

ASSIGNMENT-4

1. Explain about call by value and call by reference with suitable examples.

Call by value and call by reference are two different ways in which a function can receive arguments in C.  
Call by value: In this method, the function receives a copy of the argument's value. This means that any changes made to the argument within the function have no effect on the original value outside of the function.

Example:

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

```
int main()
{
    int a = 5;
    increment(a);
    printf("%d", a);
}
```

Output:

5.

Call by reference: In this method, the function receives a pointer to the argument. This means that any changes made to the argument within the function will affect the original value outside of the function.

Example:

```
void increment(int *x)
{
    (*x)++;
}
int main()
{
}
```

```
int a=5;
increment(&a);
printf("%d", a);
}
```

Output:

6

2. Write a C program for Multiplication of 2 Matrices.

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

```
int main()
```

```
{
```

```
int a[10][10], b[10][10], c[10][10], i, j, k, m, n, p, q;
```

```
printf("Enter no. of rows & columns of Matrix A: ");
```

```
scanf("%d %d", &m, &n);
```

```
printf("Enter elements of matrix A: ");
```

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

```
{
```

```
for (j=0; j<n; j++)
```

```
{
```

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

```
}
```

```
}
```

```
printf("Enter no. of rows & columns of Matrix B: ");
```

```
scanf("%d %d", &p, &q);
```

```
printf("Enter elements of Matrix B: ");
```

```
scanf
```

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

```
{
```

```
for (j=0; j<q; j++)
```

```
{
```

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

```
}
```

```
}
```



```

if (n != p)
{
    printf("No. of columns in Matrix A must be equal to No. of rows
    in Matrix B\n");
    return 0;
}

```

```

for (i = 0; i < m; i++)
{
    for (j = 0; j < q; j++)
    {
        c[i][j] = 0;
        for (k = 0; k < n; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}

```

```

printf("Product of given 2 Matrices: \n");
for (i = 0; i < m; i++)
{
    for (j = 0; j < q; j++)
    {
        printf("%d", c[i][j]);
    }
    printf("\n");
}
return 0;
}

```

3. Write a C program to implement Fibonacci series using recursion.

```

#include <stdio.h>
int fibonacci(int n)
{

```

```

if (n <= 0)
    return 0;
if (n == 1)
    return 1;
else
    return fibonacci(n-1) + fibonacci(n-2);
}

int main()
{
    int n, i;
    printf("Enter no. of terms: ");
    scanf("%d", &n);
    printf("Fibonacci series: ");
    for (i = 0; i < n; i++)
        printf("%d ", fibonacci(i));
    return 0;
}

```

4. Explain about string handling functions.

C provides a set of standard library functions for handling strings, which are defined in the `string.h` header file. Some of the commonly used string handling functions in C include:

- `strlen()`: This function is used to find the length of a given string.
- `strcpy()`: This function is used to copy one string to another.
- `strcat()`: This function is used to concatenate two strings.
- `strcmp()`: This function is used to compare 2 strings. It returns 0 if the strings are equal, a negative value if the first string is lexicographically less than second string, and a positive value if the first string is lexicographically greater than the second string.
- `strchr()`: This function is used to search for the first occurrence of a given character in a string.
- `strstr()`: This function is used to search for the first occurrence of a given substring in a string.



There are several other string handling functions inc, such as `strncpy()`, `strncat()`, `strncpy()`, etc. These functions work similarly to the functions mentioned above, but they accept an additional argument specifying the maximum no. of characters to be used.

5. Write a C program to sort the given set of strings.

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

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

```
#define MAX_STRINGS 10
```

```
#define MAX_LENGTH 50
```

```
void sortStrings (char strings[][MAX_LENGTH], int n)
```

```
{
```

```
    char temp[MAX_LENGTH];
```

```
    for (int i=0; i<n-1; i++)
```

```
    {
```

```
        for (int j=0; j<n; j++)
```

```
        {
```

```
            if (strcmp(strings[i], strings[j]) > 0)
```

```
            {
```

```
                strcpy(temp, strings[i]);
```

```
                strcpy(strings[i], strings[j]);
```

```
                strcpy(strings[j], temp);
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

```
int main()
```

```
{
```

```
    char strings[MAX_STRINGS][MAX_LENGTH];
```

```
    int n;
```

```
    printf("Enter the no. of strings: ");
```

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



```

printf("Enter %d strings: \n", n);
for (int i = 0; i < n; i++)
{
    scanf("%s", strings[i]);
}
sortstrings(strings, n);
printf("Sorted strings: \n");
for (int i = 0; i < n; i++)
{
    printf("%s\n", strings[i]);
}
return 0;
}

```

6. What do you mean by a function? Give the structure of user defined function and explain about the arguments & return values.

In programming, a function is a block of code that performs a specific task. The structure of a user defined function in C language typically includes the following elements:

1. The function declaration, which includes the return type, function name, and the list of parameters (if any) enclosed in parentheses.
2. The function body, which contains the statements that are executed when the function is called.

For example, the following is a simple C function that takes 2 integer arguments and returns the sum of two numbers.

```

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

```



**Arguments:** In the above example, the variables 'a' & 'b' are the arguments passed to the function. They are used to pass data into the function.

**Return Values:** In the given example, the variable 'c' is the return value of the function. It is used to return a value back to the calling code. The return statement is used to return the value of the variable 'c' to the calling code.

When the function is called, the values passed as arguments are used to perform the operations defined in the function, and the return value is <sup>to</sup> pass the results back to the calling code.

7. write a program to read, calculate average and print student marks using array of structures.

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

```
struct student
```

```
{
```

```
    int roll-no;
```

```
    char name[20];
```

```
    float marks[3];
```

```
    float average;
```

```
};
```

```
int main()
```

```
{
```

```
    int i, j, n;
```

```
    struct student s[10];
```

```
    printf("Enter the no. of students: ");
```

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

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

```
    {
```

```
        printf("Enter details for student %d : \n", i+1);
```

```
        printf("Roll number: ");
```

```
        scanf("%d", &s[i].roll-no);
```

```
        printf("Name: ");
```

```
        scanf("%s", s[i].name);
```

```

for (j = 0; j < 3; j++)
{
    printf("Marks in subject %d: ", j+1)
    printf
    scanf("%f", &S[i].marks[j]);
}
}

for (i = 0; i < n; i++)
{
    float sum = 0;
    for (j = 0; j < 3; j++)
    {
        sum += S[i].marks[j];
    }
    S[i].average = sum/3;
}

printf("\n student details: \n");
for (i = 0; i < n; i++)
{
    printf("Roll Number: %d \n", S[i].roll-no);
    printf("Name: %s \n", S[i].name);
    printf("Average Marks: %.2f \n", S[i].average);
}

return 0;
}

```

8. Differentiate between self-referential structure and nested structure with example.

In C Programming, a self-referential structure is a structure that contains a pointer to an instance of the same structure type. It is used to create linked data structures, such as linked lists and trees. For example:

```

eg:
struct node
{
    int data;

```



```
struct node * next;  
};
```

In this example, the 'node' structure contains an integer "data" and a pointer "next" to another instance of the "node" structure. This allows us to create a linked list where each node points to the next node in the list.

On the other hand, a nested structure is a structure that contains another structure as a member. It is used to group related data together and to create more complex data structures.

For example:

```
struct address{  
    char street[20];  
    char city[20];  
    char state[20];  
};
```

```
struct employee{  
    int id;  
    char name[20];  
    struct address addr;  
};
```

In this example, the "address" structure contains three character arrays for this street, city and state and the "employee" structure contains an integer 'id', a character array 'name' and a nested address structure 'addr'. This allows us to group the address details of an employee in a separate structure.

In summary, a self-referential structure is a structure that contains a pointer to an instance of the same structure type, while a nested structure is a structure that contains another structure as a member.



9. Explain three dynamic memory allocation functions with suitable examples.

In C Programming, dynamic memory allocation refers to the process of allocating memory at runtime, as opposed to compile-time. There are several functions available in the C Standard library for allocating dynamic memory, including:

1. `malloc()`: This function is used to allocate a block of memory of a specified size. It takes one argument, which is the size of the memory block in bytes. It returns a pointer to the first byte of the allocated memory block. If the memory allocation is successful, the pointer returned by `malloc()` points to the first byte of the allocated memory block, otherwise it returns a null pointer.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i;
    int *p;
    printf("Enter No. of elements: ");
    scanf("%d", &n);
    p = (int*) malloc(n * sizeof(int));
    if (p == NULL)
    {
        printf("Memory allocation failed\n");
        return 1;
    }
    for (i = 0; i < n; i++)
    {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &p[i]);
    }
```



```
printf("Entered elements are :");
```

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

```
{
```

```
    printf("%d", P[i]);
```

```
}
```

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

```
free(P);
```

```
return 0;
```

```
}
```

2. **calloc()**: This function is used to allocate a block of memory for an array of a specified number of elements, each of a specified size. It takes two arguments, the first argument is the number of elements in the array and the second argument is the size of each element in bytes. It returns a pointer to the first byte of the allocated memory block. If the memory allocation is successful, the pointer returned by `calloc()` points to the first byte of the allocated memory block, otherwise it returns a null pointer.

Example:

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

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

```
int main()
```

```
{
```

```
    int n, i;
```

```
    int *p;
```

```
    printf("Enter No. of elements: ");
```

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

```
    P=(int *)calloc(n, sizeof(int));
```

```
    if (P==NULL)
```

```
{
```

```
        printf("Memory allocation failed.\n");
```

```
        return 1;
```

```
}
```



```

for (i = 0; i < n; i++)
{
    printf("Enter element %d : ", i+1);
    scanf("%d", &P[i]);
}
printf("Entered elements are: ");
for (i = 0; i < n; i++)
{
    printf("%d", P[i]);
}
printf("\n");

free(P);
return 0;
}

```

3. realloc(): This function is used to change the size of previously allocated memory block. It takes two arguments, the first argument is a pointer to the previously allocated memory block, and the second argument is the new size of the memory block in bytes. It returns a pointer to the first byte of the re-allocated memory block. If the memory reallocation is successful, the pointer returned by realloc() points to the first byte of the re-allocated memory block.

Example:

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, new_n;
    int *p;
    printf("Enter the No. of elements: ");
    scanf("%d", &n);
    p = (int *) malloc(n * sizeof(int));
}

```



```
if (P == NULL)
```

```
{
```

```
    printf("Memory allocation failed.\n");
```

```
    return 1;
```

```
}
```

```
for (i = 0; i < n; i++) {
```

```
{
```

```
    printf("Enter element %d: ", i+1);
```

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

```
}
```

```
printf("Entered elements are: ");
```

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

```
{
```

```
    printf("%d", P[i]);
```

```
}
```

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

```
printf("Enter the new number of elements: ");
```

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

```
P = (int *) realloc(P, new_n * sizeof(int));
```

```
if (P == NULL)
```

```
{
```

```
    printf("Memory allocation failed.\n");
```

```
    return 1;
```

```
}
```

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

```
{
```

```
    printf("Enter new element %d: ", i+1);
```

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

```
}
```

```
printf("All elements are: ");
```

```
for (i = 0; i < new_n; i++) {
```

```
    printf("%d", P[i]);
```

```
}
```

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

```
free(P);
```



```
return 0;  
}
```

## 10. Explain about storage classes.

In C Programming, a storage class is a way to specify the duration and visibility of variable or function. There are 4 storage classes in C.

1. **Automatic**: These are local variables that are defined inside a function. They are also called "local variables" or "automatic variables". They are automatically created when the function is called and automatically destroyed when the function returns. They do not retain their value between function calls. They are the default storage class for local variables, if no storage class is specified.

Example:

```
void func() {  
    int x;  
    x = 5;  
    printf("i.d", x);  
}
```

2. **Register**: These are local variables that are stored in a register instead of memory. Using a register storage class can improve the performance of the program by reducing memory access time. However, the number of registers is limited, so not all variables can be stored in registers.

Example:

```
void func() {  
    register int x;  
    x = 5;  
    printf("i.d", x);  
}
```

3. **Static**: These are variables that retain their value b/w function calls. They are also used to create variables that are only visible within a specific file, rather than being



Visible throughout the entire Program. A Variable defined as Static inside a function maintains its Value between function calls.

Example:

```
void func() {  
    static int x = 0;  
    x++;  
    printf("%d", x);  
}
```

4. Extern: These are variables that are defined in one file and can be accessed in another file. They are used to share Variables between different files (or) modules in a program. An extern Variable can be defined in one source file & used in another source file.

Example:

```
//file1.c  
int x;  
x = 5;  
  
//file2.c  
extern int x;  
printf("%d", x);
```

In Summary, storage class in C specify the duration and visibility of a Variable (or) function in C has 4 types of storage classes: Automatic, Register, Static, Extern. They are used to control the lifetime and scope of Variables and functions.

11. Develop a Programme to create a library catalogue with the following members: access number, authors, name, title of book, Year of Publication and book price using structures.

```
#include <stdio.h>  
#include <string.h>  
#define MAX_BOOKS 10
```

```
struct book {  
    int access_no;  
    char author[50];  
    char title[100];
```



```

int Year;
float Price;
};

int main() {
    Struct book library[MAX_BOOKS];
    int i, n;
    Printf("Enter the no. of books: ");
    scanf("%d", &n);
    for (i=0; i<n; i++) {
        Printf("Enter details for book %d: \n", i+1);
        Printf("Access Number: ");
        scanf("%d", &library[i].access_no);
        Printf("Author: ");
        scanf("%s", library[i].author);
        Printf("title: ");
        scanf("%s", library[i].title);
        Printf("Year of Publication: ");
        scanf("%d", &library[i].year);
        Printf("Price: ");
        scanf("%f", &library[i].Price);
    }
    Printf("\n Library Catalogue: \n");
    for (i=0; i<n; i++) {
        Printf("Access Number: %d \n", library[i].access_no);
        Printf("Author: %s \n", library[i].author);
        Printf("Title: %s \n", library[i].title);
        Printf("Year of Publication: %d \n", library[i].year);
        Printf("Price: %.2f \n", library[i].Price);
    }
    return 0;
}

```



12. Explain about command line arguments with an example.

In C programming, command line arguments are Parameters Passed to a Program when it is executed from the command line. These arguments can be used to provide input to the Program or to specify options for how the program should run.

Command line arguments are Passed to the main function of a C program and are received by the program in form of an array of strings. The first element of this array (`argv[0]`) contains the name of the program, and the remaining elements contain the arguments Passed to the program. The number of arguments Passed to the program is Passed as the second argument (`argc`) to the main function.

Here is an example of a Program that takes 2 command line arguments, an input file name & an output file name, & copies the contents of the input file to the output file.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *in, *out;
    int c;
    if (argc != 3) {
        printf("Usage: %s input_file output_file\n", argv[0]);
        return 1;
    }
    in = fopen(argv[1], "r");
    if (in == NULL) {
        printf("Error: Unable to open input file %s\n", argv[1]);
        return 1;
    }
    out = fopen(argv[2], "w");
    if (out == NULL) {
        printf("Error: Unable to open output file %s\n", argv[2]);
    }
}
```



```

fclose(in);
return 1;
}
while((c = fgetc(in)) != EOF)
    fputc

```

Other Example:

```

#include <stdio.h>
int main(int argc, char *argv[]) {
    if (argc == 2) {
        printf("The argument passed is : %s\n", argv[1]);
    } else {
        printf("No argument passed.\n");
    }
    return 0;
}

```

13. What is a pointer? Explain pointer arithmetic operations with suitable examples.

A Pointer is a variable that stores the memory address of another variable. Pointers are useful for many tasks in C, including dynamic memory allocation, function pointers and passing arguments to functions by reference.

Pointer arithmetic is the manipulation of pointers to perform various operations like addition, subtraction, increment, decrement on pointers.

In C, pointer arithmetic is performed in the following way:

- 'ptr++': Increments the pointer to point to the next element of the same type.
- 'ptr--': Decrements the pointer to point to the previous element of the same type.
- 'ptr+n': Adds n to the pointer's memory address, moving it n elements ahead of the same type.
- 'ptr-n': Subtracts n from the pointer's memory address,



moving it  $n$  elements back of the same type.

Example:

```
#include <stdio.h>

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr;
    printf("Value of first element: %d\n", *ptr);
    ptr++;
    printf("Value of second element: %d\n", *ptr);
    ptr = ptr + 2;
    printf("Value of third element: %d\n", *ptr);
    ptr = ptr - 1;
    printf("Value of fourth element: %d\n", *ptr);
    return 0;
}
```

Output:

Value of first element: 1  
Value of second element: 2  
Value of third element: 4  
Value of fourth element: 3

14. What is a file? Explain different modes of operating a file.

In C, a file is a collection of data <sup>age</sup> stored on a storage device, such as a hard drive or flash drive. Files can be created, modified, and deleted by the operating system and can be used to store various types of information, such as text, images, videos and audio.

In C programming, files are accessed using the 'FILE' pointer. The 'fopen()' function is used to open a file and returns a pointer to a 'FILE' structure, which can be used to access the file.



The 'fopen()' function takes two parameters: the name of the file and the mode in which the file should be opened. The mode in which a file is opened determines what operations can be performed on the file. The different modes of opening a file in C are:

- "r": opens a text file for reading.
- "w": opens a text file for writing. If the file already exists, its contents will be truncated.
- "a": opens a text file for writing. The file is created if it does not exist. The file is opened in append mode.
- "rb": opens a binary file for reading.
- "wb": opens a binary file for writing. If the file already exists, its contents will be truncated.
- "ab": opens a binary file for writing. The file is created if it does not exist. The file is opened in append mode.

Example of Opening a file in C;

```
#include <stdio.h>
int main()
{
    FILE *fp;
    char ch;
    fp = fopen("example.txt", "r");
    if (fp == NULL)
    {
        printf("Error opening file. \n");
        return 1;
    }
    while ((ch = fgetc(fp)) != EOF) {
        printf("%c", ch);
    }
    fclose(fp);
    return 0;
}
```



15. Write a programme to demonstrate read and write operations on a file.

```
#include <stdio.h>
int main()
{
    FILE *fp; // FILE Pointer
    fp = fopen("example.txt", "w"); // Open file in writing mode
    fprintf(fp, "writing to a file in c"); // write to file
    fclose(fp); // Close the file
    fp = fopen("example.txt", "r"); // Open file in read mode
    char ch;
    while((ch = fgetc(fp)) != EOF)
    {
        printf("%c", ch);
    }
    fclose(fp); // Close the file
    return 0;
}
```

16. Explain about fscanf(), fgetc(), fprintf() and fwrite() functions with suitable examples.

'fscanf()': This function is used to read formatted input from a file. It works similarly to the scanf() function, but it takes an additional file pointer as the first argument. For example, the following code reads an integer, a string, and a float from a file called "data.txt":

```
FILE *fp;
int i;
char str[100];
float f;
fp = fopen("data.txt", "r");
```

```
fscanf(fp, "%d %s %f", &i, str, &f);  
printf("Read: %d %s %f", i, str, f);  
fclose(fp);
```

'fgetc()': This function is used to read a line of text from a file. It takes a file pointer, a buffer to store the read text, and the maximum number of characters to read as arguments. For example, the following code reads a line of text from a file called "data.txt" and prints it to the console.

```
FILE *fp;  
char line[100];  
fp = fopen("data.txt", "r");  
fgets(line, sizeof(line), fp);  
printf("Read: %s", line);  
fclose(fp);
```

'fprintf()': This function is used to write formatted output to a file. It works similarly to the printf() function, but it takes an additional file pointer as the first argument. For example, the following code writes an integer, a string, and a float to a file called "data.txt".

```
FILE *fp;  
int i = 42;  
char str[] = "Hello World";  
float f = 3.14;  
fp = fopen("data.txt", "w");  
fprintf(fp, "%d %s %f", i, str, f);  
fclose(fp);
```

'fwrite()': This function is used to write binary data to a file. It takes a pointer to the data, the size of each element, the number of elements, and a file pointer as arguments. For example, the following code writes an array of integers to a file called "data.bin":



```
FILE *fp;
int data[] = {1, 2, 3, 4, 5};
fp = fopen("data.bin", "wb");
fwrite(data, sizeof(int), sizeof(data)/sizeof(int), fp);
fclose(fp);
```

It's important to note that when reading & writing binary data, you should use "rb" and "wb" mode respectively.

17. Write a programme to copy one file contents to another.

```
#include <stdio.h>
int main()
{
    FILE *source, *target; // File pointers.
    source = fopen("source.txt", "r"); // open source file in reading mode.
    if (source == NULL) // Error handling
    {
        printf("Could not open source file.\n");
        return 1;
    }
    target = fopen("target.txt", "w"); // open target file in writing mode.
    if (target == NULL) // Error handling
    {
        printf("Could not open target file.\n");
        fclose(source);
        return 1;
    }
    char ch; // Copy contents from source to target
    while ((ch = fgetc(source)) != EOF)
    {
        fputc(ch, target);
    }
    printf("File copied successfully.");
    // Close both files.
```



```

fclose(source);
fclose(target);
return 0;
}

```

18. Explain different file handling functions with Syntaxes and Suitable examples.

C standard library provides several functions for file handling. Some of the commonly used functions are:

(i) ~~fopen~~ ~~const~~ \*filename, const char

(i) 'fopen (const char \*filename, const char \*mode)': This function is used to open a file. It takes the name of the file and the mode in which the file should be opened as arguments. The mode can be "r" for reading, "w" for writing, "a" for appending, "r+" for reading and writing, and "w+" for writing & reading.

Syntax:

```
FILE *fopen (const char *filename, const char *mode);
```

Example:

```
FILE *fp;
```

```
fp = fopen("example.txt", "r");
```

(ii) 'fclose(FILE \*fp)': This function is used to close an open file. It takes a file pointer as an argument.

Syntax:

```
int fclose(FILE *fp);
```

Example:

```
fclose(fp);
```

(iii) 'fgetc(FILE \*fp)': This function is used to read a single character from a file. It takes a file pointer as an argument and returns the character read as an int.

Syntax:

```
int fgetc(FILE *fp);
```



Example:

```
int ch;
```

```
ch = fgetc(fp);
```

- (iv) `fputc(int c, FILE *fp)`: This function is used to write a single character to a file. It takes an int and a file pointer as arguments.

Syntax:

```
int fputc(int c, FILE *fp);
```

Example:

```
fputc('A', fp);
```

- (v) `fread(void *ptr, size_t size, size_t count, FILE *fp)`: This function is used to read binary data from a file. It takes a pointer to the buffer, the size of each element, the number of elements, and a file pointer as arguments.

Syntax:

```
size_t fread(void *ptr, size_t size, size_t count, FILE *fp);
```

Example:

```
int data[100];
```

```
fread(data, sizeof(int), 100, fp);
```

- (vi) `fwrite(const void *ptr, size_t size, size_t count, FILE *fp)`: This function is used to write binary data to a file. It takes a pointer to the data, the size of each element, the number of elements, and a file pointer as arguments.

Syntax:

`sizeof`

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *fp);
```

Example:

```
int data[100] = {1, 2, 3, 4, 5};
```

```
fwrite(data, sizeof(int), 100, fp);
```

- (vii) `fprintf(FILE *fp, const char *format, ...)`: This function is used to write formatted output to a file. It takes a file pointer,



a format string, and a variable number of arguments.

### Syntax:

```
int fprintf(FILE *fp, const char *format, ...);
```

### Example:

```
FILE *fp;
```

```
int i = 42;
```

```
float f = 3.14;
```

```
char str[] = "Hello world";
```

```
fp = fopen("example.txt", "w");
```

```
fprintf(fp, "Integer: %d, Float: %.f, String: %s", i, f, str);
```

```
fclose(fp);
```