C Programming Short Notes

Part - 9

1. Little endian and Big endian

➢ Endian refers to **the byte order used to represent multi-byte data types**, such as integers, floating-point numbers, and characters, in computer memory. There are two main types of endianness: **little-endian and big-endian**.

1. Little-endian

- In a little-endian system, the least significant byte **(LSB)** of a multi-byte data type is stored at the **lowest memory address**, and the most significant byte **(MSB)** is stored at the **highest memory address**.
- This means that the data is stored in memory from **right to left**, with the least significant byte first.
- Most modern processors, including x86 and x86-64 processors, use the little-endian format.

2. Big-endian

- In a big-endian system, the most significant byte **(MSB)** of a multi-byte data type is stored at the **lowest memory address**, and the least significant byte **(LSB)** is stored at the **highest memory address**.
- This means that the data is stored in memory from **left to right**, with the most significant byte first.
- Big-endian systems are also used in some architectures, such as certain ARM processors and older mainframe systems.

Let's consider a 4-byte integer ‘0x12345678’:

In a little-endian system, it will be stored in memory as:

```
Address: 0x00  0x01  0x02  0x03
Value:    78    56    34    12
```

In a big-endian system, it will be stored in memory as:

```
Address: 0x00  0x01  0x02  0x03
Value:    12    34    56    78
```

2. Error Handling

- The specific error codes provided by '**errno**' in C can vary depending on the operating system and the C library implementation.
- Header file for error handaling

#include<errno.h>

1. Operation not premitted
2. No such file or directory
3. No such process
4. Interupted system call
5. I/O errors
6. No such devices

Example 1 :

```
int main()
{
   FILE *ptr;
   ptr=fopen("rtrr.txt","r");
   printf("Error Number : %d",errno);
   perror("\nError  ");
}
```

Output

```
Error Number : 2
Error  : No such file or directory
```

Example 2 :

```
int main()
{
   FILE *ptr;
   ptr=fopen("rtrr.txt","r");
   printf("Error : %s",strerror(errno));
}
```

Output

```
Error : No such file or directory
```

3. Command Line argument

- Command-line arguments **provide a way to pass inputs to a program from the command line when the program is executed.**
- These arguments allow programs to be more flexible and configurable without modifying the source code.

In C, command-line arguments are passed to the main function through two parameters: argc and argv.

- **argc (argument count):** It is an integer that represents the **number of command-line arguments passed** to the program, including the name of the program itself.
- **argv (argument vector):** It is an array of strings (char* argv[]) where each element is a pointer to a string representing a command-line argument. **The first element (argv[0]) is the name of the program.**

Example 1:

```
int main(int argc, char *argv[])
{
 printf("%d",argc);
}
```

Output

```
1
```

Explanation:

Default argument is 1

Example 2 :

```
int main(int argc, char *argv[])
{
   int i;
   for(i=0;i<=argc;i++)
   {
     printf(" %d = %s",i,argv[i]);
   }
}
```

Output

```
 0 = C:\Users\Admin\Desktop\Desktop\Nim\Raghul\bin\Debug\Raghul.exe 1 = 1 2 = 2 3
= 3 4 = 4 5 = 5 6 = 6 7 = (null)
```

Swapping two argument

```
int main(int argc,char *argv[])
{
   printf("%d \n",argc);
   for (int i =1 ; i<argc;i++)
   {
      printf("%s ",argv[i]);
   }
   argv[3]=argv[1];
   argv[1]=argv[2];
   argv[2]=argv[3];
   printf("\n");
   for (int i =1 ; i<argc;i++)
   {
      printf("%s ",argv[i]);
   }
}
```

Output

```
3
raghul rasim
rasim raghul
```

String to int conversion

```
int main(int argc,char *argv[])
{
   int i ,j;
   i= atoi(argv[1]);
   j= atoi(argv[2]);
   printf("%d",i+j);
}

output     arg = 3 5
           8
```

4. Union

- Union in C is a **user-defined data type that allows storing different data types in the same memory location.**
- This means that a union variable can hold different types of data at different times during execution.
- Unlike **structures, where each member has its own memory location, all members of a union share the same memory location.**
- This results in a memory-efficient way to handle multiple types of data.

Syntax :

```
#include <stdio.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main() {
    union Data data;
    data.i = 10;
    printf("data.i : %d\n", data.i);
    data.f = 220.5;
    printf("data.f : %f\n", data.f);
    strcpy(data.str, "C Programming");
    printf("data.str : %s\n", data.str);
    return 0;
}
```

Output:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

5. Enumeration

- An enumeration (enum) is a **user-defined data type** used to assign names to integral constants, making the code more readable and maintainable.
- Enumerations provide a way to create a set of named constants that can be used throughout the program.

Syntax :

```
enum enum_name {
    constant1,
    constant2,
    constant3,
    // more constants...
};
```

- **Each constant within the enumeration gets an integer value automatically assigned starting from 0, and subsequent constants are assigned values incremented by 1.**
- **However, you can explicitly assign values to the constants if needed.**

Example 1 :

```
#include <stdio.h>
                    // Defining an enumeration named Color
enum Color {
    RED,        // assigned 0
    GREEN,   // assigned 1
    BLUE       // assigned 2
};
int main() {
    enum Color c = GREEN;
    printf("The value of c is %d\n", c); // Outputs: 1, because GREEN has a value of 1
    return 0;
}
```

Output

```
The value of c is 1
```

Example 2 :

```
#include <stdio.h>
                                        // Define an enumeration named Fruit
enum Fruit {
  APPLE,
  BANANA,
  ORANGE
};
int main() {
  enum Fruit favoriteFruit;            // Declare a variable of type Fruit
  favoriteFruit = BANANA;              // Assign a value to the variable
                      // Check the value of the variable and print a message accordingly
  switch (favoriteFruit) {
    case APPLE:
      printf("Your favorite fruit is apple.\n");
      break;
    case BANANA:
      printf("Your favorite fruit is banana.\n");
      break;
    case ORANGE:
      printf("Your favorite fruit is orange.\n");
      break;
    default:
      printf("You have not chosen a favorite fruit.\n");
      break;
  }

  return 0;
}
```

Output

```
Your favorite fruit is banana.
```

Example 3 :

You can also explicitly assign values to the constants:

```
#include <stdio.h>
// Define an enumeration named Days with explicitly assigned values
enum Days {
    MON = 1,                                        // Monday is assigned 1
    TUE = 2,                                        // Tuesday is assigned 2
    WED = 4,    // Wednesday is assigned 4 (should be 3 for sequential ordering)
    THU,                                            // Thursday is assigned 5
    FRI ,                                           // Friday is assigned 6
    SAT = 9,                                        // Saturday is assigned 9
    SUN                                             // Sunday is assigned 10
};
int main() {
                                                    // Declare a variable of type Days
    enum Days today = TUE;
                                                    // Print the value of today
    printf("Today is day %d\n", today);
    return 0;
}
```

Output

```
Today is day 2
```

Enumerations in C provide a convenient way to represent sets of related constants, making the code more readable and maintainable. They are commonly used in switch statements, as well as in situations where a set of related named constants is required.

6. Bit fields

- Bit fields in C allow the programmer **to specify the size of the field in terms of the number of bits it occupies.**
- They are often used to efficiently pack data structures when memory is at a premium.

Syntax :

```
struct {

      type [member_name] : width;

};
```

Where:

- ❖ type specifies the **data type** of the bit field.
- ❖ [member_name] is an optional **identifier** for the bit field.
- ❖ width specifies the **number of bits to allocate** for the bit field.

Example :

```
#include <stdio.h>
struct student {
   int a:5;
   int b:5;
   int c:5;
} s;
int main() {
   s.a = 10;
   s.b = 11;
   s.c = 15;
   printf("%d", sizeof(s)); // Print the size of the structure
   return 0;
}
```

Output :

```
4
```

- The actual storage size of bit fields depends on the compiler and architecture.

7. Typedef

- In C, ‘typedef’ is a keyword used to **create an alias for a data type.**
- It allows programmers **to define new names for existing data types to improve code readability and maintainability.**
- With ‘typedef’, you can create custom names for data types, **including primitive types, structs, unions, and enums.** It's a powerful feature in C programming for creating custom data type names.

Syntax :

```
typedef existing_data_type new_data_type_name;
```

Example :

```
#include <stdio.h>
typedef int integer;            // Define a typedef for int
int main() {
    integer x = 10;             // Using the typedef alias 'integer' instead of 'int'
    printf("Value of x: %d\n", x);
    return 0;
}
```

Output:

```
Value of x: 10
```

➢ ‘typedef’ becomes more useful when working with complex data types, such as structures or function pointers. For example:

```
#include <stdio.h>
struct Point {                        // Define a structure
    int x;
    int y;
};
                                      // Define a typedef for struct Point
typedef struct Point Point;
int main() {
                          // Using the typedef alias 'Point' instead of 'struct Point'
    Point p1 = {10, 20};
    printf("Coordinates of p1: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

In this example, ‘typedef struct Point Point;’ creates an alias ‘Point’ for the structure ‘struct Point’.

8. Type casting

- Type casting is the process of converting a value from one data type to another.
- This can be useful in situations where you need to perform operations with variables of different types or when you want to store a value in a variable of a different type.

There are two types of type casting in C:

- ✓ Implicit Type Casting
- ✓ Explicit Type Casting

1. Implicit Type Casting:

- ❖ Implicit type casting occurs **automatically by the compiler** when values are assigned or used in expressions.
- ❖ It is also known as **automatic type conversion.**
- ❖ It typically occurs **when converting from a lower precision data type to a higher precision data type, such as from ‘int’ to ‘float’.**

2. Explicit Type Casting:

Explicit type casting is performed explicitly by the programmer using casting operators.

- ❖ It is also known as **manual type conversion**.
- ❖ **It allows the programmer to convert a value from one data type to another explicitly.**

Example:

```
#include <stdio.h>
int main() {
                                    // Implicit type casting (automatic type conversion)
   int num1 = 10;
   float num2 = num1;                    // Implicitly converts int to float
   printf("Implicit Type Casting:\n");
   printf("num1 (int): %d\n", num1);
   printf("num2 (float): %f\n\n", num2);
                                        // Explicit type casting (manual type conversion)
   float num3 = 15.5;
   int num4 = (int) num3;              // Explicitly converts float to int
   printf("Explicit Type Casting:\n");
   printf("num3 (float): %f\n", num3);
```

```
        printf("num4 (int): %d\n", num4);
        return 0;
    }
```

Output:

```
Implicit Type Casting:
num1 (int): 10
num2 (float): 10.000000

Explicit Type Casting:
num3 (float): 15.500000
num4 (int): 15
```

In this example:

- Implicit type casting occurs when assigning the value of ‘num1’ (an ‘int’) to ‘num2’ (a ‘float’).
- Explicit type casting occurs when converting the value of ‘num3’ (a ‘float’) to an ‘int’ using ‘(int)’ before ‘num3’.

9. Variable arguments

- Variable arguments (varargs) allow functions **to accept a variable number of arguments**.
- The standard library provides the **‘<stdarg.h>‘** header to support this feature.
- The key function for handling variable arguments is **‘va_start’, ‘va_arg’, and ‘va_end’**.

1. Function Declaration: Define a function with a variable number of arguments using an ellipsis (‘...’) in the argument list.

2. Initialize the Argument List: Inside the function, use ‘va_list’ to initialize the list of arguments.

3. Accessing Arguments: Use ‘va_arg’ macro to access individual arguments from the list.

4. End the Access: Use ‘va_end’ macro to clean up the argument list.

Example:

```
#include <stdio.h>
#include <stdarg.h>
int sum(int num_args, ...) {
   int total = 0;
   va_list args;
   va_start(args, num_args);                // Initialize the argument list

   for (int i = 0; i < num_args; i++) {
      total += va_arg(args, int);             // Access each argument
   }
   va_end(args);                            // End the argument list
   return total;
}
int main() {
   int result = sum(4, 10, 20, 30, 40);        // Calculate sum of 4 integers
   printf("Sum: %d\n", result);
   return 0;
}
```

Output

```
Sum: 100
```

In this example:

- The ‘sum’ function accepts a variable number of arguments.
- It starts by initializing the ‘va_list’ with ‘va_start’.
- It then uses ‘va_arg’ to access each argument one by one.
- After processing all the arguments, it cleans up the ‘va_list’ with ‘va_end’.

When you call the ‘sum’ function, you need to provide the number of arguments explicitly. In this case, **‘num_args’ is set to ‘4**’. Then, you provide **four integers (‘10’, ‘20’, ‘30’, ‘40’)**. The ‘sum’ function calculates their sum and returns it.

10.Preprocessor

File Inclusion: The #include directive is used to include the contents of header files into the source code.

Syntax:

#include <file_name.h>

When you use angle brackets < >, as in #include <file_name.h>, the preprocessor looks for the specified file in the system directories where header files are typically stored. This is typically used for including standard library header files.

#include " file_name.h "

When you use quotation marks " ", as in #include "filename.h", the preprocessor first looks for the specified file in the current directory. If the file is not found in the current directory, it then searches the system directories. This is typically used for including header files that are part of your project or located in directories specific to your project.

Example:

```
#include <stdio.h>    // Predefined header file
```

- This includes the standard input/output library (stdio.h) from the system libraries.

```
#include "userfile.h" // User-defined file
```

- This includes a user-defined file named userfile.h.
- The preprocessor will first search for this file in the current directory. If it's not found, it will search the system directories.

Macro Expansion: One of the primary tasks of the preprocessor is to expand macros defined using the #define directive. Macros are essentially symbolic constants or short code snippets that are replaced by their respective definitions in the source code.

Syntax:

```
#define MACRO_NAME replacement_text
```

Example:

```
#include<stdio.h>
        #define A 10
        int main()
        {
                printf("%d",A);
                return 0;
        }
Output:  10
```

Macro UnDefinition: The #undef directive is used to undefine macros that were previously defined using #define.

Syntax:

```
#undef MACRO_NAME
```

Example:

```
#include<stdio.h>
#define A 10
#undef A
int main()
{
        printf("%d",A);
        return 0;
}
```

Output: error: 'A' undeclared (first use in this function)

Conditional Compilation: Preprocessor directives such as #if, #ifdef, #ifndef, #elif, #else, and #endif allow for conditional compilation. Portions of code can be included or excluded from the compilation process based on certain conditions.

1. #if

- Allows conditional compilation based on the value of an expression.

Syntax:

```
#if expression
  // code if expression is true
#endif
```

Example:

```
#include<stdio.h>
#define A 20
int main()
{
   #if A==20
   printf("A value is 20");
   #endif // A
}
```

Output

```
A value is 20
```

2. #ifdef

- Checks whether a macro is defined.

Syntax:

```
#ifdef MACRO_NAME
  // code if MACRO_NAME is defined
#endif
```

Example:

```
#include<stdio.h>
#define A 20
int main()
{
   #ifdef A
   printf("A value is %d",A);
   #endif // A
}
```

Output

```
A value is 20
```

3. #ifndef

- Checks whether a macro is not defined.

Syntax:

```
#ifndef MACRO_NAME
  // code if MACRO_NAME is not defined
#endif
```

Example:

```
#include<stdio.h>
#define A 20
int main()
{
   #ifndef B
   printf("B Macro is not define");
   #endif // A
}
```

Output

```
B Macro is not define
```

4. #elif

- Acts as an "else if" within an `#if` block.

Syntax:

```
#if expression1
  // code if expression1 is true
#elif expression2
  // code if expression2 is true
#endif
```

Example:

```
#include<stdio.h>
#define A 30
int main()
{
   #if A==20
   printf("A value is 20");
   #elif A==30
   printf("A value is 30");
   #endif // A
}
```

Output

```
A value is 30
```

5. #else

- Provides an alternative code path when the condition in `#if` is false.

Syntax:

```
#if expression
  // code if expression is true
#else
  // code if expression is false
#endif
```

Example:

```
#include<stdio.h>
#define A 50
int main(){
   #if A==20
   printf("A value is 20");
   #else
   printf("A value is %d",A);
   #endif // A
```

```
                    }                    //Output: A value is 50
```

6. #endif

- Ends the conditional block started by `#if`, `#ifdef`, or `#ifndef`.

Syntax:

```
#if expression
  // code if expression is true
#endif
```

Line Control: The #line directive allows you to control the line numbering reported by the compiler. This can be useful for debugging or generating error messages.

Syntax:

```
#line LINE_NUMBER
```

Example:

```
#include<stdio.h>
int main()
{
   printf("LINE : %d\n",__LINE__);
   #line 20
   printf("LINE : %d\n",__LINE__);
}
```

Output

```
LINE : 4
LINE : 20
```

Error Reporting: The #error directive allows you to generate compiler errors with custom error messages. This can be useful for enforcing certain conditions or requirements in your code.

Syntax

```
#error ERROR_MESSAGE
```

Example

```
#include<stdio.h>
#define a 10
```

```
#define b 0
int main()
{
   #if b==0
      #error B value is 0
   #else
   printf("The division of %d / %d = %d",a,b,a/b);
   #endif
}
```

Output

```
error: #error B value is 0
```

Stringizing and Token Pasting: The # and ## operators can be used to manipulate tokens.

# Operator

The # operator converts a macro argument into a string literal

Example:

```
#include<stdio.h>
#define a(b,c) printf(#b" and "#c" are entry control loop in c\n")
#define d(e) printf( #e " is exit control loop")
int main()
{
   a(for,while);
   d(do-while);
   return 0;
}
```

Output

```
for and while are entry control loop in c
do-while is exit control loop
```

## Operator

The ## operator concatenates tokens.

Example

```
#include<stdio.h>
#define r(g) printf("r"#g "= %d" ,r##g )
#define raghul(b,c) printf( "Raghul M = %d\nRaghul E = %.2f\n",raghul##b,raghul##c)
```

```
int main()
{
   int r14=18;
   int raghulm=100;
   float raghule=99.99;
   raghul(m,e);
   r(14);
   return 0;
}
```

Output

```
Raghul M = 100
Raghul E = 99.99
r14= 18
```

Pragma Directives: The #pragma directive provides additional instructions to the compiler. These directives are compiler-specific and can be used for various purposes such as turning off warnings, specifying optimization options, etc.

11.Header file

- Header files are files that contain function prototypes, macro definitions, and other declarations that are shared across multiple source code files.
- **Extension :** ".h"
- Included at the beginning of C source files using the `#include` preprocessor directive.

Header files can generally be categorized into two main types based on their source and purpose:

- System Header Files
- User-Defined Header Files

1**. System Header Files**: These are header files that come with the C compiler or the standard C library. They provide declarations for standard functions, macros, and types defined by the C language or the operating system.

- stdio.h
- stdlib.h
- math.h
- string.h
- ctype.h

#include <stdio.h>

printf():

- **Function:** Prints formatted output to the standard output stream (usually the console).
- **Syntax:** int printf(const char *format, ...);

scanf():

- **Function:** Reads formatted input from the standard input stream (usually the keyboard).
- **Syntax:** int scanf(const char *format, ...);

getchar():

- **Function:** Reads a single character from the standard input stream.
- **Syntax:** int getchar(void);

putchar():

- **Function:** Writes a single character to the standard output stream.
- **Syntax:** int putchar(int character);

fgets():

- **Function:** Reads a line of text from the standard input stream and stores it as a string.
- **Syntax:** char *fgets(char *str, int n, FILE *stream);

fputs():

- **Function:** Writes a string to the specified output stream.
- **Syntax:** int fputs(const char *str, FILE *stream);

fprintf():

- **Function:** Writes formatted output to the specified output stream.
- **Syntax:** int fprintf(FILE *stream, const char *format, ...);

fscanf():

- **Function:** Reads formatted input from the specified input stream.
- **Syntax:** int fscanf(FILE *stream, const char *format, ...);

fclose():

- **Function:** Closes the specified file stream.
- **Syntax:** int fclose(FILE *stream);

fopen():

- **Function:** Opens a file and associates it with a file stream.
- **Syntax:** FILE *fopen(const char *filename, const char *mode);

#include<stdlib.h>

- The `stdlib.h` header file in C contains declarations for a variety of functions related to memory allocation, random number generation, string conversion, and other utility functions.

malloc():

- **Function:** Allocates a block of memory dynamically.
- **Syntax:** `void *malloc(size_t size);`

calloc():

- **Function:** Allocates memory for an array of elements, initialized to zero.
- **Syntax:** `void *calloc(size_t num, size_t size);`

realloc():

- **Function:** Reallocates memory for an existing block.
- **Syntax:** `void *realloc(void *ptr, size_t size);`

free():

- **Function:** Deallocates memory previously allocated by `malloc`, `calloc`, or `realloc`.
- **Syntax:** `void free(void *ptr);`

exit():

- **Function:** Terminates the program immediately with an optional status code.
- **Syntax:** `void exit(int status);`

rand():

- **Function:** Generates a pseudo-random integer between 0 and `RAND_MAX`.
- **Syntax:** `int rand(void);`

srand():

- **Function:** This line seeds the random number generator with the current time, ensuring that each time the program is run, a different sequence of random numbers will be generated.
- **Syntax:** `void srand(unsigned int seed);`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    srand((unsigned int)time(NULL));
    printf("5 random numbers:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d\n", rand());
    }
    return 0;
}
```

atoi():

- **Function:** Converts a string to an integer.
- **Syntax:** `int atoi(const char *str);`

atof():

- **Function:** Converts a string to a floating-point number.
- **Syntax:** `double atof(const char *str);`

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char str[] = "3.14";
    double num;
    num = atof(str);
    printf("The converted number is: %.2f\n", num);
    return 0;
}
```

Output:

```
The converted number is: 3.14
```

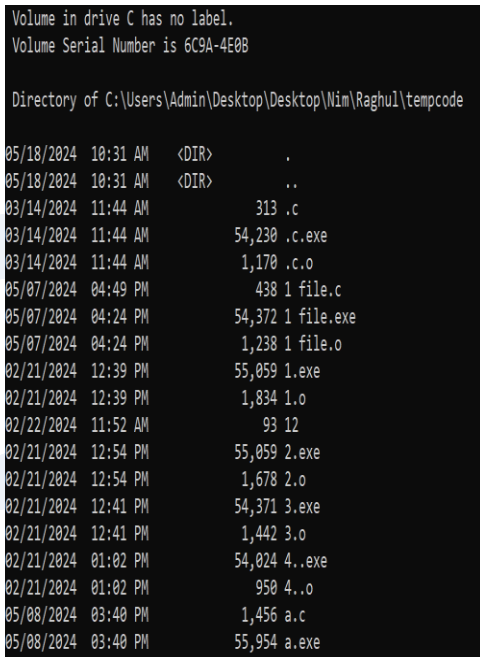
system():

- **Function:** Executes a command in the system's command processor.
- **Syntax:** `int system(const char *command);`

Example 1:

- This line invokes the system's command processor (cmd.exe on Windows) to execute the command "dir", which **lists the files and directories in the current directory.**

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int status = system("dir");
    if (status == 0) {
        printf("Command executed successfully.\n");
    } else {
        printf("Command failed to execute.\n");
    }
    return 0;
}
```

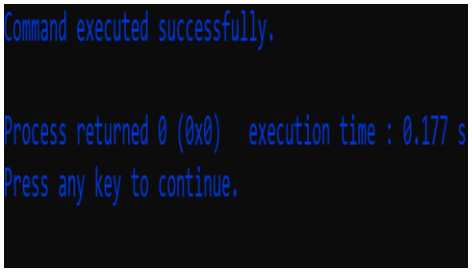


```
 Volume in drive C has no label.
 Volume Serial Number is 6C9A-4E0B

 Directory of C:\Users\Admin\Desktop\Desktop\Nim\Raghul\tempcode

05/18/2024  10:31 AM    <DIR>          .
05/18/2024  10:31 AM    <DIR>          ..
03/14/2024  11:44 AM               313 .c
03/14/2024  11:44 AM            54,230 .c.exe
03/14/2024  11:44 AM             1,170 .c.o
05/07/2024  04:49 PM               438 1 file.c
05/07/2024  04:24 PM            54,372 1 file.exe
05/07/2024  04:24 PM             1,238 1 file.o
02/21/2024  12:39 PM            55,059 1.exe
02/21/2024  12:39 PM             1,834 1.o
02/22/2024  11:52 AM                93 12
02/21/2024  12:54 PM            55,059 2.exe
02/21/2024  12:54 PM             1,678 2.o
02/21/2024  12:41 PM            54,371 3.exe
02/21/2024  12:41 PM             1,442 3.o
02/21/2024  01:02 PM            54,024 4..exe
02/21/2024  01:02 PM               950 4..o
05/08/2024  03:40 PM             1,456 a.c
05/08/2024  03:40 PM            55,954 a.exe
```

Example 2:

- The command being executed is color 1, which sets the text and background color of the console to bright blue (foreground) and black (background).

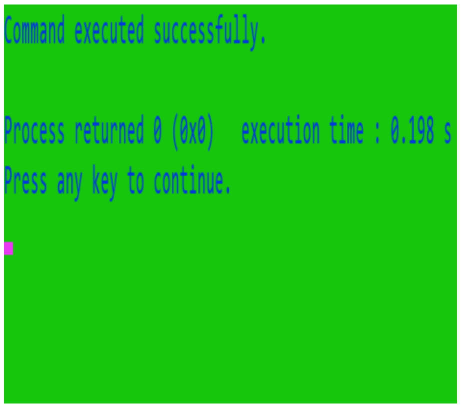
```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int status = system("color 1");
    if (status == 0) {
        printf("Command executed successfully.\n");
    } else {
        printf("Command failed to execute.\n");
    }
    return 0;
}
```



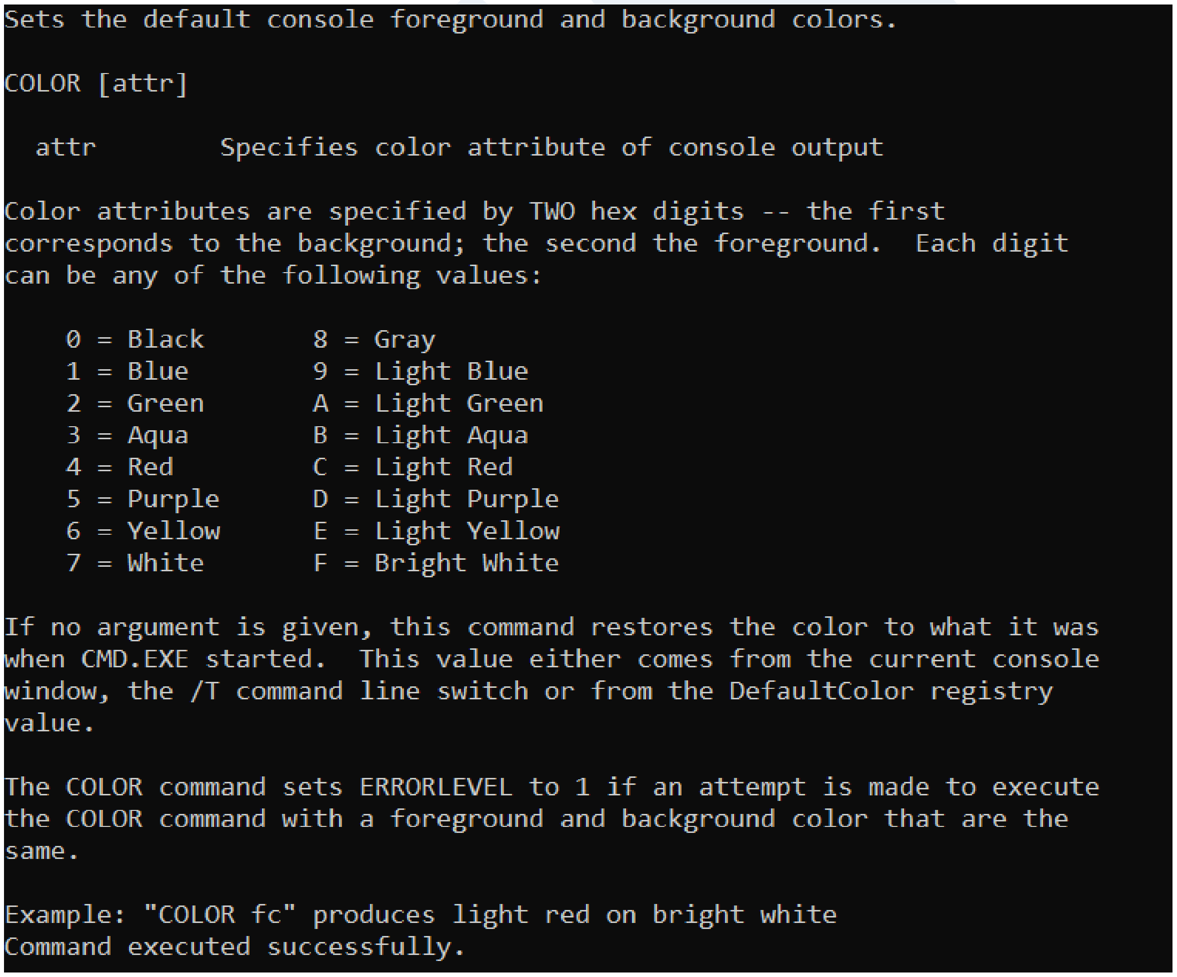
```
Command executed successfully.

Process returned 0 (0x0)   execution time : 0.177 s
Press any key to continue.
```

Example 3:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int status = system("color A1");
   if (status == 0) {
      printf("Command executed successfully.\n");
   } else {
      printf("Command failed to execute.\n");
   }
   return 0;
}
```



```
Command executed successfully.


Process returned 0 (0x0)   execution time : 0.198 s
Press any key to continue.
```



```
Sets the default console foreground and background colors.

COLOR [attr]

  attr        Specifies color attribute of console output

Color attributes are specified by TWO hex digits -- the first
corresponds to the background; the second the foreground.  Each digit
can be any of the following values:

    0 = Black       8 = Gray
    1 = Blue        9 = Light Blue
    2 = Green       A = Light Green
    3 = Aqua        B = Light Aqua
    4 = Red         C = Light Red
    5 = Purple      D = Light Purple
    6 = Yellow      E = Light Yellow
    7 = White       F = Bright White

If no argument is given, this command restores the color to what it was
when CMD.EXE started.  This value either comes from the current console
window, the /T command line switch or from the DefaultColor registry
value.

The COLOR command sets ERRORLEVEL to 1 if an attempt is made to execute
the COLOR command with a foreground and background color that are the
same.

Example: "COLOR fc" produces light red on bright white
Command executed successfully.
```

#include<math.h>

pow():

- **Definition:** Calculates the value of a base raised to the power of an exponent.
- **Syntax:** `double pow(double base, double exponent);

Example:

```
#include <stdio.h>
#include <math.h>
int main() {
    double base = 2.0, exponent = 3.0;
    double result_pow = pow(base, exponent);
    printf("pow(%lf, %lf) = %lf\n", base, exponent, result_pow);
    return 0;
}
```

Output:

```
pow(2.000000, 3.000000) = 8.000000
```

sqrt():

- **Definition:** Computes the square root of a given number.
- **Syntax:** `double sqrt(double x);

Example:

```
#include <stdio.h>
#include <math.h>
int main() {
    double number = 25.0;
    double result_sqrt = sqrt(number);
    printf("sqrt(%lf) = %lf\n", number, result_sqrt);
    return 0;
}
```

Output:

```
sqrt(25.000000) = 5.000000
```

fabs():

- **Definition:** Computes the absolute value of a floating-point number.
- **Syntax:** `double fabs(double x);

Example:

```
#include <stdio.h>
#include <math.h>
int main() {
    double value = -10.5;
    double result_fabs = fabs(value);
    printf("fabs(%lf) = %lf\n", value, result_fabs);
    return 0;
}
```

Output:

```
fabs(-10.500000) = 10.500000
```

ceil():

- **Definition:** Rounds a floating-point number up to the nearest integer greater than or equal to it.
- **Syntax:** `double ceil(double x);

Example:

```
#include <stdio.h>
#include <math.h>
int main() {
    double x = 3.14;
    double result_ceil = ceil(x);
    printf("ceil(%lf) = %lf\n", x, result_ceil);
    return 0;
}
```

Output:

```
ceil(3.140000) = 4.000000
```

floor():

- **Definition:** Rounds a floating-point number down to the nearest integer less than or equal to it.
- **Syntax:** `double floor(double x);`

Example:

```
#include <stdio.h>
#include <math.h>
int main() {
   double y = 3.14;
   double result_floor = floor(y);
   printf("floor(%lf) = %lf\n", y, result_floor);
   return 0;
}
```

Output:

```
floor(3.140000) = 3.000000
```

#include<ctype.h>

1. isalnum():

- **Definition:** Checks if a character is alphanumeric (a letter or a digit).
- **Syntax:** int isalnum(int ch);
- **Parameter:** `ch`: The character to be checked.
- **Return Value:** Returns non-zero if the character is alphanumeric, otherwise returns zero.

Example code:

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char ch = 'A';
    if (isalnum(ch)) {
        printf("%c is alphanumeric.\n", ch);
    } else {
        printf("%c is not alphanumeric.\n", ch);
    }
    return 0;
}
```

2. isalpha ():

- **Definition:** Checks if a character is alphabetic (a letter).
- **Syntax:** `int isalpha(int ch);`
- **Parameter:** `ch`: The character to be checked.
- **Return Value:** Returns non-zero if the character is alphabetic, otherwise returns zero.

Example code:

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char ch = 'A';
    if (isalpha(ch)) {
        printf("%c is alphabetic.\n", ch);
    } else {
        printf("%c is not alphabetic.\n", ch);
    }
    return 0;
}
```

3. isdigit():

- **Definition:** Checks if a character is a digit.
- **Syntax:** `int isdigit(int ch);`
- **Parameter:** `ch`: The character to be checked.
- **Return Value:** Returns non-zero if the character is a digit, otherwise returns zero.

Example code:

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char ch = '5';
    if (isdigit(ch)) {
        printf("%c is a digit.\n", ch);
    } else {
        printf("%c is not a digit.\n", ch);
    }
    return 0;
}
```

4. **islower():**

  - **Definition:** Checks if a character is a lowercase letter.
  - **Syntax:** `int islower(int ch);`
  - **Parameter:** `ch`: The character to be checked.
  - **Return Value:** Returns non-zero if the character is a lowercase letter, otherwise returns zero.

Example code:

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

int main() {
    char ch = 'a';
    if (islower(ch)) {
        printf("%c is a lowercase letter.\n", ch);
    } else {
        printf("%c is not a lowercase letter.\n", ch);
    }
    return 0;
}
```

5. **isupper() :**

  - **Definition:** Checks if a character is an uppercase letter.
  - **Syntax:** `int isupper(int ch);`
  - **Parameter:** `ch`: The character to be checked.
  - **Return Value:** Returns non-zero if the character is an uppercase letter, otherwise returns zero.

Example code:

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char ch = 'A';
    if (isupper(ch)) {
        printf("%c is an uppercase letter.\n", ch);
    } else {
        printf("%c is not an uppercase letter.\n", ch);
    }
    return 0;
}
```

6. **tolower():**
   - **Definition:** Converts an uppercase letter to its corresponding lowercase letter.
   - **Syntax:** `int tolower(int ch);`
   - **Parameter:** `ch`: The character to be converted.
   - **Return Value:** Returns the lowercase representation of the character if it is an uppercase letter, otherwise returns the character unchanged.

Example code:

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char ch = 'B';
    char lowercase_ch = tolower(ch);
    printf("Uppercase character: %c\n", ch);
    printf("Lowercase equivalent: %c\n", lowercase_ch);
    return 0;
}
```

7. **toupper():**
   - **Definition:** Converts an lowercase letter to its corresponding uppercase letter.
   - **Syntax:** `int toupper(int ch);`
   - **Parameter:** `ch`: The character to be converted.
   - **Return Value:** Returns the uppercase representation of the character if it is an lowercase letter, otherwise returns the character unchanged.

Example code:

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char ch = 'b';
    char uppercase_ch = toupper(ch);
    printf("Lowercase character: %c\n", ch);
    printf("Uppercase equivalent: %c\n", uppercase_ch);
    return 0;
}
```

System Header Files:

HEADER FILES	DESCRIPTIONS
<assert.h>	Conditionally compiled macro that compares its argument to zero
<complex.h> (since C99)	Complex number arithmetic
<ctype.h>	Functions to determine the type contained in character data
<errno.h>	Macros reporting error conditions
<fenv.h> (since C99)	Floating-point environment
<float.h>	Limits of floating-point types
<inttypes.h> (since C99)	Format conversion of integer types
<iso646.h> (since C95)	Alternative operator spellings
<limits.h>	Ranges of integer types
<locale.h>	Localization utilities
<math.h>	Common mathematics functions
<setjmp.h>	Nonlocal jumps
<signal.h>	Signal handling
<stdalign.h> (since C11)	alignas and alignof convenience macros
<stdarg.h>	Variable arguments
<stdatomic.h> (since C11)	Atomic operations
<stdbit.h> (since C23)	Macros to work with the byte and bit representations of types
<stdbool.h> (since C99)	Macros for boolean type
<stdckdint.h> (since C23)	macros for performing checked integer arithmetic
<stddef.h>	Common macro definitions
<stdint.h> (since C99)	Fixed-width integer types
<stdio.h>	Input/output
<stdlib.h>	General utilities: memory management, program utilities, string conversions, random numbers, algorithms
<stdnoreturn.h> (since C11)	noreturn convenience macro
<string.h>	String handling
<tgmath.h> (since C99)	Type-generic math (macros wrapping math.h and complex.h)
<threads.h> (since C11)	Thread library
<time.h>	Time/date utilities
<uchar.h> (since C11)	UTF-16 and UTF-32 character utilities
<wchar.h> (since C95)	Extended multibyte and wide character utilities
<wctype.h> (since C95)	Functions to determine the type contained in wide character data

2. **User-Defined Header Files:** These are **header files created by programmers to organize their code**, declare functions, macros, and types that are used across multiple source files in a project. User-defined header files typically have a `.h` extension.

User-defined header files can be created for various purposes, such as:

- Declaring function prototypes and global variables that are shared across multiple source files.
- Defining constants and macros that are used throughout the program.
- Creating custom data structures and type definitions.
- Including necessary system header files and providing additional functionality specific to the project.

In C programming, there are two ways to include header files: using angle brackets `<>` and using double quotes `""`. The difference lies in how the compiler searches for the header file.

1. Angle Brackets `< >`:

- ❖ When you include a header file using angle brackets (`<>`), the compiler searches for the file in the system directories where standard header files are located.
- ❖ Typically, standard library header files like `<stdio.h>`, `<stdlib.h>`, `<math.h>`, etc., are included using angle brackets.

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

2. Double Quotes `" "` :

- ❖ When you include a header file using double quotes (`""`), the compiler searches for the file in the current directory first, and if not found, it searches in the system directories.
- ❖ User-defined header files and header files specific to your project are typically included using double quotes.

```
#include "iies.h"
```