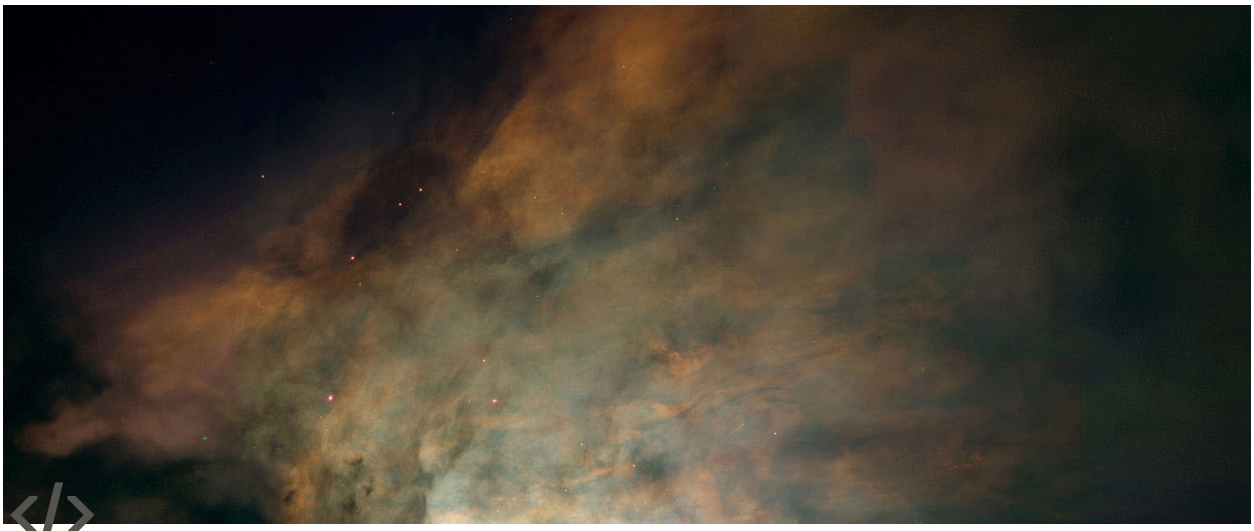


C Language Crash Course - MassCoders - Dodagatta Nihar

Learn C in Telugu By MassCoders - DodagattaNihar

⚠ Make sure you have Subscribed me on YouTube and followed me on Instagram if you like my efforts :)
Below are the reference links to our social media handles. Join us on our
Telegram or WhatsApp channel for real time updates.

YouTube → Click Here

Instagram → Click Here

LinkedIn → Click Here

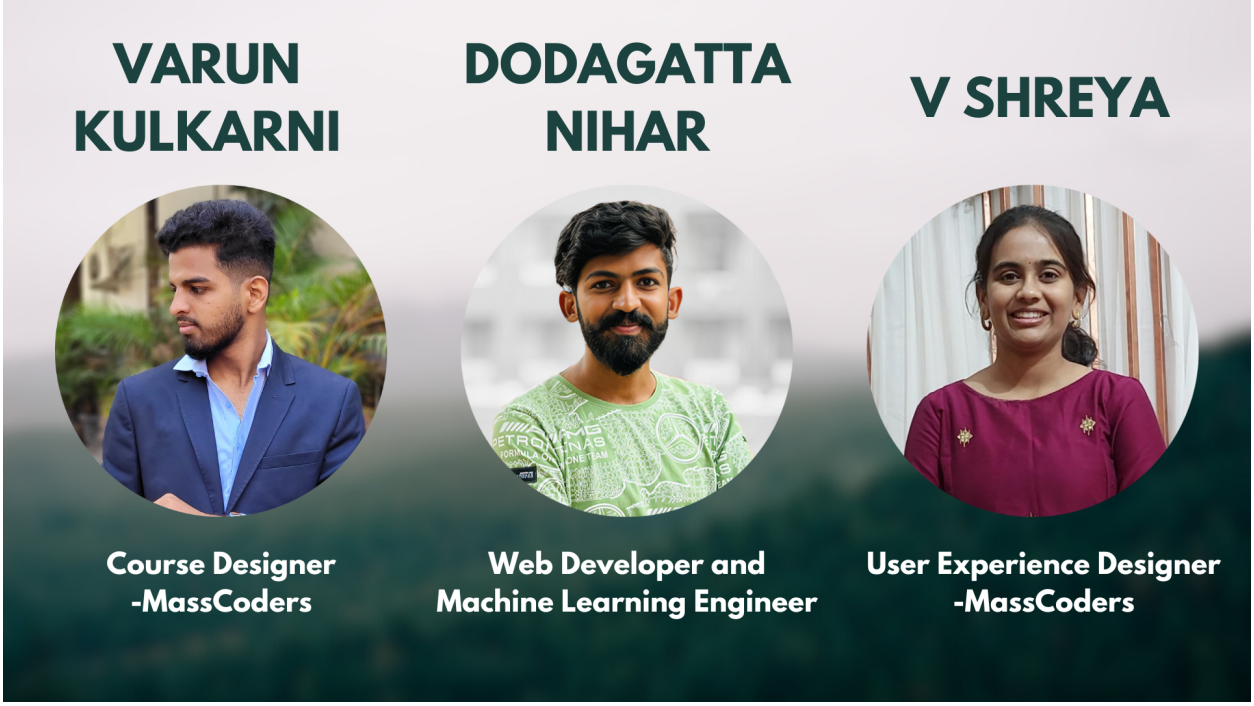
Telegram → Click Here

WhatsApp → Click Here

Discord → Click Here (Lots of Community Events Happening here, Join in to level up your career)

Pro Tip: Stay CONSISTENT everyday!🔥

Meet Varun and Shreya who put in their soul to make this notes possible.



Press on the arrow mark below of individual topics to access detailed notes.

▼ Setup of C

1. Install Visual Studio Code:

- Download and install Visual Studio Code from the official website: Visual Studio Code Download Page.
- and click on this link and follow the instructions: https://code.visualstudio.com/docs/languages/cpp

2.2: Install GCC Compiler:(iOS)

- Using Xcode Command Line Tools

1. **Install Xcode:**
    - Open the App Store on your Mac.
        - Search for "Xcode" and click on the "Get" button to download and install Xcode.
        - Once the installation is complete, open Xcode, and agree to the license terms.

1. **Install Xcode Command Line Tools:**
    - Open Terminal, which you can find using Spotlight (`Cmd + Space` and then type "Terminal").
    - Run the following command:

      ```
      xcode-select --install
      ```

    - A dialog will appear asking if you want to install the Command Line Tools. Click "Install."

2. **Verify GCC Installation:**
    - After the installation is complete, you can verify the installation by running:

```
gcc --version
```

- This command should display information about the installed GCC version.

3. Install C/C++ Extension for Visual Studio Code:

- Open VSCode.
- Go to the Extensions view (icon on the sidebar or `Ctrl+Shift+X`).
- Search for "C/C++" and click Install. This extension provides IntelliSense, debugging, and C/C++ language support.

4. Configure VSCode for C Programming:

- Create or open a C file.
- VSCode will prompt you to install recommended extensions. Click "Install All."
- If not prompted, you can still install the recommended extensions:
  - Open the Command Palette (`Ctrl+Shift+P`) and type "C/C++: Install/update language server."
  - Choose the version to install.

5. Verify the Setup:

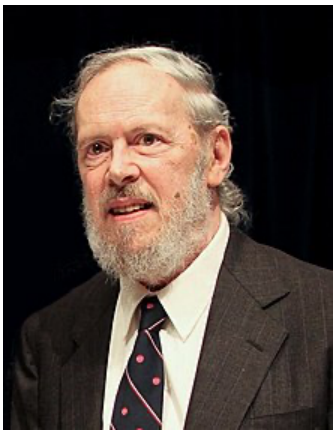
- Write a simple C program in VSCode.

```
#include <stdio.h>

int main() {
    printf("Excited to learn C from Masscoders\n");
    return 0;
}
```

- Save the file with a `.c` extension.
- Right-click in the editor and select "Run Code" or use the shortcut (`Ctrl+Alt+N`) to compile and run the program.

▼ Introduction to C

Founder of C - Dennis Ritchie - in 1970s



Introduction to C

C is a procedural, general-purpose programming language designed for system programming, emphasizing efficiency, portability, and low-level memory manipulation. Developed to provide a flexible and concise way to program various computing platforms, C has become a foundation for numerous modern programming languages.

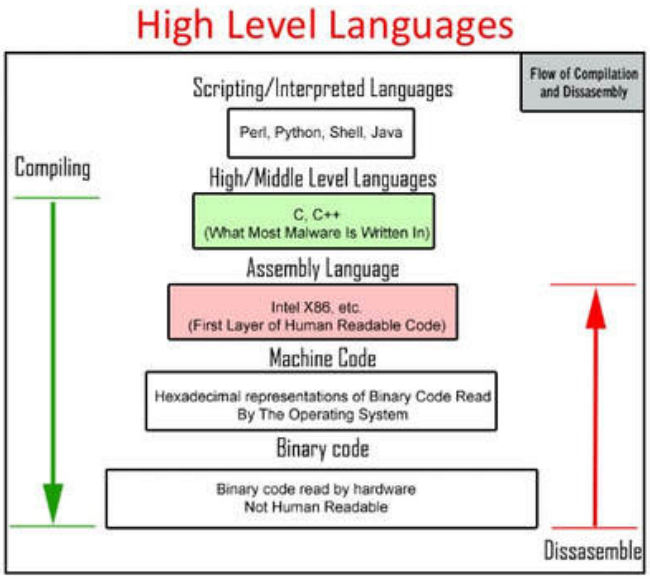
Key Technical Characteristics:

1.Procedural Paradigm:

- C follows a procedural programming paradigm, organizing code into functions that carry out specific tasks. This approach allows for modular and structured program design.

2.Efficiency and Performance:

- C provides direct access to memory and hardware, enabling developers to write code that executes efficiently. Its low-level features facilitate close control over system resources, making it suitable for performance-critical applications.



3.Portability:

- Designed with portability in mind, C allows programs written in the language to be easily adapted to different platforms and architectures with minimal modifications. This portability contributes to the language's enduring popularity.

4.Structured Programming:

- C supports structured programming through features like functions, loops, and conditionals. This aids in creating well-organized, modular, and maintainable code.

5.Pointer and Memory Management:

- Pointers, a fundamental aspect of C, provide direct memory access and allow efficient memory manipulation. This feature is crucial for tasks such as dynamic memory allocation and resource management.

6.Standard Library:

- C comes with a standard library that includes a set of functions for common tasks, ranging from input/output operations to string manipulation. This library enhances code reusability and simplifies development.

7.Low-Level Features:

- C provides low-level features, allowing for precise control over hardware and memory. This makes it well-suited for system-level programming, where direct interaction with hardware and efficient resource utilization is essential.

8.Influence on Modern Languages:

- Many modern programming languages, including C++, C#, and Objective-C, have been influenced by C. Learning C not only provides a foundation for understanding these languages but also exposes developers to fundamental concepts in software development.

Why should on learn C language?

Learning the C programming language offers several compelling reasons, making it a valuable and foundational language for aspiring programmers. Here are some key reasons why one should consider learning C:

1. Foundation for Programming:

- C serves as a fundamental language that provides a solid foundation for understanding programming concepts. It exposes learners to essential principles such as variables, data types, control structures, and functions.

2. System Programming:

- C is a go-to language for system programming tasks, including operating system development, kernel programming, and firmware development. Its low-level features enable direct interaction with hardware.

3. Embedded Systems:

- Many embedded systems, found in devices like microcontrollers and IoT devices, are programmed in C due to its efficiency and ability to handle low-level operations.

4. Influence on Other Languages:

- Learning C provides insights into the development of modern programming languages. Many popular languages, including C++, C#, and Objective-C, have been influenced by C. This knowledge facilitates easier transition to other languages.

5. Understanding Memory Management:

- C requires manual memory management, involving concepts like pointers and dynamic memory allocation. This hands-on experience provides a deep understanding of memory handling, which is valuable in programming languages with automatic memory management.

6. Versatility and Flexibility:

- C's versatility allows it to be used in various application domains, from system-level programming to high-level application development. Its flexibility makes it suitable for a wide range of projects.

7. Problem-Solving Skills:

- Programming in C often involves solving problems at a lower level, enhancing problem-solving skills and fostering a deeper understanding of how software interacts with hardware.

8. Career Opportunities:

- Proficiency in C is highly valued in industries such as systems programming, embedded systems, game development, and cybersecurity. Learning C can open doors to diverse and rewarding career opportunities.

How is a C source code compiled ?

1. Text Editing:

- Start with a C source code file (e.g., `example.c`) written in a text editor. This file contains the human-readable C code that defines the program's logic.

2. Preprocessing:

- The preprocessor (`cpp`) is invoked to handle preprocessing tasks, such as:
  - Expanding macros defined with `#define`.
  - Including header files using `#include`.
  - Removing comments.
  - Handling conditional compilation with `#ifdef`, `#ifndef`, `#else`, `#endif`.
- The result is a modified source code file without preprocessor directives.

3. Compilation:

- The modified source code is passed to the compiler (`gcc`, for example), which translates it into assembly code or an intermediate representation.
- The compiler checks for syntax errors, type errors, and performs optimization.

4. Assembly:

- The assembler (`as`) takes the intermediate representation or assembly code and translates it into machine code specific to the target architecture.
- This results in one or more object files (`.o` or `.obj files), each corresponding to a source code file.

5. Linking:

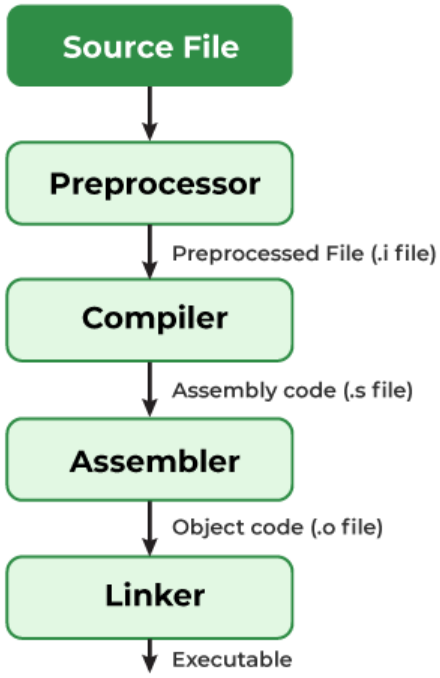
- The linker (`ld`) combines the object files with external libraries and produces the final executable file.
- It resolves addresses and symbols, linking together different parts of the program.
- If external libraries are used, the linker links the code with the necessary library files.

6. Output:

- The final output is an executable file (e.g., `.out`, `.exe`), which contains machine code that the computer's hardware can execute.
- Optionally, intermediate files (e.g., object files) may be retained for debugging or analysis purposes.

7. Execution:

- The user can run the compiled executable, and the program's instructions are executed by the computer's hardware.
- The program produces the desired output based on its logic.



▼ Basic structure of C and Syntax

Basic Structure of a C Program:

1. **Documentation section:**
   - Comments are non-executable text that are ignored by the compiler. They serve as notes for programmers to make the code more readable and understandable.
   - C supports two types of comments: single-line comments starting with `//` and multi-line comments enclosed within `/* */`.
2. **Preprocessor Directives:**
   Preprocessor directives are commands that are processed by the preprocessor before the actual compilation of code starts. They start with a hash symbol (#). Common directives include:
   - **Link section :** `#include` Used to include header files that provide declarations and definitions of functions and macros.
   - **Definition section** `#define` Used to define macros, which are symbolic constants or short functions

   Example:

   ```
   #include <stdio.h>
   #define MAX_SIZE 100
   ```

3. **Global declaration section:**
   There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This

section also declares all the
user-defined functions.

```
//syntax
//data_type  variable_name;
//done before main function
// example
int globalvar;
```

4. **Main Function:**
   The main function is the entry point of a C program. Every C program must have one main function, which returns an integer indicating the exit status of the program.

```
int main() {
    // Code goes here
    return 0;
}
```

5. **Variable Declarations:**
   Variables are declared to reserve memory space for storing data during program execution. In C, variables must be declared before they are used. Declarations typically appear at the beginning of a function, although they can appear anywhere within a block.

```
int main() {
    int num1, num2, sum;
    // Code goes here
    return 0;
}
```

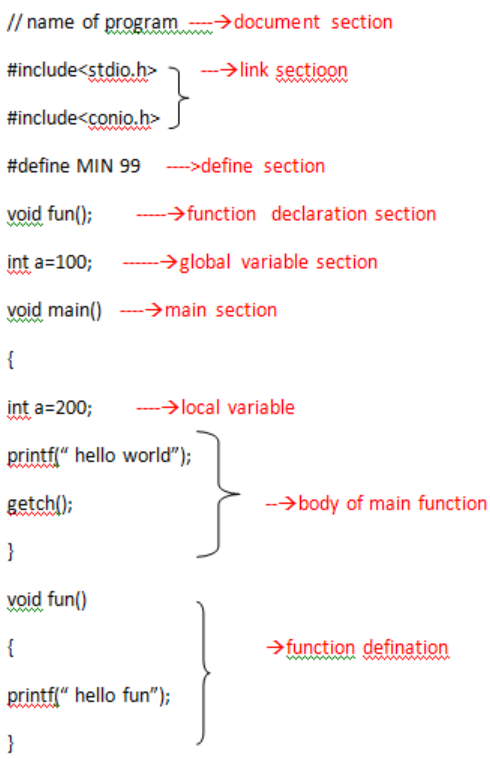
6. **Executable Statements:**
   These are the actual instructions or operations that the program will perform. Executable statements include assignments, function calls, control flow statements (such as if-else statements and loops), and other operations.

```
int main() {
    int num1, num2, sum;

    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    sum = num1 + num2;
```

```
    printf("Sum: %d\n", sum);

    return 0;
}
```

7. **Subprograms section:**

   If the program is a multi-function program then the subprogram section contains all the user-defined functions that are called in the main () function. User defined functions are generally placed immediately after the main () function, although they may appear in any order.



Example:

```
int main() {
    int num1, num2, sum;

    // Prompt the user to enter two numbers
    printf("Enter two numbers: ");

    // Read the input from the user
    scanf("%d %d", &num1, &num2);

    // Calculate the sum of the two numbers
    sum = num1 + num2;
```

```
    // Display the result
    printf("Sum: %d\n", sum);

    return 0;
}
```

▼ Data Types in C

Data Types:

In C, data types define the type of data that a variable can hold. Here are some commonly used data types:

1. **Integer Types:**
   - `int`: Represents integers (whole numbers).
   - `short`, `long`: Variations of `int` with different sizes.
   - Typically uses 4 bytes on most systems (32 bits)
   - Range: -2,147,483,648 to 2,147,483,647 for signed `int`.

```
int age = 25;
```

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

int main() {
    printf("Range of int: %d to %d\n", INT_MIN, INT_MAX);
    return 0;
}
```

2. **Floating-Point Types:**
   - `float`: Represents single-precision floating-point numbers.
   - `double`: Represents double-precision floating-point numbers.
   - Uses 4 bytes (32 bits).
   - Represents single-precision floating-point numbers.

```
float price = 19.99;
```

3. **Character Type:**
   - `char`: Represents single characters.
   - Uses 1 byte (8 bits).
   - Represents a single ASCII character.

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

4. **Void Type:**
   - `void`: Represents the absence of a type. Used in function return types where no value is returned.

```
void printMessage() {
    printf("Hello!\n");
}
```

5. **Double Type:**
   - `double`: Represents double-precision floating-point numbers, providing higher precision than `float`.
   - Uses 8 bytes (64 bits).
   - Represents double-precision floating-point numbers.

```
double preciseValue = 123.456789012345;
```

6. **Derived Types:**
   - `arrays`, `pointers`, `structures`, `unions`, and `enumerations` are derived types that allow more complex data structures.

The memory used by each data type in C can vary based on the architecture and compiler being used. However, I can provide general guidelines for common architectures like x86 and x64.

Key Points:

- The sizes mentioned are based on common architectures, but they can vary.
- Size may change on different systems and compilers.

Additional Information:

- The `sizeof` operator in C can be used to determine the size of a data type in bytes. For example:

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

It's important to be aware of the memory usage of data types, especially when working with large datasets or in memory-constrained environments. The sizes mentioned here are common, but variations exist, and programmers should consider the specific characteristics of their target platform and compiler.

Variables:

Variables are used to store and manipulate data in a C program. Here are key points about variables:

1. **Declaration and Initialization:**
   - Variables must be declared before use, specifying their data type.
   - Initialization is assigning an initial value to a variable during declaration.

```
int count;
float temperature = 98.6;
```

2. **Scope:**
   - The scope of a variable defines where it can be accessed in the program.
   - Local variables are declared within functions and have limited scope.
   - Global variables are declared outside functions and can be accessed throughout the program.
3. **Constants:**
   - Constants are fixed values and are declared using `const`.

```
const int MAX_VALUE = 100;
```

▼ Operators

Operators in C are symbols that perform operations on operands. Here's an overview of different types of operators in C:

1. Arithmetic Operators:

- Perform basic mathematical operations.

S.no	Arithmetic Operators	Operation	Example
1	+	Addition	A+B
2	–	Subtraction	A-B
3	*	multiplication	A*B
4	/	Division	A/B
5	%	Modulus	A%B

```
int a = 5, b = 2;
int sum = a + b;    // Addition
int difference = a - b;   // Subtraction
```

```
int product = a * b;   // Multiplication
int quotient = a / b;   // Division
int remainder = a % b;  // Modulus (remainder)
```

2. Relational Operators:

- Compare two values and return a boolean result.

S.no	Operators	Example	Description
1	>	x > y	x is greater than y
2	<	x < y	x is less than y
3	>=	x >= y	x is greater than or equal to y
4	<=	x <= y	x is less than or equal to y
5	==	x == y	x is equal to y
6	!=	x != y	x is not equal to y

```
int x = 10, y = 5;
printf("%d\n", x > y);   // Greater than
printf("%d\n", x < y);   // Less than
printf("%d\n", x == y);  // Equal to
printf("%d\n", x != y);  // Not equal to
```

3. Logical Operators:

- Perform logical operations on boolean values.

S.no	Operators	Name	Example	Description
1	&&	logical AND	(x>5)&&(y<5)	It returns true when both conditions are true
2	||	logical OR	(x>=10)||(y>=10)	It returns true when at-least one of the condition is true
3	!	logical NOT	!((x>5)&&(y<5))	It reverses the state of the operand –((x>5) && (y<5))‖ If –((x>5) && (y<5))‖ is true, logical NOT operator makes it false

```
int p = 1, q = 0;
printf("%d\n", p && q);  // Logical AND
printf("%d\n", p || q);  // Logical OR
printf("%d\n", !p);       // Logical NOT
```

4. Assignment Operators:

- Assign values to variables.

Operators		Example	Explanation
Simple assignment operator	=	sum = 10	10 is assigned to variable sum
Compound assignment operators	+=	sum += 10	This is same as sum = sum + 10
	-=	sum -= 10	This is same as sum = sum – 10
	*=	sum *= 10	This is same as sum = sum * 10
	/+	sum /= 10	This is same as sum = sum / 10
	%=	sum %= 10	This is same as sum = sum % 10
	&=	sum&=10	This is same as sum = sum & 10
	^=	sum ^= 10	This is same as sum = sum ^ 10

```
int num = 10;
num += 5;  // num = num + 5;
```

5. Increment and Decrement Operators:

- Increase or decrease the value of a variable by 1.

S.no	Operator type	Operator	Description
1	Pre increment	++i	Value of i is

			incremented before assigning it to variable i.
2	Post–increment	i++	Value of i is incremented after assigning it to variable i.
3	Pre decrement	— –i	Value of i is decremented before assigning it to variable i.
4	Post_decrement	i– —	Value of i is decremented after assigning it to variable i.

```
int counter = 5;
counter++;  // Increment
counter--;  // Decrement
```

6. Bitwise Operators:

- Perform operations at the bit level.

TRUTH TABLE FOR BIT WISE OPERATION BIT WISE OPERATORS

x	y	x|y	x & y	x ^ y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Operator_symbol	Operator_name
&	Bitwise_AND
|	Bitwise OR
~	Bitwise_NOT
^	XOR
<<	Left Shift
>>	Right Shift

```
int a = 5, b = 3;
int bitwiseAnd = a & b;   // Bitwise AND
int bitwiseOr = a | b;    // Bitwise OR
int bitwiseXor = a ^ b;   // Bitwise XOR
int bitwiseNot = ~a;      // Bitwise NOT
```

7. Conditional Operator (Ternary Operator):

- Provides a concise way to express a conditional statement.

```
int age = 20;
char *result = (age >= 18) ? "Adult" : "Minor";
```

8. sizeof Operator:

- Returns the size in bytes of a data type or object.

```
int size = sizeof(int);
```

C KEYWORDS:

C keywords are the words that convey a special meaning to the c compiler. The keywords cannot be used as variable names.

The list of C keywords is given below:

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

▼ Typecasting

Typecasting in C is the process of converting a variable from one data type to another. It allows you to treat a variable as if it belongs to a different data type temporarily. There are two main types of typecasting: implicit (automatic) and explicit (manual).

1. Implicit Typecasting:

Implicit typecasting, also known as automatic type conversion, occurs when the compiler automatically converts one data type to another without any explicit instructions from the programmer. This typically happens when performing operations involving mixed data types.

```
#include <stdio.h>

int main() {
    int num_int = 5;
    float num_float = 2.5;

    float result = num_int + num_float;  // Implicit typecasting from int to flo
at

    printf("Result: %f\n", result);

    return 0;
}
```

In this example, the integer `num_int` is implicitly typecast to a float when added to `num_float`.

2. Explicit Typecasting:

Explicit typecasting, also known as manual type conversion, requires the programmer to specify the type to which a variable should be converted. It is done using casting operators.

```c
#include <stdio.h>

int main() {
    float num_float = 2.5;
    int num_int;

    // Explicit typecasting from float to int
    num_int = (int)num_float;

    printf("Result: %d\n", num_int);

    return 0;
}
```

Here, the value of `num_float` is explicitly typecast to an integer using `(int)`.

Use Cases of Typecasting:

1. **Preserving Data during Assignments:**
   - When assigning a value of one data type to a variable of another data type, typecasting can be used to avoid loss of data.
2. **Handling Mixed Data Types:**
   - During arithmetic or logical operations involving variables of different data types, typecasting helps ensure consistent behavior.

It's important to note that typecasting should be done carefully, as improper type conversions can lead to data loss or unexpected behavior. It's recommended to understand the implications and potential risks before applying typecasting in your code.

Example 1: Basic Type Casting

```c
#include <stdio.h>

int main() {
    int integerNumber = 10;
    double doubleNumber;

    // Implicit type casting (int to double)
    doubleNumber = integerNumber;

    printf("Integer: %d\nDouble: %f\n", integerNumber, doubleNumber);

    return 0;
}
```

Output:

```
Integer: 10
Double: 10.000000
```

Example 2: Explicit Type Casting

```
#include <stdio.h>

int main() {
    double doubleNumber = 3.14;
    int integerNumber;

    // Explicit type casting (double to int)
    integerNumber = (int)doubleNumber;

    printf("Double: %f\nInteger: %d\n", doubleNumber, integerNumber);

    return 0;
}
```

Output:

```
Double: 3.140000
Integer: 3
```

Example 3: Type Casting with Arithmetic Operations

```
#include <stdio.h>

int main() {
    int numerator = 5;
    int denominator = 2;
    double result;

    // Type casting in arithmetic operation
    result = (double)numerator / denominator;

    printf("Result: %f\n", result);

    return 0;
}
```

Output:

```
Result: 2.500000
```

▼ Storage classes

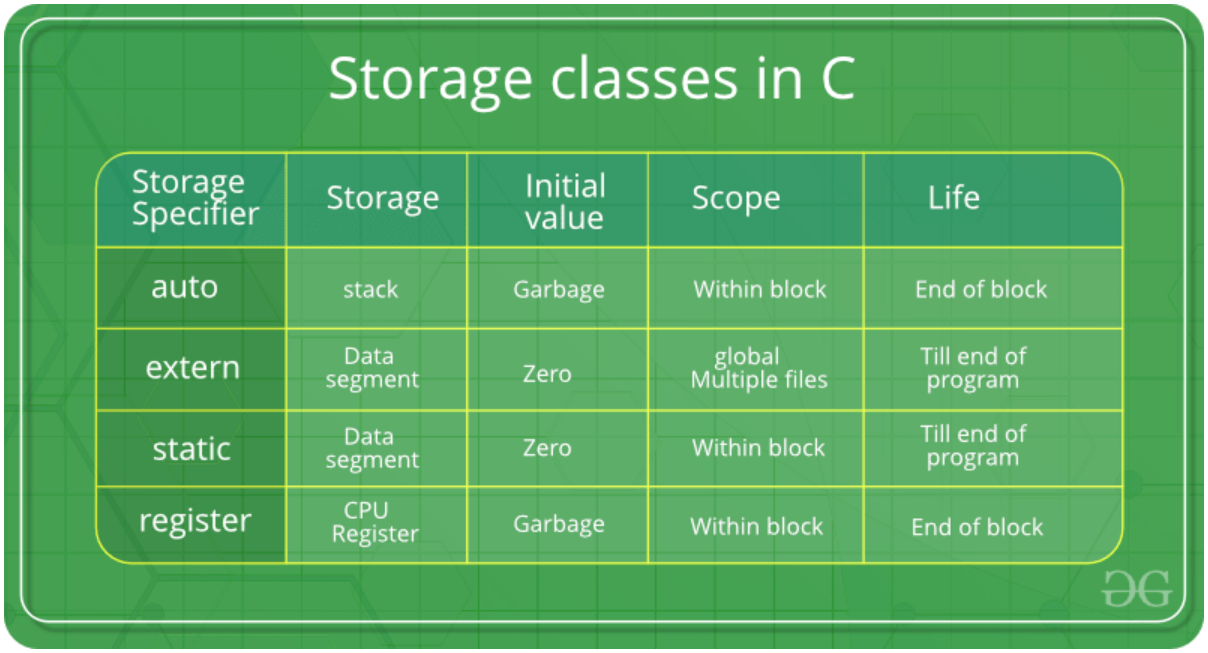
In C programming, storage classes define the scope, visibility, and lifetime of variables. C provides four storage classes.

These storage classes allow programmers to control the memory allocation and visibility of variables in their programs. Understanding storage classes is essential for writing efficient and maintainable C code.

Storage classes in C are needed for several reasons:

1. **Memory Management**: Storage classes allow programmers to control how variables are allocated and deallocated in memory. This is essential for efficient memory usage, especially in resource-constrained environments.
2. **Scope and Visibility**: Different storage classes define the scope and visibility of variables, determining where they can be accessed within the program. This helps in organizing and modularizing code by limiting access to variables based on their usage context.
3. **Lifetime**: Storage classes influence the lifetime of variables, determining how long they persist in memory. This is crucial for managing resources and ensuring that variables are available for the duration they are needed.
4. **Optimization**: Storage classes like `register` enable optimization techniques by providing hints to the compiler about variable usage patterns. This can lead to performance improvements, especially in critical code sections.
5. **Global Communication**: The `extern` storage class facilitates communication between different parts of a program by allowing variables to be accessed across multiple files.

Types of storage classes in C:



1. **auto:**

- Variables declared with the `auto` storage class are automatically created and initialized when they are declared within a block or function.
- They are local to the block or function in which they are declared.
- The `auto` storage class is rarely used in modern C programming because all local variables are automatically considered `auto` by default.
- Examples:

```
void exampleFunction() {
    auto int x = 10; // 'auto' is optional, local variable by default
    // code
}
```

```
#include <stdio.h>void exampleFunction() {
    auto int x = 10; // 'auto' is optional, local variable by default
    printf("x inside exampleFunction: %d\n", x);
}

int main() {
    exampleFunction();
    // Error: 'x' is not visible here
    // printf("x in main: %d\n", x);
    return 0;
}
```

2. **register:**
   - Variables declared with the `register` storage class are stored in CPU registers if possible.
   - They provide quick access to frequently accessed variables.
   - It's a hint to the compiler, and it may or may not store the variable in a register depending on compiler optimization.
   - Examples:

```
void exampleFunction() {
    register int x = 10;
    // code
}
```

```
#include <stdio.h>void exampleFunction() {
    register int x = 10;
```

```
    printf("x inside exampleFunction: %d\n", x);
}

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

3. **static:**
   - Variables declared with the `static` storage class retain their values between function calls.
   - If declared at the file level, they have internal linkage and can only be accessed within the same file.
   - If declared within a function or block, they retain their values between function calls and have a lifetime equal to the lifetime of the program.
   - Examples:

```
void exampleFunction() {
    static int x = 0; // Retains value between function calls
    // code
}
```

```
#include <stdio.h>
void exampleFunction() {
    static int x = 0; // Retains value between function calls
    x++; // Increment value on each call
    printf("x inside exampleFunction: %d\n", x);
}

int main() {
    exampleFunction(); // Output: 1
    exampleFunction(); // Output: 2
    exampleFunction(); // Output: 3
    return 0;
}
```

4. **extern:**
   - Variables declared with the `extern` storage class are declared in one file but can be accessed in another file.
   - It provides a way to use global variables across multiple files.
   - The actual variable is defined elsewhere in the program.
   - Examples:

```


// In one file (e.g., file1.c)
```

```
int globalVar = 10;

// In another file (e.g., file2.c)
extern int globalVar; // Declaration, not definition
```

```
// file1.c
#include <stdio.h>
int globalVar = 10;

// file2.c
#include <stdio.h>
extern int globalVar; // Declaration, not definition

int main() {
    printf("Value of globalVar from file2.c: %d\n", globalVar);
    return 0;
}
```

In summary:

- Storage classes in C define how variables are stored in memory and accessed within a program.
- Understanding storage classes helps in managing variable scope, visibility, and lifetime.
- The four main storage classes in C are `auto`, `register`, `static`, and `extern`.
- Each storage class has unique characteristics and use cases, such as automatic allocation, register optimization, static memory retention, and external linkage.
- Proper usage of storage classes can improve program efficiency and maintainability.
- Careful consideration should be given to the choice of storage class to ensure optimal program behavior and readability.

▼ Format specifiers

In C, format specifiers and escape sequences are used in printf and scanf functions for formatting input and output.

Format Specifiers in printf and scanf:

1. Integer Specifiers:

- `%d`: Print or scan integers.
- `%u`: Print or scan unsigned integers.
- `%ld`, `%lu`: Long integers.

2. Floating-Point Specifiers:

- `%f`: Print or scan floating-point numbers.

- `%lf`: Double-precision floating-point.

3. Character Specifiers:

- `%c`: Print or scan characters.
- `%s`: Print or scan strings.

4. Pointer Specifier:

- `%p`: Print memory address.

5. Width and Precision:

- `%5d`: Minimum width of 5 characters.
- `%.2f`: Two digits after the decimal point.

Escape Sequences in Strings:

1. Newline:

- `\n`: Moves the cursor to the beginning of the next line.

2. Tab:

- `\t`: Inserts a horizontal tab.

3. Double Quote and Single Quote:

- `\"`: Prints a double quote.
- `\'`: Prints a single quote.

4. Backslash:

- `\\`: Prints a backslash.

Escape Sequence Characters	Non-Printing Characters
\'	Single quote, needed for character literals.
\"	Double quote, needed for string literals.
\\	Backslash, needed for string literals.
\0	Unicode character 0.
\a	Alert.
\b	Backspace.
\f	Form feed.
\n	New line.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.
\xhh	Matches an ASCII character using hexadecimal representation (exactly two digits). For example, \x61 represents the character 'a'.
\uhhhh	Matches a Unicode character using hexadecimal representation (exactly four digits). For example, the character \u0020 represents a space.

▼ Conditional Statements

Conditional statements in C, such as `if`, `else if`, and nested `if`, allow you to control the flow of your program based on certain conditions.

1. `if` Statement:

The `if` statement is used for decision-making. It evaluates a condition and executes a block of code if the condition is true.

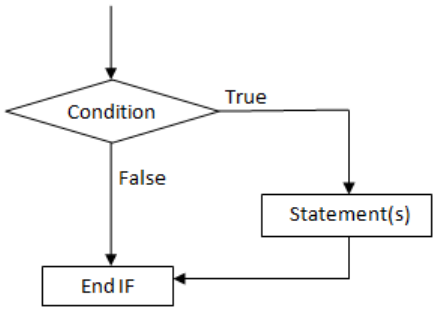


fig: Flowchart for if statement

```
int number = 10;

if (number > 0) {
    printf("The number is positive\n");
}
```

2. `else if` Statement:

The `else if` statement is used to test multiple conditions one by one. If the previous condition in the `if` or `else if` chain is false, the next condition is checked.

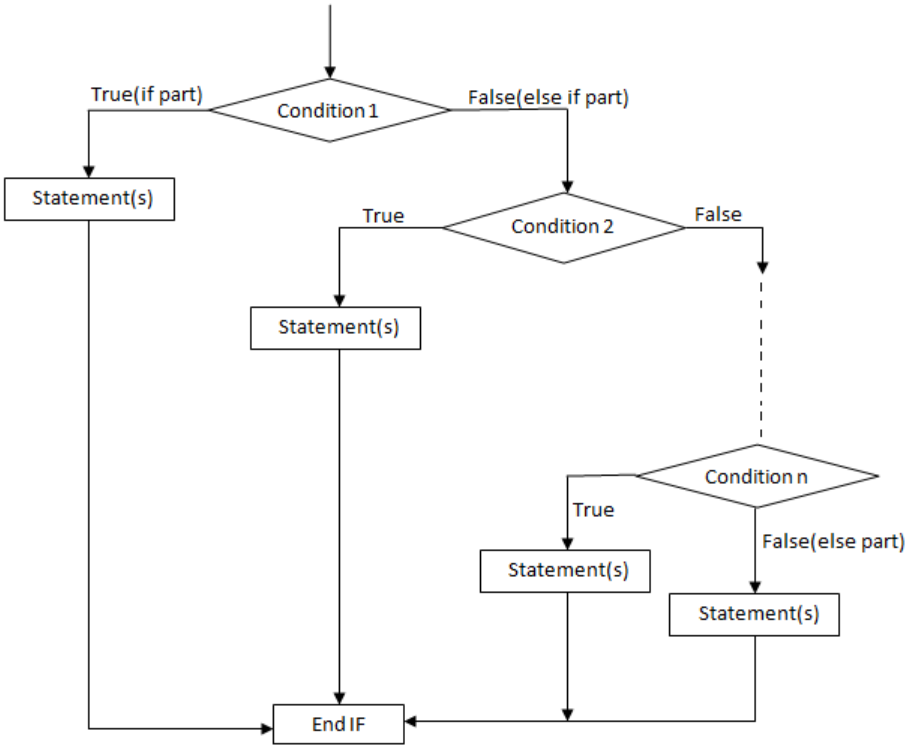


fig: Flowchart for if ... else if ... else statement

```
int number = 0;

if (number > 0) {
    printf("The number is positive.\n");
} else if (number < 0) {
    printf("The number is negative.\n");
} else {
    printf("The number is zero.\n");
}
```

3. Nested `if` Statements:

You can have `if` statements inside other `if` or `else` blocks. This is known as nested `if`.

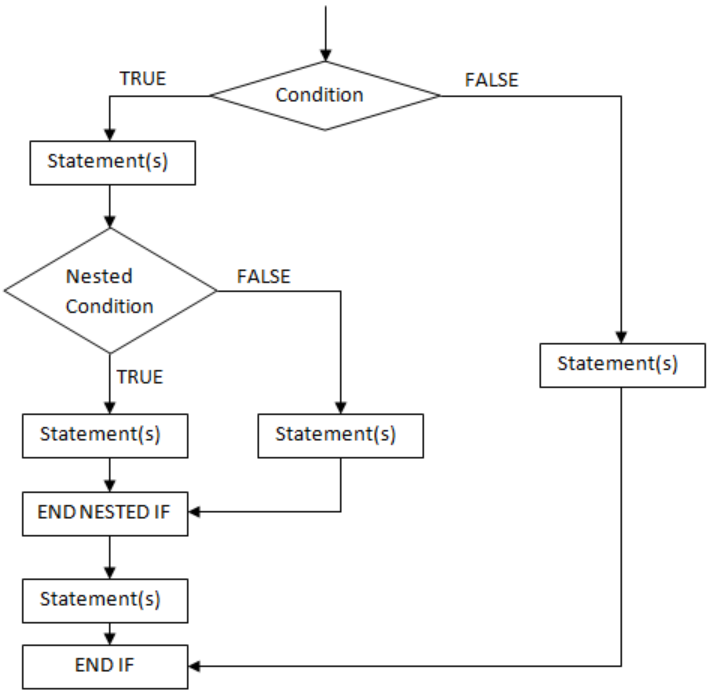


fig: Flowchart for nested if statement

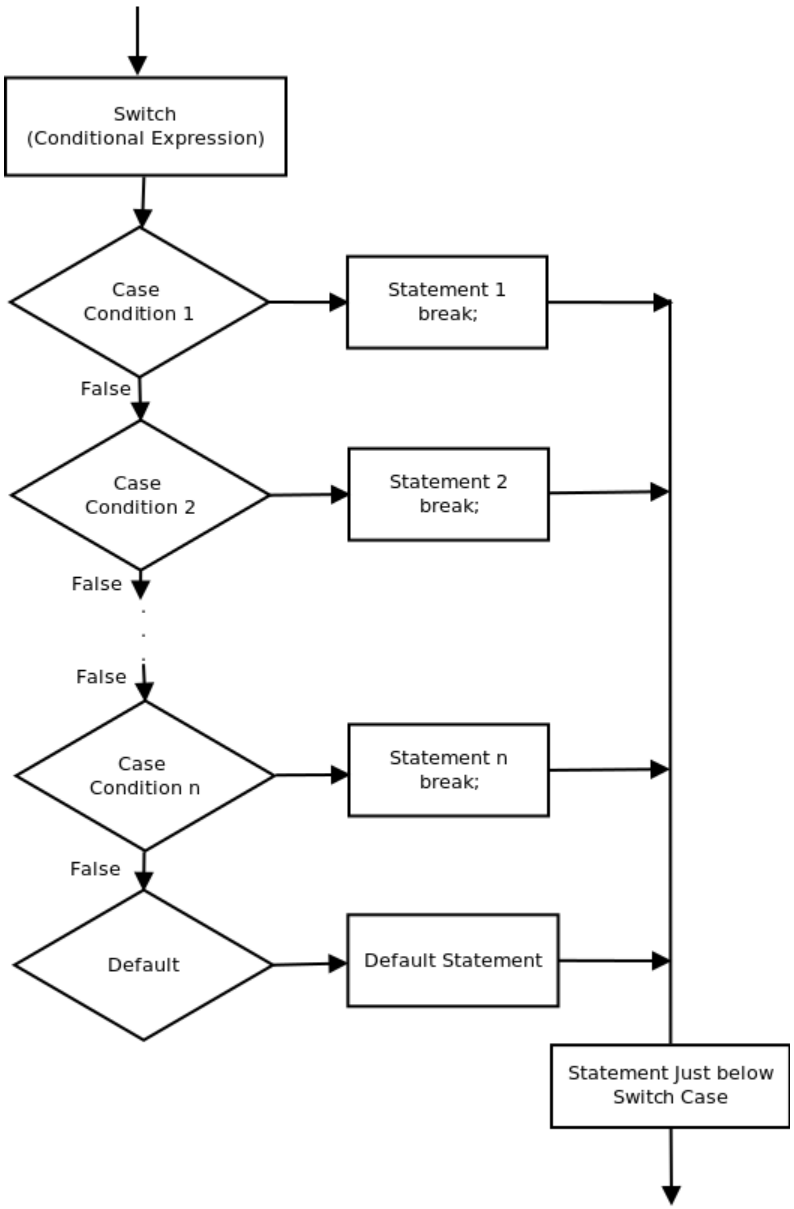
```
int age = 25;
char gender = 'M';

if (age >= 18) {
    printf("You are an adult.\n");
    if (gender == 'M') {
        printf("You are a male adult.\n");
    } else {
        printf("You are a female adult.\n");
    }
} else {
    printf("You are a minor.\n");
}
```

In this example, the nested `if` checks the gender only if the age condition is true.

Conditional statements are essential for making decisions in your program based on different situations. They provide flexibility and control over the flow of execution. Keep in mind that using too many nested `if` statements can make your code harder to read, so it's important to strike a balance between readability and logic complexity.

▼ Switch

The `switch` statement in C provides a way to make decisions based on the value of an expression. It's an alternative to using multiple `if` and `else if` statements when you need to check a variable against multiple values. Let's explore this statement with three examples of increasing complexity.



Example 1: Simple Switch Statement

In this basic example, we'll use a `switch` statement to determine the day of the week based on a numerical input.

```
#include <stdio.h>

int main() {
    int day = 3;

    switch (day) {
        case 1:
            printf("Monday\n");
            break;
```

```
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        case 4:
            printf("Thursday\n");
            break;
        case 5:
            printf("Friday\n");
            break;
        case 6:
            printf("Saturday\n");
            break;
        case 7:
            printf("Sunday\n");
            break;
        default:
            printf("Invalid day\n");
    }

    return 0;
}
```

In this example, the `switch` statement evaluates the value of `day` and executes the corresponding block of code based on the matching `case`. The `break` statements ensure that once a match is found, the program exits the `switch` statement.

Example 2: Using Fall-Through

In this example, we'll illustrate the concept of fall-through in a `switch` statement. When a `case` block lacks a `break` statement, control will fall through to the next case.

```
#include <stdio.h>

int main() {
    int month = 4;

    switch (month) {
        case 1:
            printf("January\n");
            break;
        case 2:
            printf("February\n");
            break;
        case 3:
        case 4:
```

```
        case 5:
            printf("Spring months\n");
            break;
        case 6:
        case 7:
        case 8:
            printf("Summer months\n");
            break;
        case 9:
        case 10:
        case 11:
            printf("Autumn months\n");
            break;
        case 12:
            printf("December\n");
            break;
        default:
            printf("Invalid month\n");
    }

    return 0;
}
```

Here, if `month` is 3, 4, or 5, the program prints "Spring months" due to the fall-through effect.

Example 3: Using Enumeration Constants

In this advanced example, we'll use enumeration constants with a `switch` statement. Enumeration provides symbolic names for integer values, enhancing code readability.

```
#include <stdio.h>

enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };

int main() {
    enum Day today = WEDNESDAY;

    switch (today) {
        case MONDAY:
            printf("Start of the week\n");
            break;
        case WEDNESDAY:
            printf("Midweek\n");
            break;
        case FRIDAY:
            printf("Almost there!\n");
            break;
        case SUNDAY:
```

```
            printf("Weekend\n");
            break;
        default:
            printf("Other da\n");
    }

    return 0;
}
```

Here, we use an enumeration `enum Day` to represent days of the week. The `switch` statement becomes more readable, and it's less error-prone as we use symbolic names instead of numerical values.

These examples demonstrate the flexibility of the `switch` statement in C, providing a concise and efficient way to handle multiple conditions based on the value of an expression.

Why should one use switch case over nested if?

Using `switch` statements over nested `if` statements in certain situations can improve code readability, maintainability, and performance. Here are some reasons to prefer `switch` over nested `if`:

1. **Readability:**
   - `switch` statements are particularly effective when dealing with a single variable that can take on multiple discrete values. This can make the code more readable, especially when there are many conditions to check.
2. **Simplicity and Clarity:**
   - `switch` statements are designed for scenarios where you are comparing a single expression against multiple values. This makes the code more straightforward and eliminates the need for deeply nested structures.
3. **Efficiency:**
   - In some cases, `switch` statements may be more efficient than nested `if` statements. Compilers can optimize `switch` statements more effectively, especially when the cases are contiguous integers.
4. **Ease of Maintenance:**
   - When you need to add or remove cases, the `switch` statement is typically easier to maintain. You simply add or remove cases, and there's no need to modify nested blocks of code.
5. **Fall-Through Capability:**
   - `switch` statements allow for fall-through behavior when a `case` lacks a `break`. This can be useful when multiple cases should execute the same code.

```
switch (day) {
    case 1:
    case 2:
    case 3:
        printf("Weekday\n");
        break;
    case 4:
```

```
    case 5:
        printf("Weekend approaching\n");
        break;
    default:
        printf("Invalid day\n");
}
```

6. **Enumeration Constants:**
   - `switch` statements work well with enumeration constants, providing symbolic names for values and enhancing code readability.

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };

switch (today) {
    case MONDAY:
        // ...
        break;
    case TUESDAY:
        // ...
        break;
    // ...
}
```

While `switch` statements have their advantages, there are scenarios where nested `if` statements may be more suitable, especially when dealing with more complex conditions involving logical operators. The choice between `switch` and nested `if` depends on the specific requirements of the code and the nature of the conditions being checked.

▼ Practice session

Practice Question 1:

Question:

Write a C program that simulates a simple calculator using a `switch` statement. The program should take two numbers and an operator as input from the user and perform the corresponding operation (addition, subtraction, multiplication, division). Handle division by zero and invalid operators gracefully.

Solution:

```
#include <stdio.h>

int main() {
    double num1, num2, result;
    char operator;

    // Input
    printf("Enter first number: ");
    scanf("%lf", &num1);
```

```
    printf("Enter second number: ");
    scanf("%lf", &num2);
    printf("Enter operator (+, -, *, /): ");
    scanf(" %c", &operator); // Note: Leading space to consume newline character

    // Calculation using switch
    switch (operator) {
        case '+':
            result = num1 + num2;
            break;
        case '-':
            result = num1 - num2;
            break;
        case '*':
            result = num1 * num2;
            break;
        case '/':
            if (num2 != 0) {
                result = num1 / num2;
            } else {
                printf("Error: Division by zero is not allowed.\n");
                return 1; // Exit with an error code
            }
            break;
        default:
            printf("Error: Invalid operator.\n");
            return 1; // Exit with an error code
    }

    // Output
    printf("Result: %.2lf\n", result);

    return 0;
}
```

Output:

```
Enter first number: 10
Enter operator (+, -, *, /): *
Enter second number: 5
Result: 50.00
```

Practice Question 2:

Question:
Write a C program that takes three integers as input and prints the largest among them. Use if and else if statements to handle different cases.

Solution:

```
#include <stdio.h>

int main() {
    int num1, num2, num3;

    printf("Enter three integers: ");
    scanf("%d %d %d", &num1, &num2, &num3);

    if (num1 >= num2 && num1 >= num3) {
        printf("The largest number is: %d\n", num1);
    } else if (num2 >= num1 && num2 >= num3) {
        printf("The largest number is: %d\n", num2);
    } else {
        printf("The largest number is: %d\n", num3);
    }

    return 0;
}
```

Output:

```
Enter three integers: 7 15 3
The largest number is: 15
```

Practice Question 3:

Question:
Write a C program that checks if a given year is a leap year or not. Use if and else if statements to handle different cases.

Solution:

```
#include <stdio.h>

int main() {
    int year;

    printf("Enter a year: ");
    scanf("%d", &year);

    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
        printf("%d is a leap year.\n", year);
    } else {
        printf("%d is not a leap year.\n", year);
    }
```

```
    return 0;
}
```

Output:

```
Enter a year: 2024
2024 is a leap year.
```

Practice Question 4:

Question:

Write a C program that takes the marks of a student as input and assigns a grade based on the following criteria:

- 90 and above: Grade A
- 80 - 89: Grade B
- 70 - 79: Grade C
- 60 - 69: Grade D
- Below 60: Grade F

Implement the program using conditional statements (`if`, `else if`, `else`).

Solution:

```
#include <stdio.h>

int main() {
    // Declare variables
    int marks;

    // Input marks from the user
    printf("Enter the marks: ");
    scanf("%d", &marks);

    // Assign grade based on marks
    if (marks >= 90) {
        printf("Grade:
```

```
#include <stdio.h>

int main() {
    // Assume total marks for each subject is 100
    int mathMarks, scienceMarks, englishMarks;

    // Input marks for each subject
    printf("Enter marks for Math: ");
    scanf("%d", &mathMarks);
```

```
    printf("Enter marks for Science: ");
    scanf("%d", &scienceMarks);

    printf("Enter marks for English: ");
    scanf("%d", &englishMarks);

    // Calculate total marks and average
    int totalMarks = mathMarks + scienceMarks + englishMarks;
    float averageMarks = totalMarks / 3.0;  // Using 3.0 to ensure floating-poin
t division

    // Display total marks and average
    printf("Total Marks: %d\n", totalMarks);
    printf("Average Marks: %.2f\n", averageMarks);

    // Assign grade based on average marks
    char grade;

    if (averageMarks >= 90) {
        grade = 'A';
    } else if (averageMarks >= 80) {
        grade = 'B';
    } else if (averageMarks >= 70) {
        grade = 'C';
    } else if (averageMarks >= 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }

    // Display the assigned grade
    printf("Grade: %c\n", grade);

    return 0;
}
```

In this program, the user is prompted to enter marks for Math, Science, and English. The total marks and average marks are calculated. Then, a grading system is applied based on the average marks, and the corresponding grade is displayed. Adjust the grade boundaries and criteria as needed.

Sample Output:

```
Enter marks for Math: 85
Enter marks for Science: 92
Enter marks for English: 78
Total Marks: 255
```

```
Average Marks: 85.00
Grade: A
```

▼ What is Looping? and **For** loop

Looping is a programming concept that involves repeating a set of instructions or statements multiple times until a certain condition is met. It allows for the efficient execution of repetitive tasks and is crucial for automating processes in software development. Loops provide a way to iterate over a block of code, executing it repeatedly based on a specified condition.

In C, looping is primarily achieved using three types of loops: `for`, `while`, and `do-while`. Each loop type has its own syntax and use cases, but they all serve the fundamental purpose of repetition.

Special Aspects of Looping in C:

1. **Simplicity and Efficiency:**
   - C provides a simple and efficient loop structure. The `for` loop, in particular, is concise and widely used for its simplicity in expressing iteration.
2. **Close Interaction with Memory:**
   - C has a close interaction with memory, and loops in C often involve direct manipulation of pointers and memory addresses. This allows for fine-tuned control over iteration processes.
3. **Array Processing:**
   - C is commonly used for array processing, and loops play a crucial role in traversing and manipulating arrays. This is essential for tasks like sorting, searching, and performing operations on array elements.
4. **Low-Level Control:**
   - C provides low-level control over loop constructs, allowing programmers to have precise control over loop variables, conditions, and the flow of execution. This is particularly advantageous in systems programming and tasks that require low-level optimizations.
5. **Efficient Compilation:**
   - C's simplicity and adherence to low-level details make it easier for compilers to generate efficient machine code for loops. This contributes to the language's historical reputation for producing fast and optimized executable code.
6. **Support for Nested Loops:**
   - C allows the nesting of loops, enabling the creation of more complex looping structures. This is useful in scenarios where multiple levels of iteration are required, such as processing multidimensional arrays or implementing certain algorithms.
7. **Use of Break and Continue:**
   - C provides `break` and `continue` statements within loops, offering additional control flow mechanisms. `break` terminates the loop, while `continue` skips the rest of the loop's code for the current iteration.

Overall, looping in C is characterized by its simplicity, efficiency, and low-level control. These features make C well-suited for tasks that demand performance optimization and direct interaction with system

resources. While modern programming languages provide higher-level abstractions, C's approach to looping remains influential, especially in domains where system-level control is critical.

Example 1: Basic Counting

In this example, the `for` loop is used to print numbers from 1 to 5.

```
#include <stdio.h>

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

    return 0;
}
```

Output:

```
1 2 3 4 5
```

Example 2: Sum of Numbers

This example calculates the sum of numbers from 1 to 10 using a `for` loop.

```
#include <stdio.h>

int main() {
    int sum = 0;

    for (int i = 1; i <= 10; i++) {
        sum += i;
    }

    printf("Sum of numbers from 1 to 10: %d\n", sum);

    return 0;
}
```

Output:

```
Sum of numbers from 1 to 10: 55
```

Example 3: Nested For Loop (Multiplication Table)

In this example, a nested `for` loop is used to print the multiplication table of numbers from 1 to 5.

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

```
int main() {
    for (int i = 1; i <= 5; i++) {
        for (int j = 1; j <= 10; j++) {
            printf("%d * %d = %d\t", i, j, i * j);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
1 * 1 = 1    1 * 2 = 2    1 * 3 = 3    1 * 4 = 4    1 * 5 = 5    1 * 6 = 6    1 *
2 * 1 = 2    2 * 2 = 4    2 * 3 = 6    2 * 4 = 8    2 * 5 = 10   2 * 6 = 12   2 *
3 * 1 = 3    3 * 2 = 6    3 * 3 = 9    3 * 4 = 12   3 * 5 = 15   3 * 6 = 18   3 *
4 * 1 = 4    4 * 2 = 8    4 * 3 = 12   4 * 4 = 16   4 * 5 = 20   4 * 6 = 24   4 *
5 * 1 = 5    5 * 2 = 10   5 * 3 = 15   5 * 4 = 20   5 * 5 = 25   5 * 6 = 30   5 *
```

These examples showcase different applications of the `for` loop, from simple counting to more complex scenarios involving calculations and nested loops. The `for` loop is a powerful construct for repetitive tasks in C.

▼ **While** loop

Certainly! The `while` loop in C is used to repeatedly execute a block of code as long as a specified condition is true. Here are three examples demonstrating different use cases of the `while` loop:

The same examples of For Loop now using While Loop

Example 1: Basic Counting

This example uses a `while` loop to print numbers from 1 to 5.

```
#include <stdio.h>

int main() {
    int i = 1;

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

    return 0;
}
```

Output:

```
1 2 3 4 5
```

Example 2: Sum of Numbers

This example calculates the sum of numbers from 1 to 10 using a `while` loop.

```
#include <stdio.h>

int main() {
    int i = 1;
    int sum = 0;

    while (i <= 10) {
        sum += i;
        i++;
    }

    printf("Sum of numbers from 1 to 10: %d\n", sum);

    return 0;
}
```

Output:

```
Sum of numbers from 1 to 10: 55
```

Example 3: User Input Validation

This example uses a `while` loop to repeatedly prompt the user for input until a valid option is entered.

```
#include <stdio.h>

int main() {
    int userChoice;

    while (1) {
        printf("Enter 1 for Yes, 2 for No: ");
        scanf("%d", &userChoice);

        if (userChoice == 1 || userChoice == 2) {
            break;  // Exit the loop if a valid option is entered
        } else {
            printf("Invalid choice. Try again.\n");
        }
    }

    printf("You chose: %d\n", userChoice);

    return 0;
}
```

In this example, the loop continues until the user enters either 1 or 2, providing a mechanism for input validation.

These examples illustrate different applications of the `while` loop, from simple counting to more interactive scenarios involving user input validation. The `while` loop is useful when the number of iterations is not known in advance and is determined by a condition.

Here are some new examples of While Loop:

Example 1: Factorial Calculation

This example uses a `while` loop to calculate the factorial of a number entered by the user.

```
#include <stdio.h>

int main() {
    int num, factorial = 1;

    printf("Enter a number: ");
    scanf("%d", &num);

    int i = 1;
    while (i <= num) {
        factorial *= i;
        i++;
    }

    printf("Factorial of %d is: %d\n", num, factorial);

    return 0;
}
```

Example 2: Password Validation

In this example, a simple password validation loop is implemented using a `while` loop. The loop continues until the correct password is entered.

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

int main() {
    char password[] = "secure123";
    char input[20];

    while (1) {
        printf("Enter the password: ");
        scanf("%s", input);

        if (strcmp(password, input) == 0) {
```

```
            printf("Access granted!\n");
            break;
        } else {
            printf("Incorrect password. Try again.\n");
        }
    }

    return 0;
}
```

Example 3: Generating Fibonacci Sequence

This example uses a `while` loop to generate the Fibonacci sequence up to a specified limit.

```
#include <stdio.h>

int main() {
    int limit;
    printf("Enter the limit for Fibonacci sequence: ");
    scanf("%d", &limit);

    int prev = 0, current = 1;

    printf("Fibonacci sequence up to %d: ", limit);

    while (current <= limit) {
        printf("%d ", current);

        int next = prev + current;
        prev = current;
        current = next;
    }

    printf("\n");

    return 0;
}
```

These examples showcase different applications of the `while` loop, from mathematical calculations to user input validation and sequence generation. The `while` loop is versatile and can be applied in various scenarios where iteration is required.

▼ **Do While** loop

The `do-while` loop in C is a control flow structure that executes a block of code repeatedly as long as a specified condition is true. The key characteristic of a `do-while` loop is that the condition is checked after the execution of the block, ensuring that the block is executed at least once.

While loop	Do-while loop
In this, the given condition is evaluated first and then loop body is executed	In this, the given loop body is executed first and then after the given condition is checked
It is an entry-controlled loop	It is an exit-controlled loop
The loop body would be executed, only if the given condition is true	The loop body would be executed at least once, even if the given condition is false
It allows initialization of counter variable before entering loop body	It allows initialization of counter variable before and after entering loop body
No semi-colon is used as a part of syntax, while(condition)	Semi-colon is used as a part of syntax, while(condition);
It is used when condition evaluation is required to be evaluated first, before executing loop body	D0-while is used when one-needs to enter into the loop body before evaluating condition. Eg menu driven programs

Syntax :
while(condition)
{
// loop body
}
	Syntax :
	do{
	// loop body
	} while(condition);

Example 1: Sum of Numbers

```
cCopy code
#include <stdio.h>int main() {
    int sum = 0;
    int number;

    do {
        printf("Enter a number (enter 0 to exit): ");
        scanf("%d", &number);
        sum += number;
```

```
    } while (number != 0);

    printf("Sum of the entered numbers: %d\n", sum);

    return 0;
}
```

In this example, the `do-while` loop is used to continuously prompt the user to enter numbers until they input 0. The loop iterates at least once, ensuring that the initial prompt is executed before checking the condition.

Certainly! Here are two more examples of `do-while` loops with increasing complexity:

Example 2: Factorial Calculation

```
#include <stdio.h>

int main() {
    int number, factorial = 1;

    do {
        printf("Enter a positive integer: ");
        scanf("%d", &number);

        if (number < 0) {
            printf("Please enter a positive integer.\n");
        }
    } while (number < 0);

    // Calculate factorial
    for (int i = 1; i <= number; ++i) {
        factorial *= i;
    }

    printf("Factorial of %d = %d\n", number, factorial);

    return 0;
}
```

In this example, the `do-while` loop is used to ensure that the user enters a positive integer before calculating its factorial. The loop continues until a valid input is provided.

Example 3: Guess the Number Game

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

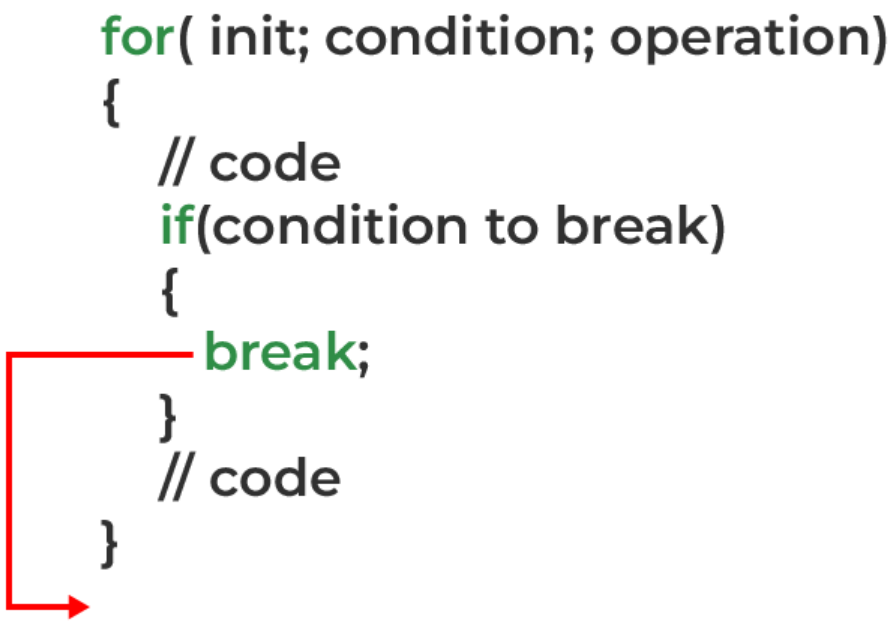
```
int main() {
    srand(time(NULL));
    int secretNumber = rand() % 100 + 1;  // Generate a random number between 1
and 100
    int guess, attempts = 0;

    printf("Guess the number between 1 and 100.\n");

    do {
        printf("Enter your guess: ");
        scanf("%d", &guess);
        attempts++;

        if (guess < secretNumber) {
            printf("Too low. Try again.\n");
        } else if (guess > secretNumber) {
            printf("Too high. Try again.\n");
        } else {
            printf("Congratulations! You guessed the number in %d attempts.\\n",
attempts);
        }
    } while (guess != secretNumber);

    return 0;
}
```

In this example, the `do-while` loop is utilized for a simple "Guess the Number" game. The user continues to guess until they correctly identify the randomly generated secret number between 1 and 100. The loop ensures that the user has at least one chance to make a guess.

▼ Conditional Branching Statements

1. `break` Statement:

- **Usage:**
  - Used within loops (`for`, `while`, `do-while`) and `switch` statements.
- **Functionality:**
  - Terminates the innermost loop or switch statement.
  - Control immediately passes to the statement following the terminated loop or switch.



Example 1: Using `break` in a Loop

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;  // Exit the loop when i is 5
    }
    printf("%d ", i);
}
```

Example 2: Using `break` in a Loop

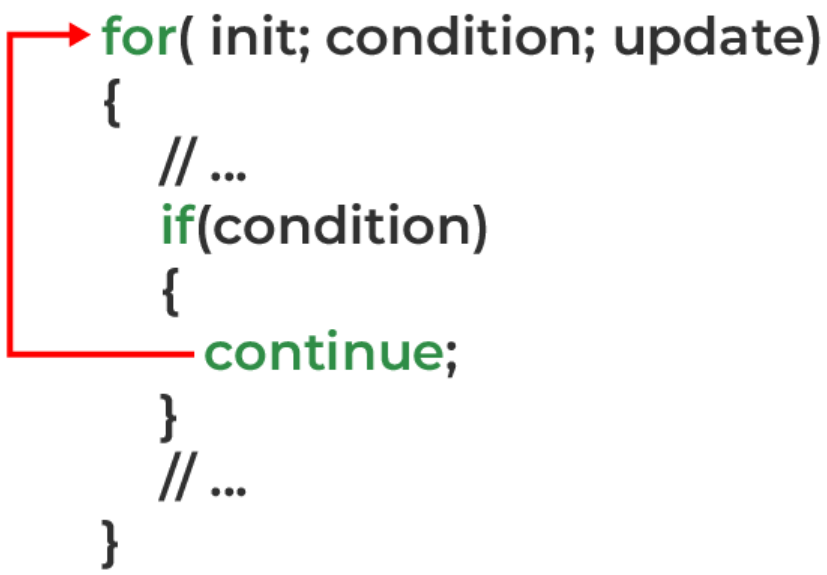
```
#include <stdio.h>

int main() {
    int i;

    // Loop to find the first even number in a sequence
    for (i = 1; i <= 10; ++i) {
        if (i % 2 == 0) {
            printf("First even number found: %d\n", i);
            // Break out of the loop once an even number is found
            break;
        }
    }
```

```
    return 0;
}
```

In this example, the loop searches for the first even number in the sequence from 1 to 10. The `break` statement is used to exit the loop as soon as an even number is found.

2. `continue` Statement:

- **Usage:**
  - Used within loops (`for`, `while`, `do-while`).
- **Functionality:**
  - Skips the rest of the code inside the loop for the current iteration.
  - Control immediately goes to the next iteration of the loop.



Example 1: Using `continue` in a Loop

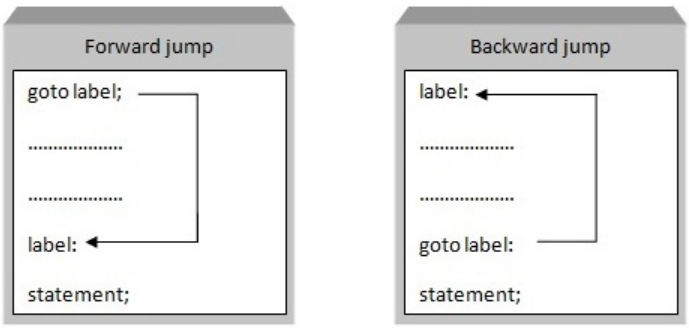
```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;  // Skip the rest of the loop for i = 3
    }
    printf("%d ", i);
}
```

Example 2: Using `continue` in a Loop

```c
#include <stdio.h>

int main() {
    int i;

    // Loop to print odd numbers in a sequence
    for (i = 1; i <= 10; ++i) {
        if (i % 2 == 0) {
            // Skip even numbers and continue with the next iteration
            continue;
        }
        printf("Odd number: %d\n", i);
    }

    return 0;
}
```

3. `goto` Statement:

- **Usage:**
  - Can be used to transfer control to a labeled statement within the same function.
- **Functionality:**
  - Jumps to the labeled statement specified by the `goto`.
  - Considered less structured and can lead to code that is harder to understand and maintain.



- **Example:**

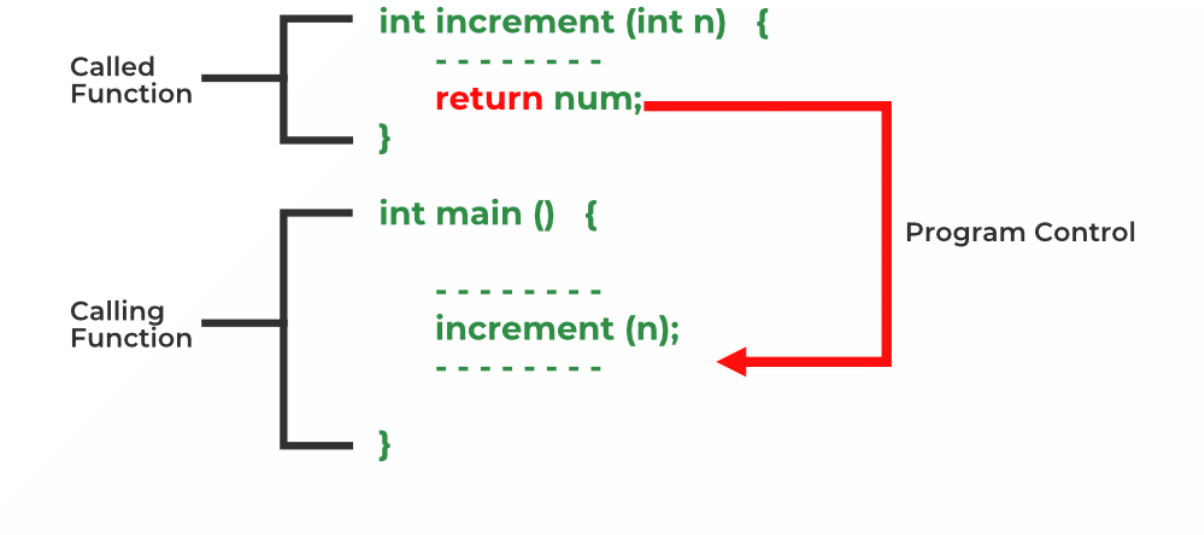
```c
int i = 1;
start:
    if (i > 5) {
        goto end;  // Jump to the 'end' label if i is greater than 5
    }
    printf("%d ", i);
```

```
    i++;
    goto start;

end:
```

It's important to note that the use of `goto` is generally discouraged in modern programming practices because it can make code less readable and harder to maintain. Structured alternatives like `break` and `continue` are preferred in most cases. `goto` is reserved for situations where it provides a clearer and more efficient solution, but such situations are rare in well-structured code.

4. Return:

- The `return` statement is essential for providing a result from a function.
- It can be used to return a value calculated within the function or to exit the function early based on certain conditions.
- The type of the value returned must match the return type declared in the function signature.



Example 1: Returning a Calculated Result

```
#include <stdio.h>

// Function to calculate the square of a number
int square(int num) {
    return num * num;  // Return the square of the input number
}

int main() {
    int result = square(5);
    printf("Square of 5 is: %d\n", result);
```

```
    return 0;
}
```

In this example, the `square` function takes an integer as an argument, calculates its square, and returns the result using the `return` statement. The calculated result is then printed in the `main` function.

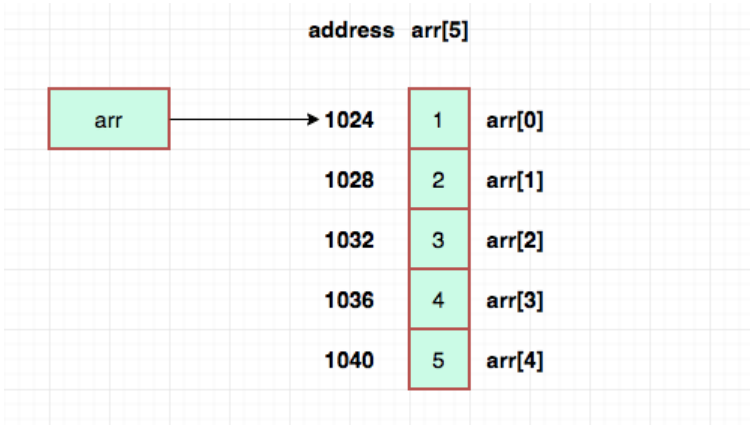
▼ Arrays in C

Arrays in C:

An array in C is a collection of elements of the same data type stored in contiguous memory locations. It provides a convenient way to represent and manipulate a group of related data items. Each element in an array is accessed using an index, which represents its position in the array.

Characteristics of Arrays:

1. **Homogeneous Data Type:**
   - All elements in an array must be of the same data type. For example, an array can consist of integers, characters, or any other data type, but all elements within that array must share the same data type.
2. **Fixed Size:**
   - The size of an array is fixed at the time of declaration. Once defined, it cannot be changed during runtime. The size determines the number of elements the array can hold.
3. **Contiguous Memory Allocation:**
   - Array elements are stored in contiguous memory locations. This sequential arrangement allows for efficient memory management and direct access to any element using its index.



Memory Management:

- When an array is declared, memory is allocated for all its elements based on the data type and size specified. The memory is allocated as a single block, ensuring that elements are stored in adjacent locations.
- The index of the first element is 0, and the index of the last element is (size - 1). For example, in an array of size 5, the indices range from 0 to 4.

Elements in an array are stored in consecutive memory locations. The memory address of the first element serves as the base address, and subsequent elements are placed at locations incremented by the size of one element. The formula for accessing the memory location of an element at index `i` in an array is:

$$\text{Address of element } i = \text{Base address} + (i \times \text{Size of one element})$$

Advantages of Arrays in C:

1. **Random Access:**
   - Arrays support direct access to any element using its index, allowing for constant-time random access. This is particularly advantageous when quick access to specific elements is required.
2. **Memory Efficiency:**
   - Arrays allocate contiguous memory for elements, reducing memory fragmentation. This contiguous memory allocation contributes to efficient use of memory.
3. **Simplicity and Compactness:**
   - Arrays provide a simple and compact way to organize and store a collection of elements of the same data type. The syntax for array declaration and access is concise.
4. **Efficient for Iterative Operations:**
   - Arrays are well-suited for operations that involve iteration through elements, such as loops. This makes them efficient for various algorithms and data manipulation tasks.
5. **Ease of Implementation:**
   - Arrays are fundamental and easy to implement in C. They serve as a building block for more complex data structures and algorithms.

Disadvantages of Arrays in C:

1. **Fixed Size:**
   - The size of an array is fixed at the time of declaration. This fixed size can be a limitation when dealing with dynamic data where the size is not known in advance.
2. **Memory Wastage:**
   - If an array is declared larger than necessary, it may lead to memory wastage. Unused memory slots may consume space without being utilized.
3. **No Built-in Bounds Checking:**
   - C arrays lack built-in bounds checking. Accessing an element beyond the array bounds can lead to undefined behavior, including segmentation faults. Programmers need to be cautious about array bounds.
4. **Inflexible for Insertion and Deletion:**
   - Arrays are not efficient for insertion and deletion operations. Inserting or deleting elements in the middle of an array requires shifting all subsequent elements, resulting in time-consuming operations.
5. **Homogeneous Data Types:**

- Arrays in C can only store elements of the same data type. This limitation may not be suitable for scenarios where a collection of different data types needs to be stored.

6. **No Dynamic Resizing:**
   - Arrays do not support dynamic resizing. Once declared, the size remains constant, and changing it requires creating a new array and copying elements.
7. **Static Memory Allocation:**
   - Arrays are statically allocated, and their memory size is determined at compile-time. This makes them less flexible for situations where dynamic memory allocation is required.

Applications:

- Storing and manipulating lists of items.
- Representing mathematical vectors and matrices.
- Implementing data structures like stacks, queues, and hash tables.

Example for Arrays:

```
#include <stdio.h>

int main() {
    // Declare an array of integers
    int numbers[5];

    // Use a loop to accept values for the array
    printf("Enter 5 integers:\n");
    for (int i = 0; i < 5; i++) {
        printf("Enter value for element %d: ", i + 1);
        scanf("%d", &numbers[i]);
    }

    // Display the entered values
    printf("\\nEntered Values: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\\n");

    return 0;
}
```

```
Enter 5 integers:
Enter value for element 1: 10
Enter value for element 2: 20
Enter value for element 3: 30
Enter value for element 4: 40
```

```
Enter value for element 5: 50

Entered Values: 10 20 30 40 50
Sum of Entered Values: 150
```

In this example, a loop is used to accept values from the user for an array of integers. The loop iterates through each element of the array, prompting the user to enter a value for each element. The entered values are then displayed. This is a common use case where looping is employed to efficiently handle repetitive tasks, such as inputting multiple values into an array.

▼ Functions in C

Functions in C:

A function is a block of code designed to perform a specific task. Functions provide modularity, code reuse, and easier maintenance by breaking down a program into smaller, manageable units. Each function has a unique name and can be called from other parts of the program.

Why Do We Need Functions?

1. **Modularity:** Functions allow breaking down a large program into smaller, more manageable parts.
2. **Code Reusability:** Once a function is defined, it can be used multiple times.
3. **Readability:** Functions make the code more readable and organized.
4. **Maintenance:** Changes or updates can be made in a specific function without affecting the entire program.

Function Declaration and Syntax:

The syntax for declaring a function in C is as follows:

```
return_type function_name(parameters);
```

- **return_type:** The type of value the function returns. It can be `int`, `void`, `double`, etc.
- **function_name:** The unique name given to the function.
- **parameters:** The input values (if any) the function accepts.

Parameters and Return Type:

1. **Parameters:**
   - **Definition:** Parameters are values passed to a function when it is called.
   - **Purpose:** Provide input to the function for its operation.
   - **Example:**

```
int add(int a, int b) {
    return a + b;
```

```
}
```

- In this example, `a` and `b` are parameters of the `add` function.

2. **Return Type:**
   - **Definition:** The return type specifies the type of value that a function will return to the caller.
   - **Purpose:** Allows a function to provide a result or output.
   - **Example:**

```
int add(int a, int b) {
    return a + b;
}
```

   - In this example, the return type is `int`, indicating that the `add` function returns an integer.

Main Function:

The `main` function is the entry point of every C program. It is called when the program starts execution. The program starts executing from the `main` function and may call other functions as needed.

```
int main() {
    // Program execution starts here
    return 0; // Indicates successful execution
}
```

User Defined Function:

a user-defined function is a function that you create yourself, rather than one provided by the C standard library or any other external source. These functions are defined by the user (programmer) to perform specific tasks within a program.

```
return_type function_name(parameter_list) {
    // Function body
    // Statements to perform the desired task
    return value; // Return statement (if applicable)
}
```

Importance of Separate User-Defined Functions:

1. **Modularity:**
   - **Reason:** Breaking down a program into smaller, modular functions makes it more organized and easier to understand.
   - **Benefit:** Each function can focus on a specific task, enhancing code readability and maintainability.
2. **Code Reusability:**
   - **Reason:** Functions can be reused in different parts of the program or even in other programs.

- **Benefit:** Reduces code duplication, saves development time, and ensures consistency across the codebase.

3. **Easier Debugging:**
   - **Reason:** Functions isolate specific functionality, making it easier to identify and fix issues.
   - **Benefit:** Faster debugging as issues are confined to smaller, manageable functions.
4. **Scoping:**
   - **Reason:** Variables declared inside functions have local scope, limiting their visibility.
   - **Benefit:** Avoids naming conflicts and unintended side effects, enhancing program stability.
5. **Readability:**
   - **Reason:** A well-designed program with meaningful function names improves overall code readability.
   - **Benefit:** Facilitates collaboration and understanding, especially in large codebases.

Key Aspects of Functions in C:

Here are the key aspects of functions in C:

1. Function Declaration and Definition:

- A function declaration specifies the function's name, return type, and parameter list. It tells the compiler about the existence of the function and how it should be called. Function declarations are typically placed at the beginning of a source file or in header files.

```
return_type function_name(parameter_list);
```

  Example:

- A function definition provides the actual implementation of the function. It specifies what the function does when it is called. Function definitions are typically placed after the main function or at the end of the source file.

```
return_type function_name(parameter_list) {
    // Function body
    // Statements to perform the desired task
    return value; // Return statement (if applicable)
}
```

  Example:

```
int sum(int a, int b) {
    return a + b; // Function definition
}
```

2. Function Parameters:

- Function parameters (also called function arguments) are variables declared inside the parentheses of a function declaration or definition. They are used to pass data into the function when it is called. Parameters allow functions to operate on different sets of data each time they are called, increasing the flexibility and reusability of the code.

  Here's the syntax for defining function parameters:

```
return_type function_name(parameter_type parameter1, parameter_type parameter
2, ...);
```

  Example:

```
int add(int x, int y); // Function declaration with parameters
```

- When calling a function with parameters, you provide values (arguments) that match the data types of the parameters declared in the function signature.

  Here's how you call a function with parameters:

```
result = function_name(argument1, argument2, ...);
```

Example:

```
int sum = add(5, 3); // Function call with arguments 5 and 3
```

3. Return Values:

- The `return` statement is used within a function to return a value back to the calling code. When a function is called, it can perform certain operations and produce a result, which can then be passed back to the part of the program that called the function using the `return` statement.

  Here's the basic syntax of the `return` statement:

```
return expression;
```

4. Function Prototypes:

- A function prototype (also known as a function declaration) is a statement that declares the function's name, return type, and parameter types to the compiler before the actual function definition. This informs the compiler about the existence of the function and its interface (the number and types of parameters it expects and the type of value it returns) without providing the function's implementation details.

The syntax for a function prototype is similar to the syntax for a function declaration within the source code, but it's usually placed at the beginning of the file, before the `main()` function, or in a header file (.h) if the function is defined in a separate source file (.c).

Here's the basic syntax of a function prototype:

```
return_type function_name(parameter_type1, parameter_type2, ...);
```

Types of Functions in C:

1. **Functions with Return Value (`int`, `double`, etc.):**
   - Example: `int add(int a, int b);`
2. **Functions without Return Value (`void`):**
   - Example: `void displayMessage();`
3. **Functions with Parameters:**
   - Example: `int multiply(int x, int y);`
4. **Functions without Parameters:**
   - Example: `void greet();`

Here are examples for different types of functions in C, each with a specific purpose, along with their output:

Example 1: Function with Return Value (Addition)

```
#include <stdio.h>

// Function Declaration
int add(int a, int b);

int main() {
    // Function Call
    int result = add(3, 4);

    // Output
    printf("Sum: %d\n", result);

    return 0;
}

// Function Definition
int add(int a, int b) {
    return a + b;
}
```

Logic and Flow:

- **Function `add`:** Takes two parameters (`a` and `b`) and returns their sum.
- `main` **Function:** Calls `add(3, 4)`, stores the result in `result`, and prints the sum.

Output:

```
Sum: 7
```

Example 2: Function without Return Value (Display Message)

```
#include <stdio.h>

// Function Declaration
void displayMessage();

int main() {
    // Function Call
    displayMessage();

    // Output
    printf("Back in the main function.\n");

    return 0;
}

// Function Definition
void displayMessage() {
    printf("This is a message from the function.\n");
}
```

Logic and Flow:

- **Function `displayMessage`:** Prints a message and has no return value.
- `main` **Function:** Calls `displayMessage`, prints a message, and continues execution.

Output:

```
This is a message from the function.
Back in the main function.
```

Example 3: Function with Parameters (Multiply)

```
#include <stdio.h>

// Function Declaration
int multiply(int a, int b);

int main() {
    // Function Call
```

```
    int result = multiply(5, 3);

    // Output
    printf("Product: %d\n", result);

    return 0;
}

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

Logic and Flow:

- **Function `multiply`:** Takes two parameters (`a` and `b`) and returns their product.
- `main` **Function:** Calls `multiply(5, 3)`, stores the result in `result`, and prints the product.

Output:

```
Product: 15
```

Example 4: Function without Parameters (Greet)

```
#include <stdio.h>

// Function Declaration
void greet();

int main() {
    // Function Call
    greet();

    // Output
    printf("Back in the main function.\n");

    return 0;
}

// Function Definition
void greet() {
    printf("Hello! Welcome to the program.\n");
}
```

Logic and Flow:

- **Function `greet`:** Prints a welcoming message and has no parameters.
- `main` **Function:** Calls `greet`, prints a message, and continues execution.

Output:

```
Hello! Welcome to the program.
Back in the main function.
```

Main Function in Detail:

The `main` function is a special function in C programming and serves as the entry point for the execution of a C program. Its primary role is to coordinate the overall flow of the program, acting as the starting point for program execution.

Key Aspects of the `main` Function:

1. **Entry Point:**
   - The program starts executing from the `main` function.
   - It is the first function that gets called when the program is run.
2. **Return Type `int`:**
   - The `main` function has a return type of `int`, indicating that it can return an integer value.
   - The return type of `int` is a convention that signals the status of program execution to the operating system.
3. **Return Value `0`:**
   - Conventionally, a return value of `0` from the `main` function indicates successful execution.
   - The `0` is often interpreted as "no errors" or "successful completion" by the operating system.

Why Does `main` Return `int` and Why `0`?

1. **Status Code:**
   - The return value of `main` is often used as a status code.
   - A return value of `0` conventionally signifies that the program executed successfully without errors.
2. **Error Signaling:**
   - Non-zero return values (e.g., `1`, `2`, etc.) are commonly used to indicate specific errors or abnormal program termination.
   - These values can be used to communicate error conditions to the operating system.

Example:

```
#include <stdio.h>

int main() {
    // Main logic of the program
```

```
    return 0; // Indicates successful execution
}
```

In this example:

- The `main` function performs the main logic of the program.
- The `return 0;` statement indicates that the program executed successfully.
- The absence of `return` statement or an explicit `return` without a value is equivalent to `return 0;` and also indicates successful execution.

Other Return Values:

- Returning a non-zero value (e.g., `return 1;`, `return 2;`) from `main` is commonly used to signal errors or abnormal terminations.
- The specific meaning of non-zero return values can be defined by the programmer or adhere to certain conventions.

▼ Recursion

Recursive Function:

A recursive function is a function that calls itself either directly or indirectly in order to solve a problem. Recursive functions have two main components:

1. **Base Case(s):** The condition(s) under which the function stops calling itself and returns a specific result without further recursion.
2. **Recursive Case(s):** The condition(s) under which the function calls itself to solve a smaller instance of the same problem.

Examples of Recursive Functions:

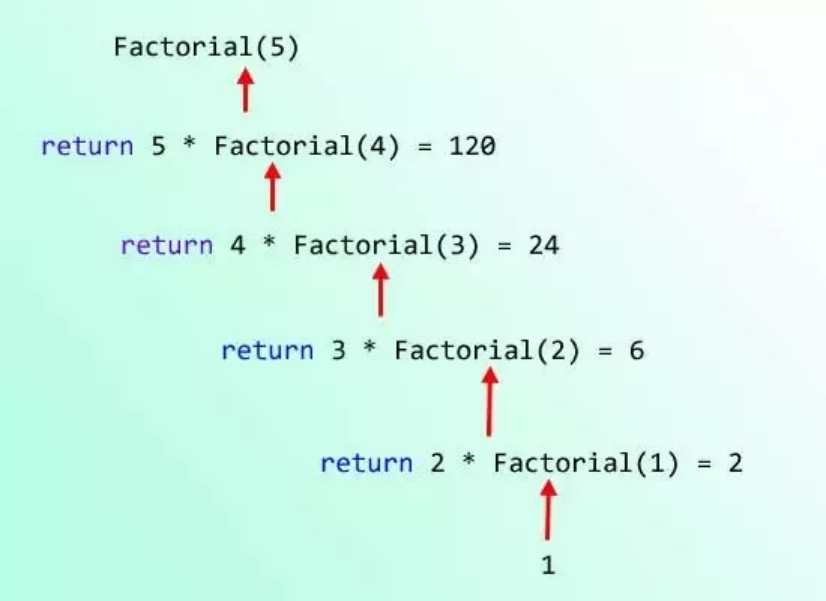
1. Factorial Calculation:

```
#include <stdio.h>

// Recursive function to calculate factorial
int factorial(int n) {
    // Base case
    if (n == 0 || n == 1) {
        return 1;
    } else {
        // Recursive case
        return n * factorial(n - 1);
    }
}

int main() {
    int num = 5;
    printf("Factorial of %d is: %d\n", num, factorial(num));
```

```
    return 0;
}
```

Logic and Flow:

1. **Base Case:** When `n` becomes 0 or 1, the base case is triggered, and the function returns 1.
2. **Recursive Case:** In each recursive call, the function multiplies `n` with the result of the factorial of `(n - 1)`. This process continues until the base case is reached.



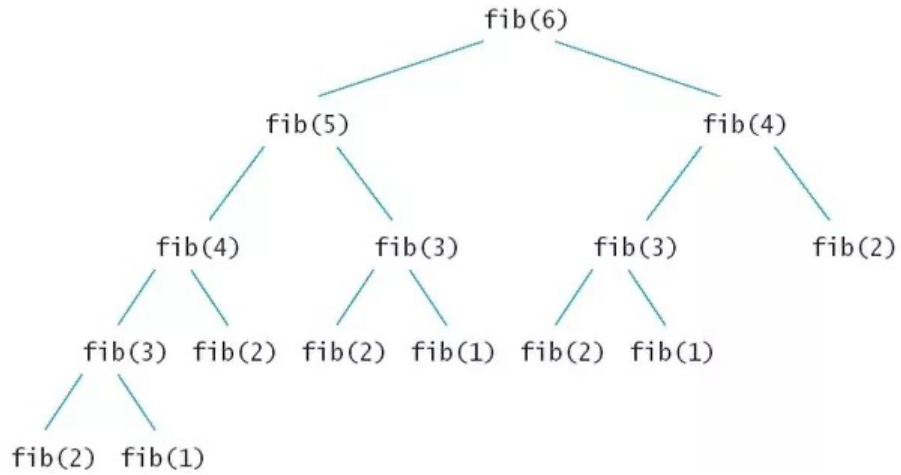
2. Fibonacci Series:

```
#include <stdio.h>

// Recursive function to generate Fibonacci series
int fibonacci(int n) {
    // Base case
    if (n <= 1) {
        return n;
    } else {
        // Recursive case
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    int terms = 6;
    printf("Fibonacci Series up to %d terms: ", terms);
    for (int i = 0; i < terms; i++) {
        printf("%d ", fibonacci(i));
    }
```

```
    printf("\n");
    return 0;
}
```

Logic and Flow:

1. **Base Case:** When `n` becomes 0 or 1, the base case is triggered, and the function returns `n`.
2. **Recursive Case:** In each recursive call, the function sums the results of the Fibonacci series for `(n - 1)` and `(n - 2)`. This process continues until the base case is reached.



3. Binary Search:

```
#include <stdio.h>

// Recursive function for binary search
int binarySearch(int arr[], int low, int high, int key) {
    // Base case
    if (low > high) {
        return -1; // Element not found
    }

    int mid = (low + high) / 2;

    // Base case: Element found
    if (arr[mid] == key) {
        return mid;
    } else if (arr[mid] > key) {
        // Recursive case: Search in the left half
        return binarySearch(arr, low, mid - 1, key);
    } else {
        // Recursive case: Search in the right half
        return binarySearch(arr, mid + 1, high, key);
    }
```

```
}

int main() {
    int arr[] = {2, 4, 6, 8, 10, 12, 14, 16};
    int key = 10;
    int size = sizeof(arr) / sizeof(arr[0]);

    int result = binarySearch(arr, 0, size - 1, key);

    if (result != -1) {
        printf("Element %d found at index %d\n", key, result);
    } else {
        printf("Element %d not found in the array\n", key);
    }

    return 0;
}
```

Advantages of Recursive Functions:

1. **Simplicity and Readability:** Recursive functions often express the solution to a problem in a clear and concise manner.
2. **Elegant Solutions:** Some problems have natural recursive structures, making recursive solutions more elegant and intuitive.
3. **Modularity:** Recursive functions can break down complex problems into simpler subproblems, promoting modular code.

Disadvantages of Recursive Functions:

1. **Performance Overhead:** Recursive calls may consume more memory and time due to function call overhead and maintaining a call stack.
2. **Stack Overflow:** Excessive recursion without proper base cases may lead to a stack overflow, causing the program to crash.
3. **Debugging Complexity:** Debugging recursive functions can be challenging due to multiple instances of the function on the call stack.

When to Choose Recursion or Looping:

Example where Looping is Preferable: Iterative Task - Calculating Factorial

```
#include <stdio.h>

// Loop-based factorial calculation
int factorialLoop(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
```

```
    }
    return result;
}

int main() {
    int num = 5;
    printf("Factorial of %d is: %d\n", num, factorialLoop(num));
    return 0;
}
```

Logic and Flow:

1. A loop iteratively multiplies the current result by each positive integer from 1 to `n`.
2. The loop-based approach avoids the overhead of multiple function calls, making it more efficient for this specific task compared to recursion.

Recursion vs. Looping for Factorial:

1. **Recursion (Like Writing Instructions):**
   - Imagine you want to find 5! (5 factorial).
   - You write down that 5! = 5 * 4! (5 times the factorial of 4).
   - Then, 4! = 4 * 3! (4 times the factorial of 3), and so on until you reach 0 or 1.
   - It's like writing a set of instructions on paper, telling yourself how to calculate each step.
2. **Looping (Direct Calculation):**
   - Now, let's do it differently.
   - You start with a blank sheet and multiply: 1 * 2 * 3 * 4 * 5.
   - Instead of writing down instructions, you directly calculate and update the result.

In Simple Terms:

- **Recursion:**
  - You write a plan on how to calculate each step.
  - Creates a set of instructions (like a storybook).
- **Looping:**
  - You directly perform the calculations.
  - No need for writing instructions; you just keep updating the result.

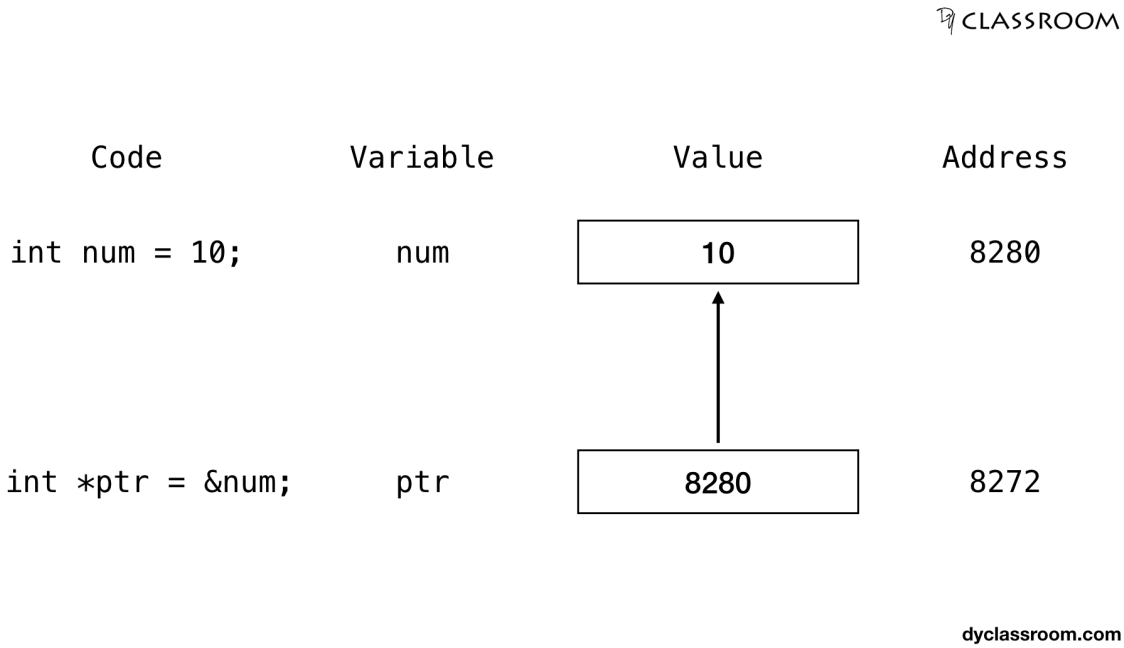
Why Looping Can Be Better:

- **Less Paper (Memory):**
  - Looping uses less "paper" (memory) because you don't keep creating new sheets of instructions.
- **Faster for Some Problems:**
  - For simple calculations like factorials, looping can be faster because it directly performs the steps without the extra overhead of managing instructions.

▼ Pointers

Pointers in C:

In C programming, a pointer is a variable that stores the memory address of another variable. Pointers are powerful features that allow manipulation of memory directly. They play a crucial role in tasks like dynamic memory allocation and efficient data access.



Example:

```
#include <stdio.h>

int main() {
    // Declare a variable and a pointer
    int num = 42;
    int *ptr;

    // Assign the address of 'num' to the pointer
    ptr = &num;

    // Access the value using the pointer
    printf("Value of num: %d\n", *ptr);

    // Modify the value using the pointer
    *ptr = 100;

    // Check the modified value
    printf("Modified value of num: %d\n", num);
```

```
    return 0;
}
```

Explanation:

1. **Pointer Declaration:**
   - `int *ptr;`: Declares a pointer `ptr` that can store the memory address of an integer.
2. **Assigning Address:**
   - `ptr = &num;`: Assigns the address of the variable `num` to the pointer `ptr`.
3. **Dereferencing:**
   - `ptr`: Dereferences the pointer, giving access to the value stored at the memory address it points to.
4. **Modifying Value:**
   - `ptr = 100;`: Modifies the value at the memory location pointed by `ptr`, affecting the original variable `num`.

Why Pointers Exist:

1. **Direct Memory Access:**
   - Pointers provide a way to directly access and manipulate memory, enabling efficient data handling.
2. **Dynamic Memory Allocation:**
   - Pointers allow dynamic allocation of memory during program execution, facilitating flexible memory management.
3. **Passing Addresses:**
   - Pointers enable functions to modify variables in the calling code by passing their addresses.
4. **Efficient Data Structures:**
   - Pointers are fundamental for implementing complex data structures like linked lists, trees, and graphs.
5. **Arrays and Strings:**
   - Arrays and strings in C are implemented using pointers.

Advantages of Pointers:

1. **Efficiency:**
   - Pointers provide direct access to memory, leading to efficient data manipulation.
2. **Dynamic Memory Allocation:**
   - Enables dynamic allocation and deallocation of memory as needed.
3. **Pass by Reference:**
   - Allows functions to modify variables in the calling code by passing their addresses.

Disadvantages of Pointers:

1. **Memory Management:**
   - Incorrect use of pointers can lead to memory leaks or segmentation faults.
2. **Complexity:**
   - Pointers add complexity to code, requiring careful handling to avoid errors.
3. **Security Risks:**
   - Improper use of pointers may lead to security vulnerabilities like buffer overflows.

Application:

- Pointers are extensively used in implementing data structures, managing memory, and optimizing algorithms. They are essential for low-level programming tasks and applications where direct memory access is crucial.

Pointer Examples:

1. Simple Pointer to an Integer:

```
#include <stdio.h>

int main() {
    // Declare an integer variable and a pointer to int
    int num = 42;
    int *ptr;

    // Assign the address of 'num' to the pointer
    ptr = &num;

    // Access the value using the pointer
    printf("Value of num: %d\n", *ptr);

    return 0;
}
```

- In this example, a pointer `ptr` is declared using `int *ptr;`. It is then assigned the address of the integer variable `num`. The value stored at the memory location pointed by `ptr` is accessed using the dereference operator ``.

2. Pointer to a Character Array (String):

```
#include <stdio.h>

int main() {
    // Declare a character array and a pointer to char
    char str[] = "Hello, World!";
    char *ptr;
```

```
    // Assign the address of the first character to the pointer
    ptr = str;

    // Access and display the characters using the pointer
    printf("String: ");
    while (*ptr != '\0') {
        printf("%c", *ptr);
        ptr++;
    }
    printf("\n");

    return 0;
}
```

- Here, a pointer `ptr` to a character (`char`) is declared. It is assigned the address of the first character in the character array `str`. The program then uses the pointer to traverse and print each character in the array until the null terminator (`'\\0'`) is encountered.

3. Pointer to a Double (Floating-Point):

```
#include <stdio.h>

int main() {
    // Declare a double variable and a pointer to double
    double pi = 3.14159;
    double *ptr;

    // Assign the address of 'pi' to the pointer
    ptr = &pi;

    // Access the value using the pointer
    printf("Value of pi: %.5f\n", *ptr);

    return 0;
}
```

- In this example, a pointer `ptr` to a double (`double *ptr;`) is declared. It is assigned the address of the double variable `pi`. The value stored at the memory location pointed by `ptr` is then accessed and printed.

why & operator not used in char pointer?

In C, the `&` operator is used to obtain the address of a variable. However, when dealing with character arrays or strings, you don't necessarily need to use the `&` operator to get the address of the array. This is because the name of an array, when used in most expressions, is automatically converted to a pointer to its first element.

For example, consider the following code:

```c
#include <stdio.h>

int main() {
    // Declare a character array and a pointer to char
    char str[] = "Hello, World!";
    char *ptr;

    // Assign the address of the first character to the pointer
    ptr = str;

    // Access and display the characters using the pointer
    printf("String: ");
    while (*ptr != '\0') {
        printf("%c", *ptr);
        ptr++;
    }
    printf("\n");

    return 0;
}
```

In this code, `ptr = str;` assigns the address of the first character of the `str` array to the pointer `ptr` without the need for the `&` operator. The reason for this is that in most contexts, the name of an array is automatically converted to a pointer pointing to its first element.

So, when working with strings in C, using the array name directly as a pointer is a common and idiomatic practice. The use of `&` with arrays is not incorrect, but it's typically unnecessary due to this automatic conversion.

▼ Passing arrays as function arguments

Passing Arrays as Function Arguments:

In C, arrays can be passed as arguments to functions. When an array is passed to a function, it is treated as a pointer to the first element of the array. This allows functions to operate on arrays without making a copy of the entire array, which can be more efficient in terms of memory and performance. When passing arrays as function arguments in C, you're essentially passing the base address of the array to the function. Unlike other data types, arrays decay into pointers when passed to functions. This means the function receives a pointer to the first element of the array. Understanding how arrays are passed to functions is important for efficient manipulation and processing of array elements within functions.

Introduction:

When passing an array to a function, the function typically receives a pointer to the array's first element. This pointer can be used to access and manipulate the array elements within the function. It provides a way to work with arrays in a modular and reusable manner.

Where is it Used:

Passing arrays as function arguments is commonly used when there is a need to perform operations on an array, and the result needs to be reflected outside the function. This is especially useful for functions

that perform tasks such as sorting, searching, or modifying array elements.

Examples:

1. Sum of Array Elements:

```
#include <stdio.h>

// Function to calculate the sum of array elements
int sumArray(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    int result = sumArray(numbers, size);

    printf("Sum of array elements: %d\n", result);

    return 0;
}
```

Output:

```
Sum of array elements: 15
```

2. Update Array Elements:

```
#include <stdio.h>

// Function to increment each element of the array by a given value
void incrementArray(int arr[], int size, int value) {
    for (int i = 0; i < size; i++) {
        arr[i] += value;
    }
}

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    int incrementValue = 10;
```

```
    incrementArray(numbers, size, incrementValue);

    printf("Updated array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Updated array elements: 11 12 13 14 15
```

3. Find Maximum Element:

```
#include <stdio.h>

// Function to find the maximum element in the array
int findMax(int arr[], int size) {
    int max = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

int main() {
    int numbers[] = {3, 8, 2, 10, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    int maxElement = findMax(numbers, size);

    printf("Maximum element in the array: %d\n", maxElement);

    return 0;
}
```

Output:

```
Maximum element in the array: 10
```

Key Points:

- Arrays are passed to functions by reference.
- The size of the array should be passed as a separate parameter.
- Functions can modify the original array passed to them.
- Array manipulation within functions is a common practice in C programming.

▼ Call by value and call by reference

Call by Value and Call by Reference in C

- These are two different ways to pass parameters to functions in C.
- **Call by Value:** The actual value is passed to the function.
- **Call by Reference:** The memory address (reference) of the variable is passed to the function.

Call by Value:

In Call by Value, when a function is called, a copy of the actual parameters (values) is passed to the function. This means that any modifications made to the parameters inside the function have no effect on the original values outside the function. It follows a straightforward and predictable model.

Call by Reference:

In Call by Reference, instead of passing copies of values, the memory address (reference) of the actual parameters is passed to the function. This allows the function to directly access and modify the original data, providing more flexibility but also introducing complexities.

Why Are They Used?

- Understanding these concepts is crucial for effective parameter passing in functions.
- They offer flexibility and efficiency in handling data within functions.

Real-Life Reference/Example:

- Think of ordering food at a restaurant.
  - **Call by Value:** You order a dish, and you get the dish delivered to your table. The dish is like a copy of the menu item.
  - **Call by Reference:** You order a dish, and the kitchen gets your table number. When the dish is ready, it's brought to your table. Any changes to the dish (like adding salt) affect what's on your table.

Example 1: Call by Value - Swapping Numbers:

```
#include <stdio.h>

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
```

```
    int x = 5, y = 10;
    printf("Before swap: x = %d, y = %d\n", x, y);

    swap(x, y);

    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;
}
```

Output:

```
Before swap: x = 5, y = 10
After swap: x = 5, y = 10
```

Logic and Flow Explanation:

- The values of `x` and `y` are passed to the `swap` function.
- Inside the function, swapping is performed, but it only affects the local copies of `a` and `b`.
- The original values of `x` and `y` remain unchanged.

Example 2: Call by Reference - Swapping Numbers:

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    printf("Before swap: x = %d, y = %d\n", x, y);

    swap(&x, &y);

    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;
}
```

Output:

```
Before swap: x = 5, y = 10
After swap: x = 10, y = 5
```

Logic and Flow Explanation:

- The addresses of `x` and `y` are passed to the `swap` function.
- Inside the function, swapping is performed using pointer dereferencing, directly modifying the original values of `x` and `y`.

Example 3: Call by Reference - Increment Array Elements:

```
#include <stdio.h>

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

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    printf("Before increment: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", numbers[i]);
    }

    incrementArray(numbers, size);

    printf("\\nAfter increment: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", numbers[i]);
    }

    return 0;
}
```

Output:

```
Before increment: 1 2 3 4 5
After increment: 2 3 4 5 6
```

Logic and Flow Explanation:

- The array `numbers` is passed to the `incrementArray` function by reference.
- Each element of the array is incremented within the function, directly affecting the original array.

Call by Value:

Advantages:

1. **Simplicity:** It's simpler to understand and implement.

2. **Predictability:** Values passed to a function remain unchanged outside the function, providing a clear boundary.
3. **Isolation:** Function cannot accidentally modify the actual parameters.

Disadvantages:

1. **Performance Overhead:** Copying large data structures may result in performance issues.
2. **Memory Usage:** Duplicate data consumes more memory.
3. **Limited Modification:** The original value cannot be modified inside the function.

Best Scenario:

- When dealing with small and immutable data types or where modification is not required.

Example:

```
#include <stdio.h>

void increment(int x) {
    x++;
}

int main() {
    int num = 5;
    increment(num);
    printf("Original value: %d\n", num);  // Output: Original value: 5
    return 0;
}
```

Call by Reference:

Advantages:

1. **Efficiency:** No need to copy large data structures.
2. **Modification:** Function can modify the original parameters.
3. **Resource Conservation:** Saves memory by avoiding duplication.

Disadvantages:

1. **Complexity:** More complex to understand and implement.
2. **Unintended Modifications:** Requires caution to avoid unintended modifications.
3. **Security Issues:** Potential for security vulnerabilities if not handled carefully.

Best Scenario:

- When dealing with large data structures or when modifications to original parameters are needed.

Example:

```
#include <stdio.h>

void increment(int *x) {
```

```
    (*x)++;
}

int main() {
    int num = 5;
    increment(&num);
    printf("Modified value: %d\n", num);  // Output: Modified value: 6
    return 0;
}
```

Choosing the Right Approach:

- **Call by Value:** Use when the function does not need to modify the original values, and simplicity is a priority.
- **Call by Reference:** Use when modifying the original values inside the function is required or when dealing with large data structures for efficiency reasons.

▼ Strings in C

Strings in C:

A string in C is an array of characters that represents a sequence of characters or text. Unlike some other programming languages, C does not have a built-in string data type. Instead, strings are represented as arrays of characters terminated by a null character (`\0`).

Characteristics of Strings:

1. **Array of Characters:**
   - A string is essentially an array of characters.
   - Example declaration: `char str[10];`
2. **Null Termination:**
   - Strings in C are null-terminated, meaning they end with a null character (`'\0'`).
   - The null character is automatically added when a string literal is initialized.
3. **Memory Usage:**
   - Each character in the string occupies one byte of memory.
   - The null character is used to mark the end of the string.
4. **Accessing Characters:**
   - Individual characters within a string can be accessed using array notation.
   - Example: `char firstChar = str[0];`

Declaration of Strings:

Strings in C can be declared in various ways:

```
// Declaration and initialization using a string literal
char greeting[] = "Hello, World!";
```

```
// Declaration and later assignment
char name[20];
```

Accessing and Manipulating Strings:

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

int main() {
    char greeting[] = "Hello, World!";

    // Accessing characters
    char firstChar = greeting[0];

    // String length
    int length = strlen(greeting);

    // Concatenation
    char name[20] = "John";
    strcat(name, " Doe");

    // String comparison
    int result = strcmp(name, "John Doe");

    // Output
    printf("First character: %c\n", firstChar);
    printf("Length of string: %d\n", length);
    printf("Concatenated string: %s\n", name);
    printf("String comparison result: %d\n", result);

    return 0;
}
```

String Manipulation in C:

String manipulation involves performing various operations on strings to modify, combine, or extract information. In C, string manipulation is typically done using standard library functions along with array and pointer operations.

Common String Manipulation Functions:

1. `strlen`:
    - Returns the length of a string (excluding the null character).
    - Example:

      ```
      char str[] = "Hello";
      int length = strlen(str);
      ```

2. `strcpy` and `strncpy`:
   - Copy one string to another. `strcpy` copies the entire string, while `strncpy` allows specifying a maximum number of characters to copy.
   - Examples:

```
char source[] = "Hello";
char destination[10];

// Using strcpy
strcpy(destination, source);

// Using strncpy
strncpy(destination, source, 3);  // Copies only the first 3 characters
```

3. `strcat` and `strncat`:
   - Concatenate one string to the end of another. `strcat` appends the entire string, while `strncat` allows specifying a maximum number of characters to concatenate.
   - Examples:

```
char str1[20] = "Hello";
char str2[] = " World";

// Using strcat
strcat(str1, str2);

// Using strncat
strncat(str1, str2, 3);  // Concatenates only the first 3 characters of st
r2
```

4. `strcmp`:
   - Compare two strings lexicographically. Returns 0 if the strings are equal.
   - Example:

```
char str1[] = "apple";
char str2[] = "banana";
int result = strcmp(str1, str2);
```

5. `strchr` and `strrchr`:
   - Locate the first or last occurrence of a character in a string.
   - Examples:

```
char str[] = "Hello, world!";
char *found = strchr(str, 'o');   // Points to the first 'o'
```

```
// Using strrchr
char *lastFound = strrchr(str, 'o');  // Points to the last 'o'
```

Additional String Operations:

1. **Substring Extraction:**
   - Extracting a substring from a larger string.
   - Example:

```
char sentence[] = "The quick brown fox";
char subString[10];
strncpy(subString, sentence + 4, 5);  // Extracts "quick"
```

2. **String Conversion:**
   - Converting a string to uppercase or lowercase.
   - Example:

```
char word[] = "Hello";
for (int i = 0; word[i] != '\0'; i++) {
    word[i] = tolower(word[i]);  // Convert to lowercase
}
```

Examples on string operations:

1. Length of a String (`strlen`):

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

int main() {
    char str[] = "Hello";
    int length = strlen(str);

    printf("Length of the string: %d\n", length);

    return 0;
}
```

2. Copying Strings (`strcpy` and `strncpy`):

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

int main() {
    char source[] = "Hello";
```

```
    char destination[10];

    // Using strcpy
    strcpy(destination, source);
    printf("Copied string using strcpy: %s\n", destination);

    // Using strncpy
    strncpy(destination, source, 3);
    destination[3] = '\0';  // Null-terminate manually
    printf("Copied string using strncpy: %s\n", destination);

    return 0;
}
```

3. Concatenating Strings (`strcat` and `strncat`):

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

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

    // Using strcat
    strcat(str1, str2);
    printf("Concatenated string using strcat: %s\n", str1);

    // Using strncat
    strncat(str1, str2, 3);
    printf("Concatenated string using strncat: %s\n", str1);

    return 0;
}
```

4. Comparing Strings (`strcmp`):

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

int main() {
    char str1[] = "apple";
    char str2[] = "banana";

    int result = strcmp(str1, str2);

    if (result == 0) {
        printf("Both strings are equal\n");
    } else {
```

```
        printf("Strings are not equal\n");
    }

    return 0;
}
```

5. Locating Characters (`strchr` and `strrchr`):

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

int main() {
    char str[] = "Hello, world!";

    // Using strchr
    char *found = strchr(str, 'o');
    printf("First 'o' found at index: %ld\n", found - str);

    // Using strrchr
    char *lastFound = strrchr(str, 'o');
    printf("Last 'o' found at index: %ld\n", lastFound - str);

    return 0;
}
```

▼ Exercises in string

Exercise 1 : To check whether the given string is a palindrome or not.

```
#include<stdio.h>
#include<conio.h>
void main ()
{
int i, pal_len=0, len=0;
char pal_str[100];
printf("Enter a string to check whether it is palindrome or not: ");
gets(pal_str);
for(i=0;i<100;i++)
{
if(pal_str[i]=='\0')
{
break;
}
len++; // Calculating length of string
}
for(i=0;i<len;i++)
```

```
{
if(pal_str[i]==pal_str[len-1-i])
{
pal_len++;
}
else
{
printf("Entered string is not palindrome");
getch();
exit(0);
}
}
if(len==pal_len)
printf("Entered string is palindrome");
}
```

Example 2: To count the following in the given string-
1. Vowels
2. Consonants
3. Digits
4. Whitespaces
5. Special chararcters

```
#include<stdio.h>
#include<conio.h>
void countCharType(char str[100])
{
int Vowels = 0, consonant = 0, WhiteSpace = 0, SpecialChar=0, digit = 0,i;
char ch;
for (i = 0; str[i]!='\0'; i++)
{
ch = str[i];
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
{
// To handle upper case letters
ch = tolower(ch);
if (ch == 'a' || ch == 'e' || ch == 'i' ||ch == 'o' || ch == 'u')
Vowels++;
else
Consonant ++;
}
else if (ch >= '0' && ch <= '9')
digit++;
else if(ch == ' ')
{
WhiteSpace++;
}
```

```
else
{
SpecialChar++;
}
}
printf("\n Vowels:%d\n ",Vowels);
printf("Consonant :%d\n ", Consonant );
printf("Digit:%d\n " , digit);
printf("WhiteSpace:%d\n ", WhiteSpace);
printf("SpecialChar:%d\n ", SpecialChar);
}
int main()
{
char str[100];
printf("Enter input : ");
gets(str);
countCharType(str); // Calling another function
return 0;
}
```

▼ Dynamic memory allocation and `malloc calloc realloc` and `free`

Dynamic Memory Allocation:

1. `malloc(size_t size)` (Memory Allocation):

- Allocates a specified number of bytes of memory.
- Returns a pointer to the first byte of the allocated memory.
- The memory content is uninitialized.

Example with `int`:

```
int *intPtr = (int *)malloc(5 * sizeof(int));
```

Example with `float`:

```
float *floatPtr = (float *)malloc(3 * sizeof(float));
```

Example with `char`:

```
char *charPtr = (char *)malloc(10 * sizeof(char));
```

2. `calloc(size_t num, size_t size)` (Contiguous Allocation):

- Allocates a block of memory for an array of elements, each with a size of `size` bytes.

- Initializes the memory content to zero.
- Returns a pointer to the first byte of the allocated memory.

Example with `int`:

```
int *intArray = (int *)calloc(5, sizeof(int));
```

3. `realloc(void *ptr, size_t size)` (Reallocation):

- Changes the size of the memory block pointed to by `ptr` to `size` bytes.
- If the block is extended, the contents are preserved up to the original size.
- If the block is reduced, the excess data may be lost.
- Returns a pointer to the newly allocated memory.

Example with `int` (after `malloc`):

```
int *newIntPtr = (int *)realloc(intPtr, 10 * sizeof(int));
```

4. `free(void *ptr)` (Freeing Memory):

- Deallocates the memory block previously allocated by `malloc`, `calloc`, or `realloc`.
- It doesn't change the value of the pointer itself; it just marks the memory as available for reuse.

Example:

```
free(intPtr);
```

Importance of `free`:

- `free` is crucial to release allocated memory, preventing memory leaks.
- Not freeing memory may lead to a gradual loss of available memory, impacting program performance.
- It is a good practice to free memory when it is no longer needed, ensuring efficient memory management.

Example (Using `free`):

```
free(intArray);
free(newIntPtr);
free(floatPtr);
free(charPtr);
```

In the above examples, we allocate memory for different data types, reallocate memory using `realloc`, and free the allocated memory using `free`. Proper memory management is essential to prevent memory leaks and optimize program performance.

Here's a simple example demonstrating the use of `malloc` and `free`:

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

int main() {
    // Allocate memory for an integer
    int *num = (int *)malloc(sizeof(int));

    if (num == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Assign a value to the allocated memory
    *num = 42;

    // Print the value
    printf("Value: %d\\n", *num);

    // Free the allocated memory
    free(num);

    return 0;
}
```

Remember to always check if the memory allocation is successful (i.e., the returned pointer is not `NULL`) and to free the allocated memory using `free()` when it is no longer needed to avoid memory leaks.

Example 1 : Dynamic Array

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int size;

    printf("Enter the size of the array: ");
    scanf("%d", &size);

    // Dynamically allocate memory for the array
    int *arr = (int*)malloc(size * sizeof(int));

    if (arr == NULL) {
        printf("Memory allocation failed!");
        return 1;
```

```
    }

    // Populate the array with user input
    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    // Display the array elements
    printf("The array elements are: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    // Free dynamically allocated memory
    free(arr);

    return 0;
}
```

Logic:

1. Ask the user to input the size of the array.
2. Dynamically allocate memory for the array based on the input size.
3. Prompt the user to enter elements for the array.
4. Populate the dynamically allocated array with user input.
5. Display the elements of the array.
6. Free the dynamically allocated memory.

Output (Sample Run):

```
Enter the size of the array: 5
Enter 5 elements:
1
2
3
4
5
The array elements are: 1 2 3 4 5
```

Example 2 : Store a character string in a block memory space created by malloc() modify the same and store a larger string.

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

```
#include<string.h>
#define NULL1 0
void main()
{
char *buffer;
/* Allocating memory */
if((buffer = (char *)malloc(6)) == NULL1)
{
printf("malloc failed.\n");
exit(1);
}
printf("Buffer of size %d created \n",_msize(buffer));
strcpy(buffer, "MYSORE");
printf("\nBuffer contains: %s \n ", buffer);
/* Realloction */
if((buffer = (char *)realloc(buffer, 25)) == NULL)
{
printf("Reallocation failed. \n");
exit(1);
}
printf("\nBuffer size modified. \n");
printf("Modified Buffer size is %d \n",_msize(buffer));
printf("\nBuffer still contains: %s \n",buffer);
strcpy(buffer, "BENGALURU");
printf("\nBuffer now contains: %s \n",buffer);
/* Freeing memory */
free(buffer);
}
```

Logic:

1. Allocate memory for a character buffer using `malloc()` with a size of 6 bytes.
2. Print the initial size of the buffer using `_msize()` function.
3. Copy the string "MYSORE" into the allocated buffer using `strcpy()`.
4. Reallocate memory for the buffer using `realloc()` to increase its size to 25 bytes.
5. Copy the string "BENGALURU" into the buffer using `strcpy()` after reallocation.
6. Free the dynamically allocated memory using `free()` to prevent memory leaks.
7. Buffer used here is variable name given to the input string.

Output:

```
Buffer of size 6 created
Buffer contains: MYSORE
```

```
Buffer size modified.
Modified Buffer size is 25

Buffer still contains: MYSORE

Buffer now contains: BENGALURU
```

▼ Structures

Structures in C:

A structure in C is a composite data type that allows you to group variables of different data types under a single name. It enables you to create a more complex and organized representation of data.

How are Structures Declared?

To declare a structure, you use the `struct` keyword followed by the structure name and a list of its member variables along with their data types. Here's the syntax:

```
struct structure_name {
    data_type member_name1;
    data_type member_name2;
    // ... more members ...
};
```

Sample:

```
#include <stdio.h>

// Declaration of a structure named 'Person'
struct Person {
    char name[50];
    int age;
    float height;
};
```

When are Structures Required?

1. **Grouping Related Data:**
   - Structures are useful when you want to represent entities that have multiple properties or attributes.
   - For example, a `Person` structure can represent an individual with a name, age, and height.
2. **Organization:**
   - Structures help in organizing and managing data in a more meaningful way.
   - You can access members of a structure using dot notation (`person.name`, `person.age`, etc.).
3. **Code Readability:**

- Using structures enhances code readability by grouping related variables together.

4. **Mathematical Operations**:
   - For handling complex numbers, distances (e.g., feet-inch system), or time periods (e.g., hour-minute-second system), structures provide a convenient way to represent and manipulate data.

Advantages of Using Structures:

1. **Heterogeneous Collection**: Structures allow you to create user-defined data types that can store items with different data types. For instance, our `Student` structure combines integers, strings, and floats.
2. **Reduced Complexity**: Instead of using separate variables for each piece of data, you can create an array of structures. This simplifies code and makes it more manageable.
3. **Increased Productivity**: Dealing with records containing heterogeneous data becomes easier, leading to improved productivity.
4. **Maintainability**: Representing complex records using a single structure name enhances code maintainability.
5. **Enhanced Readability**: Well-organized structures contribute to better code readability, especially in larger projects.
6. **Suitable for Mathematical Operations**: Structures allow you to handle complex numbers, distances, and time periods effectively.

Disadvantages of Using Structures:

1. **Increased Program Complexity**: Excessive use of structures can make the program harder to manage.
2. **Memory Overhead**: Each structure variable consumes memory, which can be inefficient if you have many instances.
3. **Padding and Alignment**: Structures may introduce padding for alignment purposes, leading to memory wastage.

Dot Operator and Structure:

The dot operator (`.`) in C is used to access the members of a structure. It allows you to refer to a specific member within a structure variable. The syntax for using the dot operator is as follows:

```
struct MyStruct {
    int member1;
    float member2;
    char member3;
};

// Declare a structure variable
struct MyStruct myVar;

// Access structure members using the dot operator
myVar.member1 = 10;
```

```
myVar.member2 = 3.14;
myVar.member3 = 'A';
```

In this example, `myVar.member1`, `myVar.member2`, and `myVar.member3` are used to access and modify the values of the members within the structure.

Here's a more detailed breakdown:

- **Declaration of Structure:**

```
struct MyStruct {
    int member1;
    float member2;
    char member3;
};
```

- **Declaration of Structure Variable:**

```
struct MyStruct myVar;
```

- **Accessing Structure Members:**

```
myVar.member1 = 10;
myVar.member2 = 3.14;
myVar.member3 = 'A';
```

In this way, the dot operator facilitates the interaction with individual members of a structure variable.

Example 1: Person Information:

```
#include <stdio.h>

// Declaration of a structure named 'Person'
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Creating an instance of the 'Person' structure
    struct Person person1;

    // Initializing structure members
    strcpy(person1.name, "John Doe");
    person1.age = 25;
    person1.height = 5.9;

    // Displaying person information
```

```
    printf("Person Information:\n");
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.2f feet\n", person1.height);

    return 0;
}
```

```
Person Information:
Name: John Doe
Age: 25
Height: 5.90 feet
```

Example 2: Book Information:

```
#include <stdio.h>

// Declaration of a structure named 'Book'
struct Book {
    char title[100];
    char author[50];
    int pages;
    float price;
};

int main() {
    // Creating an instance of the 'Book' structure
    struct Book book1;

    // Initializing structure members
    strcpy(book1.title, "The Catcher in the Rye");
    strcpy(book1.author, "J.D. Salinger");
    book1.pages = 224;
    book1.price = 12.99;

    // Displaying book information
    printf("Book Information:\n");
    printf("Title: %s\n", book1.title);
    printf("Author: %s\n", book1.author);
    printf("Pages: %d\n", book1.pages);
    printf("Price: $%.2f\n", book1.price);

    return 0;
}
```

```
Book Information:
Title: The Catcher in the Rye
Author: J.D. Salinger
Pages: 224
Price: $12.99
```

Example 3: Complex Numbers

```
#include <stdio.h>
struct Complex {
    float real;
    float imaginary;
};

int main() {
    struct Complex c1 = {3.0, 4.0};
    struct Complex c2 = {1.5, -2.0};

    struct Complex sum = {c1.real + c2.real, c1.imaginary + c2.imaginary};

    printf("Sum: %.2f + %.2fi\n", sum.real, sum.imaginary);

    return 0;
}
```

Output:

```
Sum: 4.50 + 2.00i
```

Example 4: Student Records

```
#include <stdio.h>
struct Student {
    int roll_number;
    char name[30];
    float marks;
    char address[50];
};

int main() {
    struct Student s1 = {101, "Alice", 85.5, "123 Main St"};
    struct Student s2 = {102, "Bob", 78.0, "456 Elm Ave"};

    printf("Student 1: Roll Number %d, Name: %s, Marks: %.2f\n", s1.roll_number,
s1.name, s1.marks);
    printf("Student 2: Roll Number %d, Name: %s, Marks: %.2f\n", s2.roll_number,
```

```
s2.name, s2.marks);

    return 0;
}
```

Output:

```
Student 1: Roll Number 101, Name: Alice, Marks: 85.50
Student 2: Roll Number 102, Name: Bob, Marks: 78.00
```

Structure vs Union:

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

▼ When to choose structure and when to choose union?

Choosing Structure Over Union:

Example 1: Storing Information About a Person:

```
#include <stdio.h>

// Structure to store information about a person
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Using a structure to store information about a person
    struct Person person1;
    strcpy(person1.name, "John Doe");
    person1.age = 25;
    person1.height = 5.9;
```

```
    // Displaying information
    printf("Person Information:\n");
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.2f feet\n", person1.height);

    return 0;
}
```

Output:

```
Person Information:
Name: John Doe
Age: 25
Height: 5.90 feet
```

In this case, a structure is suitable because each member (`name`, `age`, and `height`) holds different types of information about the person, and all members are relevant simultaneously.

Choosing Union Over Structure:

Example 2: Efficient Storage of Data with Different Interpretations:

```
#include <stdio.h>

// Union to efficiently store data with different interpretations
union DataUnion {
    int integerValue;
    float floatValue;
};

int main() {
    // Using a union to store data efficiently
    union DataUnion data;
    data.integerValue = 42;

    // Accessing data as an integer
    printf("Integer Value: %d\n", data.integerValue);

    // Accessing the same data as a float
    data.floatValue = 3.14;
    printf("Float Value: %.2f\n", data.floatValue);

    return 0;
}
```

Output:

```
Integer Value: 42
Float Value: 3.14
```

In this case, a union is preferred when the same memory location needs to be interpreted differently at different times. It allows efficient storage and access to data with different types.

▼ Typedef

`typedef` in C:

`typedef` in C is a keyword that allows the programmer to create an alias or a new name for an existing data type. It enhances code readability and can make complex data types more concise. The syntax for `typedef` is as follows:

```
typedef existing_data_type new_type_name;
```

Purpose of `typedef` in C:

The primary purpose of `typedef` in C is to provide a way to create aliases for existing data types. This feature serves several purposes:

1. **Code Clarity:**
   - Improves the readability of code by giving more descriptive names to data types, making the code self-documenting.
2. **Abstraction:**
   - Allows for the abstraction of underlying data types, enabling better modularization of code.
3. **Simplifying Declarations:**
   - Simplifies complex declarations, especially those involving pointers or function pointers, making them more comprehensible.
4. **Code Maintenance:**
   - Facilitates code maintenance by centralizing type definitions, making it easier to update data types throughout the codebase.

Advantages of `typedef` in C:

1. **Code Readability:**
   - `typedef` allows programmers to create more descriptive and self-explanatory names for data types, improving code readability.
2. **Abstraction:**
   - It provides a level of abstraction by hiding the implementation details of a data type, making the code more modular.
3. **Code Maintenance:**

- When the underlying data type needs to be changed, using `typedef` makes it easier to update the code in one place, reducing maintenance efforts.

4. **Portability:**
   - `typedef` can enhance code portability by abstracting away specific data type details, allowing for more flexible adaptation to different platforms.

Examples:

1. Basic Type Definition:

```
#include <stdio.h>

// Define a new name for 'int'
typedef int Integer;

int main() {
    Integer num = 42;
    printf("Value of num: %d\n", num);
    return 0;
}
```

2. Array Type Definition:

```
#include <stdio.h>

// Define a new name for an array of integers
typedef int IntegerArray[5];

int main() {
    IntegerArray numbers = {1, 2, 3, 4, 5};

    // Access and print array elements
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    return 0;
}
```

3. Structure Type Definition:

```
#include <stdio.h>

// Define a new name for a structure
typedef struct {
```

```
    char name[20];
    int age;
} Person;

int main() {
    // Declare and initialize a structure variable
    Person student = {"John Doe", 21};

    // Access and print structure members
    printf("Name: %s\nAge: %d\n", student.name, student.age);

    return 0;
}
```

In these examples, `typedef` simplifies the usage of existing data types (`int`, array, and structure) by introducing more descriptive names (`Integer`, `IntegerArray`, and `Person`). This can lead to more readable and self-explanatory code.

Disadvantages of `typedef` in C:

1. **Overuse Complexity:**
   - Overusing `typedef` for every data type can lead to code complexity and reduced readability if not used judiciously.
2. **Namespace Pollution:**
   - Excessive use of `typedef` may introduce a large number of new names into the global namespace, potentially causing naming conflicts.
3. **Dependency on Naming Conventions:**
   - The readability gains heavily depend on choosing meaningful names, and if poor naming conventions are used, the code may become more confusing.

In summary, `typedef` is a tool that, when used judiciously, enhances code readability and maintainability by providing more meaningful names for data types, abstracting implementation details, and simplifying complex declarations.

▼ File Handling

File Handling:

File handling in C involves manipulating files stored on the computer's storage system. This capability allows C programs to read data from files, write data to files, and perform various operations like appending, deleting, or modifying files. File handling in C is achieved through the use of the `stdio.h` library, which provides functions for file operations.

Components of File Handling in C:

1. **File Pointer**: A file pointer is a reference or handle used to access a file. It tracks the current position within the file during read and write operations.

2. **File Stream**: A file stream is a flow of data between the program and the file. It represents the connection between the program and the file.
3. **File Descriptor**: In low-level file handling, a file descriptor is an integer associated with an open file. It's used by the operating system to perform file operations.

Usage of File Handling in C:

1. **Opening Files**: Use the `fopen()` function to open a file with a specified mode.
2. **Reading from Files**: Use functions like `fread()` or `fscanf()` to read data from a file.
3. **Writing to Files**: Use functions like `fwrite()` or `fprintf()` to write data to a file.
4. **Closing Files**: Always close the file using the `fclose()` function after performing operations to release system resources.
5. **Moving within Files**: Use functions like `fseek()` and `rewind()` to move the file pointer to a specific position within the file.
6. **Checking for End of File**: Use the `feof()` function to check if the end of file has been reached.
7. **Error Handling**: Check for errors returned by file handling functions and handle them appropriately using techniques like checking return values and setting `errno`.

Advantages of File Handling in C:

1. **Data Persistence**: Data stored in files remains available even after the program is terminated or the system is shut down.
2. **Versatility**: Files can store different types of data, including text, binary, and formatted data.
3. **Data Sharing**: Files allow data to be shared between different programs or different instances of the same program.
4. **Efficient Storage**: Files provide an efficient way to store and organize large volumes of data.
5. **Backup and Recovery**: Files can be backed up and restored, providing a means of data recovery in case of system failure.

Disadvantages of File Handling in C:

1. **File Corruption**: Files can get corrupted due to various reasons like system crashes or hardware failures, leading to data loss.
2. **Security Risks**: Files can be vulnerable to unauthorized access, leading to security breaches.
3. **Performance Overhead**: File operations involve I/O operations, which can be slower compared to in-memory operations.
4. **Platform Dependency**: File handling functions may behave differently on different operating systems, leading to portability issues.
5. **Complexity**: Handling files requires understanding concepts like file pointers, modes, and error handling, which can add complexity to the program.

Modes in File Handling:

1. Read Mode (`"r"`):

- Opens a file for reading.
- If the file doesn't exist, fopen() returns NULL.
- The file pointer is positioned at the beginning of the file.
- Use for operations where reading from the file is the primary task.

2. Write Mode (`"w"`):

- Opens a file for writing.
- If the file doesn't exist, it creates a new file. If the file exists, it truncates it to zero length.
- Use for operations where writing to the file is the primary task and existing content can be discarded.

3. Append Mode (`"a"`):

- Opens a file for appending data.
- If the file doesn't exist, it creates a new file.
- The file pointer is positioned at the end of the file, allowing data to be added without overwriting existing content.
- Use when adding data to the end of the file is the primary task.

4. Read/Write Mode (`"r+"`):

- Opens a file for both reading and writing.
- The file must exist, or fopen() returns NULL.
- The file pointer is positioned at the beginning of the file.
- Use when both reading from and writing to the file are required.

5. Write/Read Mode (`"w+"`):

- Opens a file for both reading and writing.
- If the file doesn't exist, it creates a new file. If the file exists, it truncates it to zero length.
- Use when both reading from and writing to the file are required, and existing content can be discarded.

6. Append/Read Mode (`"a+"`):

- Opens a file for both reading and appending.
- If the file doesn't exist, it creates a new file.
- The file pointer is positioned at the end of the file.
- Use when both reading from and appending to the file are required.

7. Binary Read Mode (`"rb"`):

- Opens a file for binary reading.
- Similar to `"r"`, but specifically for binary files.
- Ensures that the file is treated as a binary file, avoiding any text interpretation.

8. Binary Write Mode (`"wb"`):

- Opens a file for binary writing.
- Similar to `"w"`, but specifically for binary files.
- Ensures that the file is treated as a binary file, avoiding any text interpretation.

9. Binary Append Mode (`"ab"`):

- Opens a file for binary appending.
- Similar to `"a"`, but specifically for binary files.
- Ensures that the file is treated as a binary file, avoiding any text interpretation.

10. Binary Read/Write Mode (`"rb+"`):

- Opens a file for both binary reading and writing.
- Similar to `"r+"`, but specifically for binary files.
- Ensures that the file is treated as a binary file, avoiding any text interpretation.

11. Binary Write/Read Mode (`"wb+"`):

- Opens a file for both binary reading and writing.
- Similar to `"w+"`, but specifically for binary files.
- Ensures that the file is treated as a binary file, avoiding any text interpretation.

12. Binary Append/Read Mode (`"ab+"`):

- Opens a file for both binary reading and appending.
- Similar to `"a+"`, but specifically for binary files.
- Ensures that the file is treated as a binary file, avoiding any text interpretation.

File Handling Methods:

1. `fopen()` :

- Syntax: `FILE *fopen(const char *filename, const char *mode);`
- Opens a file with the specified filename and mode.
- Returns a file pointer to the opened file, or NULL if an error occurs.

2. `fclose()` :

- Syntax: `int fclose(FILE *stream);`
- Closes the file associated with the given file pointer.
- Returns 0 on success, EOF on failure.

3. `fread()` :

- Syntax: `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
- Reads data from the file into the specified buffer.
- Returns the number of elements read successfully.

4. `fwrite()`:
   - Syntax: `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
   - Writes data from the specified buffer to the file.
   - Returns the number of elements written successfully.
5. `fseek()`:
   - Syntax: `int fseek(FILE *stream, long int offset, int whence);`
   - Moves the file pointer to the specified position within the file.
   - Returns 0 on success, nonzero on failure.
6. `ftell()`:
   - Syntax: `long int ftell(FILE *stream);`
   - Returns the current position of the file pointer within the file.
7. `rewind()`:
   - Syntax: `void rewind(FILE *stream);`
   - Resets the file pointer to the beginning of the file.
8. `feof()`:
   - Syntax: `int feof(FILE *stream);`
   - Checks if the end of the file has been reached.
   - Returns nonzero if the end of the file has been reached, 0 otherwise.

Error Handling:

- Functions in file handling return specific values to indicate success or failure.
- Use `errno` or return values to handle errors appropriately, such as file not found or permission denied.

Here's a simple example of file handling in C that demonstrates how to read data from a file, process it, and then write the processed data to another file:

```
#include <stdio.h>
int main() {
    FILE *input_file, *output_file;
    char character;

    // Open the input file in read mode
    input_file = fopen("input.txt", "r");
    if (input_file == NULL) {
```

```
        printf("Error opening the input file!\n");
        return 1;
    }

    // Open the output file in write mode
    output_file = fopen("output.txt", "w");
    if (output_file == NULL) {
        printf("Error opening the output file!\n");
        fclose(input_file);
        return 1;
    }

    // Read characters from the input file and write them to the output file
    while ((character = fgetc(input_file)) != EOF) {
        // Process the character (for example, convert uppercase to lowercase)
        if (character >= 'A' && character <= 'Z') {
            character += 32; // Convert uppercase to lowercase
        }
        // Write the processed character to the output file
        fputc(character, output_file);
    }

    // Close the input and output files
    fclose(input_file);
    fclose(output_file);

    printf("File processing completed successfully!\n");

    return 0;
}
```

In this example:

- We include the necessary header file `stdio.h`.
- We declare two file pointers `input_file` and `output_file`.
- We open the input file `"input.txt"` in read mode (`"r"`) using `fopen()` and check if the file was opened successfully.
- We open the output file `"output.txt"` in write mode (`"w"`) using `fopen()` and check if the file was opened successfully.
- We read characters from the input file using `fgetc()` in a loop until `EOF` (end of file) is reached.
- We process each character (e.g., convert uppercase to lowercase) and write the processed character to the output file using `fputc()`.
- Finally, we close both the input and output files using `fclose()`.

Sample output:

Suppose the contents of `input.txt` are:

```
Hello, World!
This is a sample input file.
```

After running the program, the contents of `output.txt` would be:

```
Hello, World!
This is a sample input file.
```

Explanation:

- The program reads each character from `input.txt`.
- It converts any uppercase letters to lowercase.
- It writes the processed characters to `output.txt`.

Exercise Questions

▼ Exercise 1

Find the square root of a quadratic equation:

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

void main() {
    float a, b, c, d, x, x1, x2, realpart, imagipart;

    printf("Enter the coefficients of the quadratic equation\n");
    scanf("%f %f %f", &a, &b, &c);

    if (a == 0) {
        x = -b / c;
        printf("Only root x=%7.3f", x);
    }

    d = b * b - 4 * a * c;

    if (d > 0) {
        printf("Real and distinct roots: ");
        x1 = (-b + sqrt(d)) / (2 * a);
        x2 = (-b - sqrt(d)) / (2 * a);
        printf("x1 = %7.3f \n x2 = %7.3f ", x1, x2);
    } else if (d == 0) {
        printf("Repeated roots are");
```

```
        x1 = -b / (2 * a);
        x2 = x1;
        printf("x1 = x2 = %7.3f ", x1, x2);
    } else {
        d = sqrt(fabs(d));
        realpart = -b / (2 * a);
        imagipart = d / (2 * a);
        printf("Complex roots are:\n");
        printf("x1 = %7.3f + i%7.3f", realpart, imagipart);
        printf("\n x2 = %7.3f - i%7.3f", realpart, imagipart);
    }
}
```

I've added indentation to improve code readability. Note that `getch()` is not a standard function in C; you might want to use `getchar()` or other appropriate functions depending on your compiler and environment.

Logic:

1. **Input**:
   - Prompt the user to enter the coefficients of the quadratic equation.
   - Read the values of 'a', 'b', and 'c' using `scanf()`.
2. **Check if 'a' is zero**:
   - If 'a' is zero, it's not a quadratic equation but a linear equation, so calculate and print the root directly.
3. **Calculate Discriminant (d)**:
   - Compute the discriminant using the formula: `d = b * b - 4 * a * c`.
4. **Check Discriminant**:
   - If the discriminant is positive, there are two real and distinct roots.
   - If the discriminant is zero, there are repeated roots.
   - If the discriminant is negative, there are complex roots.
5. **Calculate Roots**:
   - For real roots, use the quadratic formula to compute 'x1' and 'x2'.
   - For repeated roots, compute 'x1'.
   - For complex roots, compute the real and imaginary parts separately.

Sample Output:

```
Enter the coefficients of the quadratic equation (a, b, c): 1 -3 2
Real and distinct roots: x1 = 2.000, x2 = 1.000
```

▼ Exercise 2

Compute mean, variance and standard deviation for N real numbers

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

#define SIZE 100 // Symbolic Constant

void main() {
    int n, i;
    float a[SIZE], sum = 0, mean = 0, variance = 0, deviation = 0;
    clrscr();

    printf("Enter the size of array ");
    scanf("%d", &n);

    printf("Enter the numbers in array \n");
    for (i = 0; i < n; i++) {
        scanf("%f", &a[i]);
    }

    // Find mean value
    for (i = 0; i < n; i++) {
        sum = sum + a[i];
    }
    mean = sum / n;
    printf("\nMean (Average) = %f", mean);

    // Find variance value
    sum = 0;
    for (i = 0; i < n; i++) {
        sum = sum + (a[i] - mean) * (a[i] - mean);
    }
    variance = sum / n;
    printf("\nVariance = %f", variance);

    // Find standard deviation
    deviation = sqrt(variance);
    printf("\nDeviation = %f \n", deviation);
}
```

Logic:

1. **Input Size and Elements**:

- Prompt the user to enter the size of the array (up to 100).
- Read the size 'n' using `scanf()`.
- Prompt the user to enter 'n' numbers into the array.
- Read the numbers into the array 'a[]' using a loop.

2. **Calculate Mean (Average)**:
   - Initialize a variable `sum` to store the sum of all elements.
   - Iterate through the array and add each element to `sum`.
   - Calculate the mean by dividing the sum by the total number of elements 'n'.
3. **Calculate Variance**:
   - Reuse the variable `sum` to store the sum of squared differences from the mean.
   - Iterate through the array and add the squared difference of each element from the mean to `sum`.
   - Divide the sum by 'n' to get the variance.
4. **Calculate Standard Deviation**:
   - Use the calculated variance.
   - Calculate the standard deviation by taking the square root of the variance using the `sqrt()` function from the `<math.h>` library.
5. **Output Results**:
   - Print the calculated mean, variance, and standard deviation to the console.
6. **End of Program**:
   - End the program after displaying the results.

Sample Output:

```
Enter the size of array 5
Enter the numbers in array
1
5
10
15
20
Mean (Average)= 10.200000
Variance = 46.159996
Deviation = 6.794115
```

▼ Exercise 3

Dynamic Memory Allocation (2D Array):

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

```
>#include <stdlib.h>
int main() {
    int rows, cols;

    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    printf("Enter the number of columns: ");
    scanf("%d", &cols);

    int **matrix = (int **)malloc(rows * sizeof(int *));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int *)malloc(cols * sizeof(int));
    }

    // Initializing matrix elements
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = i + j;
        }
    }

    // Printing matrix elements
    printf("Matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    // Freeing dynamically allocated memory
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);

    return 0;
}
```

Logic:

1. **Input Rows and Columns**:
    - Prompt the user to enter the number of rows and columns for the 2D array.
    - Read the values of `rows` and `cols` using `scanf()`.
2. **Dynamic Memory Allocation**:
    - Allocate memory for an array of `rows` integer pointers to represent the rows of the 2D array.

- For each row, allocate memory for an array of `cols` integers to represent the columns.

3. **Initialize Matrix Elements**:
   - Iterate through each element of the 2D array.
   - Assign a value to each element based on its row and column index. Here, we're assigning the sum of row and column indices.
4. **Print Matrix Elements**:
   - Print the elements of the 2D array row by row.
   - Nested loops are used to iterate through each row and each column, printing the corresponding element.
5. **Free Memory**:
   - Deallocate the dynamically allocated memory to prevent memory leaks.
   - First, free the memory allocated for each row individually.
   - Then, free the memory allocated for the array of row pointers.
6. **Return**:
   - Exit the program with a return value of 0, indicating successful execution.

Sample Output:

```
Enter the number of rows: 3
Enter the number of columns: 4
Matrix:
0 1 2 3
1 2 3 4
2 3 4 5
```

In this sample output:

- The user inputs 3 for the number of rows and 4 for the number of columns.
- The program dynamically allocates memory for a 3×4 integer matrix.
- The matrix elements are initialized with values based on their row and column indices.
- Finally, the matrix is printed row by row.

▼ Exercise 4

Linear Search:

```
#include <stdio.h>
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target)
```

```
            return i; // Return the index if the target is found
    }
    return -1; // Return -1 if the target is not found
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 13;

    printf("Array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    int index = linearSearch(arr, n, target);

    if (index != -1)
        printf("%d found at index %d.\n", target, index);
    else
        printf("%d not found.\n", target);

    return 0;
}
```

Logic:

1. **Linear Search Algorithm**:
   - Linear search is a simple searching algorithm that sequentially checks each element of the array until it finds the target element or reaches the end of the array.
   - The algorithm starts from the beginning of the array and compares each element with the target.
   - If the target is found, the algorithm returns the index of the target.
   - If the target is not found after iterating through the entire array, the algorithm returns -1.
2. **linearSearch Function**:
   - Accepts an integer array `arr[]`, its size `n`, and the target element `target` as parameters.
   - Implements the linear search algorithm to find the target element in the given array.
   - Returns the index of the target element if found, otherwise returns -1.
3. **Main Function**:
   - Declares an integer array `arr[]` with some initial values.
   - Calculates the size of the array `n`.
   - Specifies the target element to be searched.

- Prints the original array.
- Calls the `linearSearch` function to search for the target element.
- Prints the search result.

Sample Output:

```
Array: 12 11 13 5 6
13 found at index 2.
```

In this sample output:

- The original array `{12, 11, 13, 5, 6}` is printed.
- The target element `13` is found at index `2`.

▼ Exercise 5

Insertion Sort:

```
#include <stdio.h>
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are greater than key, to one posit
ion ahead of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    int i;

    printf("Original array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
```

```
    insertionSort(arr, n);

    printf("Sorted array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

Logic:

1. **Insertion Sort Algorithm**:
   - In insertion sort, the array is divided into a sorted and an unsorted part.
   - The sorted part is initially empty, and elements are picked one by one from the unsorted part and placed at the correct position in the sorted part.
   - The algorithm iterates over each element of the array, starting from the second element.
   - For each element, it compares it with the elements to its left in the sorted part and shifts the larger elements to the right until it finds the correct position for insertion.
   - The process continues until the entire array is sorted.
2. **insertionSort Function**:
   - Accepts an integer array `arr[]` and its size `n` as parameters.
   - Implements the insertion sort algorithm on the given array.
3. **Main Function**:
   - Declares an integer array `arr[]` with some initial values.
   - Calculates the size of the array `n`.
   - Prints the original array.
   - Calls the `insertionSort` function to sort the array.
   - Prints the sorted array.

Sample Output:

```
Original array: 12 11 13 5 6
Sorted array: 5 6 11 12 13
```

In this sample output:

- The original array `{12, 11, 13, 5, 6}` is printed.

- After sorting using insertion sort, the sorted array `{5, 6, 11, 12, 13}` is printed.

▼ Exercise 6

Selection Sort:

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[i]) {
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        // Display the array after each pass
        printf("Pass %d:", i + 1);
        for (int k = 0; k < n; k++) {
            printf(" %d", arr[k]);
        }
        printf("\n");
    }
}

int main() {
    int size;

    // Input the size of the array
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int numbers[size];

    // Input the array elements
    printf("Enter %d two-digit numbers for the array: ", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &numbers[i]);
    }

    // Perform selection sort
    selectionSort(numbers, size);

    // Display the final sorted array
    printf("Final Sorted Array:");
```

```
    for (int i = 0; i < size; i++) {
        printf(" %d", numbers[i]);
    }
    printf("\n");

    return 0;
}
```

Logic:

1. **Input Size and Array Elements**:
   - Prompt the user to input the size of the array.
   - Read the size of the array from the user.
   - Declare an integer array `numbers[]` of size `size` to store the input elements.
   - Prompt the user to input `size` two-digit numbers for the array.
   - Read the input numbers and store them in the array.
2. **Selection Sort Function (`selectionSort`)**:
   - This function accepts an integer array `arr[]` and its size `n` as arguments.
   - It implements the selection sort algorithm to sort the array in ascending order.
   - The outer loop iterates over each element of the array except the last one (`n - 1` times).
   - The inner loop searches for the minimum element in the unsorted portion of the array.
   - If a smaller element is found, it swaps it with the current element at index `i`.
   - After each pass, it prints the array to display the progress of sorting.
3. **Main Function**:
   - Declare a variable `size` to store the size of the array.
   - Input the size of the array from the user.
   - Declare an array `numbers[]` of size `size` to hold the input elements.
   - Input the elements of the array from the user.
   - Call the `selectionSort` function to sort the array.
   - After sorting, print the final sorted array.

Sample Output:

```
Enter the size of the array: 5
Enter 5 two-digit numbers for the array: 32 18 45 27 54
Pass 1: 18 32 45 27 54
Pass 2: 18 27 32 45 54
Pass 3: 18 27 32 45 54
Pass 4: 18 27 32 45 54
Pass 5: 18 27 32 45 54 (Final Sorted Array)
```

▼ Exercise 7

Bubble Sort:

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap if the element found is greater than the next element
                swap(&arr[j], &arr[j + 1]);
            }
        }

        // Display the array after each pass
        printf("Pass %d: ", i + 1);
        for (int k = 0; k < n; k++) {
            printf("%d ", arr[k]);
        }
        printf("\n");
    }
}

int main() {
    int n;

    // Input the size of the array
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int arr[n];

    // Input array elements
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Perform Bubble Sort
    bubbleSort(arr, n);
```

```
    // Display the sorted array
    printf("\nSorted Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Example Input:

```
Enter the size of the array: 5
Enter 5 integers:
67 23 45 12 89
```

Output (After Each Pass):

```
Pass 1: 23 45 12 67 89
Pass 2: 23 12 45 67 89
Pass 3: 12 23 45 67 89
Pass 4: 12 23 45 67 89
```

Final Sorted Array:

```
Sorted Array: 12 23 45 67 89
```

This demonstrates the step-by-step process of Bubble Sort on a set of 2-digit numbers.

▼ Exercise 8

Arrays and Functions:

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

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
```

```
                arr[j + 1] = temp;
            }
        }
    }
}

int binarySearch(int arr[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }

    return -1;
}

int main() {
    int size = 10;
    int arr[size];

    srand(time(NULL));
    printf("Original Array: ");
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 100;
        printf("%d ", arr[i]);
    }
    printf("\n");

    bubbleSort(arr, size);

    printf("Sorted Array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    int target = arr[rand() % size];
    printf("Searching for %d...\n", target);
    int index = binarySearch(arr, 0, size - 1, target);
    if (index != -1)
        printf("%d found at index %d.\n", target, index);
    else
```

```
        printf("%d not found.\n", target);

    return 0;
}
```

Logic:

1. **BubbleSort Function**:
   - This function takes an integer array `arr[]` and its size `n` as arguments.
   - It implements the bubble sort algorithm to sort the elements of the array in ascending order.
   - The algorithm repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
   - This process is repeated until the array is sorted.
2. **BinarySearch Function**:
   - This function takes an integer array `arr[]`, left index `left`, right index `right`, and a target value `target` as arguments.
   - It performs binary search on the sorted array to find the index of the target element.
   - Binary search works by dividing the sorted array into halves and repeatedly narrowing down the search interval until the target element is found or the interval is empty.
3. **Main Function**:
   1. **Array Initialization**:
      - `int size = 10;`: Declares an integer variable `size` and initializes it to 10, representing the size of the array.
      - `int arr[size];`: Declares an integer array `arr[]` with a size of 10. This array will store randomly generated integers.
   2. **Random Number Generation**:
      - `srand(time(NULL));`: Seeds the random number generator with the current time to produce different random numbers on each program execution.
      - `for (int i = 0; i < size; i++) { ... }`: Loops through the array indices from 0 to 9.
      - `arr[i] = rand() % 100;`: Generates a random integer between 0 and 99 using the `rand()` function and assigns it to the current element of the array.
   3. **Original Array Printing**:
      - `printf("Original Array: ");`: Prints a message indicating the original array is being printed.
      - `for (int i = 0; i < size; i++) { printf("%d ", arr[i]); }`: Loops through the array indices and prints each element separated by a space.

4.
Sorting the Array:
   - `bubbleSort(arr, size);`: Calls the `bubbleSort` function to sort the array `arr[]` in ascending order.
5.
Sorted Array Printing:
   - `printf("Sorted Array: ");`: Prints a message indicating the sorted array is being printed.
   - `for (int i = 0; i < size; i++) { printf("%d ", arr[i]); }`: Loops through the sorted array indices and prints each element separated by a space.
6.
Target Element Selection:
   - `int target = arr[rand() % size];`: Generates a random index within the range of the array and selects the corresponding element as the target for searching.
7.
Binary Search:
   - `int index = binarySearch(arr, 0, size - 1, target);`: Calls the `binarySearch` function to search for the target element within the sorted array.
   - The function parameters are the array `arr[]`, the left index (0), the right index (`size - 1`), and the target value.
8.
Search Result Display:
   - `if (index != -1) { printf("%d found at index %d.\n", target, index); }`: If the target element is found (`index != -1`), it prints the target value and its index in the array.
   - Otherwise, it prints that the target element was not found.
9.
Return Statement:
   - `return 0;`: Indicates successful program execution. The program terminates here.

Sample Output:

```
Original Array: 67 12 90 34 20 80 56 45 68 99
Sorted Array: 12 20 34 45 56 67 68 80 90 99
Searching for 80...
80 found at index 7.
```

In this output:

- An array of 10 random integers between 0 and 99 is generated and printed.

- The array is then sorted in ascending order using the bubble sort algorithm.
- The sorted array is printed.
- A random target value (in this case, 80) is selected for searching.
- The program performs a binary search to find the index of the target value in the sorted array.
- The result indicates that the target value (80) was found at index 7.

Hope this notes was helpful to you! Share our social channels among your friends circle and contribute expanding MassCoders.
Love you all 3000💚