1. Write a recursive function in C to generate a Fibonacci series.

Certainly! Here's a simple recursive function in C that generates the Fibonacci series up to a given number of terms:

```c
#include <stdio.h>

// Function to calculate the nth Fibonacci number recursively
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    int terms, i;

    printf("Enter the number of terms in the Fibonacci series: ");
    scanf("%d", &terms);

    printf("Fibonacci Series: ");
    for (i = 0; i < terms; i++) {
        printf("%d ", fibonacci(i));
    }

    return 0;
}
```

To use this program, compile and run it. It will ask for the number of terms you want in the Fibonacci series, and then it will print out the series up to that number of terms. Keep in mind that this recursive approach is not the most efficient for large numbers due to the repeated calculations involved. For larger series, an iterative approach or memoization would be more efficient.

2. Write an algorithm and draw a corresponding flow chart to check whether the given number is prime or not.

Certainly! Here's a simple algorithm to check if a number is prime, followed by a flowchart representation:

Algorithm:

1. Start
2. Input the number `n`
3. If `n` is less than 2, it is not prime. End.
4. Set `flag` to true
5. For `i` from 2 to `sqrt(n)`
   - If `n` modulo `i` is 0, set `flag` to false and break the loop
6. If `flag` is true, `n` is prime
7. Else, `n` is not prime
8. End

Flowchart:

```
  Start
    |
    v
 Input n
    |
    v
n < 2? -----> No ----> Set flag = true
    |                      |
   Yes                     v
    |                For i = 2 to sqrt(n)
    v                      |
 Not Prime                 v
    |               n mod i == 0?
    |                      |
    v                     Yes
  End <---- No <---- Set flag = false
                       |
                      Yes
                       |
                       v
                   flag == true?
                       |
                  No /    \ Yes
                    /      \
                  v         v
              Not Prime   Prime
                  |         |
                  v         v
                  End      End
```

This algorithm checks divisibility starting from 2 up to the square root of `n` because a larger factor of `n` must be a multiple of a smaller factor that has been already checked.

The flowchart visually represents the decision-making process involved in determining if a number is prime. It's a helpful way to understand the algorithm's structure at a glance. Remember, this is a basic algorithm and optimizations can be made for larger numbers or specific use cases.

3. Write an interactive C program to perform the following operation on a 3 x 3 matrix with appropriate validation checks:

C = A * B

Certainly! Below is an interactive C program that multiplies two 3x3 matrices A and B, and stores the result in matrix C. It includes validation checks to ensure that only numerical inputs are accepted.

```
#include <stdio.h>

#define SIZE 3

// Function to input matrix
void inputMatrix(int mat[SIZE][SIZE], char name)
    { printf("Enter the elements of matrix %c:\n",
    name);for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (scanf("%d", &mat[i][j]) != 1) {
                printf("Invalid input! Please enter an integer.\n");
                // Clear the input buffer
                while (getchar() != '\n');
                j--; // Decrement j to re-enter the last value
            }
        }
    }
}

// Function to multiply matrices
void multiplyMatrices(int matA[SIZE][SIZE], int matB[SIZE][SIZE], int matC[SIZE][SIZE])
    { for (int i = 0; i < SIZE; i++)
        {for (int j = 0; j < SIZE; j++)
            { matC[i][j] = 0;
            for (int k = 0; k < SIZE; k++)
                { matC[i][j] += matA[i][k] *
                matB[k][j];
            }
        }
    }
}

// Function to print matrix
void printMatrix(int mat[SIZE][SIZE], char name)
    { printf("Matrix %c:\n", name);
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++)
            { printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

    // Input matrices A and B
    inputMatrix(A, 'A');
    inputMatrix(B, 'B');

    // Multiply A and B, result in C
    multiplyMatrices(A, B, C);

    // Print the result
    printMatrix(C, 'C');

    return 0;
}
```

4. Differentiate between call-by-value and call-by-reference using an example program for each.

Certainly! In programming, **call-by-value** and **call-by-reference** are two methods of passing arguments to functions. Here's a brief explanation and example programs for each:

Call-by-Value

In call-by-value, the function receives copies of the arguments, so any changes made within the function do not affect the original variables. Here's a simple C program demonstrating call-by-value:

```
#include <stdio.h>

void swapx(int x, int y);

int main() {
    int a = 10, b = 20;
    printf("Before swap in main: a = %d, b = %d\\n", a, b);
    swapx(a, b);
    printf("After swap in main: a = %d, b = %d\\n", a, b);
    return 0;
}

void  swapx(int  x,  int  y)
    { int t;
    t  =  x;
    x  =  y;
    y = t;
    printf("Inside swapx function: x = %d, y = %d\\n", x, y);
}
```

Output:

```
Before swap in main: a = 10, b = 20
Inside swapx function: x = 20, y = 10
After swap in main: a = 10, b = 20
```

As you can see, the values of `a` and `b` remain unchanged in the main function after the swap.

Call-by-Reference

In call-by-reference, the function receives references (or addresses) to the actual arguments, so changes made within the function affect the original variables. Here's a simple C program demonstrating call-by-reference using pointers:

```
#include <stdio.h>

void swapx(int* x, int* y);

int main() {
    int a = 10, b = 20;
    printf("Before swap in main: a = %d, b = %d\\n", a, b);
    swapx(&a, &b);
    printf("After swap in main: a = %d, b = %d\\n", a, b);
    return 0;
}

void swapx(int* x, int* y)
    { int t;
    t = *x;
    *x = *y;
    *y = t;
    printf("Inside swapx function: x = %d, y = %d\\n", *x, *y);
}
```

Output:

```
Before swap in main: a = 10, b = 20
Inside swapx function: x = 20, y = 10
After swap in main: a = 20, b = 10
```

Here, the values of `a` and `b` are changed in the main function after the swap because the function operates on their addresses.

These examples should help illustrate the difference between call-by-value and call-by-reference. In summary, call-by-value protects the original variables from being altered by the function, while call-by-reference allows the function to modify the original variables directly.

5. Using pointers, write a program in C to count the number of occurrences of each character in a given string.

Sample I/P: ARRANGE

O/P:

A — 2 times

G — 1 time

R — 2 times

E — 1 time

N — 1 time

Certainly! Below is a simple C program that uses pointers to count the number of occurrences of each character in a given string:

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

#define MAX 256

int main() {
    char str[] = "ARRANGE";
    int count[MAX] = {0};
    char *ptr = str;

    // Counting each character in the string
    while(*ptr != '\0') {
        count[*ptr++]++;
    }

    // Displaying the occurrences of each character
    for(int i = 0; i < MAX; i++) {
        if(count[i] > 0) {
            printf("%c — %d time%s\n", i, count[i], (count[i] > 1) ? "s" : "");
        }
    }

    return 0;
}
```

This program initializes an array `count` to keep track of the occurrences of each character. It uses a pointer `ptr` to traverse the string `str`. For each character pointed to by `ptr`, it increments the corresponding index in the `count` array. Finally, it prints out the characters and their counts, adding an ‘s’ to “time” if a character appears more than once.

You can modify the `str` array with your input string to count the occurrences of characters in different strings. Remember to compile the program with a C compiler before running it.

6. Using file handling concepts, write a C program to read a file and count the number of lines in the file.

Certainly! Here's a simple C program that uses file handling to read a file and count the number of lines:

```c
#include <stdio.h>

int main() {
    FILE *file;
    char filename[100];
    char ch;
    int linesCount = 0;

    // Prompt user to enter the filename
    printf("Enter the name of the file: ");
    scanf("%s", filename);

    // Open the file in read mode
    file = fopen(filename, "r");

    // Check if the file exists
    if (file == NULL) {
        printf("Could not open file %s", filename);
        return 1;
    }

    // Read character by character and check for new line
    while ((ch = fgetc(file)) != EOF) {
        // Increment count if this character is newline
        if (ch == '\n') linesCount++;
    }

    // Increment lines count if last line doesn't end with a new line character
    if (ch != '\n' && linesCount != 0) linesCount++;

    // Close the file
    fclose(file);

    // Print the number of lines
    printf("The file %s has %d lines\n", filename, linesCount);

    return 0;
}
```

This program will prompt the user to enter the name of the file, then it will count the number of lines in the file and print the result. Make sure the file you want to read is in the same directory as your program or provide the full path to the file. Compile and run the program in your C environment to see the output.

7. Define a function in C. List and explain various categories of functions. Also, illustrate a function to find the square root of a given number.

In C programming, a function is a block of code that performs a specific task. It has a name and can be called from other parts of the program to execute the code within the function. Functions help in structuring a program into logical pieces and can be reused, which makes the code more manageable and modular.

Categories of Functions in C:

1. **Standard Library Functions**: These are built-in functions provided by C's standard library, such as `printf()`, `scanf()`, `sqrt()`, etc.
2. **User-Defined Functions**: Functions that programmers create to perform specific tasks that are not included in the standard library.
3. **Recursive Functions**: A function that calls itself directly or indirectly is known as a recursive function. It's used to solve problems that can be broken down into smaller, repetitive problems.

Here's an example of a user-defined function to find the square root of a given number using the Newton-Raphson method:

```
#include <stdio.h>

double squareRoot(double number) {
    double error = 0.00001; // Define the precision of your result
    double s = number;

    // Using Newton-Raphson method to find the square root
    while ((s - number / s) > error) { // Continue until the precision is satisfactory
        s = (s + number / s) / 2;
    }
    return s;
}

int main() {
    double number = 16.0;
    printf("Square root of %.2f is %.2f\n", number, squareRoot(number));
    return 0;
}
```

This function `squareRoot` takes a `double` type number as an argument and returns its square root. The `main` function demonstrates how to call this `squareRoot` function and print the result.

8. Explain the following user-defined data types:

(i) typedef

(ii) enum

Certainly! Here's an explanation of the two user-defined data types you've asked about:

1. **typedef**: The `typedef` keyword in C and C++ is used to create an alias for an existing data type. This can make complex data type declarations easier to understand and work with. For example, if you have a struct for a date:

```
struct Date {
    int day;
    int month;
    int year;
};
```

You can use `typedef` to create a simpler name for this type:

```
typedef struct Date Date;
```

Now you can use `Date` to declare variables of this struct type, like so:

```
Date today;
```

1. **enum**: The `enum` (enumeration) keyword is used to define a set of named integer constants. An `enum` can increase code clarity and make it easier to manipulate sets of related constants. Here's an example:

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

This defines a new type called `Day`, which can hold one of the seven named values. You can then use this type to declare variables:

```
Day today = SUNDAY;
```

Each name in an `enum` is associated with an integer, starting from 0 by default. You can also assign specific values to the names:

```
enum StatusCode { OK = 200, NOT_FOUND = 404, ERROR = 500 };
```

In this case, `OK` will have the value 200, `NOT_FOUND` will be 404, and `ERROR` will be 500.

These user-defined data types help in writing more readable and maintainable code.

9. What is the difference between a high-level language and a low-level language? Why is the C language referred to as a middle-level language?

High-level languages and low-level languages serve different purposes and have distinct characteristics:

- **High-Level Languages**:
  - **Abstract from hardware**: They are more abstracted from the details of the computer's hardware.
  - **Easier to learn and use**: They are designed to be easy for humans to read and write.
  - **Portable**: Programs written in high-level languages can often be run on different types of computers with few or no changes.
  - **Require compilers/interpreters**: They need to be translated into machine code by compilers or interpreters.
  - **Examples**: C, C++, Java, Python.
- **Low-Level Languages**:
  - **Close to hardware**: They are closer to the machine's hardware and are less abstracted.
  - **Harder to understand**: They are more difficult for humans to read and write.
  - **Machine-dependent**: Programs are often specific to a particular type of computer.
  - **Efficient**: They can be more efficient because they allow for more control over what the computer is doing.
  - **Examples**: Assembly language, machine code.

The **C language** is often referred to as a **middle-level language** because it includes features of both high-level and low-level languages. It can be used for system programming (like operating systems) as well as application programming (like spreadsheets). C provides the ability to manipulate bits, bytes, and addresses—features that are characteristic of low-level languages. At the same time, C offers constructs like functions, loops, and conditionals, which are typical of high-level languages.

This combination makes C a versatile language that can be used in a wide range of applications, from system software to application software, which is why it's considered a middle-level language.

10. Write a program using pointers to swap the values of two variables.

Certainly! Here's a simple C program that uses pointers to swap the values of two variables:

```
#include <stdio.h>

void swap(int *x, int *y)
    { int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;

    printf("Before swap: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After swap: a = %d, b = %d\n", a, b);

    return 0;
}
```

In this program, `swap` is a function that takes two integer pointers `x` and `y` as arguments. It dereferences the pointers to access the variables they point to and swaps their values using a temporary variable `temp`. The `main` function demonstrates the swap by printing the values of `a` and `b` before and after the call to `swap`. Remember to pass the addresses of `a` and `b` to the `swap` function using the address-of operator `&`.

11. Explain type-cast and sizeof operators in C with an example for each.

Certainly! In C, **type-casting** is used to convert a variable from one data type to another. It can be done explicitly by the programmer or implicitly by the compiler. Here's an example of explicit type-casting:

```c
#include <stdio.h>

int main() {
    int i = 10;
    float f = (float)i; // The integer 'i' is being type-casted to a float
    printf("The value of f is: %.2f\n", f);
    return 0;
}
```

In this example, the integer `i` is cast to a float `f`. This is necessary when you want to ensure that operations involving `i` are carried out as floating-point operations.

The `sizeof` operator, on the other hand, is used to determine the size, in bytes, of a data type or a variable. Here's an example:

```c
#include <stdio.h>

int main() {
    int i;
    printf("The size of int is: %zu bytes\n", sizeof(i)); // Using sizeof to find the size of an int
    return 0;
}
```

In this code, `sizeof(i)` returns the size of the variable `i`, which is of type `int`. The `%zu` format specifier is used for size_t type returned by `sizeof`.

Remember, `sizeof` is a compile-time operator, so its argument is not evaluated. For example, `sizeof(1/0)` will not produce a runtime error; it simply returns the size of `int`.

12. Define a preprocessor in C. How is it implemented? Explain with the help of an example.

In C programming, a **preprocessor** is a tool that processes your source code before the compiler takes over. It's essentially a text substitution tool that instructs the compiler to do certain pre-processing based on specific directives.

The preprocessor directives all start with a `#` symbol and include commands like `#define`, `#include`, `#if`, `#else`, `#endif`, and more. These directives can define macros, include header files, or conditionally compile different parts of your program.

Here's a simple example using the `#define` directive to create a macro:

```c
#include <stdio.h>

// Define a macro named LIMIT with a value of 5
#define LIMIT 5

int main() {
    for (int i = 0; i < LIMIT; i++)
        { printf("%d\\n", i);
    }
    return 0;
}
```

In this example, the preprocessor will replace every instance of `LIMIT` with `5` before the compilation process begins. So, the loop will run from `0` to `4`, printing each number on a new line.

The preprocessor is implemented as part of the C compiler system. When you compile a C program, the first step is preprocessing, where all the preprocessor directives are executed. The output is an expanded source code file with all macros replaced and all included files inserted. This expanded file is then compiled into object code, and finally, the linker creates the executable.

Copilot

13. Write a C program using structures to find the total, average, and grade for 5 students.

Note: Assumptions can be made wherever necessary and write them.

Certainly! Below is a simple C program that uses structures to calculate the total, average, and grade for 5 students. I'm assuming each student has marks for 3 subjects, and the grading system is as follows: A for average >= 90, B for average >= 75, C for average >= 60, D for average >= 50, and F for average < 50.

```
#include<stdio.h>

#define STUDENTS

5
#define SUBJECTS 3

struct Student
   {int id;
   char name[100];
   int
   marks[SUBJECTS];
   int  total;
   float average;
   char grade;
};

 void calculateGrades(struct Student *s)
{
   s->total = 0;
   for (int i = 0; i < SUBJECTS;
      i++) { s->total += s-
      >marks[i];
   }
   s->average = s->total / (float)SUBJECTS;

   if (s->average >= 90) s->grade = 'A';
   else  if  (s->average  >=  75)  s-
   >grade   =   'B';   else   if   (s-
   >average >= 60) s->grade  =  'C';
   else  if  (s->average  >=  50)  s-
   >grade  =  'D';  else  s->grade  =
   'F';
}

int main() {
   struct Student students[STUDENTS];

   students for (int i = 0; i <
   STUDENTS; i++) {
      calculateGrades(&students[i]);
   }

   printf("ID\tName\tTotal\tAverage\tGr
   ade\n"); for (int i = 0; i <
   STUDENTS; i++) {
      printf("%d\t%s\t%d\t%.2f\t%c\n", students[i].id, students[i].name, students[i].total,
      students[i].average, students[i].grade);
   }

   return 0;
}
```

Assumptions:

- Each student has a unique ID and name.
- The marks for each student are already known and entered into the system.•The grading system is fixed and based on the average marks as described.

14. Write short notes on the following with the help of an example mentioning their use in programs:

(a) getch( )

(b) void

(c) gets( )

(d) ++ (increment operator)

(e) -- (decrement operator)

(f) % operator

(g) break statement

(h) #define

(i) fseek( )

(j) goto statement

Certainly! Here are short notes on each of the listed programming concepts with examples:

(a) `getch()`: This function is used to get a character from the keyboard. It does not echo the character to the console. It's often used to pause the program until the user presses a key.

```
#include <conio.h>
int main() {
    char ch;
    printf("Press a key");
    ch = getch();  // Waits for user input
    return 0;
}
```

(b) `void`: The `void` keyword is used to specify that a function does not return a value.

```
void displayMessage()
    { printf("Hello,
    World!");
}
```

© **`gets()`**: This function reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or EOF (End of File).

```
char str[50];
gets(str);  // Reads a string from the user until a newline is encountered
```

(d) `++` (Increment Operator): This operator increases the value of a variable by 1. It can be used as a prefix or postfix.

```
int i = 10;
i++;  // i becomes 11
```

(e) `--` (Decrement Operator): This operator decreases the value of a variable by 1. It can be used as a prefix or postfix.

```
int i = 10;
i--;  // i becomes 9
```

(f) `%` Operator: The modulus operator returns the remainder of the division of two numbers.

```
int remainder = 10 % 3;  // remainder is 1
```

(g) `break` Statement: Used to exit from a loop or switch statement before it has completed its normal sequence.

```
for (int i = 0; i < 10; i++)
    { if (i == 5) {
        break;  // Exits the loop when i is 5
    }
}
```

(h) `#define`: This preprocessor directive is used to define a macro or a constant value that replaces a token in the code.

```
#define PI 3.14
double area = PI * radius * radius;
```

(i) `fseek()`: This function sets the file position of the stream to the given offset. It is used for random access in files.

```
FILE *file = fopen("example.txt", "r");
fseek(file, 5, SEEK_SET);  // Moves the file pointer to the 6th byte from the beginning
```

(j) `goto` Statement: Allows the program to jump to a labeled statement.

```
if (error) {
    goto errorHandler;
}
// ...
errorHandler:
    printf("An error occurred");
```

These concepts are fundamental in C programming and are used for various purposes such as input/output operations, flow control, and file handling. Remember that functions like `gets()` are considered unsafe due to buffer overflow risks and should be avoided in favor of safer alternatives like `fgets()`.

15. Define an array. How do we declare and initialize a single-dimensional array and a 2-dimensional array? Explain with an example for each.

An array is a collection of elements, all of the same type, stored in contiguous memory locations. It allows us to access elements using an index.

Here's how you declare and initialize arrays in C++:

Single-dimensional array:

```
// Declaration
int myArray[5];

// Initialization
myArray[0] = 10;
myArray[1] = 20;
myArray[2] = 30;
myArray[3] = 40;
myArray[4] = 50;

// Declaration and initialization at the same time
int myArray[5] = {10, 20, 30, 40, 50};
```

2-dimensional array:

```
// Declaration
int my2DArray[2][3];

// Initialization
my2DArray[0][0] = 1;
my2DArray[0][1] = 2;
my2DArray[0][2] = 3;
my2DArray[1][0] = 4;
my2DArray[1][1] = 5;
my2DArray[1][2] = 6;

// Declaration and initialization at the same time
int my2DArray[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

In the single-dimensional array example, `myArray` has 5 elements of type `int`. In the 2-dimensional array, `my2DArray` is an array of 2 arrays, each containing 3 `int` elements. The first index represents the row, and the second index represents the column.

16. Write a C program to implement STRING COPY operation that copies a string "str1" to another string "str2" without using a library function.

Certainly! Below is a simple C program that demonstrates how to copy a string from `str1` to `str2` without using library functions like `strcpy`.

```
#include <stdio.h>

void stringCopy(char *dest, const char *source)
    { while(*source) {
        *dest = *source;
        dest++;
        source++;
    }
    *dest = '\0'; // Null-terminate the destination string
}

int main() {
    char str1[] = "Hello, World!";
    char str2[20]; // Ensure this is large enough to hold str1

    stringCopy(str2, str1); // Copy str1 into str2

    printf("str1: %s\n", str1);
    printf("str2: %s\n", str2);

    return 0;
}
```

This program includes a function `stringCopy` that takes two pointers to characters as its parameters. It copies the characters from the source string (`str1`) to the destination string (`str2`) one by one until it encounters the null character `\0` that marks the end of the string. After copying, it manually adds a null character at the end of `str2` to ensure it is a properly terminated string. The `main` function demonstrates the usage of `stringCopy` by copying `str1` to `str2` and then printing both strings. Remember to always ensure that the destination array is large enough to hold the source string plus the null terminator.

17. Discuss dynamic memory allocation in C. Write and explain dynamic allocation functions in C.

Dynamic memory allocation in C is a powerful feature that allows you to allocate memory during the runtime of the program. This is particularly useful when you don't know the exact amount of memory that you will need while writing the program. C provides several functions for dynamic memory allocation, and they are all found in the `<stdlib.h>` header file.

Here are the primary dynamic memory allocation functions in C:

1. **`malloc`**: This function allocates a single block of memory of the specified size. It returns a pointer to the beginning of the block. The memory is not initialized, which means it will contain garbage values.

   ```
   void* malloc(size_t size);
   ```

   Example:

   ```
   int *ptr = (int*)malloc(10 * sizeof(int)); // Allocates space for 10 integers
   ```

2. **`calloc`**: Similar to `malloc`, but it also initializes the allocated memory to zero.

   ```
   void* calloc(size_t num, size_t size);
   ```

   Example:

   ```
   int *ptr = (int*)calloc(10, sizeof(int)); // Allocates and initializes space for 10 integers
   ```

3. **`realloc`**: This function is used to resize the memory block that was previously allocated. It can make the memory block larger or smaller.

   ```
   void* realloc(void* ptr, size_t size);
   ```

   Example:

   ```
   ptr = realloc(ptr, 20 * sizeof(int)); // Resizes the previously allocated space to hold 20 integers
   ```

4. **`free`**: This function deallocates the memory block that was previously allocated by `malloc`, `calloc`, or `realloc`.

   ```
   void free(void* ptr);
   ```

   Example:

   ```
   free(ptr); // Frees the allocated memory
   ```

It's important to note that you should always check if the memory allocation was successful by checking if the pointer returned by `malloc`, `calloc`, or `realloc` is not `NULL`. Also, it's good practice to free any dynamically allocated memory once you're done using it to avoid memory leaks.

18. Write a C program to find the sum of diagonal elements of a 3 x 3 matrix.

Certainly! Here's a simple C program that calculates the sum of the diagonal elements of a 3 x 3 matrix:

```
#include <stdio.h>

int main() {
    int matrix[3][3];
    int i, j, sum = 0;

    // Input elements in the matrix
    printf("Enter elements in 3x3 matrix:\n");
    for(i = 0; i < 3; i++) {
        for(j = 0; j < 3; j++)
            { scanf("%d",
            &matrix[i][j]);
        }
    }

    // Calculate the sum of diagonal elements
    for(i = 0; i < 3; i++) {
        sum += matrix[i][i];
    }

    // Display the sum
    printf("Sum of diagonal elements = %d\n", sum);

    return 0;
}
```

This program first takes the input for the 3 x 3 matrix from the user. Then, it calculates the sum of the diagonal elements where the row index and column index are the same (i.e., `matrix[0][0]`, `matrix[1][1]`, `matrix[2][2]`). Finally, it prints out the sum of these elements. Make sure to compile and run this program in a C environment. Let me know if you need further assistance!

19. Using recursion, write a C program to find the factorial of a given number.

Certainly! Here's a simple C program that uses recursion to calculate the factorial of a given number:

```c
#include <stdio.h>

// Function to calculate the factorial recursively
long long int factorial(int n) {
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive case
}

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

    // Ensure the user enters a positive number
    if (number < 0) {
        printf("Factorial of a negative number doesn't exist.\n");
    } else {
        printf("Factorial of %d is %lld\n", number, factorial(number));
    }

    return 0;
}
```

To use this program, simply compile it with a C compiler and run the executable. When prompted, enter a positive integer to get its factorial. Remember, factorials can grow very large very quickly, so `long long int` is used to accommodate larger numbers. However, even `long long int` has its limits, so be mindful of the input size to avoid overflow.

20. Write a function to print the sum of the following series:

$1 + 2^2 + 3^3 + ... + n^n$

where "n" is passed as an argument to the function.

Certainly! Here's a C function that calculates and prints the sum of the series ($1 + 2^2 + 3^3 + \ldots + n^n$), where (n) is the input to the function:

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

// Function to calculate the sum of the series
long long sumOfSeries(int n) {
    long long sum = 0;
    for (int i = 1; i <= n; i++)
        { sum += pow(i, i);
    }
    return sum;
}

int main() {
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);

    // Calculate and print the sum of the series
    printf("The sum of the series is: %lld\n", sumOfSeries(n));

    return 0;
}
```

This function uses a loop to iterate from 1 to (n), calculating (i^i) at each step using the `pow` function from the `math.h` library, and adding it to the `sum`. The `sumOfSeries` function returns the final sum, which is then printed in the `main` function.

Please note that for very large values of (n), the result may not be accurate due to the limitations of the `long long` data type and the `pow` function. For such cases, additional considerations for handling large numbers would be required.

21. Write a program to calculate the number of vowels (a, e, i, o, u) separately in the entered string and display their individual count.

Certainly! Below is a C program that calculates and displays the count of each vowel (a, e, i, o, u) in an entered string:

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

int main() {
    char str[100];
    int count_a = 0, count_e = 0, count_i = 0, count_o = 0, count_u = 0;
    int i;

    // Prompt user to enter a string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    // Calculate the length of the string
    int length = strlen(str);

    // Iterate through each character of the string
    for(i = 0; i < length; i++) {
        // Convert character to lowercase for comparison
        char ch = tolower(str[i]);

        // Check and count vowels
        switch(ch) {
            case 'a':
                count_a++;
                break;
            case 'e':
                count_e++;
                break;
            case 'i':
                count_i++;
                break;
            case 'o':
                count_o++;
                break;
            case 'u':
                count_u++;
                break;
            default:
                // Not a vowel, do nothing
                break;
        }
    }

    // Display individual vowel counts
    printf("Count of 'a': %d\n", count_a);
    printf("Count of 'e': %d\n", count_e);
    printf("Count of 'i': %d\n", count_i);
    printf("Count of 'o': %d\n", count_o);
    printf("Count of 'u': %d\n", count_u);

    return 0;
}
```

This program uses a `for` loop to iterate through each character of the string. It converts each character to lowercase using `tolower()` function to ensure case-insensitive comparison. Then, it uses a `switch` statement to check if the character is a vowel and increments the corresponding counter if it is.

To run the program, simply compile it with a C compiler (like `gcc`) and execute the resulting binary. When prompted, enter the string you want to analyze, and the program will output the count of each vowel.

22. Write a C program to find the substring in a string without using a library function.

Certainly! Below is a simple C program that finds a substring within a string without using any library functions. It uses two nested loops to compare the substring with the main string character by character.

```
#include <stdio.h>

int main() {
    char string[100], substring[100];
    int position = -1, i, j, flag;

    printf("Enter the main string: ");
    gets(string);
    printf("Enter the substring to find: ");
    gets(substring);

    for(i = 0; string[i] != '\0'; i++)
        { if(string[i] == substring[0])
            {flag = 1;
            for(j = 0; substring[j] != '\0'; j++)
                { if(string[i + j] != substring[j])
                    {flag = 0;
                    break;
                }
            }
            if(flag == 1)
                { position = i;
                break;
            }
        }
    }

    if(position != -1) {
        printf("Substring found at position: %d\n", position);
    } else {
        printf("Substring not found in the string.\n");
    }

    return 0;
}
```

In this program:

- `string` is the main string where we search for the `substring`.
- `position` keeps track of the starting index of the substring within the main string. It's initialized to `-1` to indicate that the substring is not found by default.
- The outer loop goes through each character of the main string.
- The inner loop compares the substring with the main string starting from the current index of the outer loop.
- If the substring is found, `flag` remains `1` and the position is updated.
- After the loops, the program checks if `position` is not `-1` to confirm the substring was found and prints its position.

Remember to compile the program with a C compiler and run it to test with different strings and substrings. Let me know if you need further assistance!

23. Explain the switch statement with the help of a program segment. Also, write its syntax.

Certainly! The `switch` statement in C is a control structure that allows you to perform different actions based on the value of an integer or character expression. Here's the syntax for a `switch` statement:

```
switch (expression)
  { case constant1:
    // statements
    break;
  case constant2:
    // statements
    break;
  // more cases...
  default:
    // default statements
}
```

And here's a simple program segment that demonstrates how a `switch` statement can be used:

```
#include <stdio.h>

int main() {
    char grade = 'B';

    printf("Your grade is %c\n", grade);

    switch (grade) {
        case 'A':
            printf("Excellent!\n");
            break;
        case 'B':
        case 'C':
            printf("Well done\n");
            break;
        case 'D':
            printf("You passed\n");
            break;
        case 'F':
            printf("Better try again\n");
            break;
        default:
            printf("Invalid grade\n");
    }

    printf("Your grade is %c\n", grade);

    return 0;
}
```

In this example, the `switch` statement checks the value of the variable `grade`. Depending on its value, it prints out a corresponding message. If `grade` is 'A', it prints "Excellent!", and so on. If `grade` doesn't match any of the `case` constants, the `default` case is executed, printing "Invalid grade". The `break` statement is used to exit the `switch` block once a matching case is found and executed. Without `break`, the program would continue executing the subsequent cases, which is usually not the desired behavior.

24. How is "#define" used to create functional macros? Illustrate with the help of a C program segment.

In C programming, `#define` is used to create macros, which are essentially a piece of code in a program that is given a name. You can define a macro for a constant value or to create function-like macros. Here's how you can use `#define` to create a functional macro:

```c
#include <stdio.h>

// Define a macro that calculates the square of a number
#define SQUARE(x) ((x) * (x))

int main() {
    int value = 5;
    // Use the SQUARE macro
    printf("The square of %d is %d\n", value, SQUARE(value));
    return 0;
}
```

In this example, `SQUARE(x)` is a functional macro that calculates the square of a number `x`. When you call `SQUARE(value)`, it expands to `((value) * (value))`, which computes the square of `value`.

Key points to remember when creating functional macros:

- Always enclose the macro arguments and the entire macro definition in parentheses to avoid unexpected behavior due to operator precedence.
- Be cautious with side effects; for example, if you pass an expression with a side effect like `i++` as an argument to a macro, it might get evaluated multiple times.
- Macros are replaced by their definition before the program is compiled, not at runtime.

25. Differentiate between sequential and random access files.

Sequential and random access files are two methods of organizing and accessing data within files:

- **Sequential Access Files**:
  - Data is stored and retrieved in a **sequential order**.
  - Suitable for scenarios that involve processing data sequentially, like sorting or merging data.
  - Accessing a specific piece of data requires processing all previous data.
  - Commonly used in tape drives and for continuous data streaming applications.

- **Random Access Files**:
  - Allows **direct access** to any record within the file.
  - Efficient for searching, updating, and retrieving specific records without traversing the entire file.
  - Commonly used in databases and file systems where frequent direct access or modification of specific records is required.

In summary, sequential access is linear and well-suited for tasks with a predictable flow of data, while random access provides flexibility and efficiency for tasks requiring frequent and direct manipulation of specific data points.

26. Write short notes on the following:

(i) Structure

(ii) Union

Certainly! Here's a brief explanation of both structures and unions in C:

(i) Structure: A structure in C is a user-defined data type that allows you to combine data items of different kinds. Structures are used to represent a record. Suppose you want to keep track of books in a library. You might want to track the following attributes about each book: title, author, and ISBN. Here's how you can create a structure for it:

```
struct Book {
    char title[50];
    char author[50];
    int isbn;
};
```

In this structure, `title` and `author` are arrays of characters, while `isbn` is an integer. Each element within a structure is called a member of the structure.

Key Points:

- You can create variables of a structure type to store information for different entities.
- Each structure variable will have its own copy of the data members.
- The size of a structure is the sum of the sizes of all its members.

(ii) Union: A union in C is also a user-defined data type similar to a structure, but it stores different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time.

Here's an example of a union that can store either an integer, a float, or a character array:

```
union Data {
    int integer;
    float decimal;
    char str[20];
};
```

Key Points:

- All members of a union share the same memory location. This means that at any time, a union can contain no more than one of its members from its definition.
- The size of a union is the size of its largest member.
- Unions are useful in situations where you want to use the same memory location for different purposes.

Both structures and unions are essential for creating complex data types that are closer to how we think about data in the real world. They help in organizing data in a logical and manageable way.

27. Explain the function "call by reference" in C language. Give advantages and disadvantages of it.

In C language, "call by reference" is a method of passing arguments to a function where the function receives the **addresses** of the parameters rather than their values. This allows the function to modify the actual arguments used in the call. Here's how it works and its pros and cons:

Functioning of Call by Reference:

```
void swap(int *x, int *y)
    { int temp;
    temp = *x; // save the value at address x
    *x = *y;   // put y into x
    *y = temp; // put temp into y
}

int main() {
    int a = 100;
    int b = 200;

    // Before swap, value of a and b
    swap(&a, &b); // passing addresses of a and b

    // After swap, values are swapped
    return 0;
}
```

In the above example, `swap` function takes pointers to integers as its parameters, and swapping is done using the addresses, which reflects back in the original variables `a` and `b`.

Advantages:

- **Efficiency**: No copying of arguments, which can lead to better performance, especially with large data structures.
- **Modification**: Allows the function to modify the caller's variable, making it useful for returning multiple values.
- **Memory**: Saves memory as no additional space is required for the copied variables.

Disadvantages:

- **Safety**: The function can unintentionally modify the input data, which can lead to bugs if not handled carefully.
- **Null Pointers**: Requires ensuring that the pointers are not null before using them, as dereferencing null pointers can lead to undefined behavior.
- **Lifetime Issues**: The function must ensure that the pointers refer to valid memory locations throughout the function's execution.

In summary, call by reference can be more efficient and flexible but requires careful handling to avoid errors related to pointer use.

28. Write a short note on the ternary operator with an illustration.

The ternary operator in C is a concise way to perform conditional operations. It's represented as `? :` and is used as a shorthand for an `if-else` statement. The syntax for the ternary operator is:

```
condition ? expression_if_true : expression_if_false;
```

Here's how it works:

1. The `condition` is evaluated first.
2. If the `condition` is true, the `expression_if_true` is executed.
3. If the `condition` is false, the `expression_if_false` is executed.

Illustration in C:

Let's say we want to find the maximum of two numbers:

```c
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    int max;

    // Using ternary operator to find the maximum
    max = (a > b) ? a : b;

    printf("The maximum number is %d\n", max);
    return 0;
}
```

In this example, `(a > b)` is the condition. If `a` is greater than `b`, `a` is assigned to `max`; otherwise, `b` is assigned to `max`. The `printf` statement then prints out the maximum number.

29. Explain the difference between "call by value" and "call by reference" with the help of examples.

Certainly! In C, "call by value" and "call by reference" are two different ways a function can access variables passed to it.

Call by Value: In call by value, a copy of the actual parameter's value is passed to the function. Changes made to the parameter inside the function have no effect on the actual parameter.

```c
#include <stdio.h>

void square(int n)
    { n = n * n;
    printf("Inside square function: %d\n", n);
}

int main() {
    int number = 4;
    square(number);
    printf("Inside main function: %d\n", number);
    return 0;
}
```

Output:

```
Inside square function: 16
Inside main function: 4
```

The `number` variable remains unchanged in the `main` function because only a copy was modified in the `square` function.

Call by Reference: In call by reference, a reference (pointer) to the actual parameter is passed to the function. Changes made to the parameter inside the function affect the actual parameter.

```c
#include <stdio.h>

void square(int *n) {
    *n = (*n) * (*n);
    printf("Inside square function: %d\n", *n);
}

int main() {
    int number = 4;
    square(&number);
    printf("Inside main function: %d\n", number);
    return 0;
}
```

Output:

```
Inside square function: 16
Inside main function: 16
```

The `number` variable is changed in the `main` function because the `square` function modified the actual value through its pointer.

In summary, **call by value** protects the original data from being altered by the function, while **call by reference** allows the function to modify the original data.

30. Write the syntax and explain the use of the following functions in C:

(i) malloc

(ii) fputc

(iii) strcat

(iv) fclose

(v) calloc

Certainly! Here's the syntax and explanation for each of the C functions you've asked about:

1. **malloc**

   ```
   void* malloc(size_t size);
   ```

   `malloc` stands for "memory allocation" and is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type `void` which can be cast into a pointer of any form.

2. **fputc**

   ```
   int fputc(int char, FILE *stream);
   ```

   `fputc` writes the character value of the argument `char` to the specified `stream`. It returns the written character as an `unsigned char` cast to an `int` or `EOF` on error.

3. **strcat**

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

   `strcat` appends the string pointed to by `src` to the end of the string pointed to by `dest`. The end of the string is determined by the null-terminating character. The `dest` string must have enough space to hold the result.

4. **fclose**

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

   `fclose` closes the stream. All buffers are flushed to the operating system and the file is closed. It returns zero on success, or `EOF` if an error occurs.

5. **calloc**

   ```
   void* calloc(size_t num, size_t size);
   ```

   `calloc` stands for "contiguous allocation" and is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value `0`.

These functions are fundamental for dynamic memory management, file operations, and string manipulation in C programming.

31. Write a C program to open an existing file "My.TXT" and read its content and display it.

Certainly! Below is a simple C program that opens an existing file named “My.TXT”, reads its content, and displays it on the screen:

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

int main() {
    FILE *file;
    char ch;

    // Open the file in read mode
    file = fopen("My.TXT", "r");

    // Check if the file exists
    if (file == NULL) {
        printf("Cannot open file or file does not exist.\n");
        exit(0);
    }

    // Read and display the content of the file
    while ((ch = fgetc(file)) != EOF)
        putchar(ch);

    // Close the file
    fclose(file);

    return 0;
}
```

This program uses `fopen` to open the file in read mode (`"r"`). It checks if the file exists and is accessible. If the file cannot be opened, it prints an error message and exits. Otherwise, it reads characters one by one using `fgetc` until the end of file (`EOF`) is reached, and prints them using `putchar`. Finally, it closes the file using `fclose`.

Make sure that “My.TXT” is in the same directory as your C program, or provide the full path to the file when calling `fopen`. Also, remember to compile your C program with a C compiler like `gcc` before running it.

32. Write an algorithm to calculate simple interest.

(Hint: SI = (P * T * R) / 100)

Certainly! Here's a simple algorithm to calculate Simple Interest (SI):

```
Algorithm to Calculate Simple Interest:
1. Start
2. Initialize variables P (Principal), T (Time), R (Rate)
3. Read values of P, T, R
4. Calculate SI using the formula: SI = (P * T * R) / 100
5. Display the calculated SI
6. Stop
```

◎ Macro…

`#define` preprocessor directive example…

What C code would print the following pattern when executed?

C

CO

COB

COBO

COBOL

COBOL

COBO

COB

CO

C

```
Answer:

#include <stdio.h>

#define COBOL \
    for (int i = 0; i < 5; i++) { \
        for (int j = 0; j <= i; j++) { \
            printf("%c", "COBOL"[j]); \
        } \
        printf("\n"); \
    } \
    for (int i = 4; i >= 0; i--) { \
        for (int j = 0; j <= i; j++) { \
            printf("%c", "COBOL"[j]); \
        } \
        printf("\n"); \
    }

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

33. Write a C program to create a structure to store the name, roll number, address, and course (BCA, MCA) of ten students. Use an array of structures to display the details of the students.

Certainly! Below is a C program that defines a structure to store the details of ten students and displays their information using an array of structures.

```c
#include <stdio.h>

// Define the structure for student details
struct Student {
    char name[50];
    int rollNo;
    char address[100];
    char course[4]; // BCA or MCA
};

int main() {
    // Declare an array of structures for 10 students
    struct Student students[10];

    // Input details for each student
    for(int i = 0; i < 10; i++) {
        printf("Enter details for student %d\n", i + 1);
        printf("Name: ");
        scanf("%49s", students[i].name); // Using 49 to leave space for null terminator
        printf("Roll Number: ");
        scanf("%d", &students[i].rollNo);
        printf("Address: ");
        scanf("%99s", students[i].address); // Using 99 to leave space for null terminator
        printf("Course (BCA/MCA): ");
        scanf("%3s", students[i].course); // Using 3 to leave space for null terminator
    }

    // Display details for each student
    printf("\nDetails of the students:\n");
    for(int i = 0; i < 10; i++) {
        printf("Student %d\n", i + 1);
        printf("Name: %s\n", students[i].name);
        printf("Roll Number: %d\n", students[i].rollNo);
        printf("Address: %s\n", students[i].address);
        printf("Course: %s\n\n", students[i].course);
    }

    return 0;
}
```

This program will prompt the user to enter the details for each student one by one. After the details for all ten students have been entered, it will print out the details for each student. Please note that for simplicity, the program assumes that the name and address do not contain spaces. If you need to handle spaces in those fields, you would need to use a different method for input, such as `fgets`.

Remember to compile the program with a C compiler before running it. If you have any questions or need further assistance, feel free to ask!

34. Write a program to display the string "INFINITY" in the following pattern (using a "LOOP" control statement):

I

I N

I N F

I N F I

I N F I N

I N F I N I

I N F I N I T

I N F I N I T Y

Certainly! Below is a simple C program that uses a loop control statement to display the string "INFINITY" in the pattern you've described:

```c
#include <stdio.h>

int main() {
    char word[] = "INFINITY";
    int length = sizeof(word) - 1; // Subtract 1 to ignore the null terminator

    for(int i = 0; i < length; i++)
        { for(int j = 0; j <= i; j++)
            {printf("%c ", word[j]);
        }
        printf("\n");
    }

    return 0;
}
```

This program uses a nested loop where the outer loop runs for each character in the string "INFINITY", and the inner loop prints each character up to the current position of the outer loop, creating the desired pattern.

35. What are palindrome numbers & words? Give an example of it in C.

Palindrome numbers and words are sequences that read the same forwards and backwards. For example, the number **121** and the word "level" are palindromes because they remain unchanged when their order is reversed.

Here's an example of how you can check for a palindrome number in C:

```
#include <stdio.h>

int main() {
    int num, reversed = 0, original, remainder;

    printf("Enter an integer: ");
    scanf("%d", &num);

    original = num;

    // Reversing the number
    while (num != 0) {
        remainder = num % 10;
        reversed = reversed * 10 + remainder;
        num /= 10;
    }

    // Palindrome check
    if (original == reversed)
        printf("%d is a palindrome.", original);
    else
        printf("%d is not a palindrome.", original);

    return 0;
}
```

This program takes an integer as input, reverses it, and then checks if the original number and the reversed number are the same. If they are, it means the number is a palindrome.

For palindrome words, you would follow a similar logic but compare characters in a string from the beginning and end, moving towards the center. Would you like to see an example of that as well?

36. Write a C function for sorting techniques: Bubble sort, Quick sort, Insertion sort.

Certainly! Below are the C functions for each of the sorting techniques you've asked for:

Bubble Sort:

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

Quick Sort:

```
void quickSort(int arr[], int low, int high)
    { if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int partition(int arr[], int low, int high)
    { int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high- 1; j++)
        { if (arr[j] < pivot)
            {i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

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

Insertion Sort:

```
void insertionSort(int arr[], int n)
    { int i, key, j;
    for (i = 1; i < n; i++)
        { key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
            { arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```