1. Overview of C Programming:

→C language developed by **Dennis Ritchie** in the late 1960s and early 1970s.

→C is a high-level, general-purpose programming language.

→C is very powerful, fast and flexible language, it has been used to develop operation system, databases, applications etc.

Importance of C Programming

→ C has had a profound impact on the development of many other programming languages.

→ For example, C++, Java, Python, and many others are directly or indirectly influenced by C.

Why C is Still Used Today:

1. Low-Level Access:

→ C allows programmers to manipulate memory directly, giving them more control over how a program uses system resources.

2. Speed and Performance:

→ Programs written in C tend to execute faster because the language is compiled directly into machine code.

3. Legacy Systems:

→ A large portion of legacy systems, particularly operating systems (like UNIX, Linux), databases, and embedded systems, is written in C.

4. Continued Development:

→ This ensures that C remains up-to-date with modern programming needs while retaining its core strengths.

2. Setting Up Environment

DevC++ installing step:

Step1: Download DevC++ installer

Step2: Visit the DevC++ website→ Go to the official DevC++ download page

Step3: Download the installer: Click the green "Download" button to get the latest version of DevC++ for Windows. The file will usually be installer.

Step4: Install the program: Follow the remaining steps in the installer and wait for the installation process to complete.

Step5: Open DevC++ from the Start Menu or desktop shortcut.

Step6: The IDE should launch, and you will be ready to write and compile C/C++ code.

Step7: Verify Installation

Step8: Create a simple C program in DevC++

```
#include<stdio.h>
main()
{
        printf("\n hello world!");
}
```

Step9: Click the Execute menu, then Compile & Run (or press F11).

Step10: DevC++ will use the GCC compiler to compile the code and then run it in the output window.

3. Basic Structure of a C Program

A. Including header:

→These are lines that begin with # and are processed by the actual compilation starts.
→Headers provide access to predefined functions, constants, and types
Ex: #include<stdio.h> //Standard input-output library

B. Main function:

→Every C program must have a main() function.
→This is the entry point where execution begins.
Ex: main()
{
}

C. Comments:

→Comments are non-executable lines of text used to describe or explain code.
→This use for code more readable and understanding.
→There are two types of comments in C language:

1. Single-line Comments: Start with **“ // ”** and extend to the end of the line.

2. Multi-line Comments: Enclosed between **“ /* and */ ”**and can span multiple lines.

Ex: // This is a single-line comment.
/* This is a mu
lti-line comment that spans multiple lines */

D. Data Types:

→In C variables must be declared with a specific data type.
→There are many type of data type.
→Explain the basic data type in c language.
1. int: Used for integers (numbers).
2. float: Used for floating-point (decimal) numbers.

3. char: Used for single characters.

4. double: Used for double-precision floating-point numbers.

5. void: Used when a function does not return any value.

Ex: int age; // integer type variable

float height; // floating-point type variable

char grade; // character type variable

E. Variables:

→A variable in C is used to store data, and each variable has a type that defines what kind of data it can hold.

Syntax:

data_type variable_name;

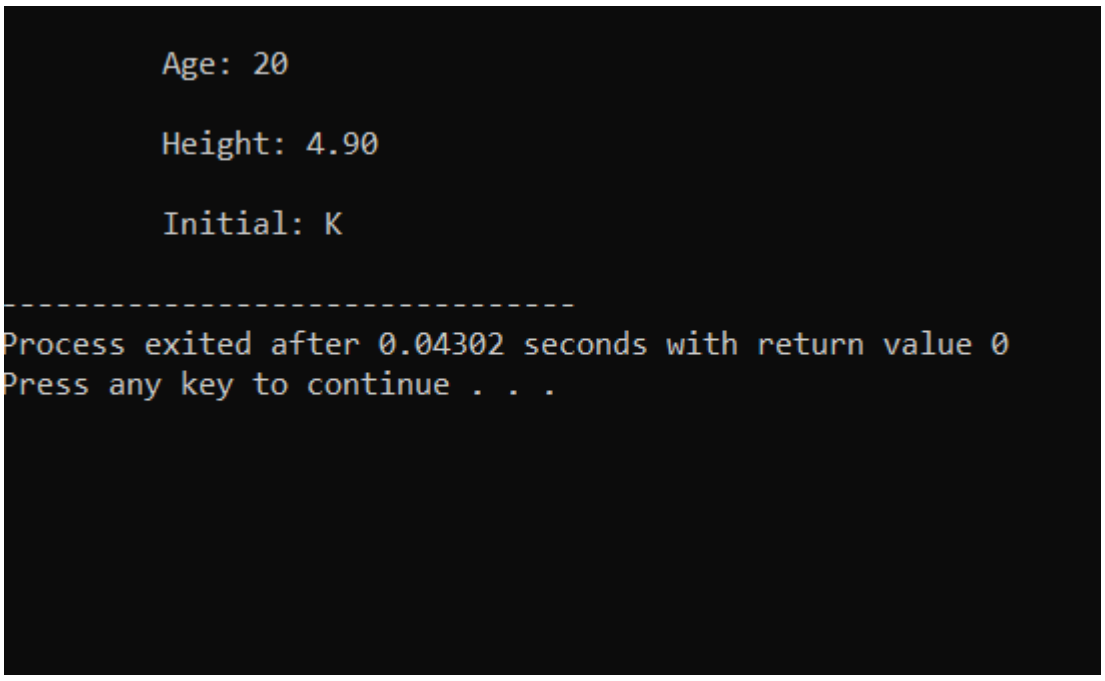
Ex:

```
int age = 25;            // Declare integer variable
float height = 5.9;      // Declare float variable
char grade = 'A';        // Declare char variable
```

Full example of a c program:

```
#include<stdio.h>

main()

{

        int age = 20;

        float height = 4.9;

        char initial = 'K';

        printf("\n\t Age: %d\n", age);

        printf("\n\t Height: %.2f\n", height);

        printf("\n\t Initial: %c\n", initial);

}
```

OUTPUT:



4. Operators in C

→There are many operators in c programing.

→Below is a detailed explanation of each type of operator, with examples.

1. Arithmetic operators

Arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, division, and modulus.

List of Arithmetic operators:

+ : Addition

- : Subtraction

* : Multiplication

/ : Division

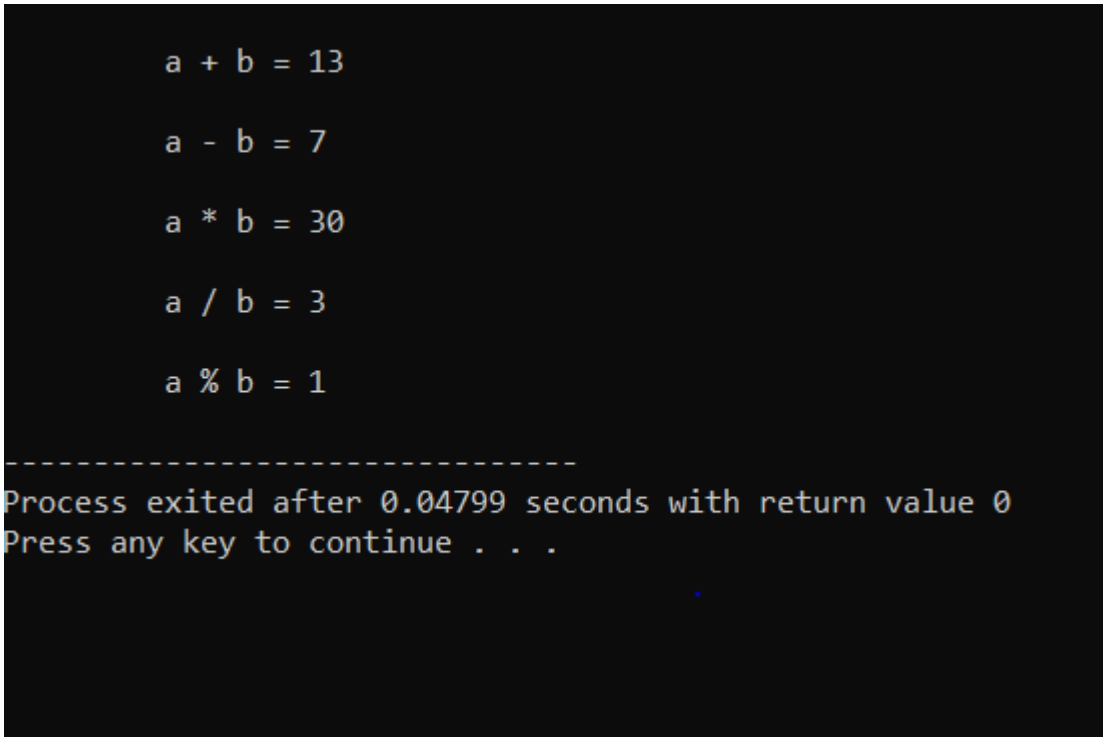
% : Modulus

Example:

```
#include <stdio.h>

main()

{

        int a = 10, b = 3;

         printf("\n\t a + b = %d\n", a + b);

         printf("\n\t a - b = %d\n", a - b);

         printf("\n\t a * b = %d\n", a * b);

         printf("\n\t a / b = %d\n", a / b);
```

```
    printf("\n\t a %% b = %d\n", a % b);
}
```

OUTPUT:



2. Relational operators:

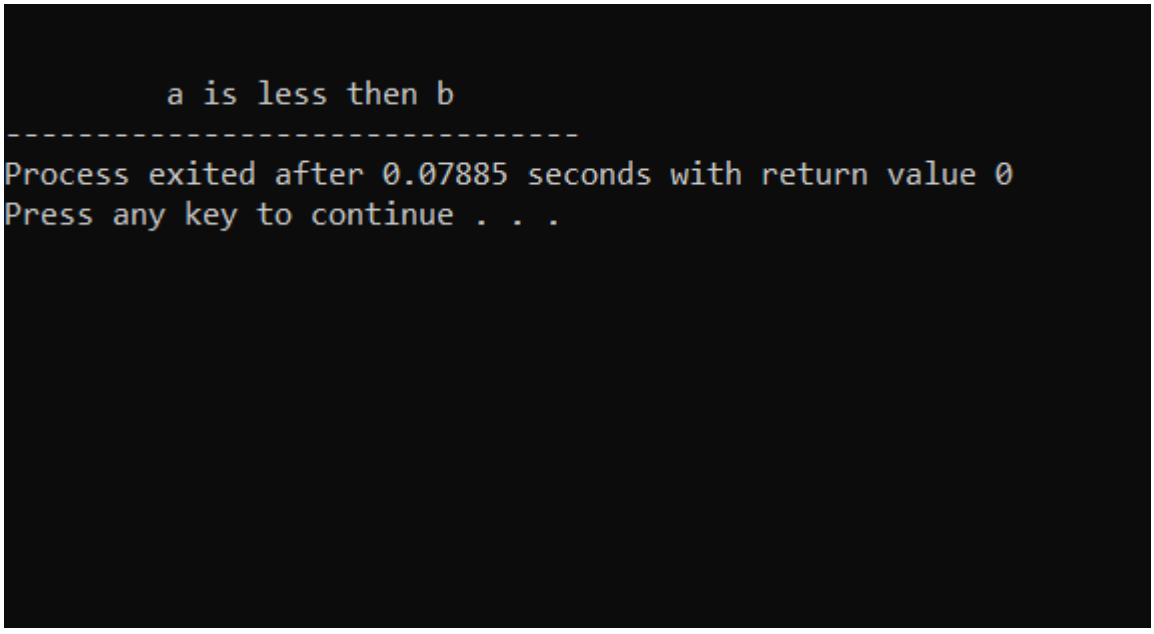
→These operators are used to compare two values. They return either true (1) or false (0).

== : Equal to (checks if two operands are equal)
!= : Not equal to (checks if two operands are not equal)
> : Greater than (checks if the left operand is greater than the right)
< : Less than (checks if the left operand is less than the right)
>= : Greater than or equal to (checks if the left operand is greater than or equal to the right)
<= : Less than or equal to (checks if the left operand is less than or equal to the right)

Ex:

```
int a = 10, b = 20;

if (a < b)

{

        printf("\n\n\t a is less then b ");

}
```

OUTPUT:



3. logical operators

Logical operators are used to perform logical operations, typically in conditional statements. They return Boolean values.

&& : Logical AND (e.g., a > 0 && b < 10)

|| : Logical OR (e.g., a > 0 || b < 10)

! : Logical NOT (e.g., !a checks if a is zero)

EX:

```
#include<stdio.h>
main()
{
   int a = 10, b = 5, c = 20;
   if (a > b && c > a)
   {
         printf("\n\n\t Both conditions are true: a > b and c > a\n");
    }
     else
    {
         printf("\n\n\t At least one of the conditions is false\n");
     }
  }
```

OUTPUT:

```
        Both conditions are true: a > b and c > a

--------------------------------
Process exited after 0.061 seconds with return value 0
Press any key to continue . . .
```

4. Assignment operators

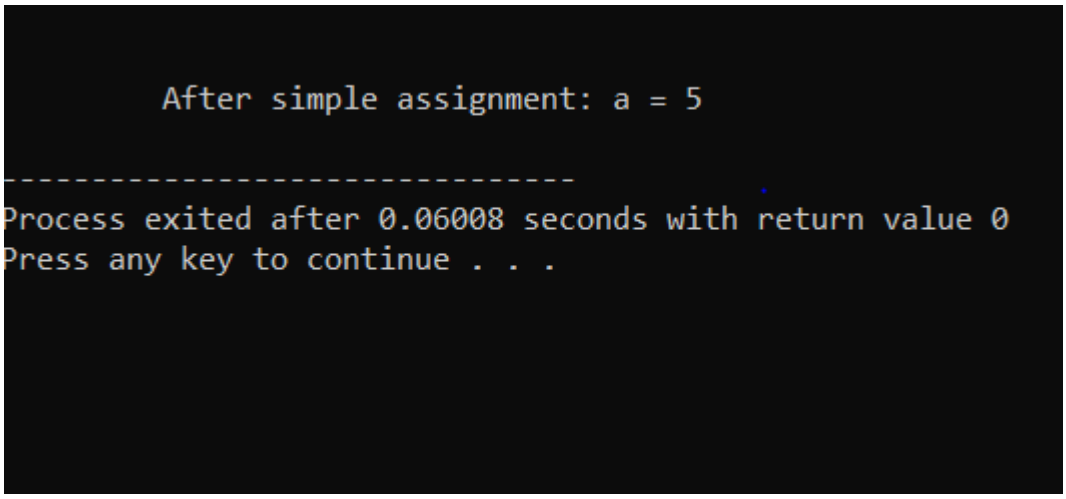
→Assignment operators are used to assign values to variables.

= : Simple assignment (e.g., a = 5)
+= : Add and assign (e.g., a += 5 is equivalent to a = a + 5)
-= : Subtract and assign (e.g., a -= 5 is equivalent to a = a - 5)
*= : Multiply and assign (e.g., a *= 5 is equivalent to a = a * 5)
/= : Divide and assign (e.g., a /= 5 is equivalent to a = a / 5)
%= : Modulus and assign (e.g., a %= 5 is equivalent to a = a % 5)

Ex:

```
#include <stdio.h>
main()
{
   int a = 10, b = 5;
   a = b;
   printf("\n\n\t After simple assignment: a = %d", a);
}
```

OUTPUT:



5. Increment/decrement operators

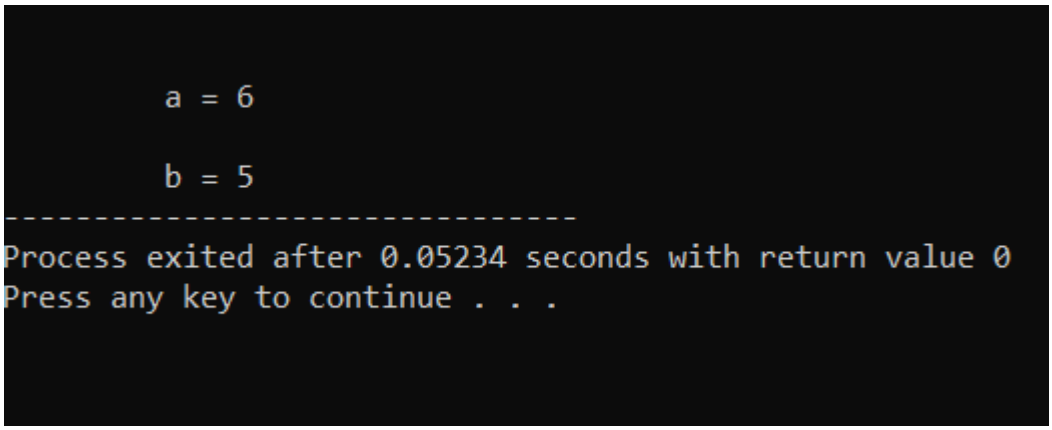
→These operators are used to increase or decrease the value of a variable by 1.

++ : Increment (e.g., a++ or ++a)
-- : Decrement (e.g., a-- or --a)

EX:

```
#include <stdio.h>

main()

{

   int a = 5;

   int b = a++;

   printf("\n\n\t a = %d", a);

   printf("\n\n\t b = %d", b);

}
```

OUTPUT:



6. bitwise operator

→Bitwise operators operate on the individual bits of integers. They perform operations bit by bit.

& : Bitwise AND

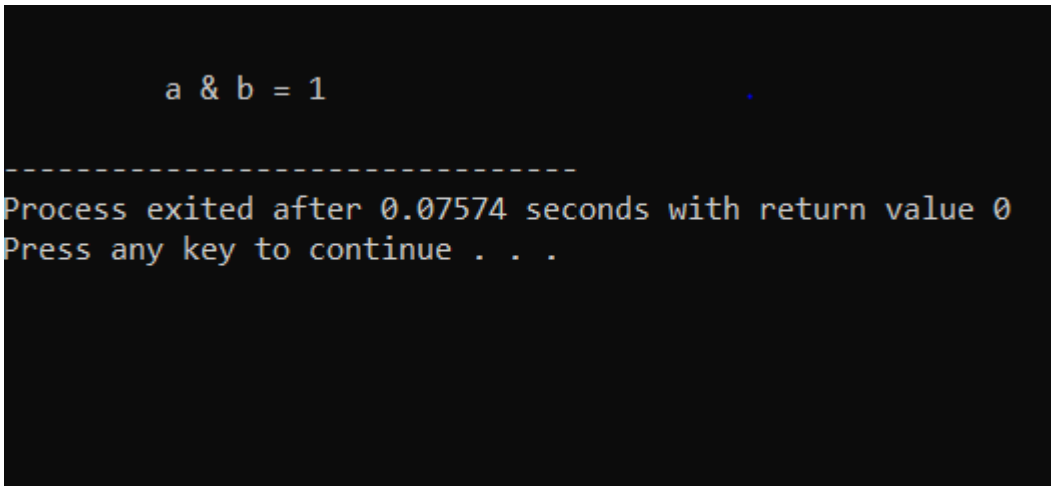
| : Bitwise OR

^ : Bitwise XOR
~ : Bitwise NOT
<< : Left shift
>> : Right shift

Ex:

```
#include <stdio.h>
main()
{
   int a = 5;
   int b = 3;

   int result = a & b;
   printf("\n\n\t a & b = %d\n", result);
}
```

OUTPUT:



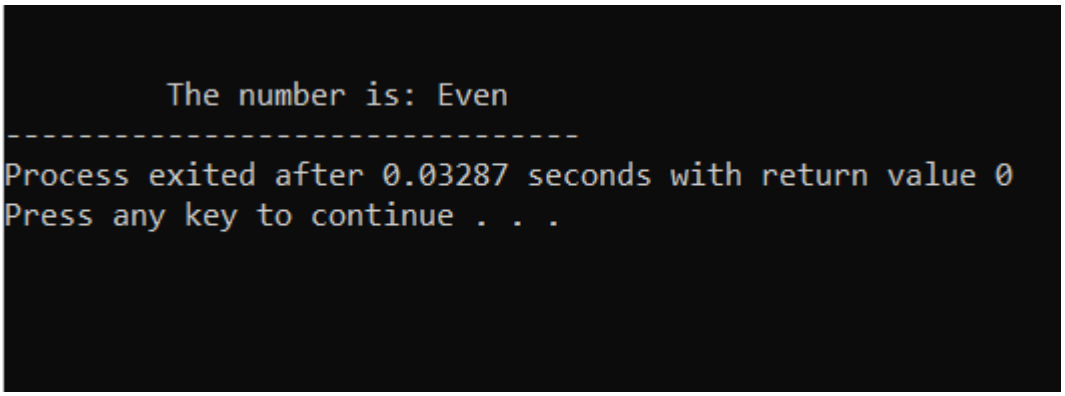
7. conditional operator

→The conditional (ternary) operator is a shorthand for if-else statements.

Ex:

```
#include <stdio.h>

main()

{

   int num = 4;

   const char* result = (num % 2 == 0) ? "Even" : "Odd";

   printf("\n\n\t The number is: %s", result);

}
```

OUTPUT:



5. Control Flow Statements in C

→Explain the one by one statement:

1. if statement:

→The if statement allows you to execute a block of code only if a specified condition is true.

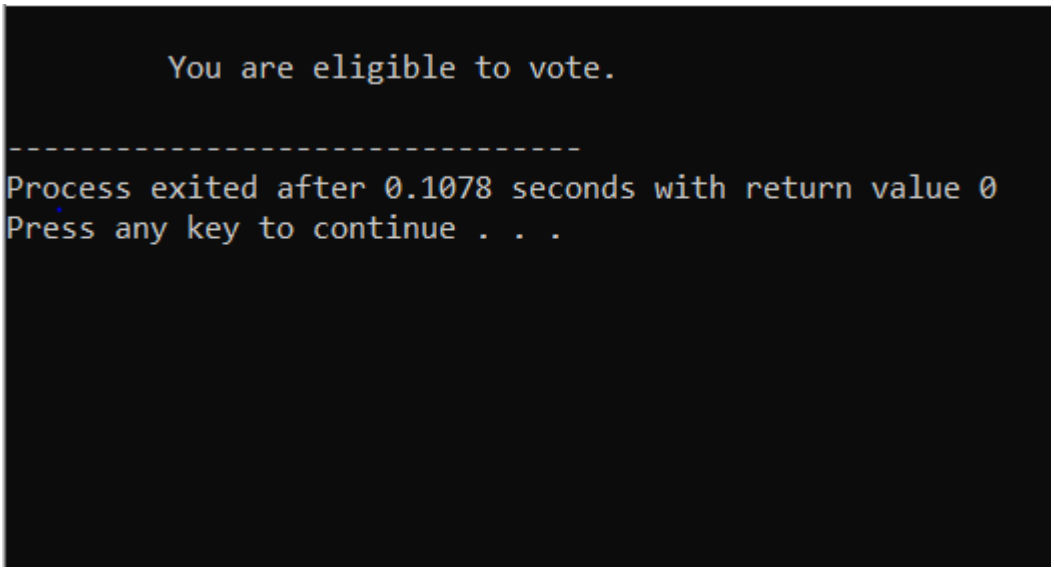
Syntax:

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

Ex:

```
#include <stdio.h>
main()
 {
        int age = 18;
        if (age >= 18)
                {
                        printf("\n\t You are eligible to vote.\n");
                }
 }
```

OUTPUT:



2. else Statement

→The else statement provides an alternative block of code to execute if the condition in the if statement is false.

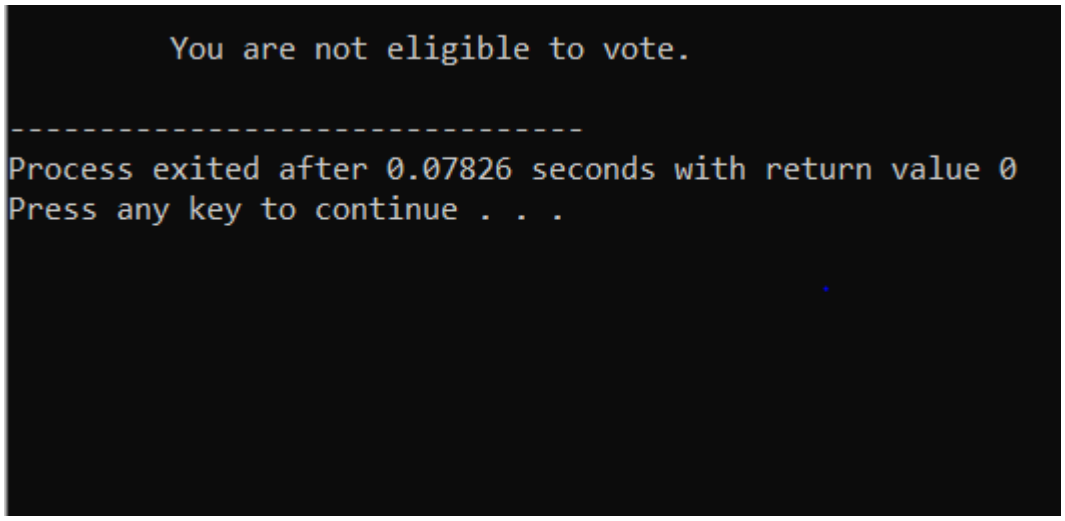
Syntax:

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

Ex:

```
#include <stdio.h>

main()

{

        int age = 16; if (age >= 18)

         {

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

          }

          else

         {

                 printf("\n\t You are not eligible to vote.\n");

          }

    }
```

OUTPUT:



3. else if statement

→The else if statement is used when you want to check multiple conditions.

→If the first if condition is false, it will check the else if condition.

Syntax:

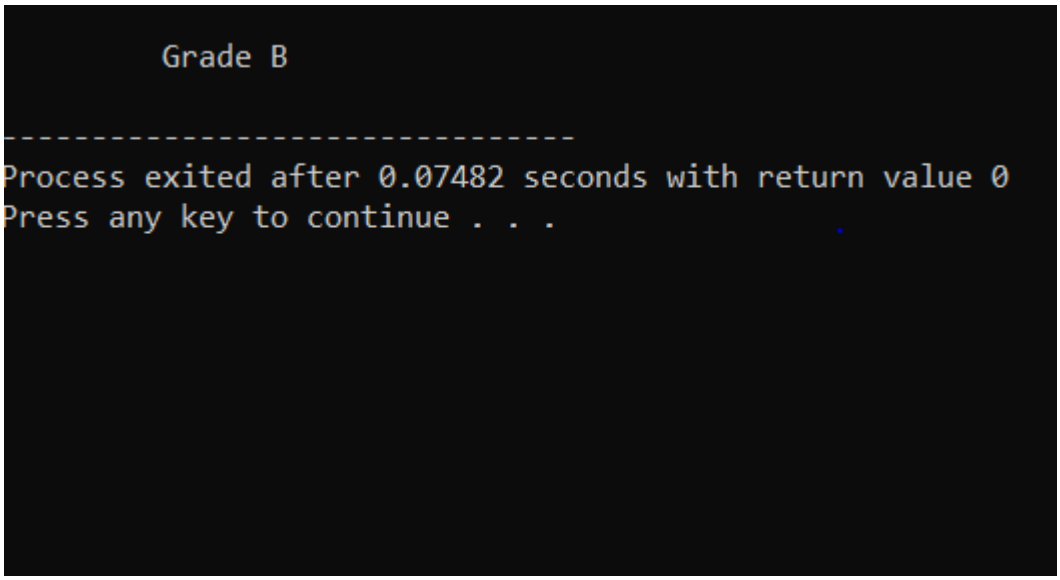
```
if (condition1) {
                    // Code to be executed if condition1 is
                }
                else if (condition2)
                {
                    // Code to be executed if condition2 is true
                }
                else
                 {
                    // Code to be executed if neither condition1 nor
                condition2 is true
                }
```

Ex:

```
#include <stdio.h>
 main()
 {
   int score = 85;
   if (score >= 90) {
      printf("\n\t Grade A\n");
   } else if (score >= 80) {
```

```
        printf("\n\t Grade B\n");
    } else if (score >= 70) {
        printf("\n\t Grade C\n");
    } else {
        printf("\n\t Grade F\n");
    }

}
```

OUTPUT:



4. Nested if Statements

→You can also nest if statements inside another if, allowing for more complex decision-making.

Syntax:

```
if (condition1)

{

        if (condition2)

        {

         // Code to execute if both condition1 and condition2 are true
```

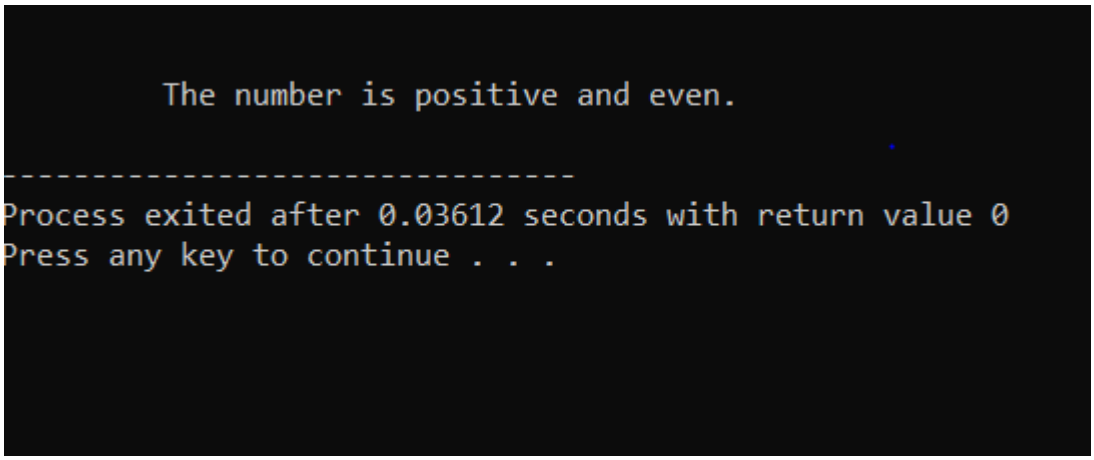
```
        }
}
```

Ex:

```
#include <stdio.h>
main()
{
   int num = 10;
   if (num > 0)
   {
      if (num % 2 == 0)
       {
              printf("\n\n\t The number is positive and even.\n");
       }
        else
       {
             printf("\n\n\t The number is positive but odd.\n");
        }
   }
   else
  {
       printf("\n\n\t The number is not positive.\n");
```

```
    }
}
```

OUTPUT:



5. Switch statement

→It is typically used when you have multiple possible values for a single variable.

Syntax:

```
switch (expression)
{
        case value1:
                    // Code to execute if expression == value1
                    Break;
        case value2:
                    // Code to execute if expression == value2
                    Break;
        default:
```

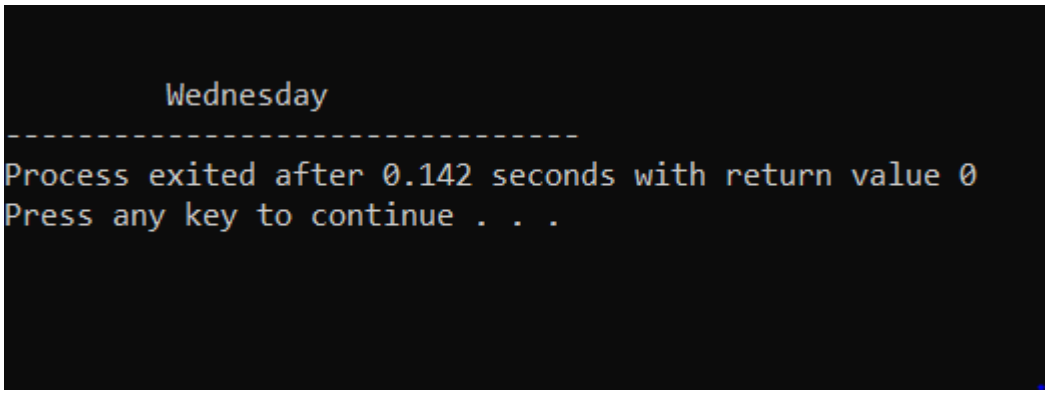
```
}
```

Ex:

```
#include <stdio.h>
 main()
{
   int day = 3;
   switch (day)
   {
      case 1:
         printf("\n\n\t Monday");
         break;
      case 2:
         printf("\n\n\t Tuesday");
         break;
      case 3:
         printf("\n\n\t Wednesday");
         break;
      case 4:
         printf("\n\n\t Thursday");
         break;
      case 5:
```

```
            printf("\n\n\t Friday");
            break;
        case 6:
            printf("\n\n\t Saturday");
            break;
        case 7:
            printf("\n\n\t Sunday");
            break;
        default:
            printf("\n\n\t Invalid day");
    }
}
```

OUTPUT:



6. Looping in C:

1. **While Loop:**

→The condition is checked before executing the loop body.

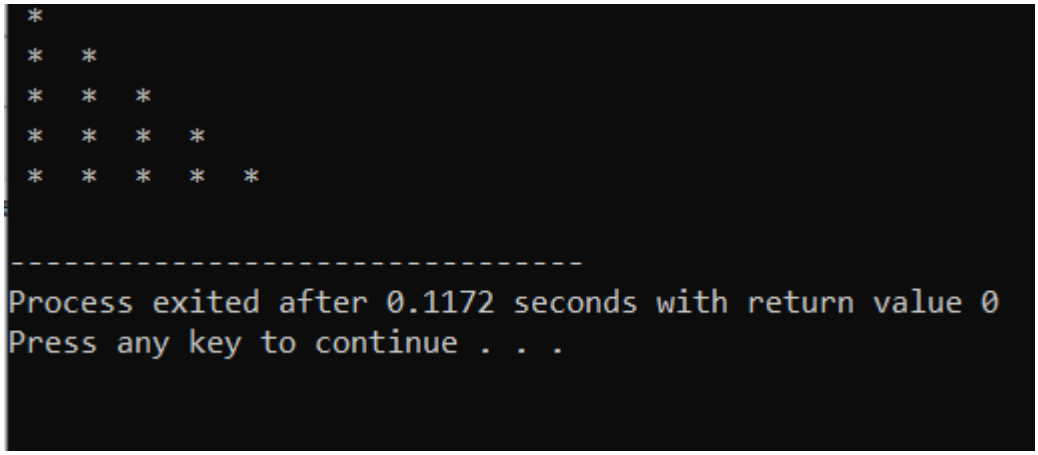
→A while loop keeps repeating a block of code while a condition is true.

Syntax: while condition{#code to execute }

Ex:

```
#include<stdio.h>
main()
{
    int r,c;
    r=1;
    while(r<=5)
    {
        c=1;
        while(c<=r)
        {
            printf(" * ");
            c++;
        }
        printf("\n");
        r++;
    }
}
```

OUTPUT:



2. For loop:

→A for loop is a way to repeat a block of code multiple times.

→It automatically goes through a list, range, or sequence of items and executes the code for each item.

Syntax:

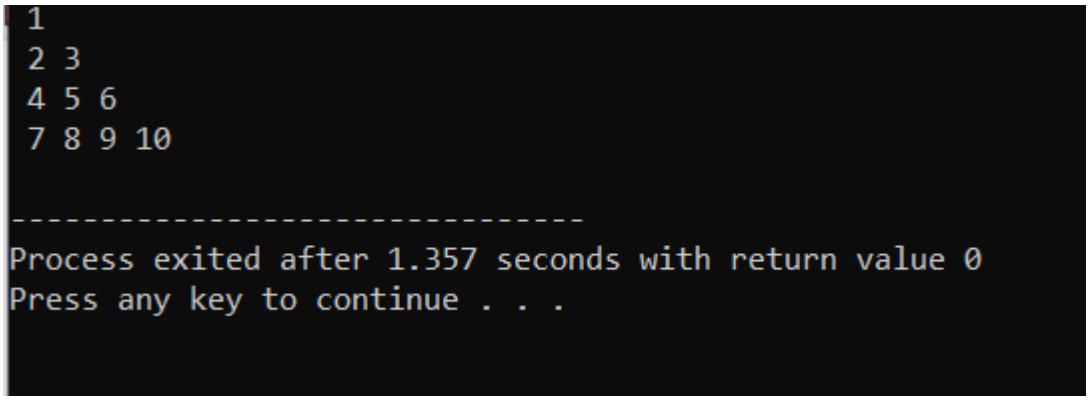
```
For(initialization ; condition ; inc/dec)
{
        Block of code;
}
```

Ex:

```
#include<stdio.h>
main()
{
        int r,c,num=1;
        for(r=1 ; r<=4 ; r++)
        {
```

```
            for(c=1 ; c<=r ; c++)
            {
                printf(" %d",num);
                num++;
            }
            printf("\n");
        }
}
```

OUTPUT:



3. do while loop:

→A do-while loop runs the code first, then checks if it should keep running based on a condition.

→It always runs at least once

Syntax:

Initialization:

do

{

Block of code;

Increment/decrement;

}while(condition);

Ex:

```
#include<stdio.h>
main()
{
    int r=1;
    do
    {
        printf(" %d ",r);
        r++;
    }while(r<=10);
}
```

OUTPUT:

```
 1  2  3  4  5  6  7  8  9  10
--------------------------------
Process exited after 0.05382 seconds with return value 0
Press any key to continue . . .
```

7. Loop Control Statements:

1. Break statement:

→The break statement is used to exit from a loop or a switch statement prematurely.

Ex:

```
#include<stdio.h>
main()
{
   for(int i=1; i<=10; i++)
  {
      if(i==5)
      {
         break;
      }
      printf("%d ", i);
   }
   printf("\nLoop terminated.\n");
}
```

OUTPUT:

```
1 2 3 4
Loop terminated.

--------------------------------
Process exited after 0.09376 seconds with return value 0
Press any key to continue . . .
```

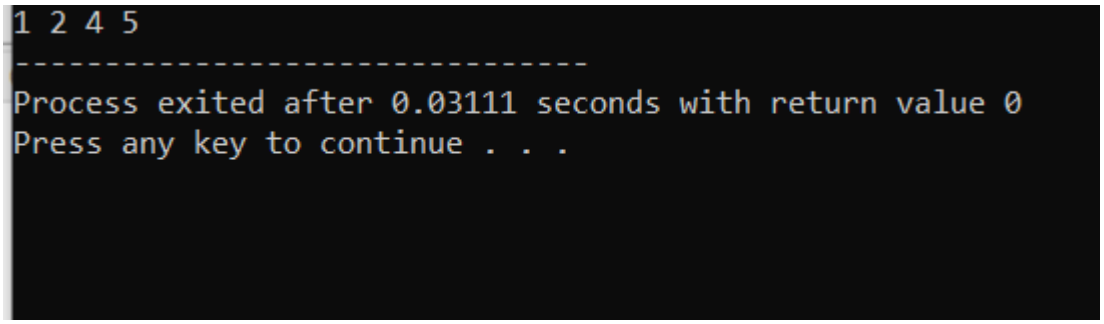
2. Continue statement:

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

Ex:

```
#include<stdio.h>
main()
{
   for(int i=1; i<=5; i++)
   {
      if(i==3)
      {
          continue;
      }
      printf("%d ", i);
   }
}
```

OUTPUT:



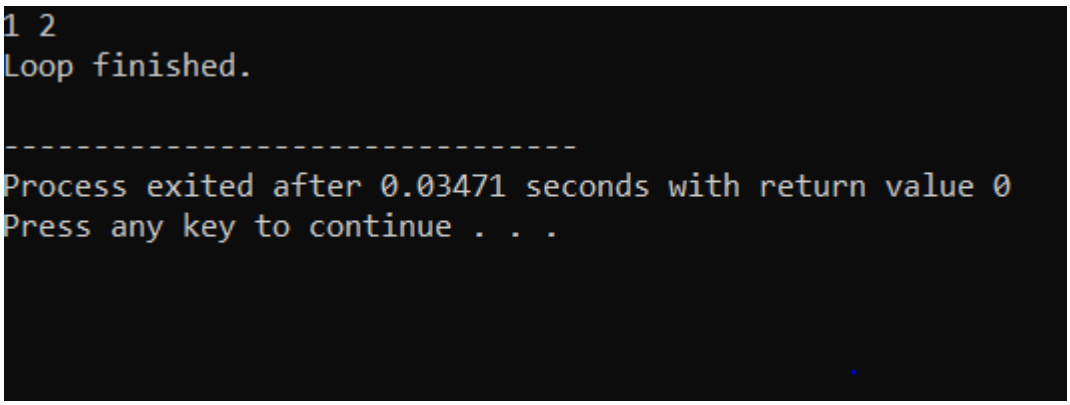
3. goto statement:

→The goto statement is used to jump to another part of the program, usually within the same function.

Ex:

```
#include<stdio.h>
main(){
    int i=0;
    start_loop:
    i++;
    if(i==3)
    {
        goto end_loop;
    }
    printf("%d ", i);
    goto start_loop;
  end_loop:
    printf("\nLoop finished.");
}
```

OUTPUT:



8. functions in c:

→ A function in C is a block of code that performs a specific task.

→In C programming, functions are blocks of code designed to perform a specific task.

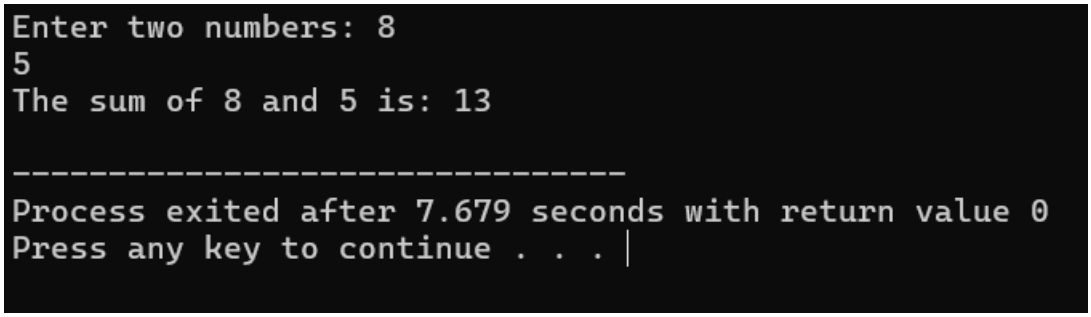
Syntax:

```
return_type function_name(parameters_list);
{
        // body of the function
}
```

Ex:

```
#include<stdio.h>
int add(int, int);
int main()
{
   int num1, num2, sum;
   printf("Enter two numbers: ");
   scanf("%d %d", &num1, &num2);
   sum = add(num1, num2);
   printf("The sum of %d and %d is: %d\n", num1, num2, sum);
   return 0;
}
int add(int a, int b)
{
   return a + b;
}
```

Output:



9. Arrays in C:

What is an array?

→An array is a collection of variables, all of the same type, that are stored in contiguous memory locations.

One-dimensional array:

→ A one-dimensional array in C is just a simple list of values that are all of the same type (like integers, characters, etc.).

→ One-Dimensional means it is a single line or row of elements.

Syntax:

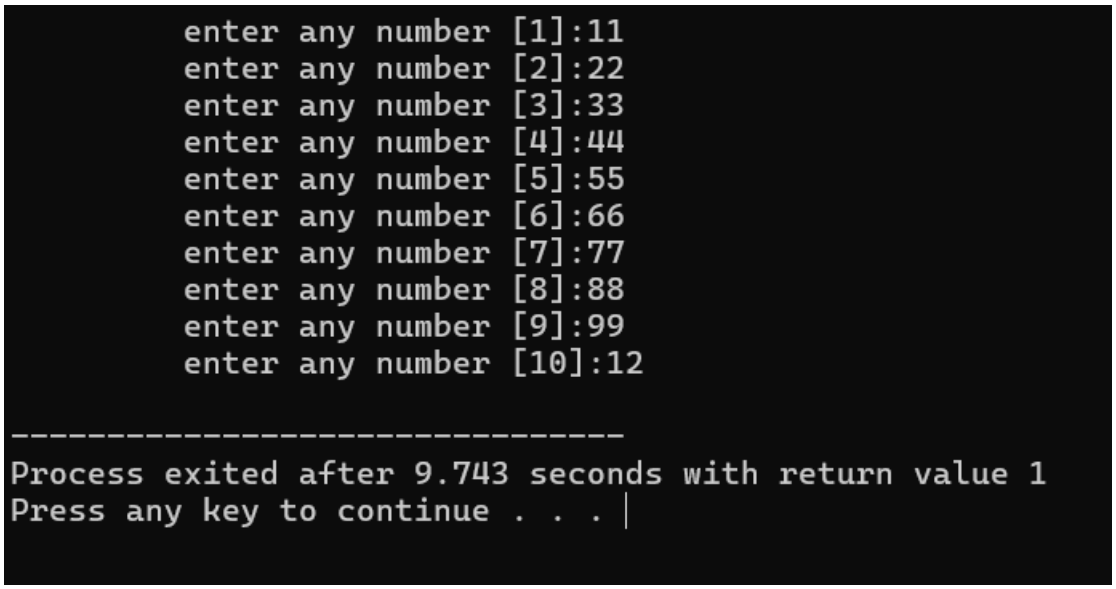
type arrayName[size];

Ex:

```
#include<stdio.h>
main()
{
        int arr[10],i;
        for(i=0;i<10;i++)
        {
                printf("\t enter any number [%d]:",i+1);
```

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

Output:



2.multi-Dimensional arrays:

- A two-dimensional array is an array of arrays. You can think of it like a matrix or grid with rows and columns.

Syntax:

data_type array_name[rows][columns];

Example:

```
#include<stdio.h>
main()
{
        int arr[3][3];
        int r,c;

        for(r=0;r<3;r++)
```

```
        {
                for(c=0;c<3;c++)
                {
                        printf("\n\n\t arr[%d][%d]",r,c);
                        scanf("%d",&arr[r][c]);
                }
        }


        for(r=0;r<3;r++)
        {
                for(c=0;c<3;c++)
                {
                        printf("%d",arr[r][c]);
                }
                printf("\n");
        }

}
```

Output:

arr[0][0]2
arr[0][1]3
arr[0][2]4
arr[1][0]5
arr[1][1]6
arr[1][2]7
arr[2][0]5
arr[2][1]3
arr[2][2]4
234
567
534

3.three-dimensional arrays:

- A three-dimensional array can be thought of as an array of arrays of arrays.
- It is often used to represent data that can be described with three coordinates

Syntax:

data_type array_name[depth][rows][columns];

Example:

```
#include<stdio.h>
main()
{
	int mat[3][3][3];
	int m, r, c;
	for(m=0;m<3;m++)
	{
		printf("\n\n\t Matrix [%d] : ", m+1);
		for(r=0;r<3;r++)
		{
			for(c=0;c<3;c++)
			{
				printf("\n\n\t Input mat[%d][%d][%d] : ",
m, r, c);
				scanf("%d",&mat[m][r][c]);
			}
		}
	}

	for(m=0;m<3;m++)
	{
		printf("\n\n\t Matrix [%d] : \n\n", m+1);
		for(r=0;r<3;r++)
		{
			for(c=0;c<3;c++)
			{
				printf(" %d", mat[m][r][c]);
```

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

Output:

```
Matrix [1] :
        Input mat[0][0][0] : 1
        Input mat[0][0][1] : 2
        Input mat[0][0][2] : 3
        Input mat[0][1][0] : 4
        Input mat[0][1][1] : 5
        Input mat[0][1][2] : 6
        Input mat[0][2][0] : 7
        Input mat[0][2][1] : 8
        Input mat[0][2][2] : 9

 Matrix [2] :
        Input mat[1][0][0] : 9
        Input mat[1][0][1] : 8
        Input mat[1][0][2] : 7
        Input mat[1][1][0] : 6
        Input mat[1][1][1] : 5
        Input mat[1][1][2] : 4
        Input mat[1][2][0] : 3
        Input mat[1][2][1] : 2
        Input mat[1][2][2] : 1

 Matrix [3] :
        Input mat[2][0][0] : 1
        Input mat[2][0][1] : 2
        Input mat[2][0][2] : 3
        Input mat[2][1][0] : 4
        Input mat[2][1][1] : 5
        Input mat[2][1][2] : 9
```

```
Input mat[2][2][0] : 8
Input mat[2][2][1] : 7
Input mat[2][2][2] : 6

Matrix [1] :
                1 2 3
                4 5 6
                7 8 9
Matrix [2] :
                9 8 7
                6 5 4
                3 2 1
Matrix [3] :
                1 2 3
                4 5 9
                8 7 6
```

Key differences between one-dimensional and multi-Dimensional arrays:

Feature	**One-Dimensional Array**	**Multi-Dimensional Array**
Structure	A single row of elements	Multiple rows and columns (like a matrix or table)
Size	Defined by a single value (number of elements)	Defined by multiple values (e.g., rows and columns)
Indexing	Single index (e.g., arr[0])	Multiple indices (e.g., matrix[0][0])
Example	int arr[5] = {1, 2, 3, 4, 5};	int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
Use Case	Storing a list of elements	Storing tables or grids of data (rows and columns)

10.Pointers in C:

→ In C, a pointer is a variable that stores the memory address of another variable.
→ Instead of holding a data value (like an integer or a character), a pointer holds the location where the value is stored in the computer's memory.

1. Declaring Pointers:

→ To declare a pointer in C, you need to specify the type of data it will point to followed by an asterisk '*'.

Syntax:
data_type *pointer_name;

Example: int *ptr;
char *ch;

2. Initializing Pointers:

→ You can initialize a pointer by assigning it the address of an existing variable using the address-of operator (&).

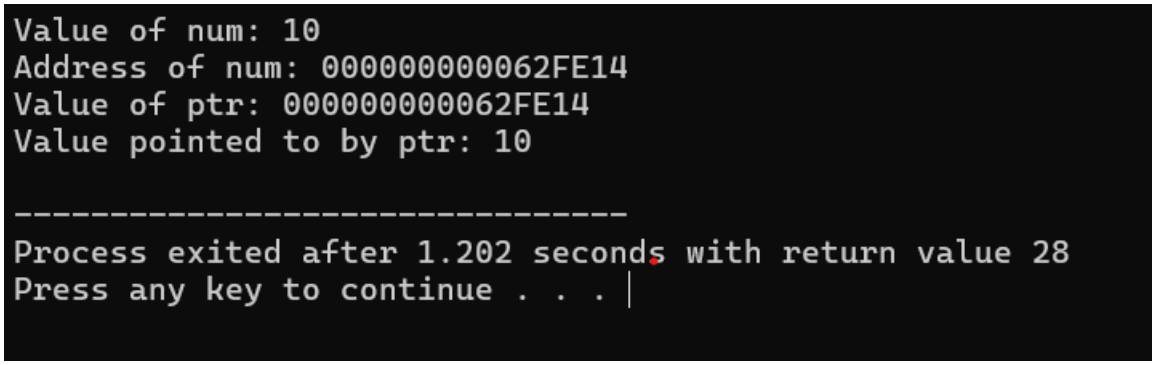
Syntax:
pointer_name = &variable_name;

Ex:

```
#include <stdio.h>
main()
{
   int num=10;
   int *ptr;
   ptr=&num;
```

```
    printf("Value of num: %d\n", num);
    printf("Address of num: %p\n", (void*)&num);
    printf("Value of ptr: %p\n", (void*)ptr);
    printf("Value pointed to by ptr: %d\n", *ptr);
}
```

Output:



1. Access Memory Directly

→ Pointers give you the ability to directly access and change the values stored in memory.

→ This allows for better control over how data is managed.

2. Make Programs Faster and More Efficient

→ Instead of copying large amounts of data, you can pass a pointer (memory address) to a function, which is faster and uses less memory.

3.Work with Arrays and Strings Easily

→ Arrays and strings in C are treated as pointers, so you can use pointers to navigate through and modify these data structures more easily.

4.Create Complex Data Structures

→ Pointers let you create advanced data structures like linked lists, trees, and graphs, which are essential for many types of programs.

5. Dynamic Memory Management

→ Pointers allow you to allocate memory while the program is running, giving you more flexibility when working with large or unpredictable amounts of data.

11. Strings in C:

1.strlen()

→ **Purpose**: This function returns the number of characters in a string (excluding the null terminator \0).

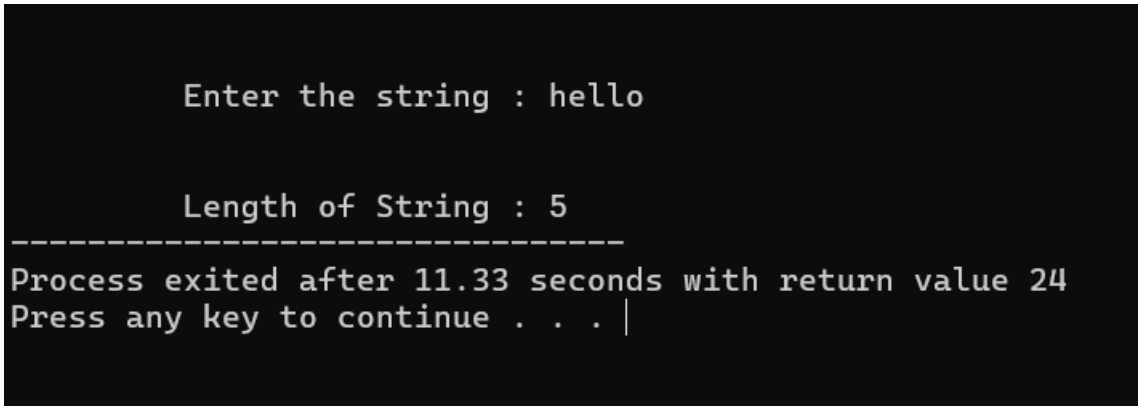
Syntax:

size_t strlen(const char *str);

Example:

```
#include <stdio.h>
main()
{
   char str[20];
   int length = strlen(str);

   printf("\n\n\t Enter the string : ");
   scanf("%s",&str);
        printf("\n\n\t Length of String : %d", strlen(str));
}
```

Output:



2.strcpy():

→ strcpy(destination, source): It copies the content of the source string into the destination string.

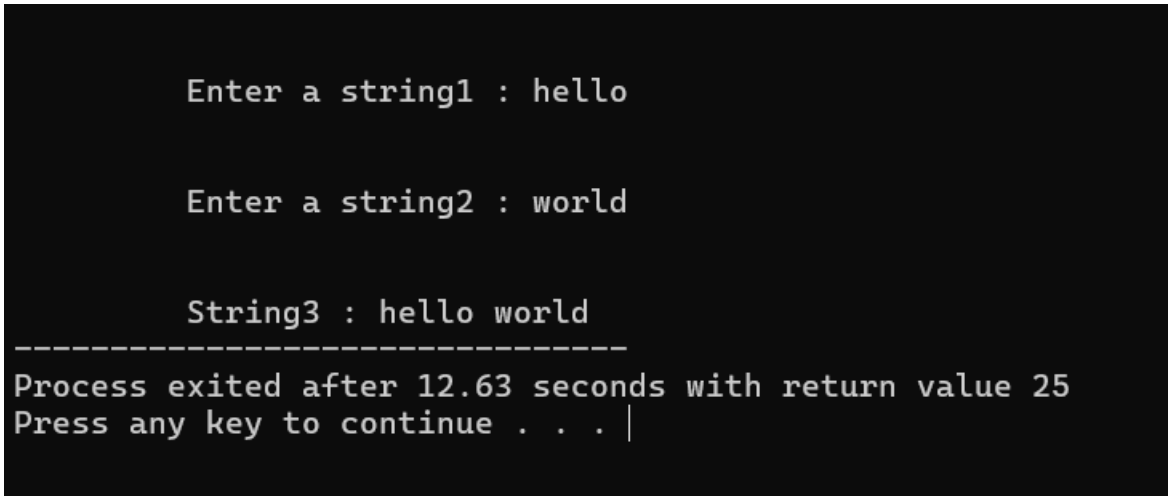
Syntax:

char *strcpy(char *dest, const char *src);

Example:

```
#include <stdio.h>
int main()
{
	char str1[20], str2[20], str3[40];

	printf("\n\n\t Enter a string1 : ");
	gets(str1);
	printf("\n\n\t Enter a string2 : ");
	gets(str2);

	strcpy(str3, strcat(str1, str2));
	printf("\n\n\t String3 : %s", str3);

}
```

Output:



3.strcat():

→strcat(destination, source): It appends the content of the source string to the end of the destination string.

Syntax:

char *strcat(char *dest, const char *src);

4.strcmp():

→strcmp(str1, str2) compares the two strings str1 and str2 character by character.

Syntax:

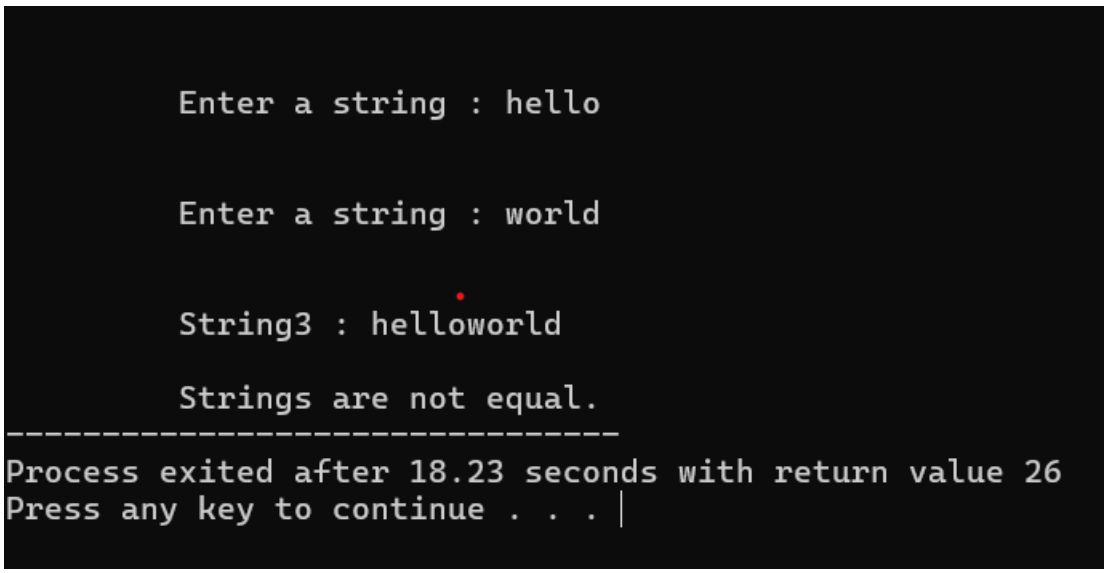
int strcmp(const char *str1, const char *str2);

Example:

```
#include <stdio.h>
int main()
{
        char str1[20], str2[20],str3[40];
        printf("\n\n\t Enter a string : ");
        scanf("%s",&str1);
        printf("\n\n\t Enter a string : ");
        scanf("%s",&str2);
```

```
    strcpy(str3, strcat(str1, str2));
        printf("\n\n\t String3 : %s", str3);

        if(stricmp(str1, str2)==0)
            printf("\n\n\t Strings are equal.");
        else
            printf("\n\n\t Strings are not equal.");
}
```

Output:



5. strchr()

→strchr(str, ch) searches for the first occurrence of the character ch in the string str.
→It returns a pointer to the first occurrence of ch in str.

Syntax:

char *strchr(const char *str, int ch);

12. Structures in C:

→A structure in C is a user-defined data type that allows you to group different types of variables under a single name.

How to Declare a Structure?

→To declare a structure, you use the struct keyword followed by the structure name and its members.

Syntax:

→ struct StructureName
{
    dataType member1;
  dataType member2;
};

Example:

```
#include <stdio.h>
struct Book
{
        char title[50];
        char author[50];
        float price;
};
```

How to Initialize a Structure?

→ You can initialize the structure at the time of declaration or later when you create an instance of it.

Example:

struct Book book1 = {"C Programming", "Dennis Ritchie", 299.99};

How to Access Structure Members?

→ You can access the members of a structure using the dot operator (.).

Example:

```
#include <stdio.h>
struct Book
{
   char title[50];
   char author[50];
   float price;
};
main()
{
   struct Book book1;

   strcpy(book1.title, "C Programming");
   strcpy(book1.author, "Dennis Ritchie");
   book1.price = 299.99;

   printf("Book Title: %s\n", book1.title);
   printf("Book Author: %s\n", book1.author);
   printf("Book Price: %.2f\n", book1.price);
}
```

Output:

```
Book Title: C Programming
Book Author: Dennis Ritchie
Book Price: 299.99

--------------------------------
Process exited after 3.494 seconds with return value 19
Press any key to continue . . .
```

13. File Handling in C:

→ Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.
→ Save data permanently for future use.

1. Opening Files

→ To work with a file, you need to open it first using the built-in open() function.

Syntax:

```
file = open('file.txt', 'r')  # Open file in read mode
```

3. Reading from a File

→ Once a file is opened, you can read its contents using various methods:
→ **read()**: Reads the entire file into a single string.

Syntax:

```
content = file.read()
print(content)
```

3. Writing to a File

→ To write to a file, you need to open it in write ('w'), append ('a'), or exclusive creation ('x') mode.
→ **write()**: Writes a string to the file.
Python.

Syntax:

```
file = open('file.txt', 'w') # Open for writing file.write('Hello, World!\n')
```

4. Closing a File

→ After performing file operations, it's important to close the file to free up resources.

→ This is done using the close() method.

Syntax:

file.close()

5. Using with Statement (Context Manager)

→ To avoid forgetting to close a file or handling exceptions properly, Python provides a with statement.

Syntax:

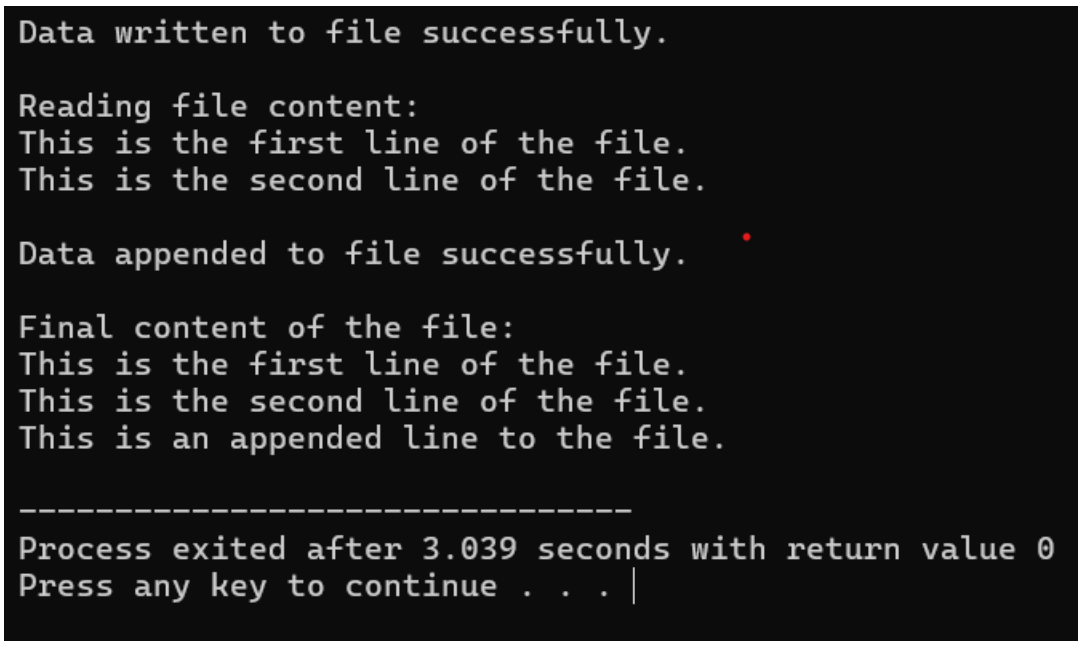
with open('file.txt', 'r') as file: content = file.read()
print(content)

Example:

```
#include <stdio.h>
main()
{
   FILE*file;
   file=fopen("example.txt", "w");
   if(file==NULL)
         {
               printf("Error opening file for writing.\n");
               return 1;
          }
         fprintf(file, "This is the first line of the file.\n");
         fprintf(file, "This is the second line of the file.\n");
        printf("Data written to file successfully.\n");

         fclose(file);
        file=fopen("example.txt", "r");
        if(file==NULL)
        {
```

```
            printf("Error opening file for reading.\n");
             return 1;
        }
        char ch;
         printf("\n Reading file content:\n");
        while((ch=fgetc(file))!=EOF)
        {
             putchar(ch);
          }
         fclose(file);

        file=fopen("example.txt", "a");
        if(file==NULL)
        {
            printf("Error opening file for appending.\n");
             return 1;
        }
        fprintf(file, "This is an appended line to the file.\n");
        printf("\n Data appended to file successfully.\n");

        fclose(file);
        file=fopen("example.txt", "r");
         if(file==NULL)
        {
            printf("Error opening file for reading.\n");
            return 1;
        }
         printf("\n Final content of the file:\n");
        while((ch=fgetc(file))!=EOF)
        {
            putchar(ch);
         }
         fclose(file);
}
```

Output:


Data written to file successfully.
Reading file content:
This is the first line of the file.
This is the second line of the file.
Data appended to file successfully.
Final content of the file:
This is the first line of the file.
This is the second line of the file.
This is an appended line to the file.

Process exited after 3.039 seconds with return value 0
Press any key to continue . . .

****