**Experiment No: 1 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Familiarize with Turbo C Compiler or equivalent compiler features.**

**AIM**

Familiarize with Turbo C Compiler features

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Turbo C Compiler Features**

Turbo C was a software development tool for writing programs in the C language.  
As an **Integrated Development Environment (IDE)**, it included:

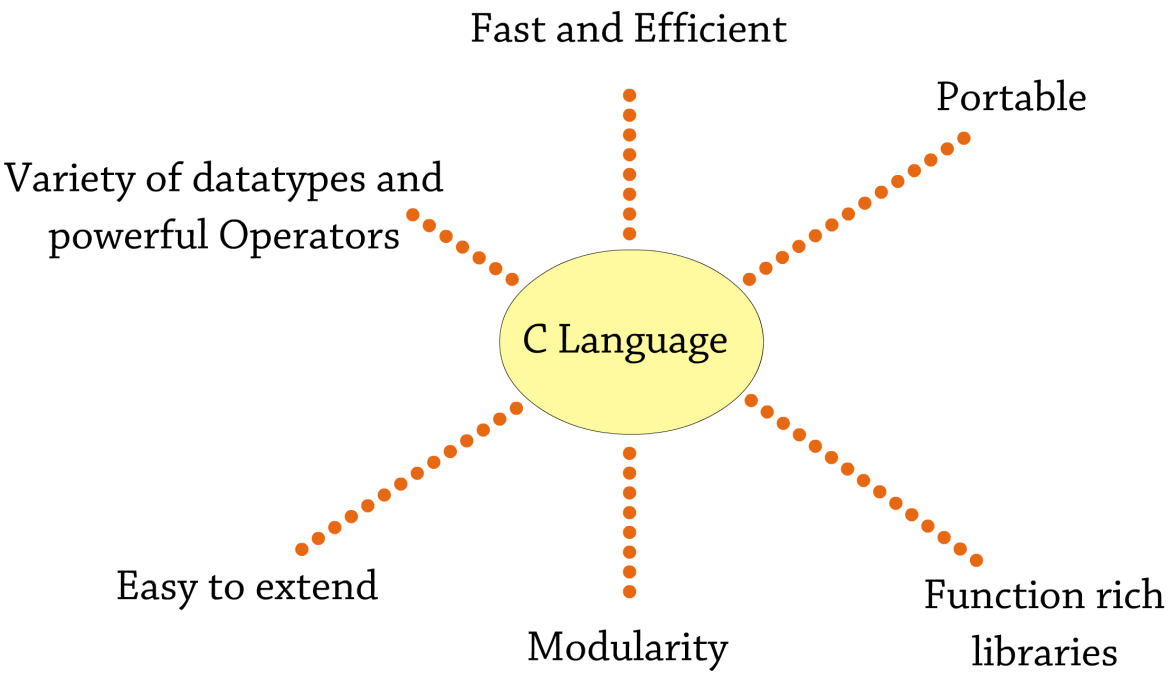
* **Source code editor**
* **Fast compiler**
* **Linker**
* **Offline help file** for reference
* **Built-in debugger** (added in Version 2)

Turbo C followed Borland’s **Turbo Pascal**, popular in educational institutions for teaching programming.  
Although originally developed by another company, Turbo C shared many interface and debugging features with Turbo Pascal.

Despite competition from Microsoft C, Watcom C, and Lattice C, Turbo C remained attractive because of its **high compile speed** and **affordable price**.

**Historical Notes**

* **First release:** 13 May 1987
* **Significance:** First *edit–compile–run* environment for IBM PCs
* **Origin:** Initially called **Wizard C**, created by **Bob Jervis**, later purchased by Borland
* **Interface influence:** Turbo Pascal adopted Turbo C–style pull-down menus starting with Version 4



**Fig1:Turbo C features**

**Key C Language Features (as supported by Turbo C)**

* Rich set of built-in functions and operators for complex programs
* Combines **assembly-language power** with **high-level language features**
* Programs are **efficient and fast** due to diverse data types and powerful operators (often faster than BASIC)
* **Portability:** Programs can run on different machines with little or no modification
* **Extensibility:** Ability to create new functions and add them to the C library
* Widely used in **operating systems** and **embedded systems** development
* Simple syntax with solid functionality—ideal for beginners

**Support for All Memory Models**

Because 16-bit processors used a segmented memory architecture (each segment limited to 64 KB), Turbo C supported several memory models:

* **Tiny:** Program and data in a single 64 KB segment
* **Small:** Program and data each in separate 64 KB segments
* **Medium, Large, Huge:** For programs or data larger than 64 KB

*(32-bit processors use a flat memory model and do not have this limitation.)*

**Speed or Size Optimization**

The compiler could generate executables optimized either for **speed** or for **smaller size**, but not both at the same time.

**Constant Folding**

Turbo C could evaluate constant expressions at **compile time** rather than at run time, improving performance.

**Understanding Compilers**

A **compiler** converts source code files (e.g., .c, .cpp, .cc) into a runnable program.  
Popular compilers include **gcc**, **g++**, **Microsoft Visual C++**, **Borland**, and others.

**Common Terminology**

* **Compile:** Convert source code into object (intermediate) code
* **Link:** Combine object code and libraries to create an executable
* **Build:** Entire process of compiling and linking
* **Linker:** Program that performs the linking
* **IDE:** Integrated environment combining editor, compiler, and related tools

**Compilation Process**

Compilation and linking together produce the final executable.  
IDEs streamline this process and help locate errors quickly.

**Limitations of the Turbo C Compiler**

* Access limited to **1 MB memory**
* **Stack and heap share** the same memory segment
* No support for **dynamically loaded libraries**
* **Multi-process** or **multi-threaded** programming not possible
* **Socket/network programming** unsupported
* No **ODBC** (database) programming
* Limited/poor **graphics support**; no 32-bit graphics
* Maximum of **20 open files** at a time
* **Heap segment size** only 64 KB
* Pointer addressable space limited to **1 MB**
* Physical addresses are 20 bits, but Turbo C uses 16-bit registers; memory models manage pointers
* No **locking mechanisms**, **system call implementations**, or **memory protection**
* No separate **kernel** and **user space**
* **Buffer size** limited to 512 bytes
* Lacks **advanced program-analysis tools**

**Result**

Familiarized with Turbo C Compiler or equivalent compiler features

**Experiment No: 2 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Practice formatted input/output (printf and scanf) functions.**

**AIM**

To practice formatted input/output (printf and scanf) functions

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Formatted Input / Output Functions in C**

The **formatted input/output functions** allow data to be entered and displayed in a specific format, improving the readability of both input and output.

**1. scanf()**

* The scanf() function in C is a standard library input function defined in **stdio.h** that reads data entered from the keyboard and stores it in variables specified by the programmer.
* It uses format specifiers to determine the expected data types, and each variable’s address must be provided with the address-of operator (&) so the function can place the input value in the correct memory location.
* For example, scanf("%d %f", &age, &height); reads an integer into age and a floating-point number into height.
* scanf() is essential for interactive programs where user input drives computation, offering a straightforward way to capture different data types in a single call.

**scanf() – Formatted Input**

* **Purpose:** Reads data from the keyboard.
* **Header File:** #include <stdio.h> is required.
* **Description:** scanf() belongs to the C standard library and can be used to accept multiple inputs with specified format specifiers.

**Syntax**

scanf("list\_of\_format\_specifiers", &var1, &var2, ..., &varn);

**Example**

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

Reads an integer into a and a float into b.

**2. printf()**

* The printf() function in C is a standard library output function defined in **stdio.h** that displays formatted text and variable values on the screen.
* It allows you to mix literal text, escape sequences (such as \n for a new line or \t for a tab), and format specifiers like %d for integers, %f for floats, or %s for strings in a single statement.
* For example, printf("Result = %d\n", sum); prints both the label and the value of the variable sum neatly.
* Because it supports precise formatting—such as setting field width or decimal precision—printf() is widely used to produce clear, readable program output.

**printf() – Formatted Output**

* **Purpose:** Displays output on the screen in a user-defined format.
* **Header File:** #include <stdio.h> is required.
* **Description:** printf() belongs to the C standard library and can use escape sequences (e.g., \n, \t) and format specifiers.

**Syntax**

printf("list\_of\_format\_specifiers\_and\_escape\_sequences", var1, var2, ..., varn);

**Example**

printf("%d %f", a, b);

Displays the integer value of a and the float value of b.

**3. Example Program**

#include <stdio.h>

#include <conio.h>

void main()

{

int a, b, c;

clrscr(); // Clears the screen (specific to some old compilers)

printf("Enter the A and B value: ");

scanf("%d %d", &a, &b); // Read two integers

c = a + b;

printf("The value of c = %d", c); // Display result

getch(); // Wait for a key press (specific to some old compilers)

}

#include <stdio.h>

int main()

{

int a, b, c;

printf("Enter the A and B value: ");

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

c = a + b;

printf("The value of c = %d", c);

return 0;

}

**Sample Output**

Enter the A and B value: 5 6

The value of c = 11

**Result**

Practiced formatted input/output (printf and scanf) functions

**Experiment No: 3 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Practice with various operators in C**

**AIM**

To practice with various operators in C

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Introduction to Operators and Expressions**

Operators are **symbolic characters** that specify the action to be performed on their associated operands.

* **Operands** are the values or variables on which operators act.
* Together, an operator and its operands form an **expression**, which produces a result.

**General Form**

[operand] operator [operand]

* **Unary Operators:** Operate on **one** operand (e.g., -a).
* **Binary Operators:** Operate on **two** operands (e.g., a + b).

**Table of Operator Symbols**

|  |  |  |
| --- | --- | --- |
| **Category** | **Operators** | **Description / Use** |
| **Arithmetic** | + - \* / % | Addition, subtraction, multiplication, division, modulo (remainder) |
| **Unary** | + - ++ -- ! | Unary plus, unary minus, increment, decrement, logical NOT |
| **Relational** | == != > < >= <= | Equal to, not equal to, greater than, less than, greater or equal, less or equal |
| **Logical** | && ` | Logical AND |
| **Bitwise** | & ` | ^ ~ << >>` |
| **Assignment** | = += -= \*= /= %= | Simple assignment and compound assignments |
| **Conditional (Ternary)** | ?: | Conditional expression (if–else in a single statement) |
| **Member/Pointer** | . -> | Access structure or object members, access through pointer |
| **Others** | sizeof , & (address-of) \* (dereference) | Size of object, comma operator, address-of, pointer dereference |

**Classification of Expressions**

1. **Arithmetic Expression**
   * Contains arithmetic operators (+, -, \*, /, %).
   * Produces a **numeric result**.
   * Example:
   * SET expr = 1 + 2
   * SET expr = a + b
2. **String Expression**
   * Contains string operators.
   * Produces a **string result**.
   * Example:
   * SET expr = "hello"
   * SET expr = "hello" \_ x
3. **Logical Expression**
   * Uses relational and logical operators (>, <, &&, ||, etc.).
   * Produces a **boolean result**: **TRUE (1)** or **FALSE (0)**.
   * Example:
   * SET expr = a && b
   * SET expr = a > b
4. **Object Expression**
   * Produces an **object reference** as a result.
   * Example:
   * SET expr = employee.Company
   * SET expr = ##class(Person).%New()

**Writing Expressions Clearly**

When writing expressions, add spaces before and after every operator to improve readability:

a = 234 \* (45 - 56.0 / 34)

* A string used in an arithmetic expression is treated as a numeric value (or **0** if it is not numeric).
* Using the **unary plus (+)** converts a string to a numeric value.

**C Program: Demonstrating Arithmetic Operators**

#include <stdio.h>

#include <conio.h>

void main() {

int a = 10, b = 100;

clrscr(); // Clears the screen (Turbo C specific)

printf("The Addition value = %d\n", (a + b));

printf("The Subtraction value = %d\n", (a - b));

printf("The Multiplication value = %d\n", (a \* b));

printf("The Division value = %d\n", (b / a));

printf("The Modulo value = %d\n", (b % a));

getch(); // Wait for a key press (Turbo C specific)

}

**Output**

The Addition value = 110

The Subtraction value = -90

The Multiplication value = 1000

The Division value = 10

The Modulo value = 0

**Result**

Practiced with various operators in C

**Experiment No: 4 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Practice with decision & control Statements**

**(if, if-else, nested if–else)**

**AIM**

To Practice with decision & control Statements (if, if-else, nested if–else)

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Decision Making and Control Statements**

In programming, the **order of execution of instructions** may need to change depending on certain conditions.  
This requires **decision making**, where the program evaluates a condition and decides which statements to execute.

* Such statements are called **control statements** because they control the flow of execution.
* A decision is typically based on a condition whose result is either **TRUE (non-zero)** or **FALSE (0)**.
* In C, the value **0 is considered FALSE**, and any **non-zero value is TRUE**.

**if Statement**

* The if statement is a powerful **decision-making** construct.
* It **tests a logical condition** and executes a block of code only when the condition is true.

**Key Points**

* If the **test expression is TRUE** (non-zero), the statements inside the body of if execute.
* If the **test expression is FALSE** (0), the statements inside the body of if are skipped.

**Syntax (if Statement)**

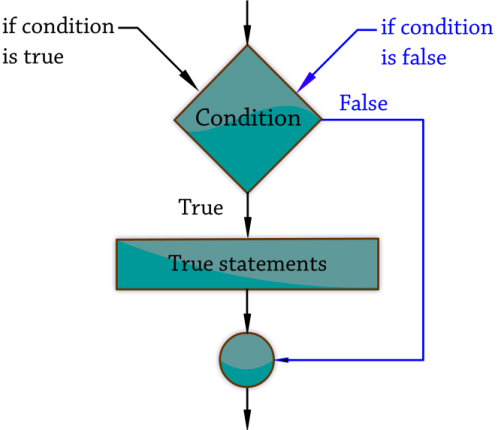
if (condition) // test expression

{

statement;

}

**Flowchart (if Statement)**



**if-else Statement**

The **if-else statement** is also called a **double-block conditional statement**.

* It has **two parts**:
  1. **True part:** Executed if the condition evaluates to true.
  2. **False part:** Executed if the condition evaluates to false.
* It is used to control the flow of execution based on a condition.

**Syntax (if-else Statement)**

if (condition)

{

// Statement(s) executed if the condition is TRUE

}

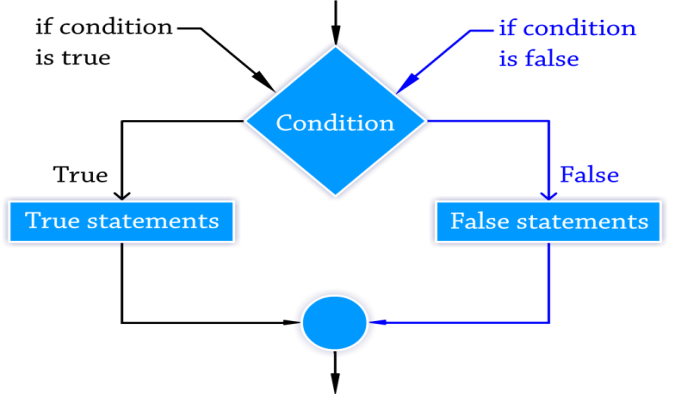
else

{

// Statement(s) executed if the condition is FALSE

}

**Flowchart (if-else Statement)**



**Nested if…else Statement**

When more than one **if…else** statement is used inside another **if…else**, it is called a **nested if…else statement**.

**Syntax (Nested if…else Statement)**

if (condition1)

{

if (condition2)

{

// True statement 2

}

else

{

// False statement 2

}

}

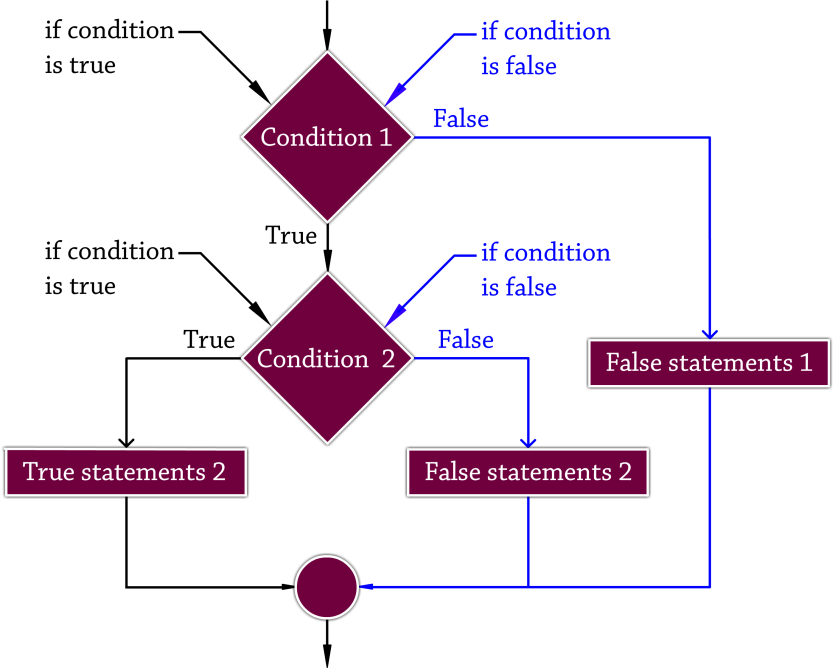
else

{

// False statement 1

}

**Flowchart (Nested if…else Statement)**

****

**PROGRAM**

**Example Program (if Statement)**

#include <stdio.h>

#include <conio.h>

void main() {

int a;

clrscr(); // Clears the screen (Turbo C specific)

printf("Enter the value: ");

scanf("%d", &a);

if (a < 10) // Test expression is true if value is less than 10

{

printf("The value is less than 10\n");

}

printf("The if statement is easy");

getch(); // Waits for a key press (Turbo C specific)

}

**Sample Output**

**Case 1: Input less than 10**

Enter the value: 8

The value is less than 10

The if statement is easy

**Case 2: Input greater than or equal to 10**

Enter the value: 17

The if statement is easy

**Program: Check Even or Odd (if Statement)**

#include <stdio.h>

#include <conio.h>

void main()

{

int num;

clrscr(); // Clears the screen (Turbo C specific)

printf("Enter an integer: ");

scanf("%d", &num);

// If remainder of num divided by 2 is 0, then it's even

if (num % 2 == 0)

{

printf("%d is an EVEN number.\n", num);

}

printf("Program finished successfully.");

getch(); // Waits for a key press (Turbo C specific)

}

**Explanation**

* The condition num % 2 == 0 checks whether the number is evenly divisible by 2.
* If the condition is **TRUE**, the message stating that the number is even is printed.
* If the condition is **FALSE**, the program simply skips that statement and only prints “Program finished successfully.”

**Sample Output**

**Case 1: Input is 8**

Enter an integer: 8

8 is an EVEN number.

Program finished successfully.

**Case 2: Input is 7**

Enter an integer: 7

Program finished successfully.

**Example Program (if-else Statement)**

#include <stdio.h>

#include <conio.h>

void main() {

int a;

clrscr(); // Clears the screen (Turbo C specific)

printf("Enter the value: ");

scanf("%d", &a);

if (a < 10) // Test expression is TRUE if value is less than 10

{

printf("The value is less than 10\n");

}

else // Executed if test expression is FALSE

{

printf("The value is greater than 10\n");

}

getch(); // Waits for a key press (Turbo C specific)

}

**Sample Output**

**Case 1: Input less than 10**

Enter the value: 8

The value is less than 10

**Case 2: Input greater than or equal to 10**

Enter the value: 17

The value is greater than 10

**Program: Find the Biggest of Three Numbers (Nested if…else Statement)**

#include <stdio.h>

#include <conio.h>

void main()

{

int a, b, c, big;

clrscr(); // Clears the screen (Turbo C specific)

printf("Enter the a value: ");

scanf("%d", &a);

printf("Enter the b value: ");

scanf("%d", &b);

printf("Enter the c value: ");

scanf("%d", &c);

if (a > b)

{

if (a > c)

{

big = a;

}

else

{

big = c;

}

}

else

{

if (b > c)

{

big = b;

}

else

{

big = c;

}

}

printf("The biggest of three numbers: %d", big);

getch(); // Waits for a key press (Turbo C specific)

}

**Sample Input**

Enter the a value: 15

Enter the b value: 10

Enter the c value: 20

**Sample Output**

The biggest of three numbers: 20

**Explanation**

* The program takes three integers as input: **a**, **b**, and **c**.
* It first compares **a** and **b**.
  + If **a > b**, it checks whether **a > c** to decide if **a** or **c** is the largest.
  + Otherwise, it compares **b** and **c** to decide if **b** or **c** is the largest.
* Finally, it prints the largest value among the three.

**Result:**

Practiced with decision & control Statements (if, if-else, nested if–else)

**Experiment No: 5 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Practice with decision control**

**(switch–case structure) statements**

**AIM**

To practice with decision control (switch–case structure) statements

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Decision Control – Switch–Case Structure**

The **switch–case** statement is a **multiple-condition checking statement**.  
It is commonly used in **menu-driven programs**, where the user must select **one option out of several**.

* The number of case statements inside a switch matches the number of menu options.
* Each **case** performs **only one specific task** at a time.
* The **default** section is **optional** and is generally used for **error handling**.
* It executes automatically when the user enters an invalid or out-of-range choice.

**Syntax (switch–case structure)**

switch (expression)

{

case 1:

statement1;

break;

case 2:

statement2;

break;

...

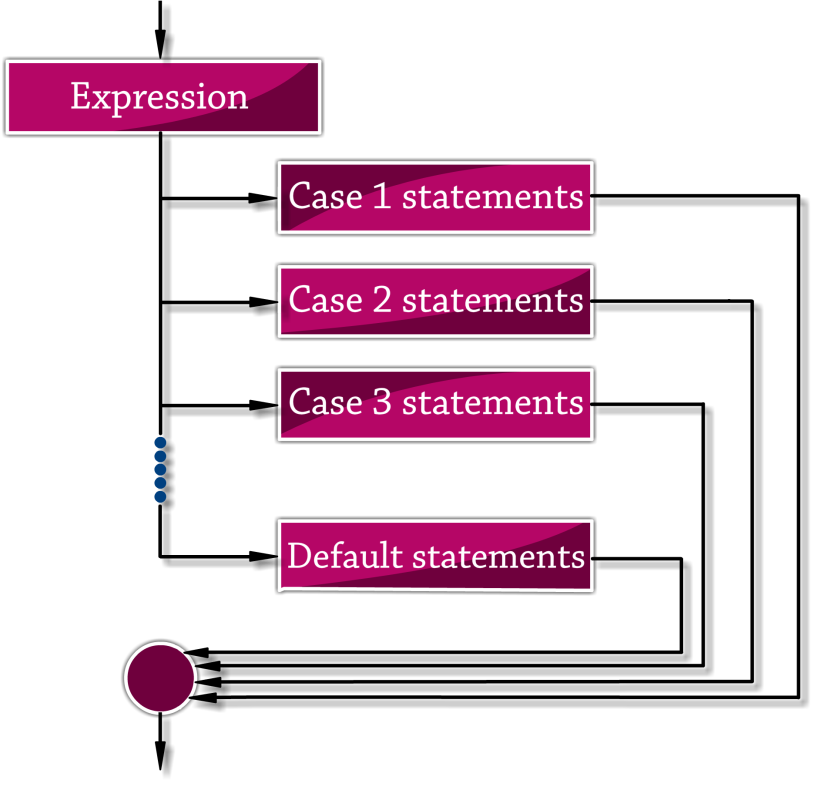
default:

statement\_for\_invalid\_choice;

break;

}

**Flowchart (switch–case structure)**



**Program: Switch–Case Example**

#include <stdio.h>

#include <conio.h>

void main() {

int n;

clrscr(); // Clears the screen (Turbo C specific)

printf("Enter the value: ");

scanf("%d", &n);

switch (n) {

case 1:

printf("I am in college");

break;

case 2:

printf("I am in home");

break;

default:

printf("I am in hospital");

}

getch(); // Waits for a key press (Turbo C specific)

}

**Sample Output**

**Input:** 1

I am in college

**Input:** 2

I am in home

**Input:** 3

I am in hospital

**Explanation**

* The user enters an integer value.
* The switch statement checks the value of **n**:
* If **n = 1**, it prints **“I am in college.”**
* If **n = 2**, it prints **“I am in home.”**
* For any other value, the **default** case runs and prints **“I am in hospital.”**

**Result:**

Practiced with decision control (switch–case structure) statements

**Experiment No: 6 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Practice with loop control statements**

**(while, do…while, and for)**

**AIM**

To practice with loop control statements (while, do…while, and for)

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Looping Statements**

A **loop** is a block of statements that are executed **repeatedly** for a specified number of times or until a condition becomes false.  
It allows **sequential execution** of a program with repetition.

**Key Steps in a Loop**

1. **Initialization** of a control variable.
2. **Testing** the control condition.
3. **Updating** the control variable after each iteration.

**while Statement**

* The test condition is **evaluated first**.
* If the condition is **true**, the body of the loop executes.
* This process repeats until the test condition becomes **false**.
* When the test expression is **false**, the loop **terminates**.

**Syntax (while Statement)**

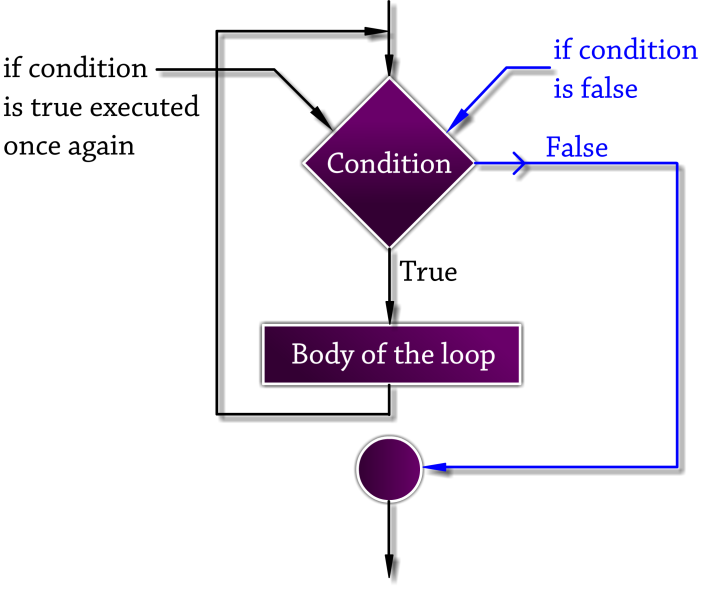
while (condition)

{

// Body of the loop

}

**Flowchart (while Statement)**

****

**do…while Statement**

* In a **do…while** loop, the **test condition is checked at the end** of the loop.
* The statements inside the braces are **always executed at least once**, even if the condition is false the first time.
* Because the condition is tested **after** the body executes, it is called an **exit-controlled loop**.

**Syntax (do…while Statement)**

do

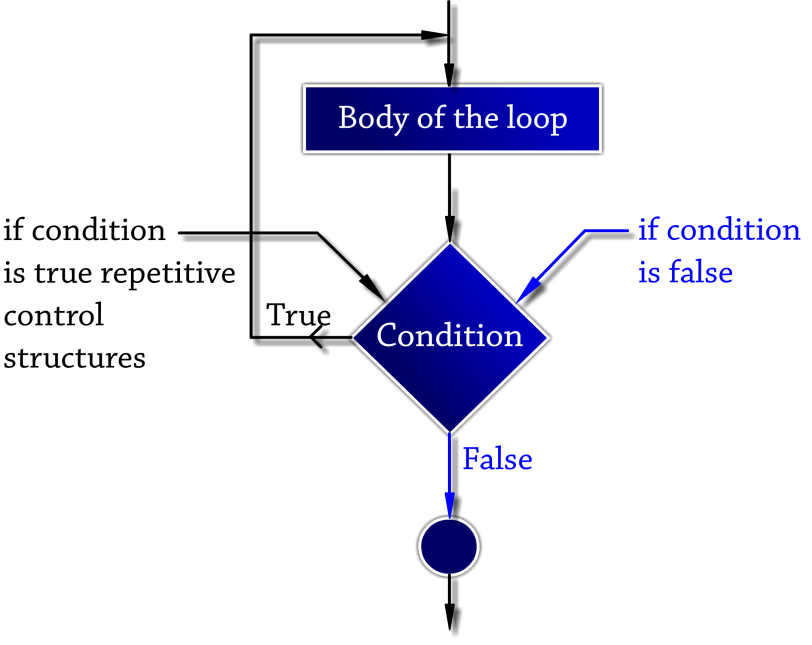
{

// Body of the loop

}

while (condition);

**Flowchart (do…while Statement)**

****

**Key Difference Between do…while, while, and for**

|  |  |  |  |
| --- | --- | --- | --- |
| **Loop Type** | **When Condition is Checked** | **Minimum Executions** | **Typical Use Case** |
| **while** | Before loop body | 0 or more | When the number of iterations isn’t known. |
| **do…while** | After loop body | At least 1 | When you need the body to run **once** always. |
| **for** | Before each iteration | 0 or more | When the number of iterations is **known**. |

**For Loop**

* The **for loop** allows a block of code to be **repeatedly executed**.
* It is a **repetitive control structure** that continues execution until the condition becomes **false**.
* A for loop has **three parts**:
  1. **Initialization** – Sets the starting value of the counter variable.
  2. **Condition** – Checked before each iteration; if true, the loop body runs.
  3. **Increment/Decrement** – Updates the counter variable after each iteration.

**Working Steps**

1. **Initialization:** Counter variable is initialized.
2. **Condition Check:**
   * If the condition is **true**, the statements inside the loop execute.
   * If the condition is **false**, the loop terminates and control moves out of the loop.
3. **Update:** After the body executes, the counter variable is incremented or decremented.

**Syntax (For Loop Statement)**

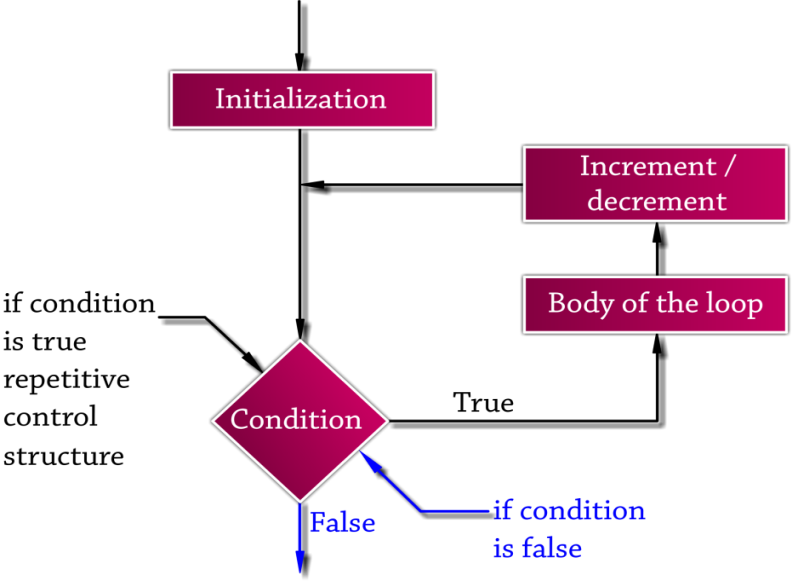
for (initialization; condition; increment/decrement)

{

// Body of the loop

}

**Flowchart (For Loop Statement)**



**Difference Between for and while Loop**

|  |  |  |
| --- | --- | --- |
| **Feature** | **for loop** | **while loop** |
| Initialization | Performed **once** in the loop header. | Done **before** the loop starts. |
| Condition Checking | Checked **before every iteration**. | Checked **before every iteration**. |
| Increment/Decrement | Declared in the header; automatic each pass. | Must be written **inside** the loop body. |
| Best Use Case | When the **number of iterations is known**. | When the **number of iterations is unknown**. |

**Program: Sum of Numbers from 1 to 10 (while Statement)**

#include <stdio.h>

#include <conio.h>

void main()

{

int a = 1, sum = 0;

clrscr(); // Clears the screen (Turbo C specific)

while (a <= 10)

{

sum = sum + a; // Add current value of a to sum

a++; // Increment a by 1

}

printf("The sum of 1 to 10 is : %d", sum);

getch(); // Waits for a key press (Turbo C specific)

}

**Output**

The sum of 1 to 10 is : 55

**Explanation**

* The variable **a** starts at 1 and is incremented by 1 in each loop.
* The loop continues until **a > 10**.
* The variable **sum** accumulates the total of numbers from 1 to 10.
* Final output displays the total as **55**.

**Program: Sum of Numbers from 1 to 10 (do…while Statement)**

#include <stdio.h>

#include <conio.h>

void main()

{

int a = 1, sum = 0;

clrscr(); // Clears the screen (Turbo C specific)

do

{

sum = sum + a; // Add current value of a to sum

a++; // Increment a by 1

} while (a <= 10);

printf("The sum is: %d", sum);

getch(); // Waits for a key press (Turbo C specific)

}

**Output**

The sum is: 55

**Explanation**

* The loop **executes the body first**, adding a to sum and incrementing a.
* After each execution, the condition a <= 10 is checked.
* The process continues until the condition is false.
* Even if the condition were false at the start, the body would still execute **once**.

**Example Program (For Loop Statement)**

#include <stdio.h>

#include <conio.h>

void main()

{

int a, sum = 0;

clrscr(); // Clears the screen (Turbo C specific)

for (a = 1; a <= 10; a++)

{ // Loop from 1 to 10

sum = sum + a; // Add each value of a to sum

}

printf("The sum of 1 to 10 is: %d", sum);

getch(); // Waits for a key press (Turbo C specific)

}

**Output**

The sum of 1 to 10 is: 55

**Explanation:**

* The loop starts with a = 1, checks the condition a <= 10, and adds a to sum.
* After each iteration, a is incremented by 1.
* When a becomes 11, the condition fails and the loop stops, printing the total sum.

**Result**

Practiced with loop control statements (while, do…while, and for)

**Experiment No: 7 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Practice the use of functions in C**

**AIM**

To practice the use of functions in C. understand the functions on demonstrating prototyping, parameter passing, function returning values

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**User-Defined Functions in C**

* **User-defined functions** are functions created by the programmer to perform a specific task.
* They improve code **reusability**, **modularity**, and make programs easier to read and maintain.
* The main() function itself is an example of a user-defined function.
* Every user-defined function must be **declared**, **defined**, and **called**.

**Steps to Use a Function**

1. **Function Definition** – The actual code that performs the task.
2. **Function Declaration** – Informs the compiler about the function name, return type, and parameters.
3. **Function Call** – Invokes the function when required.

**Function Definition**

The function definition contains the code needed to complete the task.

**Syntax:**

return\_type function\_name(list\_of\_parameters)

{

// Function body

}

A function definition includes:

* **Function Header**
  + **Function Type (Return Type):** Specifies the type of value returned (e.g., int, float, void).
    - If not specified, C assumes int.
    - Use void when no value is returned.
  + **Function Name:** Must be a valid C identifier and meaningful.
  + **List of Parameters (Arguments):** Variables that receive input from the calling program.
    - If no parameters are required, use void inside the parentheses.
* **Function Body**
  + **Local Declarations:** Variables used inside the function.
  + **Function Statements:** Code to perform the task.
  + **Return Statement:** Returns a value to the caller (if the function type is not void).

**Function Declaration**

A function declaration tells the compiler about the function’s name, return type, and parameters **before** it is used.

**Syntax:**

return\_type function\_name(parameter\_list);

End the declaration with a semicolon (;).

**Function Call**

A function is called by writing its name followed by parentheses. If parameters exist, pass them inside the parentheses.

**Syntax:**

function\_name();

function\_name(parameter\_list);

return\_value = function\_name(parameter\_list);

**Example Program**

#include <stdio.h>

#include <conio.h>

void sum(); // Function declaration

void main()

{

int a = 10, b = 20, c;

clrscr(); // Clears the screen (Turbo C specific)

sum(); // Function call

}

void sum()

{ // Function definition

int a = 10, b = 20, c;

c = a + b;

printf("Sum: %d", c);

}

**Output**

Sum: 30

**Explanation:**

* The function sum() is **declared** before main(), **called** inside main(), and **defined** after main().
* It calculates the sum of two numbers and prints the result.

## Function with **No Arguments** and **No Return Type**

### Program

#include <stdio.h>

void sum(); // Function declaration

void main()

{

sum(); // Function call

}

void sum()

{ // Function definition

int a, b, s;

printf("Enter two numbers: ");

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

s = a + b;

printf("Sum = %d\n", s);

}

### Sample Output

Enter two numbers: 5 7

Sum = 12

### Explanation

* **No arguments:** Numbers are taken inside the function itself.
* **No return type:** The function prints the result directly; nothing is sent back to main().

## Function with **Arguments** and **No Return Type**

### Program

#include <stdio.h>

void sum(int x, int y); // Function declaration

void main()

{

int a, b;

printf("Enter two numbers: ");

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

sum(a, b); // Passing arguments to function

}

void sum(int x, int y)

{ // Function definition

printf("Sum = %d\n", x + y);

}

### Sample Output

Enter two numbers: 10 15

Sum = 25

### Explanation

* Numbers are read in main() and **passed as arguments** to sum().
* The function only **displays** the sum and returns nothing.

## Function with **Arguments** and **Return Type**

### Program

#include <stdio.h>

int sum(int x, int y); // Function declaration

void main()

{

int a, b, result;

printf("Enter two numbers: ");

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

result = sum(a, b); // Function call with arguments

printf("Sum = %d\n", result);

}

int sum(int x, int y)

{ // Function definition

return x + y; // Returns the sum to main

}

### Sample Output

Enter two numbers: 8 20

Sum = 28

### Explanation

* Numbers are entered in main() and sent to sum().
* sum() **returns the result** to main(), which then prints it.

## Function with **No Arguments** and **Return Type**

### Program

#include <stdio.h>

int sum(); // Function declaration

void main()

{

int result;

result = sum(); // Function call

printf("Sum = %d\n", result);

}

int sum()

{ // Function definition

int a, b;

printf("Enter two numbers: ");

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

return a + b; // Returns the sum to main

}

### Sample Output

Enter two numbers: 4 9

Sum = 13

### Explanation

* **No arguments:** The numbers are read inside the function.
* **Return type:** The computed sum is **returned to** main(), which prints it.

### Result

Practiced the use of functions in C. understand the functions on demonstrating prototyping, parameter passing, function returning values

**Experiment No: 8 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Write and run small programs using**

**single-dimensional integer arrays**

**AIM**

To Write and run small programs using single-dimensional integer arrays

**Apparatus Required**

Hardware Required

Windows version 7/9/10

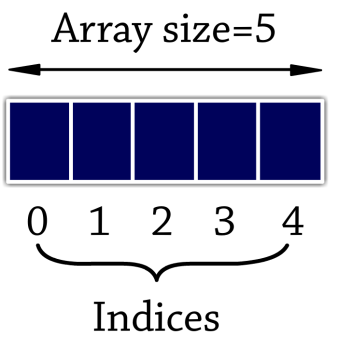
Software Required

turbo C

**Theory**

**Array in C**

An **array** is a collection of data items that stores a **fixed number of values of the same type** under a single name.  
C language provides the capability to create a **set of similar data types** using arrays.



**One-Dimensional Array**

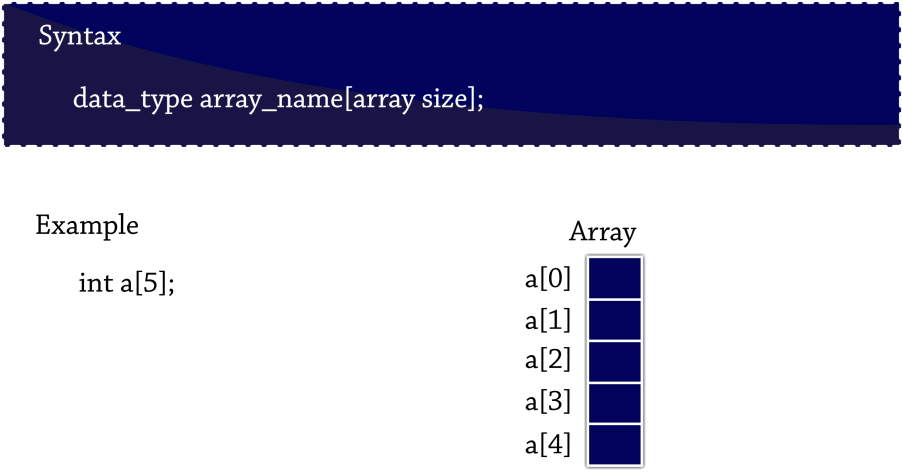
A **one-dimensional array** is a list of data items that can be accessed using a **single subscript**.  
This allows multiple values to be stored under one variable name.

**Array Declaration**

Arrays are declared like ordinary variables but must specify the **size** (number of elements).

**Syntax:**

data\_type array\_name[size];



Example:

int a[5];

**Initialization of One-Dimensional Array**

Array values can be initialized **when the array is defined** or **during runtime**.

**Compile-Time Initialization**

Values are assigned directly at the time of declaration.

**Syntax:**

data\_type array\_name[size] = {list\_of\_values};

**Example:**

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

**Run-Time Initialization**

Values are assigned by reading user input during program execution.

**Example:**

int a[2];

scanf("%d %d", &a[0], &a[1]);

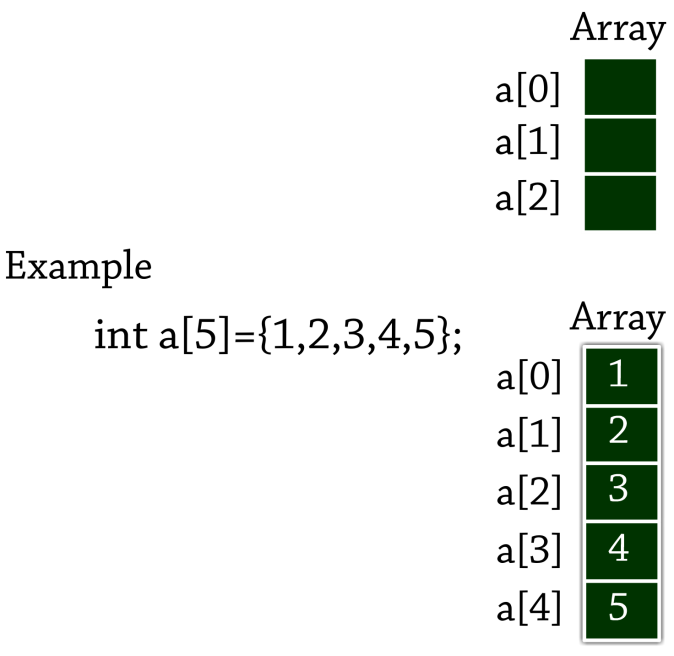


Fig: Initialization of one-Dimensional Array

**Program: Sum of 5 Numbers Using 1-D Array**

#include <stdio.h>

#include <conio.h>

void main() {

int a[5], sum = 0, i;

clrscr();

printf("Enter 5 numbers: ");

for (i = 0; i <= 4; i++) {

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

sum = sum + a[i];

}

printf("The sum of given numbers is %d", sum);

getch();

}

**Sample Output**

Enter 5 numbers:

1

2

3

4

5

The sum of given numbers is 15

**Program: Sum of Predefined Array Elements**

#include <stdio.h>

#include <conio.h>

void main() {

int sum = 0, a[5] = {5, 10, 15, 20, 25}, i;

clrscr();

for (i = 0; i <= 4; i++) {

sum = sum + a[i];

}

printf("The sum of given numbers is %d", sum);

getch();

}

**Sample Output**

The sum of given numbers is 75

**Result**

Practiced single-dimensional integer arrays

**Experiment No: 9 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Write and run small programs using**

**Multidimensional integer arrays**

**AIM**

To write and run small programs using multidimensional integer arrays.

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Multi-Dimensional Array**

A **multi-dimensional array** is a collection of data items stored under a **single variable name** using **multiple subscripts (indices)**.  
Each dimension adds a level of indexing.

**Array Declaration**

Arrays are declared similarly to ordinary variables, but with **multiple sizes** specified for each dimension.

**Syntax:**

data\_type array\_name[size1][size2][size3]...[sizeN];

**Example:**

int a[3][3][3];

This example declares a **3×3×3** three-dimensional array.

**Example Program: 4-Dimensional Array**

#include <stdio.h>

void main() {

int array\_4d[3][3][3][3];

int a, b, c, d;

// Assign values to the 4D array

for (a = 0; a < 3; a++)

for (b = 0; b < 3; b++)

for (c = 0; c < 3; c++)

for (d = 0; d < 3; d++)

array\_4d[a][b][c][d] = a + b + c + d;

// Display the array elements

for (a = 0; a < 3; a++) {

printf("\n");

for (b = 0; b < 3; b++) {

for (c = 0; c < 3; c++) {

for (d = 0; d < 3; d++)

printf("%3d", array\_4d[a][b][c][d]);

printf("\n");

}

}

}

}

**Sample Output**

012

123

234

123

234

345

234

345

456

123

234

345

234

345

456

345

456

567

234

345

456

345

456

567

456

567

678

**Explanation of Output:**

* Each number printed is the **sum of the indices**: a + b + c + d.
* For example:
  + When a=0, b=0, c=0, d=0 → value = 0+0+0+0 = **0**
  + When a=1, b=0, c=1, d=1 → value = 1+0+1+1 = **3**
* The program displays these sums layer by layer for each combination of indices.

The program prints the sum of indices (a + b + c + d) layer by layer.  
Each “block” corresponds to one fixed a and b pair.

a = 0, b = 0

0 1 2

1 2 3

2 3 4

a = 0, b = 1

1 2 3

2 3 4

3 4 5

a = 0, b = 2

2 3 4

3 4 5

4 5 6

a = 1, b = 0

1 2 3

2 3 4

3 4 5

a = 1, b = 1

2 3 4

3 4 5

4 5 6

a = 1, b = 2

3 4 5

4 5 6

5 6 7

a = 2, b = 0

2 3 4

3 4 5

4 5 6

a = 2, b = 1

3 4 5

4 5 6

5 6 7

a = 2, b = 2

4 5 6

5 6 7

6 7 8

* Each 3×3 grid shows all values for the inner c and d indices.
* Notice how numbers increase because they are simply the sum of the four indices.

**Ressult**

Practiced multidimensional integer arrays

**Experiment No: 10 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Write and run small programs using string functions for string comparison, copying and concatenation**

**AIM**

To write and run small programs using string functions for string comparison, copying and concatenation.

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**String Library Functions**

**Strings**

* A string is a **collection of characters**.
* You **cannot** directly add or join one string to another.
* You **cannot** use logical operators directly on strings.
* **Header File:**
* #include <string.h>
* Invalid examples (not allowed):
* string3 = string1 + string2;
* string2 = string1 + "hello";

**Common String Manipulation Functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| strcat() | Concatenates (joins) two strings |
| strcmp() | Compares two strings |
| strcpy() | Copies one string into another |
| strlen() | Finds the length of a string |
| strrev() | Reverses a string |

All these functions are declared in **<string.h>**.

**1. String Compare – strcmp()**

**Purpose:**  
Compares two strings.

**Return Value:**

* Returns **0** if both strings are equal.
* Returns a **non-zero value** if they are not equal.

**Syntax:**

strcmp(string1, string2);

**Example:**

char name1[] = "Computer";

char name2[] = "Engineering";

strcmp(name1, "Engineer");

strcmp("Computer", name1);

**Example Program**

#include <stdio.h>

#include <conio.h>

#include <string.h>

void main() {

char name1[] = "Computer";

char name2[] = "Engineering";

int a, b;

clrscr();

a = strcmp(name1, "Computer");

b = strcmp(name1, name2);

printf("The String is %d\n", a);

printf("The String is %d\n", b);

getch();

}

**Sample Output**

The String is 0

The String is 1

(Here 0 means strings match, 1 means they differ.)

**2. String Copy – strcpy()**

**Purpose:**  
Copies one string into another.

**Syntax:**

strcpy(destination, source);

**Example:**

char name1[] = "drsmn";

char name2[20];

strcpy(name2, name1);

**Example Program**

#include <stdio.h>

#include <conio.h>

#include <string.h>

void main() {

char name1[] = "drsmn";

char name2[20];

clrscr();

strcpy(name2, name1);

printf("The String name1 is %s\n", name1);

printf("The String name2 is %s\n", name2);

getch();

}

**Sample Output**

The String name1 is drsmn

The String name2 is drsmn

**3. String Concatenate – strcat()**

**Purpose:**  
Concatenates (joins) two strings.

**Syntax:**

strcat(string1, string2);

**Example:**

strcat("Engineering", "foru"); // Result: "Engineeringforu"

**Example Program**

#include <stdio.h>

#include <conio.h>

#include <string.h>

void main() {

char name1[40] = "Engineering";

char name2[] = "For u";

clrscr();

printf("The String name1 is %s\n", name1);

printf("The String name2 is %s\n", name2);

strcat(name1, name2);

printf("Concatenated String is %s\n", name1);

getch();

}

**Sample Output**

The String name1 is Engineering

The String name2 is For u

Concatenated String is EngineeringFor u

**Result**

Practiced small programs using string functions for string comparison, copying and concatenation.

**Experiment No: 11 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Write and run small programs using pointers in C**

**AIM**

To write and run small programs using C, understand and demonstrate use of pointers as function arguments, functions returning pointers.

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Introduction to Pointers**

Generally, a computer uses memory to store instructions and variable values.  
Pointers are special variables whose **values are memory addresses**.  
A pointer stores the address where a program’s instructions or data are located, and can be used to **access and manipulate data** directly in memory.

**Concept of Pointers**

* A **pointer is a variable** that holds the memory address of another variable.
* It is declared like any other variable but is always denoted by the \* (asterisk) operator.
* A pointer variable’s **value is an address**, not a regular data value.
* Every variable has two attributes:
  1. **Address** – the location in memory.
  2. **Value** – the data stored at that location.

Example explanation:

* Suppose we have an integer variable with value 39 stored at address 3977.
* A pointer can store the address (3977) and access the value (39) indirectly.

**Features of Pointers**

Benefits of using pointers include:

* Efficient in handling data and arrays.
* Reduces length and complexity of programs.
* Enables **dynamic memory allocation** (assigns and releases memory during runtime).
* Faster execution since operations work directly with memory addresses.
* Allows **functions to be passed as arguments** to other functions.

**Pointer Syntax**

data\_type \*pointer\_name;

Example:

int \*a;

int b = 10;

a = &b;

Here:

* b is a normal integer variable.
* a is a pointer to an integer that stores the address of b.

**Sample Program**

#include <stdio.h>

#include <conio.h>

void main() {

int \*a, b = 10;

clrscr();

a = &b;

printf("The address a = %u\n", a);

printf("The value of a = %d\n", \*a);

getch();

}

**Sample Output**

The address a = 2201 // (Address will vary at runtime)

The value of a = 10

**Advantages of Pointers**

* Save memory space.
* Reduce length and complexity of programs.
* Support **dynamic memory allocation** (assign and release memory).
* Enable faster execution as data is accessed by address.
* Allow references to functions and passing functions as arguments.

**Result**

small programs using C, understand and demonstrate use of pointers as function arguments, functions returning pointers

**Experiment No: 12 Date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_\_\_**

**Write and run small programs using structures and unions in C**

**AIM**

To write and run small programs using structures in C, using C preprocessor directives and to practice command line arguments in C

**Apparatus Required**

Hardware Required

Windows version 7/9/10

Software Required

turbo C

**Theory**

**Structure in C**

**Definition**

A **structure** is a collection of data elements of **different data types** that are logically grouped together and referenced under the **same name**.

* It is **heterogeneous** in nature.
* A structure can include **other structures**, **arrays**, **pointers**, or **unions** as some of its members.

**Syntax**

struct structure\_name

{

type variable\_list1;

type variable\_list2;

...

type variable\_listN;

};

* Memory is **not allocated** for the members when the structure is defined.
* The structure definition simply informs the compiler about the members of the structure.
* When a program contains several structures, they are distinguished by their **structure\_name**.

**Declaration of Structure Variable(s)**

* The struct keyword is used to declare a structure.
* Members of a structure are enclosed within **curly braces { }**.
* Declaring a structure **does not reserve space**; memory is allocated only when a structure variable is created.

**Syntax:**

struct structure\_name structure\_variable(s);

**Example:**

struct emp // structure definition

{

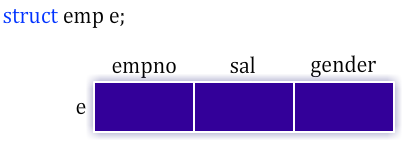
int empno;

float sal;

char gender;

};

struct emp e; // structure variable declaration



**Combined Definition and Declaration**

Structure definition and variable declaration can be combined.

struct emp // structure definition

{

int empno;

float sal;

char gender;

} s; // structure variable declaration

* The **structure name is optional** if the variable is declared at the time of defining the structure itself:

struct

{

int empno;

float sal;

char gender;

} s; // no structure name used

**Initialization of Structure**

* Initialization can be done at **compile time**.
* The **individual members cannot be initialized separately** during the definition.
* Values are enclosed in **braces { }**.
* Uninitialized members get **default values**:
  + 0 for int and float
  + '\0' for char and strings

**Syntax:**

struct structure\_name

{

type member1;

type member2;

...

} variable = { value1, value2, ... };

**Example:**

struct std

{

char name[10];

int age;

};

struct std s = {"bala", 25};

Alternatively:

struct std

{

char name[10];

int age;

} s = {"bala", 25};

**Accessing Structure Members**

To access any member of a structure, use the **dot operator ( . )**, also called the **structure member operator**.

**Syntax:**

structure\_variable.member\_name;

**Examples:**

e.empno; // gives content of empno

e.sal; // gives content of sal

e.gender; // gives content of gender

## **Introduction to Preprocessor in C**

The **preprocessor** is a program that processes the source code **before compilation**. It handles tasks such as including files, defining macros, and conditional compilation.

### **General Rules**

* Every preprocessor directive **must start with** #.
* It is always placed **before the** main() **function**.
* It **does not end with a semicolon (**;**)**.
* #define statements **do not use an assignment operator**.
* Conditional macros (like #ifdef, #endif) **must be properly terminated**.
* After preprocessing, the **source program is moved for compilation**.

### **Types of Preprocessor Directives**

There are **three main types** of preprocessor directives:

1. **File Inclusion**
2. **Macro Substitution**
3. **Conditional Inclusion**

### **Example Program: Cube Calculation**

#include <stdio.h>

#define cube(n) (n \* n \* n)

void main()

{

int a = 10, b;

b = cube(a);

printf("The cube of a = %d", b);

}

**Output:**

The cube of a = 1000

### **Common Preprocessor Directives**

|  |  |
| --- | --- |
| **Directive** | **Purpose** |
| #include | Includes a file (header or custom). |
| #define | Defines symbolic constants or macros. |
| #ifdef | Tests whether a macro is defined. |
| #else | Specifies alternative code when #ifdef is false. |
| #ifndef | Checks whether a macro is **not** defined. |

**Command Line Arguments in C**

**Introduction**

* An **executable program** that performs a specific task for the operating system is called a **command**.
* When **arguments** are associated with the command, they are called **command-line arguments**.
* Command-line arguments allow you to **supply parameters to a program at runtime**, passing information when the program is executed.

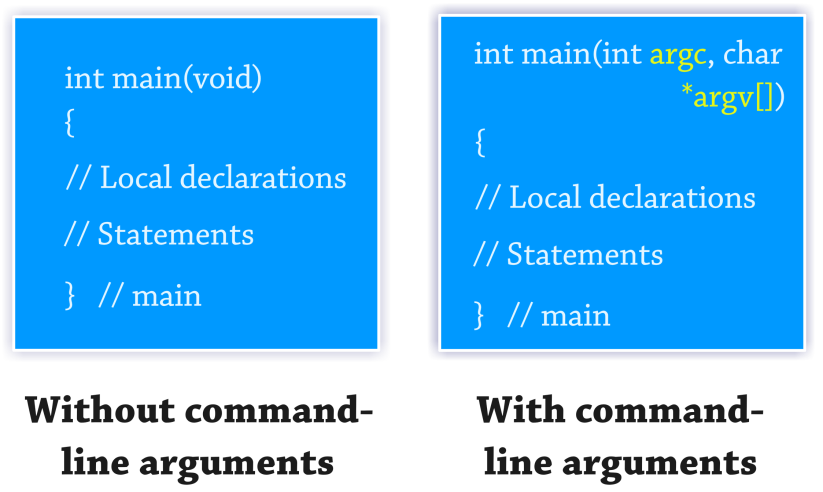
**Example in Daily Use**

If you have used **DOS**, you may be familiar with the copy command:

copy src\_file dest\_file

* src\_file → Name of the **source file**
* dest\_file → Name of the **destination file**

Both of these inputs are **command-line arguments** entered at the command prompt.



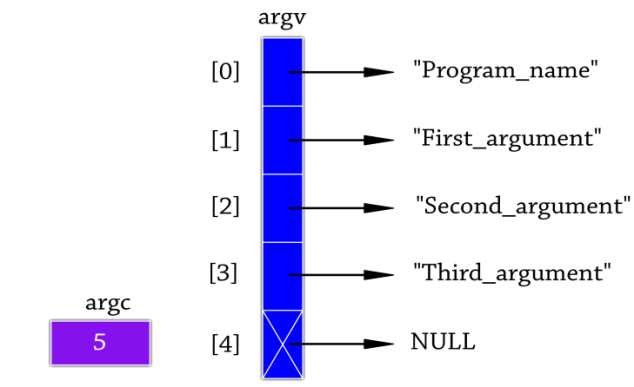
**Main Function with Arguments**

In C, the main function can be defined **with or without arguments**:

int main(void); // No arguments

int main(int argc, char \*argv[]); // With arguments

* **argc**: Argument Counter
  + An integer that contains the **number of arguments** passed from the command line.
* **argv**: Argument Vector
  + An **array of character pointers (strings)**, holding each argument passed to the program.



**Details about argc and argv**

* argv[0] → Points to the **program’s name** (i.e., the executable filename).
* argv[1] ... argv[argc-1] → Point to the **user-supplied arguments**.
* The **last element** is a **NULL pointer**, indicating the end of the list.

**Example Program**

/\* Demonstrates the use of command-line arguments \*/

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

printf("The number of arguments: %d\n", argc);

printf("The name of the program: %s\n", argv[0]);

for (int i = 1; i < argc; i++)

printf("User Value No. %d: %s\n", i, argv[i]);

return 0;

}

**Execution Command (Example):**

C:> pet\_animals cat dog

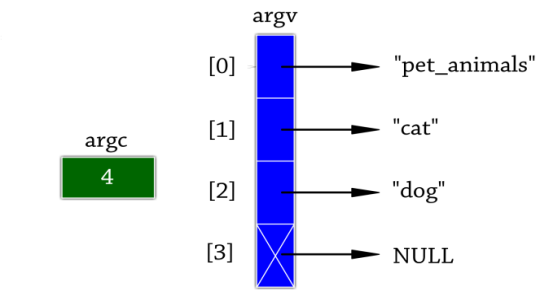
**Output:**

The number of arguments: 4

The name of the program: pet\_animals

User Value No. 1: cat

User Value No. 2: dog



**Other Preprocessor Commands**

**1 #line Command**

* Used to set the **line number** and optionally the **filename** for the program.
* Can be checked using predefined macros like \_\_LINE\_\_ and \_\_FILE\_\_.

**Examples:**

#line 100

Sets the next line number to **100**.

#line 100 "MyProgram.c"

Sets the next line number to **100** and assigns the program name **"MyProgram.c"**.

**2 #error Command**

* Used to **generate a compile-time error message** detected by the preprocessor.

**Syntax:**

#error message

**Example:**

#error CHECK THE INPUT AGAIN

This displays the message **"CHECK THE INPUT AGAIN"** and stops compilation.

**Summary Table**

|  |  |
| --- | --- |
| **Directive** | **Purpose** |
| argc | Holds the **number of command-line arguments**. |
| argv | Array of **string pointers** representing each argument. |
| #line | Sets the next line number and optional filename. |
| #error | Prints a **custom error message** at compile time. |

**Result**

small programs using structures in C, using C preprocessor directives and to practice command line arguments in C