Developing a computer program typically involves a series of structured steps. Following these steps helps ensure a well-designed, functional, and maintainable program.

1. Problem Definition/Analysis:

- Clearly understand what the program needs to achieve.
- Identify the inputs, outputs, and processing requirements.
- What data will the program need? What results should it produce?

2. Algorithm Development:

- An *algorithm* is a step-by-step procedure for solving a problem.
- It's a logical sequence of instructions that, when followed, will lead to the desired output.
- Algorithms can be expressed using natural language, pseudocode, or flowcharts.

3. Flowcharting (Optional but Recommended):

- A *flowchart* is a graphical representation of an algorithm.

- It uses standard symbols to depict the flow of control and operations.
- Helps visualize the program logic and identify potential errors before coding.

4. Coding (Implementation):

- Translate the algorithm (and/or flowchart) into a specific programming language (in this case, C).
- Write the source code using the syntax and rules of the chosen language.

5. Compilation/Interpretation:

- For compiled languages like C, the source code is translated into machine-readable code (executable file) by a *compiler*.
- The compiler checks for syntax errors.

6. Debugging:

- The process of finding and fixing errors (bugs) in the program.
- Errors can be syntax errors (caught by the compiler), logical errors (program doesn't do what it's supposed to), or runtime errors (errors that occur during execution).

7. Testing:

- Running the program with various inputs to ensure it produces the correct outputs for all expected scenarios, including edge cases.
- Verify that the program meets all the initial requirements.

8. Documentation:

- Writing comments within the code to explain its purpose and logic.
- Creating external documentation like user manuals, design specifications, and technical guides. This helps others understand and maintain the code.

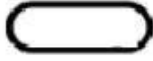
9. Maintenance:

- Ongoing process of updating and improving the program after deployment.
- Includes fixing new bugs, adding new features, and adapting to changes in the environment.

Flowcharts

A flowchart is a visual representation of the sequence of steps and decisions needed to perform a process. It uses standard symbols, each representing a specific type of operation or step in the process.

Common Flowchart Symbols:

Symbol	Symbol Name	Description
	Flow Lines	Used to connect symbols
	Terminal	Used to start, pause or halt in the program logic
	Input/output	Represents the information entering or leaving the system
	Processing	Represents arithmetic and logical instructions
	Decision	Represents a decision to be made
	Connector	Used to Join different flow lines
	Sub function	used to call function

Example Flowchart: To calculate the sum of two numbers.

A[Start] --> B(Input num1, num2)

B --> C{Sum = num1 + num2}

C --> D[Print Sum]

D --> E[End]

Algorithm Development

An algorithm is a finite set of unambiguous instructions that, when performed, accomplishes a specific task. It's language-independent and focuses on the logic.

Characteristics of a Good Algorithm:

- **Finiteness:** It must terminate after a finite number of steps.
- **Definiteness:** Each step must be precisely and unambiguously defined.
- **Input:** It must have zero or more well-defined inputs.
- **Output:** It must have one or more well-defined outputs.
- **Effectiveness:** Each step must be sufficiently basic that it can be carried out, at least in principle, by a person using pencil and paper.

Example Algorithm: To calculate the sum of two numbers.

1. **START**

2. Declare two variables, num1 and num2, to store the numbers.
3. Declare a variable sum to store the result.
4. Read the value of num1 from the user.
5. Read the value of num2 from the user.
6. Calculate $\text{sum} = \text{num1} + \text{num2}$.
7. Display the value of sum.
8. **END**

1.2 Structure and Syntax of C Programme, Basic Preprocessor Directives and Library Functions, Program Debugging, I/O Statements, Constants, Variables and Data Types

Structure and Syntax of a C Program

Every C program follows a basic structure. Understanding this structure is crucial for writing correct C code.

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

// Global variable declaration (optional)
int globalVar = 10;

// Function declaration (optional)
void myFunction();

int main() { // main function: Entry point of the program

    // Local variable declaration
    int a = 5;
    char myChar = 'A';

    // Executable statements
    printf("Hello, World!\n"); // Output statement
    myFunction(); // Function call
    return 0; // Indicates successful execution
}
```

```
// Function definition

void myFunction() {

    printf("This is inside myFunction.\n");

}
```

Explanation of Elements:

1. Preprocessor Directives (#include, #define, etc.):

- These are instructions to the C preprocessor, which is a program that processes the source code *before* it is passed to the compiler.
- They typically start with a # symbol.
- #include <stdio.h>: This directive tells the preprocessor to include the contents of the stdio.h (Standard Input/Output) header file into the program. This file contains declarations for standard input/output functions like printf() and scanf().

2. Global Variable Declaration (Optional):

- Variables declared outside any function are called global variables.
- They can be accessed from any function in the program.

3. Function Declaration (Prototype - Optional for main and if defined before use):

- A function declaration tells the compiler about a function's name, return type, and parameters *before* it's actually defined. This is necessary if a function is called before its definition in the code.

4. main() Function:

- Every C program **must** have a main() function.
- It is the entry point of the program. Execution always begins from main().
- int main(): Indicates that the main function returns an integer value.
- The code within the curly braces {} of main() is executed sequentially.

5. Local Variable Declaration:

- Variables declared inside a function (like a and myChar in main) are called local variables.
- They are only accessible within the function in which they are declared.

6. Executable Statements:

- These are the instructions that perform operations.
- Each statement in C typically ends with a semicolon (;).

7. **return 0;;**

- In the main function, return 0; indicates that the program executed successfully. A non-zero value typically indicates an error.

8. **Comments:**

- Used to explain the code. They are ignored by the compiler.
- Single-line comments: // This is a single-line comment.
- Multi-line comments: /* This is a multi-line comment. It can span across multiple lines. */

Basic Preprocessor Directives

Besides #include, another common preprocessor directive is #define.

- **#define:** Used to define macros. Macros are essentially text substitutions that happen before compilation.

C

```
#define PI 3.14159 // Defines a constant PI
```

```
#define MAX(a, b) ((a) > (b) ? (a) : (b)) // Defines a macro for finding the maximum of two numbers
```

```
int main() {  
    float circumference = 2 * PI * 5;  
    int result = MAX(10, 20);  
    printf("Circumference: %f\n", circumference);  
    printf("Max: %d\n", result);  
    return 0;  
}
```

Library Functions

C provides a rich set of built-in functions, known as *library functions*, grouped into various header files. These functions perform common tasks, saving you from writing them from scratch.

- **stdio.h (Standard Input/Output):**
 - printf(): For formatted output to the console.
 - scanf(): For formatted input from the console.
 - getchar(), putchar(), gets(), puts(): For character and string I/O.

- **math.h (Mathematical Functions):**
 - sqrt(): Calculates square root.
 - pow(): Calculates power.
 - sin(), cos(), tan(): Trigonometric functions.
- **stdlib.h (Standard Library):**
 - malloc(), free(): For dynamic memory allocation.
 - exit(): To terminate the program.
 - rand(), srand(): For random number generation.
- **string.h (String Manipulation):**
 - strlen(): Calculates string length.
 - strcpy(): Copies a string.
 - strcat(): Concatenates strings.
 - strcmp(): Compares strings.

Program Debugging

Debugging is the process of identifying and fixing errors (bugs) in a program.

Types of Errors:

1. **Syntax Errors:**
 - Violations of the grammar or rules of the C language.
 - Caught by the compiler during the compilation phase.
 - Examples: Missing semicolon, misspelled keywords, unmatched parentheses.
 - The compiler provides error messages that help pinpoint the location and type of error.
2. **Logical Errors:**
 - The program compiles and runs without errors, but produces incorrect or unexpected results.
 - The logic of the program is flawed.
 - Examples: Incorrect formula, wrong condition in an if statement, infinite loop.
 - Debugging logical errors often involves tracing the program's execution, printing intermediate values, or using a debugger.

3. Runtime Errors:

- Errors that occur during the execution of the program.
- Often lead to program crashes or abnormal termination.
- Examples: Division by zero, accessing an array out of bounds, memory leaks, stack overflow.
- These errors might not be caught by the compiler and only manifest during execution.

Debugging Techniques:

- **Read Compiler Error Messages:** They are your first clue for syntax errors.
- **printf() Debugging:** Insert printf() statements at various points in your code to print the values of variables and trace the program's flow. This is a simple but effective technique.
- **Using a Debugger:** Integrated Development Environments (IDEs) like Code::Blocks, Visual Studio Code (with C/C++ extensions), or specialized debuggers (like GDB) allow you to:
 - Set **breakpoints**: Pause execution at specific lines.
 - **Step through code**: Execute the program line by line.
 - **Inspect variables**: View the current values of variables.
 - **Watch expressions**: Monitor the value of an expression as the program runs.
- **Code Review:** Have another programmer review your code to spot errors.
- **Test Cases:** Create specific input scenarios (test cases) that are designed to reveal errors.

I/O Statements (Input/Output)

I/O statements are used to communicate with the user (or files).

- **printf() (Formatted Output):**
 - Used to display data on the console.
 - Syntax: `printf("format string", arg1, arg2, ...);`
 - **Format Specifiers:**
 - `%d` or `%i`: integer
 - `%f`: float
 - `%lf`: double
 - `%c`: character
 - `%s`: string
 - `%p`: pointer address

- %x or %X: hexadecimal
- %o: octal
- **Escape Sequences:**
 - \n: Newline
 - \t: Tab
 - \\: Backslash
 - \": Double quote
 - \': Single quote
 - \b: Backspace

Example:

```
int age = 25;
```

```
float height = 1.75;
```

```
char initial = 'J';
```

```
char name[] = "John"; // C-style string
```

```
printf("My name is %s, my initial is %c.\n", name, initial);
```

```
printf("I am %d years old and %.2f meters tall.\n", age, height); // .2f for 2 decimal places
```

- **scanf() (Formatted Input):**

- Used to read data from the console.
- Syntax: `scanf("format string", &variable1, &variable2, ...);`
- **Important:** Use the & (address-of) operator before variable names to pass their memory addresses, so scanf can store the input value directly into them.

Example:

```
int num1, num2;
```

```
float price;
```

```
printf("Enter two integers: ");
```

```
scanf("%d %d", &num1, &num2); // Read two integers
```

```
printf("Enter the price: ");
```

```
scanf("%f", &price); // Read a float
```

```
printf("You entered: %d, %d, %.2f\n", num1, num2, price);
```

Constants, Variables, and Data Types

Constants

Constants are fixed values that do not change during the execution of a program.

1. **Integer Constants:** Whole numbers (e.g., 10, -5, 0, 12345).
2. **Floating-point Constants:** Numbers with a decimal point or an exponent (e.g., 3.14, -0.001, 1.2e-5).
3. **Character Constants:** Single characters enclosed in single quotes (e.g., 'A', 'z', '5', '\n').
4. **String Literals:** A sequence of characters enclosed in double quotes (e.g., "Hello World", "C Programming"). String literals are actually arrays of characters terminated by a null character \0.
5. **const Keyword:** Used to declare a variable as a constant. Its value cannot be changed after initialization.

Example:

```
const float PI = 3.14159; // PI is a constant float
```

```
const int MAX_USERS = 100; // MAX_USERS is a constant integer
```

6. **#define Preprocessor Directive:** As discussed earlier, used to define symbolic constants.

Example:

```
#define PI_DEF 3.14159 // PI_DEF is a symbolic constant
```

Variables

Variables are named memory locations used to store data. Their values can change during program execution.

- **Declaration:** Before using a variable, you must declare it, specifying its data type.
 - `dataType variableName;`
 - `int age;`
 - `float salary;`
- **Initialization:** Assigning an initial value to a variable during declaration.
 - `int count = 0;`
 - `char grade = 'A';`

Data Types

Data types classify the kind of values a variable can hold and the operations that can be performed on them.

Basic Data Types in C:

1. **int (Integer):**

- Used to store whole numbers (positive, negative, or zero) without a decimal point.
- Typically occupies 2 or 4 bytes of memory, depending on the system/compiler.
- Range: Dependent on size (e.g., for 4 bytes, approximately $\pm 2 \times 10^9$).
- **Modifiers for int:**
 - short int: Smaller range, typically 2 bytes.
 - long int: Larger range, typically 4 or 8 bytes.
 - long long int: Even larger range, typically 8 bytes (C99 standard).
 - unsigned int: Stores only non-negative numbers, effectively doubling the positive range.
 - signed int: Stores both positive and negative numbers (default for int).

Example:

```
int score = 100;
```

```
unsigned int population = 1300000000;
```

```
long long int distance_to_moon = 384400000000LL;
```

2. **float (Floating-Point):**

- Used to store single-precision floating-point numbers (numbers with a decimal point).
- Typically occupies 4 bytes.
- Provides about 6-7 decimal digits of precision.

Example:

```
float temperature = 25.5f; // 'f' suffix is good practice for float literals
```

3. **double (Double Precision Floating-Point):**

- Used to store double-precision floating-point numbers.
- Typically occupies 8 bytes.

- Provides about 15-17 decimal digits of precision. Preferred for calculations requiring higher accuracy.

Example:

```
double pi_value = 3.1415926535;
```

4. char (Character):

- Used to store a single character.
- Internally, characters are stored as their ASCII (or other character set) integer values.
- Typically occupies 1 byte.

Example:

```
char grade = 'A';
```

```
char newline_char = '\n';
```

Size and Range (Illustrative - actual values depend on compiler/system):

Data Type	Bytes (typical)	Approximate Range	Format Specifier
char	1	-128 to 127 or 0 to 255 (signed/unsigned)	%c
short int	2	-32,768 to 32,767	%hd
int	4	-2,147,483,648 to 2,147,483,647	%d, %i
long int	4 or 8	Same as int or larger	%ld
long long int	8	$\pm 9 \times 10^{18}$	%lld
float	4	$\pm 3.4 \times 10^{38}$ (6-7 decimal digits)	%f
double	8	$\pm 1.7 \times 10^{308}$ (15-17 decimal digits)	%lf

You can use sizeof() operator to find the actual size of a data type on your system:

```
printf("Size of int: %zu bytes\n", sizeof(int));
```

1.3 Operators & Expressions, Unformatted and Formatted Input Output Statements, Data Type Casting

Operators & Expressions

An **operator** is a symbol that tells the compiler to perform specific mathematical or logical manipulations. An **expression** is a combination of operators and operands (variables, constants, function calls) that evaluates to a single value.

Types of Operators in C:

1. **Arithmetic Operators:** Perform mathematical calculations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	10 / 3	3 (integer division)
%	Modulo (Remainder)	10 % 3	1

- **Integer Division:** When both operands of / are integers, the result is an integer (truncates the decimal part).
- **Modulo Operator:** Works only with integer operands.

2. **Relational Operators:** Used to compare two operands. They return 1 (true) or 0 (false).

Operator	Description	Example (a=5, b=3)	Result
==	Equal to	a == b	0 (false)
!=	Not equal to	a != b	1 (true)
>	Greater than	a > b	1 (true)

Operator	Description	Example (a=5, b=3)	Result
<	Less than	a < b	0 (false)
>=	Greater than or equal to	a >= b	1 (true)
<=	Less than or equal to	a <= b	0 (false)

3. **Logical Operators:** Used to combine or negate boolean expressions.

Operator	Description	Example (a=1, b=0)	Result
&&	Logical AND	a && b	0 (false)
			Logical OR
!	Logical NOT	!a	0 (false)

- && returns true if both operands are true.
- || returns true if at least one operand is true.
- ! negates the truth value of an operand.
- In C, any non-zero value is considered true, and 0 is considered false.

4. **Assignment Operators:** Used to assign values to variables.

Operator	Description	Example	Equivalent to
=	Simple Assignment	x = 10	
+=	Add and Assign	x += 5	x = x + 5
-=	Subtract and Assign	x -= 2	x = x - 2
*=	Multiply and Assign	x *= 3	x = x * 3
/=	Divide and Assign	x /= 2	x = x / 2

Operator	Description	Example	Equivalent to
%=	Modulo and Assign	x %= 3	x = x % 3

5. **Increment and Decrement Operators:** Used to increase or decrease a variable's value by 1.

Operator	Description	Example	Explanation
++	Increment	x++ (postfix) ++x (prefix)	Adds 1 to x
--	Decrement	x-- (postfix) --x (prefix)	Subtracts 1 from x

- **Prefix (++x, --x):** Increments/decrements the value *before* the expression is evaluated.
- **Postfix (x++, x--):** Increments/decrements the value *after* the expression is evaluated.

Example:

```
int i = 5, j = 5;
```

```
int a, b;
```

```
a = ++i; // i becomes 6, a becomes 6
```

```
b = j++; // j becomes 6, b becomes 5
```

```
printf("a=%d, i=%d\n", a, i); // Output: a=6, i=6
```

```
printf("b=%d, j=%d\n", b, j); // Output: b=5, j=6
```

6. **Bitwise Operators:** Perform operations on individual bits of integer operands. (More advanced topic, but good to be aware of).

- & (Bitwise AND)
- | (Bitwise OR)
- ^ (Bitwise XOR)
- ~ (Bitwise NOT)
- << (Left Shift)
- >> (Right Shift)

7. Special Operators:

- `sizeof()`: Returns the size in bytes of a variable or data type.
- `&` (Address-of operator): Returns the memory address of a variable.
- `*` (Pointer dereference operator): Accesses the value at a memory address (used with pointers).
- `? :` (Ternary/Conditional operator): A shorthand for if-else.

Example:

```
// Ternary operator example
```

```
int num = 10;
```

```
char *result = (num > 5) ? "Greater" : "Smaller";
```

```
printf("%s\n", result); // Output: Greater
```

Operator Precedence and Associativity:

Operators have a specific order of evaluation (precedence) and direction of evaluation (associativity) when multiple operators are present in an expression. For example, multiplication and division have higher precedence than addition and subtraction. Parentheses () can be used to override precedence.

Unformatted and Formatted Input/Output Statements

We've already covered `printf()` and `scanf()` which are **formatted** I/O functions. They use format specifiers to interpret and display data.

Unformatted Input/Output Functions:

These functions are simpler and typically deal with single characters or strings without explicit format specifiers.

1. Character I/O:

- **`getchar()`**: Reads a single character from the standard input (keyboard). Returns an int (the ASCII value of the character) or EOF (End of File).

Example:

```
char ch;
```

```
printf("Enter a character: ");
```

```
ch = getchar();
```

```
printf("You entered: %c\n", ch);
```

- **`putchar()`**: Writes a single character to the standard output (screen). Takes an int argument (the ASCII value of the character).

Example:

```
char myChar = 'X';  
  
printf("Outputting a character: ");  
  
putchar(myChar);  
  
putchar('\n'); // Newline character
```

2. String I/O:

- **gets() (Avoid using in new code!):** Reads a string from the standard input until a newline character is encountered. It doesn't perform bounds checking, which can lead to buffer overflows (a security vulnerability).

Example

```
// DON'T USE THIS IN PRODUCTION CODE  
  
char name[50];  
  
printf("Enter your name: ");  
  
gets(name); // UNSAFE!  
  
printf("Hello, %s\n", name);
```

- **puts():** Writes a string to the standard output, automatically adding a newline character at the end.

Example:

```
char message[] = "Hello from puts!";  
  
puts(message); // Prints "Hello from puts!\n"
```

- **Safer Alternatives for String Input:**
 - **fgets():** Reads a string from a specified stream (e.g., stdin), with a size limit to prevent buffer overflows.

Example:

```
char safeName[50];  
  
printf("Enter your name (safe): ");  
  
// Read up to 49 characters + null terminator  
  
fgets(safeName, sizeof(safeName), stdin);  
  
// Remove trailing newline if present  
  
safeName[strcspn(safeName, "\n")] = 0;  
  
printf("Hello (safe), %s\n", safeName);
```

Data Type Casting (Type Conversion)

Data type casting is the process of converting a value from one data type to another.

1. Implicit Type Conversion (Automatic Conversion/Coercion):

- Performed automatically by the compiler when different data types are involved in an expression.
- Conversions happen from a "smaller" type to a "larger" type (promotion) to avoid loss of data.
- Example: int to float, float to double.

Example:

```
int i = 10;
```

```
float f = 3.5;
```

```
double d = i + f; // i (int) is implicitly converted to float, then added to f. Result is float, then promoted to double for d.
```

```
printf("d = %f\n", d); // Output: d = 13.500000
```

2. Explicit Type Conversion (Type Casting):

- Performed by the programmer using the cast operator (dataType).
- Allows you to force a conversion, even if it might involve data loss (e.g., float to int).

Example:

```
float numFloat = 10.75;
```

```
int numInt;
```

```
numInt = (int)numFloat; // Explicitly cast float to int. Decimal part is truncated.
```

```
printf("numInt = %d\n", numInt); // Output: numInt = 10
```

```
int numerator = 10;
```

```
int denominator = 3;
```

```
float result;
```

```
result = (float)numerator / denominator; // Cast numerator to float before division to get float result
```

```
printf("Result = %f\n", result); // Output: Result = 3.333333
```

```
// Without casting:
```

```
// result = numerator / denominator; // Integer division: result would be 3.000000
```

When to use type casting:

- To avoid integer division when one operand is an integer and you need a floating-point result.
- To convert between compatible but different data types when explicit control is needed.
- When passing arguments to functions that expect a specific type.

1.4 Decision Making with IF – Statement, IF – Else and Nested IF

Decision-making statements allow a program to execute different blocks of code based on certain conditions.

1. if Statement

The if statement is the simplest decision-making statement. It executes a block of code if a specified condition is true.

Syntax:

Example:

```
if (condition) {
```

```
    // Code to be executed if the condition is true
```

```
}
```

```
// Code outside the if block (always executes regardless of condition)
```

- The condition is an expression that evaluates to either true (non-zero) or false (zero).
- The curly braces {} define the block of code. If there's only one statement, the braces are optional but recommended for clarity.

Example:

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

```
int main() {
```

```
    int age = 18;
```

```
    if (age >= 18) {
```

```
        printf("You are eligible to vote.\n");
```

```
    }
```

```
    printf("Program continues here.\n");
```

```
    return 0;
}
```

2. if-else Statement

The if-else statement provides two paths of execution: one if the condition is true, and another if the condition is false.

Syntax:

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

Example:

```
#include <stdio.h>

int main() {
    int num;

    printf("Enter an integer: ");
    scanf("%d", &num);

    if (num % 2 == 0) {
        printf("%d is an even number.\n", num);
    } else {
        printf("%d is an odd number.\n", num);
    }

    return 0;
}
```

3. if-else if-else Ladder (Chained if-else)

This structure is used when you have multiple conditions to check, and each condition leads to a different action.

Syntax:

```

if (condition1) {
    // Code if condition1 is true
} else if (condition2) {
    // Code if condition1 is false AND condition2 is true
} else if (condition3) {
    // Code if condition1 and condition2 are false AND condition3 is true
} else {
    // Code if all preceding conditions are false
}

```

Example: Grading System

```

#include <stdio.h>

int main() {
    int score;

    printf("Enter your score: ");
    scanf("%d", &score);

    if (score >= 90) {
        printf("Grade: A\n");
    } else if (score >= 80) {
        printf("Grade: B\n");
    } else if (score >= 70) {
        printf("Grade: C\n");
    } else if (score >= 60) {
        printf("Grade: D\n");
    } else {
        printf("Grade: F\n");
    }

    return 0;
}

```

4. Nested if Statements

When an if or else if statement contains another if or if-else statement, it's called nesting. This allows for more complex decision logic.

Syntax:

```
if (condition1) {  
    // Outer if block  
    if (nested_condition1) {  
        // Code if condition1 AND nested_condition1 are true  
    } else {  
        // Code if condition1 is true AND nested_condition1 is false  
    }  
} else {  
    // Outer else block  
    if (nested_condition2) {  
        // Code if condition1 is false AND nested_condition2 is true  
    } else {  
        // Code if condition1 is false AND nested_condition2 is false  
    }  
}
```

Example: Checking Eligibility for a Ride (Age and Height)

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

```
int main() {  
    int age;  
    float height;  
    printf("Enter your age: ");  
    scanf("%d", &age);  
    printf("Enter your height in meters: ");  
    scanf("%f", &height);  
  
    if (age >= 12) {  
        if (height >= 1.5) {
```

```

    printf("You are eligible for the ride!\n");
} else {
    printf("You are old enough, but not tall enough for the ride.\n");
}
} else {
    printf("You are not old enough for the ride.\n");
}
return 0;
}

```

Important Considerations for if statements:

- **Semicolon after if:** Never put a semicolon immediately after the if (condition) or else statement, as it would terminate the statement, making the block of code always execute or be associated incorrectly.
 - if (condition); { /* code */ } is usually a bug. The semicolon makes the if statement an empty statement, and the block {} will always execute.
- **Braces {}:** Always use curly braces for code blocks, even if there's only one statement. This prevents logical errors when adding more statements later.
- **Boolean Values:** In C, 0 is considered false, and any non-zero value (typically 1) is true.

1.5 While and do-while, for loop, Break, Continue, goto and switch statements

Looping statements (or iteration statements) allow a block of code to be executed repeatedly as long as a certain condition is met.

1. while Loop

The while loop repeatedly executes a block of code as long as a given condition is true. The condition is checked *before* each iteration.

Syntax:

```

while (condition) {
    // Code to be executed repeatedly
    // (Must contain something that eventually makes the condition false)
}

```

Example: Print numbers from 1 to 5

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

```
int main() {
```

```

int i = 1; // Initialization
while (i <= 5) { // Condition
    printf("%d ", i);
    i++; // Update/Increment
}

printf("\n");

return 0;
}

```

2. do-while Loop

The do-while loop is similar to the while loop, but the code block is executed at least once *before* the condition is checked.

Syntax:

```

do {
    // Code to be executed repeatedly

    // (Must contain something that eventually makes the condition false)
} while (condition); // Don't forget the semicolon here!

```

Example: User input validation (gets at least one input)

```

#include <stdio.h>

int main() {
    int num;

    do {
        printf("Enter a positive number (or 0 to exit): ");
        scanf("%d", &num);

        if (num < 0) {
            printf("Invalid input! Please enter a positive number.\n");
        }
    } while (num < 0); // Loop continues if number is negative

    printf("You entered: %d\n", num);

    return 0;
}

```

3. for Loop

The for loop is typically used when you know in advance how many times you want to iterate, or when the loop's control variable has a clear starting point, ending point, and increment/decrement.

Syntax:

```
for (initialization; condition; update) {  
    // Code to be executed repeatedly  
}
```

- **Initialization:** Executed once at the beginning of the loop.
- **Condition:** Checked before each iteration. If true, the loop body executes. If false, the loop terminates.
- **Update:** Executed after each iteration of the loop body.

Example: Print numbers from 1 to 5

```
#include <stdio.h>  
  
int main() {  
    for (int i = 1; i <= 5; i++) {  
        printf("%d ", i);  
    }  
    printf("\n");  
    return 0;  
}
```

Nested for loops: One loop inside another, often used for working with 2D structures (like matrices) or patterns.

// Example: Print a 3x3 square of asterisks

```
for (int i = 0; i < 3; i++) { // Outer loop for rows  
    for (int j = 0; j < 3; j++) { // Inner loop for columns  
        printf("* ");  
    }  
    printf("\n"); // Move to the next line after each row  
}
```

4. break Statement

The break statement is used to immediately terminate the innermost loop (or switch statement) in which it appears. Program control resumes at the statement immediately following the loop/switch.

Example: Searching for a number

```
#include <stdio.h>

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int search_num = 30;
    int found = 0;
    for (int i = 0; i < 5; i++) {
        if (numbers[i] == search_num) {
            printf("%d found at index %d\n", search_num, i);
            found = 1;
            break; // Exit the loop once found
        }
    }
    if (!found) {
        printf("%d not found.\n", search_num);
    }
    return 0;
}
```

5. continue Statement

The continue statement is used to skip the rest of the current iteration of the loop and proceed to the next iteration.

Example: Print odd numbers from 1 to 10

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) { // If number is even
            continue; // Skip the rest of this iteration
        }
        printf("%d ", i);
    }
    printf("\n");
}
```

```

    }

    printf("%d ", i); // Only odd numbers will be printed
}

printf("\n");

return 0;
}

```

6. goto Statement (Avoid using if possible)

The goto statement is an unconditional jump statement that transfers program control to a specified label within the same function. While it exists in C, its use is generally discouraged as it can lead to spaghetti code, making programs hard to read, debug, and maintain.

Syntax:

```

goto label;

// ... some code ...

label:

    // Code to jump to

```

Example (Illustrative, not recommended):

```

#include <stdio.h>

int main() {
    int i = 1;

start_loop:
    if (i <= 5) {
        printf("%d ", i);

        i++;

        goto start_loop; // Jump back to start_loop
    }

    printf("\nLoop finished.\n");

    return 0;
}

```

Why goto is generally avoided:

- **Readability:** Makes code difficult to follow the flow of execution.
- **Maintainability:** Changes in one part of the code can have unexpected consequences due to distant jumps.
- **Debugging:** Tracing logic becomes much harder.
- Structured programming constructs (if, for, while, functions) almost always provide better and clearer alternatives.

7. switch Statement

The switch statement is a multi-way decision-making statement that allows a variable (or expression) to be tested for equality against a list of values. It provides an alternative to a long if-else if-else ladder when you are comparing a single value against multiple constant possibilities.

Syntax:

```
switch (expression) {  
    case constant1:  
        // Code to execute if expression == constant1  
        break; // Important! Exits the switch  
    case constant2:  
        // Code to execute if expression == constant2  
        break;  
    // ... more cases ...  
    default:  
        // Code to execute if no case matches (optional)  
        break; // Good practice, though not strictly necessary here  
}
```

- The expression must evaluate to an integer type (including char).
- case labels must be unique constant integer expressions.
- The break statement is crucial. Without it, execution will "fall through" to the next case label, executing its code as well (this is known as "fall-through behavior" and is often a source of bugs if not intended).
- The default case is optional and executes if none of the case values match the expression.

Example: Simple Calculator using switch

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

```
int main() {
```

```
    char operator;
```

```
    double n1, n2, result;
```

```
    printf("Enter an operator (+, -, *, /): ");
```

```
    scanf(" %c", &operator); // Note the space before %c to consume leftover newline
```

```
    printf("Enter two operands: ");
```

```
    scanf("%lf %lf", &n1, &n2);
```

```
    switch (operator) {
```

```
        case '+':
```

```
            result = n1 + n2;
```

```
            printf("%.2lf + %.2lf = %.2lf\n", n1, n2, result);
```

```
            break;
```

```
        case '-':
```

```
            result = n1 - n2;
```

```
            printf("%.2lf - %.2lf = %.2lf\n", n1, n2, result);
```

```
            break;
```

```
        case '*':
```

```
            result = n1 * n2;
```

```
            printf("%.2lf * %.2lf = %.2lf\n", n1, n2, result);
```

```
            break;
```

```
        case '/':
```

```
            if (n2 != 0) {
```

```
                result = n1 / n2;
```

```
                printf("%.2lf / %.2lf = %.2lf\n", n1, n2, result);
```

```
            } else {
```

```
                printf("Error: Division by zero is not allowed.\n");
```

```
            }
```

```
        break;
```

default:

```
printf("Error: Invalid operator.\n");
```

```
break;
```

```
}
```

```
return 0;
```

```
}
```

This concludes UNIT I of your C Programming class notes. Make sure to practice these concepts thoroughly by writing and running various programs.

Unit-2: Introduction to C Pointers

A Pointer in C language is a variable that holds a memory address. This memory address is the address of another variable(mostly) of same data type. In simple words, if one variable stores the address of second variable then the first variable can be said to **point towards** the second variable.

Before we start understanding what pointers are and what they can do, let's start by understanding what does "**Address of a memory location**" means?

What is a Memory Address in C?

Whenever a variable is defined in C language, a memory location is assigned for it, in which it's value gets stored. We can check this memory address, using the & symbol.

If *var* is the name of the variable, then *&var* will give it's address.

Let's write a small program to see memory address of any variable that we define in our program.

```
#include<stdio.h>

void main()
{
    int var = 7;
    printf("Value of the variable var is: %d\n", var);
    printf("Memory address of the variable var is: %x\n", &var);
}
```

Output:

```
Value of the variable var is: 7
Memory address of the variable var is: bcc7a00
```

Also while using the `scanf()` function, we mention `&var` to take user input for any variable `var`

```
scanf("%d", &var);
```

This is used to store the user input value to the memory address of the variable `var`.

What is a Pointer in C?

A Pointer in C language is a variable that holds a memory address. Pointers are used to access memory of a variable and manipulate the value stored in it.

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. Although pointers may appear a little confusing and complicated in the beginning, but trust me, once you understand the concept, you will be able to do so much more with C language.

This is used to store the user input value to the memory address of the variable var.

Whenever a **variable** is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we saw in the program above.

Let us assume that system has allocated memory location 80F for a variable a.

```
int a = 10;
```

We can access the value 10 either by using the variable name a or by using its address 80F.

The question is how we can access a variable using its address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**.

A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.

Pointer Variable in C

Like we mentioned above that a pointer is also a variable, but with a little twist, that is, it only stores address of other variables.

So if you have to define a pointer variable, the syntax is a little different.

Following is the **syntax** for declaring a variable as a pointer:

```
type *name;
```

Here, **type** is the data type of the pointer, and the **name** is the name of the pointer variable.

And the * operator with the name, informs the compiler that the variable is a pointer.

The data type of the pointer variable should be the same as of the variable to which the pointer is pointing.

Pointer Operators in C

There are two pointer operators in C, they are:

1. * operator
2. & operator

We have covered operators in C in detail separately.

The & operator returns the memory address of its operand. For example,

```
a = &b;
```

In the variable a the memory address of the variable b will get stored.

The * operators is the complement of &. This operator returns the value located at the given address.

For example, if a contains the memory address of the variable b, then the code,

```
c = *a; //will store the value of the variable b into c.
```

Example of Pointers in C

Let's see a basic code example where we will create a pointer and assign it a value.

```
#include<stdio.h>

int main(void)
{
    int x = 99;
    // declare a pointer
    int *ptr;
    // assign value to pointer
    ptr = &x;
    printf("Value at ptr is: %d \n", *ptr);
    printf("Address pointed by ptr is: %p \n", ptr);
    return 0;
}
```

Output:

Value at ptr is: 99

Address pointed by ptr is: 0x7fff99c0e6c4

The %p format specifier is used for pointer variable.

Dereference a Pointer

We have to first **dereference** the pointer to access the value present at the memory address. This is done with the help of **dereferencing operator(*)** (same operator used in declaration).

```
#include <stdio.h>

int main() {
    int var = 10;    // Store address of var variable
    int* ptr = &var; // Dereferencing ptr to access the value
    printf("%d", *ptr);
    return 0;
}
```

Output

10

Note: Earlier, we used %d for printing pointers, but C provides a separate format specifier **%p** for printing pointers.

Size of Pointers

The **size of a pointer in C** depends on the **architecture (bit system)** of the machine, **not the data type** it points to.

- On a **32-bit system**, all pointers typically occupy **4 bytes**.
- On a **64-bit system**, all pointers typically occupy **8 bytes**.

The size remains **constant regardless of the data type** (int*, char*, float*, etc.). We can verify this using the sizeof operator.

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

```
int main() {  
    int *ptr1;  
    char *ptr2; // Finding size using sizeof()  
    printf("%zu\n", sizeof(ptr1));  
    printf("%zu", sizeof(ptr2));  
    return 0;  
}
```

Output

10

Note: Earlier, we used %d for printing pointers, but C provides a separate format specifier **%p** for printing pointers.

Size of Pointers

The **size of a pointer in C** depends on the **architecture (bit system)** of the machine, **not the data type** it points to.

- On a **32-bit system**, all pointers typically occupy **4 bytes**.
- On a **64-bit system**, all pointers typically occupy **8 bytes**.

The size remains **constant regardless of the data type** (int*, char*, float*, etc.). We can verify this using the sizeof operator.

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

```
int main() {  
    int *ptr1;  
    char *ptr2; // Finding size using sizeof()  
    printf("%zu\n", sizeof(ptr1));  
    printf("%zu", sizeof(ptr2));  
}
```

```
    return 0;
}
```

Output

8

8

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

Note: The actual size of the pointer may vary depending on the **compiler and system architecture**, but it is always **uniform across all data types** on the same system.

Special Types of Pointers

There are 4 special types of pointers that used or referred to in different contexts:

NULL Pointer

The NULL Pointers are those pointers that do not point to any memory location. They can be created by assigning **NULL** value to the pointer. A pointer of any type can be assigned the NULL value.

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

```
int main() {
```

```
    // Null pointer
```

```
    int *ptr = NULL;
```

```
    return 0;
```

```
}
```

Output

10

Note: Earlier, we used %d for printing pointers, but C provides a separate [format specifier](#) %p for printing pointers.

Size of Pointers

The **size of a pointer in C** depends on the **architecture (bit system)** of the machine, **not the data type** it points to.

- On a **32-bit system**, all pointers typically occupy **4 bytes**.
- On a **64-bit system**, all pointers typically occupy **8 bytes**.

The size remains **constant regardless of the data type** (int*, char*, float*, etc.). We can verify this using the sizeof operator.

```
#include <stdio.h>

int main()
{
    int *ptr1;

    char *ptr2; // Finding size using sizeof()

    printf("%zu\n", sizeof(ptr1));

    printf("%zu", sizeof(ptr2));

    return 0;
}
```

Output

8

8

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

Note: *The actual size of the pointer may vary depending on the **compiler and system architecture**, but it is always **uniform across all data types** on the same system.*

Void Pointer

The [void pointers](#) in C are the pointers of type **void**. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

```
#include <stdio.h>

int main() {
    // Void pointer
    void *ptr;

    return 0;
}
```

Wild Pointers

The wild pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values are updated using wild pointers, they could cause data abort or data corruption.

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

```
int main()
```

```
{
```

```
    // Wild Pointer
```

```
    int *ptr;
```

```
    return 0;
```

```
}
```

Dangling Pointer

A pointer pointing to a memory location that has been deleted (or freed) is called a [dangling pointer](#). Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

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

```
#include <stdlib.h>
```

```
int main() {
```

```
    int* ptr = (int*)malloc(sizeof(int)); // After below free call, ptr becomes a dangling pointer
```

```
    free(ptr);
```

```
    printf("Memory freed\n"); // removing Dangling Pointer
```

```
    ptr = NULL;
```

```
    return 0;
```

```
}
```

Output

Memory freed

What is the use of Pointers in C?

Below we have listed a few benefits and use cases of using pointers:

1. Pointers are more efficient in handling Arrays in C and Structures in C.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. Pointers also provide means by which a function in C can change its calling arguments.
4. It reduces length of the program and its execution time as well.
5. It allows C language to support Dynamic Memory management.

FUNCTION

A function is a self contained block of codes or sub programs with a set of statements that perform some specific task or coherent task when it is called. It is something like to hiring a person to do some specific task like, every six months servicing a bike and hand over to it.

Any 'C' program contain at least one function i.e main().

There are basically two types of function those are

1. Library function

2. User defined function

The user defined functions defined by the user according to its requirement. System defined function can't be modified, it can only read and can be used. These function are supplied with every C compiler. Source of these library function are pre compiled and only object code get used by the user by linking to the code by linker.

Here in system defined function description:

Function definition : predefined, precompiled, stored in the library

Function declaration : In header file with or function prototype.

Function call : By the programmer

User defined function

Syntax:-

Return type name of function (type 1 arg 1, type2 arg2, type3 arg3)

Return type function name argument list of the above syntax

So when user gets his own function three thing he has to know, these are.

Function declaration

Function definition

Function call

These three things are represented like

```
int function(int, int, int); /*function declaration*/
```

```

main() /* calling function*/
{
function(arg1,arg2,arg3);
}
int function(type 1 arg 1,type2 arg2,type3, arg3) /*function definition/*
{
Local variable declaration;
Statement;
Return value;
}

```

Function declaration:-

Function declaration is also known as function prototype. It inform the compiler about three thing, those are name of the function, number and type of argument received by the function and the type of value returned by the function.

While declaring the name of the argument is optional and the function prototype always terminated by the semicolon.

Function definition:-

Function definition consists of the whole description and code of the function. It tells about what function is doing what are its inputs and what are its out put It consists of two parts function header and function body

Syntax:-

```

return type function(type 1 arg1, type2 arg2, type3 arg3) /*function header*/
{
Local variable declaration;
Statement 1;
Statement 2;
Return value
}

```

The return type denotes the type of the value that function will return and it is optional and if it is omitted, it is assumed to be int by default. The body of

the function is the compound statements or block which consists of local variable declaration statement and optional return statement.

The local variable declared inside a function is local to that function only. It can't be used anywhere in the program and its existence is only within this function.

The arguments of the function **definition** are known as **formal arguments**.

Function Call

When the function get called by the calling function then that is called, function call. The compiler execute these functions when the semicolon is followed by the function name.

Example:-

```
function(arg1,arg2,arg3);
```

The argument that are used inside the function call are called **actual argument**

Ex:-

```
int S=sum(a, b); //actual arguments
```

Actual argument

The arguments which are mentioned or used inside the function call is knows as actual argument and these are the original values and copy of these are actually sent to the called function. It can be written as constant, expression or any function call like

```
Function (x);
```

```
Function (20, 30);
```

```
Function (a*b, c*d);
```

```
Function(2,3,sum(a, b));
```

Formal Arguments

The arguments which are mentioned in function definition are called formal arguments or dummy arguments.

These arguments are used to just hold the copied of the values that are sent by the calling function through the function call.

These arguments are like other local variables which are created when the function call starts and destroyed when the function ends. The basic difference between the formal argument and the actual argument are

- 1) The formal argument are declared inside the parenthesis where as the local variable declared at the beginning of the function block.
- 2). The **formal argument** are automatically initialized when the copy of actual arguments are passed while other local variable are assigned values through the statements. Order number and type of actual arguments in the function call should be match with the order number and type of the formal arguments.

Return type

It is used to return value to the calling function. It can be used in two way as
Return Or return(expression);

Ex:- return (a);

return (a*b);

return (a*b+c);

Here the 1st return statement used to terminate the function without returning any value

Ex:- /*summation of two values*/

```
int sum (int a1, int a2);
```

```
main()
```

```
{
```

```
int a,b;
```

```
printf("enter two no");
```

```
scanf("%d%d",&a,&b);
```

```
int S=sum(a,b);
```

```
printf("summation is = %d",s);
```

```
}
```

```
int sum(intx1,int y1)
```

```
{
```

```
int z=x1+y1;
```

```
Return z;
```

```
}
```

Advantage of function

By using function large and difficult program can be divided into sub programs and solved. When we want to perform some task repeatedly or some code is to be used more than once at different place in the program, then function avoids this repetition or rewritten over and over.

Due to reducing size, modular function it is easy to modify and test

Notes:-

C program is a collection of one or more function.

A function is get called when function is followed by the semicolon.

A function is defined when a function name followed by a pair of curly braces

Any function can be called by another function even main() can be called by other function.

```
main()
{
function1()
}
function1()
{
Statement;
function2;
}
function 2()
{
}
```

So every function in a program must be called directly or indirectly by the main() function. A function can be called any number of times. A function can call itself again and again and this process is called **recursion**. A function can be called from other function **but** a function can't be defined in another function.

Call by value and call by reference

There are two way through which we can pass the arguments to the function such as **call by value** and **call by reference**.

1. Call by value

In the call by value copy of the actual argument is passed to the formal argument and the operation is done on formal argument.

When the function is called by 'call by value' method, it doesn't affect content of the actual argument.

Changes made to formal argument are local to block of called function so when the control back to calling function the changes made is vanish.

Example:-

```
main()
{
int x,y;
change(int,int);
printf("enter two values:¥n");
scanf("%d%d",&x,&y);
change(x ,y);
printf("value of x=%d and y=%d¥n",x ,y);
}

change(int a,int b);
{
int k;
k=a;
a=b;
b=k;
}
```

Output: enter two values: 12

23

Value of x=12 and y=23

2. Call by reference

Instead of passing the value of variable, address or reference is passed and the function operate on address of the variable rather than value.

Here formal argument is alter to the actual argument, it means formal arguments calls the actual arguments.

Example:-

```
void main()
{
int a,b;
change(int *,int*);
printf("enter two values:¥n");
scanf("%d%d",&a,&b);
change(&a,&b);
printf("after changing two value of a=%d and b=%d¥n:"a,b);
}

change(int *a, int *b)
{
int k;
k=*a;
*a=*b;
*b= k;
printf("value in this function a=%d and b=%d¥n",*a,*b);
}
```

Output: enter two values: 12 32

Value in this function a=32 and b=12

After changing two value of a=32 and b=12

So here instead of passing value of the variable, directly passing address of the variables. Formal argument directly access the value and swapping is possible even after calling a function.

Recursion

When function calls itself (inside function body) again and again then it is called as recursive function. In recursion calling function and called function are same. It is powerful technique of writing complicated algorithm in easiest way. According to recursion problem is defined in term of itself. Here

statement with in body of the function calls the same function and same times it is called as circular definition. In other words recursion is the process of defining something in form of itself.

Syntax:

```
main ()  
{  
  rec(); /*function call*/  
  rec();  
  rec();
```

Ex:- /*calculate factorial of a no.using recursion*/

```
int fact(int);  
void main()  
{  
  int num;  
  printf("enter a number");  
  scanf("%d",&num);  
  f=fact(num);  
  printf("factorial is =%d¥n",f);  
}  
fact (int num)  
{  
  If (num==0||num==1)  
  return 1;  
  else  
  return(num*fact(num-1));  
}
```

Monolithic Programming

The program which contains a single function for the large program is called monolithic program. In monolithic program not divided the program, it is huge long pieces of code that jump back and forth doing all the tasks like single thread of execution, the program requires. Problem arise in monolithic

program is that, when the program **size** increases it leads inconvenience and difficult to maintain such as testing, debugging etc. Many disadvantages of monolithic programming are:

1. Difficult to check error on large programs size.
2. Difficult to maintain because of huge size.
3. Code can be specific to a particular problem. i.e. it cannot be reused.

Many early languages (FORTRAN, COBOL, BASIC, C) required one huge workspace with labelled areas that may does specific tasks but are not isolated.

Modular Programming

The process of subdividing a computer program into separate sub-programs such as functions and subroutines is called Modular programming. **Modular programming sometimes also called as structured programming.** It enables multiple programmers to divide up the large program and debug pieces of program independently and tested. . Then the linker will link all these modules to form the complete program. This principle dividing software up into parts, or modules, where a module can be changed, replaced, or removed, with minimal effect on the other software it works with. Segmenting the program into modules clearly defined functions, it can determine the source of program errors more easily. Breaking down program functions into modules, where each of which accomplishes one function and contains all the source code and variables needed to accomplish that function. Modular program is the solution to the problem of very large program that are difficult to debug, test and maintain. A program module may be rewritten while its inputs and outputs remain the same. The person making a change may only understand a small portion of the original program.

Object-oriented programming (OOP) is compatible with the modular programming.

Function Advantages

- A single procedure can be developed for reuse, eliminating the need to retype the code many times.
- Programs can be designed more easily because a small team deals with only a small part of the entire code.
- Modular programming allows many programmers to collaborate on the same application.
- The code is stored across multiple files.
- Code is short, simple and easy to understand and modify, make simple to figure out how the program is operate and reduce likely hood of bugs.
- Errors can easily be identified, as they are localized to a subroutine or function or isolated to specific module.
- The same code can be reused in many applications.
- The scoping of variables and functions can easily be controlled.

Disadvantages

- Recursion can be slow, confusing, and can cause stack overflows. It also uses more memory because a function call is added to the call stack until it returns a value.
- Some dangerous functions in C, such as *strcpy*, *strcat*, *sprintf*, and *gets*, do not check bounds. If these functions are used, the user must ensure that the bounds will not be exceeded
- Cannot return multiple values.
- Memory and time overhead due to stack frame allocation and transfer of program control.

ARRAY

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string. Each data item of an array is called an element. And each element is unique and located in separated memory location. Each of elements of an array share a variable but each element having different index no. known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. One dimensional array is known as vector and two dimensional arrays are known as matrix.

ADVANTAGES: array variable can store more than one value at a time where other variable can store one value at a time.

Example:

```
int arr[100];
```

```
int mark[100];
```

DECLARATION OF AN ARRAY :

Its syntax is :

Data type array name [size];

```
int arr[100];
```

```
int mark[100];
```

```
int a[5]={10,20,30,100,5}
```

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte for each element etc. The size of the array operates the number of elements that can be stored in an array and it may be a int constant or constant int expression.

We can represent individual array as :

```
int ar[5];
```

```
ar[0], ar[1], ar[2], ar[3], ar[4];
```

Symbolic constant can also be used to specify the size of the array as:

```
#define SIZE 10;
```

INITIALIZATION OF AN ARRAY:

After declaration element of local array has garbage value. If it is global or static array then it will be automatically initialize with zero. An explicitly it can be initialize that

Data type array name [size] = {value1, value2, value3...}

Example:

in ar[5]={20,60,90, 100,120}

Array subscript always start from zero which is known as lower bound and upper value is known as upper bound and the last subscript value is one less than the size of array. Subscript can be an expression i.e. integer value. It can be any integer, integer constant, integer variable, integer expression or return value from functional call that yield integer value. So if i & j are not variable then the valid subscript are

ar [i*7],ar[i*i],ar[i++],ar[3];

The array elements are standing in continuous memory locations and the amount of storage required for hold the element depend in its size & type.

Total size in byte for 1D array is:

Total bytes=size of (data type) * size of array.

Example : if an array declared is:

int [20];

Total byte= 2 * 20 =40 byte.

ACCESSING OF ARRAY ELEMENT:

/*Write a program to input values into an array and display them*/

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

```
int main()
```

```
{
```

```
int arr[5],i;
```

```
for(i=0;i<5;i++)
```

```
{
```

```
printf("enter a value for arr[%d] \n",i);
```

```
scanf("%d",&arr[i]);
```

```
}
```

```

printf("the array elements are: \n");
for (i=0;i<5;i++)
{
printf("%d\t",arr[i]);
}
return 0;
}

```

OUTPUT:

Enter a value for arr[0] = 12

Enter a value for arr[1] =45

Enter a value for arr[2] =59

Enter a value for arr[3] =98

Enter a value for arr[4] =21

The array elements are 12 45 59 98 21

Example: From the above example value stored in an array are and occupy its memory addresses 2000, 2002, 2004, 2006, 2008 respectively.

a[0]=12, a[1]=45, a[2]=59, a[3]=98, a[4]=21

ar[0] ar[1] ar[2] ar[3] ar[4]

12 45 59 98 21

2000 2002 2004 2006 2008

Example 2:

/* Write a program to add 10 array elements */

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

```
void main()
```

```
{
```

```
int i;
```

```
int arr [10];
```

```
int sum=o;
```

```
for (i=0; i<=9; i++)
```

```
{
```

```
printf ("enter the %d element \n", i+1);
```

```
scanf ("%d", &arr[i]);
}
for (i=0; i<=9; i++)
{
sum = sum + a[i];
}
printf ("the sum of 10 array elements is %d", sum);
}
```

OUTPUT:

Enter a value for arr[0] =5

Enter a value for arr[1] =10

Enter a value for arr[2] =15

Enter a value for arr[3] =20

Enter a value for arr[4] =25

Enter a value for arr[5] =30

Enter a value for arr[6] =35

Enter a value for arr[7] =40

Enter a value for arr[8] =45

Enter a value for arr[9] =50

Sum = 275

while initializing a single dimensional array, it is optional to specify the size of array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

For example:-

```
int marks[]={99,78,50,45,67,89};
```

If during the initialization of the number the initializers is less then size of array, then all the remaining elements of array are assigned value zero .

For example:-

```
int marks[5]={99,78};
```

Here the size of the array is 5 while there are only two initializers so After this initialization, the value of the rest elements are automatically occupied by zeros such as

```
Marks[0]=99 , Marks[1]=78 , Marks[2]=0, Marks[3]=0, Marks[4]=0
```

Again if we initialize an array like

```
int array[100]={0};
```

Then the all the element of the array will be initialized to zero. If the number of initializers is more than the size given in brackets then the compiler will show an error.

For example:-

```
int arr[5]={1,2,3,4,5,6,7,8};//error
```

we cannot copy all the elements of an array to another array by simply assigning it to the other array like, by initializing or declaring as

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

```
int b[5];
```

```
b=a;//not valid
```

(**note**:-here we will have to copy all the elements of array one by one, using for loop.)

Single dimensional arrays and functions

```
/*program to pass array elements to a function*/
```

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

```
void main()
```

```
{
```

```
int arr[10],i;
```

```
printf("enter the array elements¥n");
```

```
for(i=0;i<10;i++)
```

```
{
```

```
scanf("%d",&arr[i]);
```

```
check(arr[i]);
```

```
}
```

```
}
```

```
void check(int num)
```

```

{
if(num%2=0)
{
printf("%d is even ¥n",num);
}
else
{
printf("%d is odd ¥n",num);
}
}

```

Two dimensional arrays

Two dimensional array is known as matrix. The array declaration in both the array i.e. in single dimensional array single subscript is used and in two dimensional array two subscripts are used.

Its syntax is Data-type array name[row][column]; Or we can say 2-d array is a collection of 1-D array placed one below the other.

Total no. of elements in 2-D array is calculated as **row*column**

Example:-

```
int a[2][3];
```

Total no of elements=row*column is $2*3 = 6$

It means the matrix consist of 2 rows and 3 columns

For example:-

20 2 7

8 3 15

Positions of 2-D array elements in an array are as below

00 01 02

10 11 12

a [0][0] a [0][1] a [0][2] a [1][0] a [1][1] a [1][2]

20 2 7 8 3 15

2000 2002 2004 2006 2008

Accessing 2-d array /processing 2-d arrays

For processing 2-d array, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

For example

```
int a[4][5];
```

for reading value:-

```
for(i=0;i<4;i++)
{
    for(j=0;j<5;j++)
    {
        scanf("%d",&a[i][j]);
    }
}
```

For displaying value:-

```
for(i=0;i<4;i++)
{
    for(j=0;j<5;j++)
    {
        printf("%d",a[i][j]);
    }
}
```

Initialization of 2-d array:

2-D array can be initialized in a way similar to that of 1-D array. for example:-

```
int mat[4][3]={11,12,13,14,15,16,17,18,19,20,21,22};
```

These values are assigned to the elements row wise, so the values of elements after this initialization are

Mat[0][0]=11, Mat[1][0]=14, Mat[2][0]=17 Mat[3][0]=20

Mat[0][1]=12, Mat[1][1]=15, Mat[2][1]=18 Mat[3][1]=21

Mat[0][2]=13, Mat[1][2]=16, Mat[2][2]=19 Mat[3][2]=22

While initializing we can group the elements row wise using inner braces.

for example:-

```
int mat[4][3]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};
```

And while initializing , it is necessary to mention the 2nd dimension where 1st dimension is optional.

```
int mat[][3];
```

```
int mat[2][3];
```

```
int mat[][];
```

```
int mat[2][]; invalid
```

If we **initialize an array** as

```
int mat[4][3]={{11},{12,13},{14,15,16},{17}};
```

Then the compiler will assume its all rest value as 0, which are not defined.

```
Mat[0][0]=11, Mat[1][0]=12, Mat[2][0]=14, Mat[3][0]=17
```

```
Mat[0][1]=0, Mat[1][1]=13, Mat[2][1]=15 Mat[3][1]=0
```

```
Mat[0][2]=0, Mat[1][2]=0, Mat[2][2]=16, Mat[3][2]=0
```

In memory map whether it is 1-D or 2-D, elements are stored in one contiguous manner. We can also give the size of the 2-D array by using symbolic constant Such as

```
#define ROW 2;
```

```
#define COLUMN 3;
```

```
int mat[ROW][COLUMN];
```

String

Array of character is called a string. It is always terminated by the NULL character. String is a one dimensional array of character.

We can initialize the string as

```
char name[]={'j','o','h','n','\0'};
```

Here each character occupies 1 byte of memory and last character is always NULL character. Where '\0' and 0 (zero) are not same, where **ASCII** value of '\0' is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation.

From the above we can represent as;

```
J o h N '\0'
```

The terminating NULL is important because it is only the way that the

function that work with string can know, where string end.

String can also be **initialized** as;

```
char name[]="John";
```

Here the NULL character is not necessary and the compiler will assume it automatically.

String constant (string literal)

A string constant is a set of character that enclosed within the double quotes and is also called a literal. Whenever a string constant is written anywhere in a program it is stored somewhere in a memory as an array of characters terminated by a NULL character (' ').

Example – "m"

"Tajmahal"

"My age is %d and height is %f n"

The string constant itself becomes a pointer to the first character in array.

Example-

```
char crr[20]="Taj mahal";
```

It is called base address.

Unit IV: Structure and Unions

UNION

Union is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place of memory.

Syntax of union:

```
union student
{
    datatype member1;
    datatype member2;
};
```

Like structure variable, union variable can be declared with definition or separately such as
union union name

```
{
    Datatype member1;
}var1;
```

Example:- union student s;

Union members can also be accessed by the dot operator with union variable and if we have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:- struct student

```
struct student
{
    int i;
    char ch[10];
};struct student s;
```

Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

Nested of Union

When one union is inside another union it is called nested of union.

Example:-

```
union a
{
    int i;
    int age;
};
union b
{
    char name[10];
    union a aa;
}; union b bb;
```

There can also be union inside structure or structure in union.

Example:-

```

void main()
{
struct a
{
int i;
char ch[20];
};
struct b
{
int i;
char d[10];
};
union z
{
struct a a1;
struct b b1;
}; union z z1;
z1.b1.j=20;
z1.a1.i=10;
z1.a1.ch[10]= " i";
z1.b1.d[0]="j ";
printf(" ");

```

Dynamic memory Allocation

The process of allocating memory at the time of execution or at the runtime, is called dynamic memory location. Two types of problem may occur in static memory allocation. If number of values to be stored is less than the size of memory, there would be wastage of memory. If we would want to store more values by increase in size during the execution on assigned size then it fails. Allocation and release of memory space can be done with the help of some library function called dynamic memory allocation function. These library function are called as **dynamic memory allocation function**. These library function prototype are found in the header file, "alloc.h" where it has defined. Function take memory from memory area is called heap and release when not required.

Pointer has important role in the dynamic memory allocation to allocate memory.

malloc():

This function use to allocate memory during run time, its declaration is

```
void*malloc(size);
```

malloc ()

returns the pointer to the 1st byte and allocate memory, and its return type is void, which can be type cast such as:

```
int *p=(datatype*)malloc(size)
```

If memory location is successful, it returns the address of the memory chunk that was allocated and it returns null on unsuccessful and from the above declaration a pointer of type(**datatype**) and size in byte.

And **datatype** pointer used to typecast the pointer returned by malloc and this typecasting is necessary since, malloc() by default returns a pointer to void.

Example `int*p=(int*)malloc(10);`

So, from the above pointer p, allocated IO contiguous memory space address of 1st byte and is stored in the variable.

We can also use, the size of operator to specify the size, such as

p=(int)malloc(5*size of int) Here, 5 is the no. of data.

Moreover , it returns null, if no sufficient memory available , we should always check the malloc return such as, **if(p==null)**

printf("not sufficient memory");

Example:

```
/*calculate the average of mark*/
void main()
{
    int n , avg,i,*p,sum=0;
    printf("enter the no. of marks ");
    scanf("%d",&n);
    p=(int *)malloc(n*size(int));
    if(p==null)
        printf("not sufficient");
    exit();
}
for(i=0;i<n;i++)
    scanf("%d",(p+i));
for(i=0;i<n;i++)
    Printf("%d",*(p+i));
sum=sum+*p;
avg=sum/n;
printf("avg=%d",avg);
```

calloc()

Similar to malloc only difference is that calloc function use to allocate multiple block of memory .two arguments are there

1st argument specify number of blocks

2nd argument specify size of each block.

Example:-

int *p= (int*) calloc(5, 2);

int*p=(int *)calloc(5, size of (int));

Another difference between malloc and calloc is by default memory allocated by malloc contains garbage value, where as memory allocated by calloc is initialized by zero(but this initialisation) is not reliable.

realloc()

The function realloc use to change the size of the memory block and it alter the size of the memory block without loosing the old data, it is called reallocation of memory.

It takes two argument such as;

int *ptr=(int *)malloc(size);

int*p=(int *)realloc(ptr, new size);

The new size allocated may be larger or smaller.

If new size is larger than the old size, then old data is not lost and newly allocated

bytes are uninitialized. If old address is not sufficient then starting address contained in pointer may be changed and this reallocation function moves content of old block into the new block and data on the old block is not lost.

Example:

```
#include<stdio.h>
#include<alloc.h>
void main()
int i,*p;
p=(int*)malloc(5*size of (int));
if(p==null)
{
printf("space not available");
exit();
printf("enter 5 integer");
for(i=0;i<5;i++)
{
scanf("%d",(p+i));
int*ptr=(int*)realloc(9*size of (int) );
if(ptr==null)
{
printf("not available");
exit();
}
printf("enter 4 more integer");
for(i=5;i<9;i++)
scanf("%d",(p+i));
for(i=0;i<9;i++)
printf("%d",*(p+i));
}
```

free()

Function free() is used to release space allocated dynamically, the memory released by free() is made available to heap again. It can be used for further purpose.

Syntax for free declaration .

void(*ptr)

Or

free(p)

When program is terminated, memory released automatically by the operating system. Even we don't free the memory, it doesn't give error, thus lead to memory leak.

We can't free the memory, those didn't allocated.

Dynamic array

Array is the example where memory is organized in contiguous way, in the dynamic memory allocation function used such as malloc(), calloc(), realloc() always made up of contiguous way and as usual we can access the element in two ways as:

Subscript notation

Pointer notation

Example:

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

```
#include<alloc.h>
void main()
{
printf("enter the no.of values");
scanf("%d",&n);
p=(int*)malloc(n*size of int);
If(p==null)
printf("not available memory");
exit();
}
for(i=0;i<n;i++)
{
printf("enter an integer");
scanf("%d",&p[i]);
for(i=0;i<n;i++)
{
printf("%d",p[i]);
}
}
}
```

Unit V: File Handling

File handling

File: the file is a permanent storage medium in which we can store the data permanently.

Types of file can be handled

we can handle three type of file as

(1) **sequential file**

(2) **random access file**

(3) **binary file**

File Operation

opening a file:

Before performing any type of operation, a file must be opened and for this `fopen()` function is used.

syntax:

```
file-pointer=fopen("FILE NAME ", "Mode of open");
```

example:

```
FILE *fp=fopen("ar.c", "r");
```

If `fopen()` unable to open a file than it will return `NULL` to the file pointer.

File-pointer: The file pointer is a pointer variable which can be store the address of a special file that means it is based upon the file pointer a file gets opened.

Declaration of a file pointer:-

```
FILE* var;
```

Modes of open

The file can be open in three different ways as

Read mode 'r'/rt

Write mode 'w'/wt

Appened Mode 'a'/at

Reading a character from a file

getc() is used to read a character into a file

Syntax:

```
character_variable=getc(file_ptr);
```

Writing a character into a file

putc() is used to write a character into a file

```
puts(character-var,file_ptr);
```

CLOSING A FILE

fclose() function close a file.

```
fclose(file_ptr);
```

fcloseall () is used to close all the opened file at a time

File Operation

The following file operation carried out the file

(1)creation of a new file

(3)writing a file

(4)closing a file

Before performing any type of operation we must have to open the file.c, language communicate with file using A new type called **file pointer**.

Operation with fopen()

File pointer=fopen(“FILE NAME”,”mode of open”);

If **fopen()** unable to open a file then it will return **NULL** to the file-pointer.

Reading and writing a characters from/to a file

fgetc() is used for reading a character from the file

Syntax:

character variable= fgetc(file pointer);

fputc() is used to writing a character to a file

Syntax:

fputc(character,file_pointer);

/*Program to copy a file to another*/

#include<stdio.h>

void main()

{

FILE *fs,*fd;

char ch;

If(fs=fopen(“scr.txt”,”r”)==0)

{

printf(“sorry....The source file cannot be opened”);

return;

}

If(fd=fopen(“dest.txt”,”w”)==0)

{

printf(“Sorry.....The destination file cannot be opened”);

fclose(fs);

return;

}

while(ch=fgets(fs)!=EOF)

fputc(ch,fd);

fcloseall();

}

Reading and writing a string from/to a file

getw() is used for reading a string from the file

Syntax:

gets(file pointer);

putw() is used to writing a character to a file

Syntax:

fputs(integer,file_pointer);

#include<stdio.h>

#include<stdlib.h>

void main()

{

FILE *fp;

int word;

/*place the word in a file*/

fp=fopen(“dgt.txt”,”wb”);

If(fp==NULL)

```

{
printf("Error opening file");
exit(1);
}
word=94;
putw(word,fp);
If(ferror(fp))
printf("Error writing to file\n");
else
printf("Successful write\n");
fclose(fp);
/*reopen the file*/
fp=fopen("dgt.txt","rb");
If(fp==NULL)
{
printf("Error opening file");
exit(1);
}
/*extract the word*/
word=getw(fp);
If(ferror(fp))
printf("Error reading file\n");
else
printf("Successful read:word=%d\n",word);
/*clean up*/
fclose(fp);
}

```

Reading and writing a string from/to a file

fgets() is used for reading a string from the file

Syntax:

fgets(string, length, file pointer);

fputs() is used for writing a character to a file

Syntax:

fputs(string,file_pointer);

#include<string.h>

#include<stdio.h>

void main(void)

{

FILE*stream;

char string[]="This is a test";

char msg[20];

/*open a file for update*/

stream=fopen("DUMMY.FIL","w+");

/*write a string into the file*/

fwrite(string,strlen(string),1,stream);

/*seek to the start of the file*/

fseek(stream,0,SEEK_SET);

```
/*read a string from the file*/  
fgets(msg,strlen(string)+1,stream);  
/*display the string*/  
printf("%s",msg);  
fclose(stream);  
}
```

BOOKS:

- 1 E.Balagurusamy “Programming in C”. Tata McGraw Hill
 - 2 Y. Kanetkar “Let Us C”. BPB publication
 - 3 Ashok N. Kamthane “Programming with ANSI and TURBO C”. Pearson Education
 - 4 Programming in C, a complete introduction to the programming language, Stephan G. Kocham, third edition
 - 5 C in Depth, S.K Srivastava and Deepali Srivastava
- 127 *Under revision