**PPS IMPORTANT QUESTIONS**

What are storage classes? Explain static storage class with example.

Storage classes in C programming language are used to specify the lifetime, visibility, and storage location of variables. [There are four types of storage classes in C: auto, extern, register, and static](https://www.guru99.com/c-storage-classes.html)[1](https://www.guru99.com/c-storage-classes.html).

[The static storage class is used to declare variables that have a **lifetime throughout the program** and are **not destroyed** even when the function call is over 2](https://www.w3schools.in/c-programming/storage-classes). Here’s an example of how to use the static storage class in C:

#include <stdio.h>

void func() {

static int x = 0;

printf("x is %d\n", x);

x++;

}

int main() {

func();

func();

func();

return 0;

}

In this example, the variable x is declared as static inside the function func(). The first time func() is called, x is initialized to 0. Each time func() is called, the value of x is incremented by 1 and printed to the console. Since x is declared as static, its value is preserved between function calls, and the output of this program is:

x is 0

x is 1

x is 2

What is recursion? WAP to find factorial of a number and Fibonacci series using recursion

**Recursion** is a technique in programming where a function calls itself to solve a problem. [It is a powerful tool that can simplify complex problems and make code more elegant 1](https://www.geeksforgeeks.org/fibonacci-series/).

Here’s an example of how to find the **factorial of a number** using recursion in C:

#include <stdio.h>

int factorial(int n) {

if (n == 0) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

int main() {

int n = 5;

printf("The factorial of %d is %d\n", n, factorial(n));

return 0;

}

In this example, the factorial() function is called recursively to calculate the factorial of a number. If the input number is 0, the function returns 1. Otherwise, it multiplies the input number by the factorial of the number one less than it. The output of this program is:

The factorial of 5 is 120

Here’s an example of how to generate the **Fibonacci series** using recursion in C:

#include <stdio.h>

int fibonacci(int n) {

if (n == 0) {

return 0;

} else if (n == 1) {

return 1;

} else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

int main() {

int n = 10;

printf("The first %d numbers in the Fibonacci series are: ", n);

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

printf("%d ", fibonacci(i));

}

printf("\n");

return 0;

}

In this example, the fibonacci() function is called recursively to generate the Fibonacci series. If the input number is 0, the function returns 0. If the input number is 1, the function returns 1. Otherwise, it returns the sum of the previous two numbers in the series. The output of this program is:

The first 10 numbers in the Fibonacci series are: 0 1 1 2 3 5 8 13 21 34

Difference between pointer to an array and array of pointers

[In C programming, there are two ways to declare pointers that can be used to access arrays: **pointer to an array** and **array of pointers** 1](https://www.geeksforgeeks.org/difference-between-pointer-to-an-array-and-array-of-pointers/).

A **pointer to an array** is a pointer that points to the first element of an array. We can use this pointer to access the elements of the array. Here’s an example of how to declare a pointer to an array in C:

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

int (\*ptr)[5] = &arr;

In this example, ptr is a pointer to an array of 5 integers. We initialize it to the address of the first element of the array arr. We can use the pointer ptr to access the elements of the array arr.

An **array of pointers** is an array where each element is a pointer to another array or variable. Here’s an example of how to declare an array of pointers in C:

int arr1[3] = {1, 2, 3};

int arr2[2] = {4, 5};

int \*arr[2] = {arr1, arr2};

In this example, arr is an array of 2 pointers. The first pointer points to the array arr1, and the second pointer points to the array arr2. We can use these pointers to access the elements of the arrays.

WAP to find whether a number is Armstrong or not.

An **Armstrong number** is a number that is equal to the sum of its digits raised to the power of the number of digits. [For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 153 1](https://www.programiz.com/c-programming/examples/check-armstrong-number).

Here’s an example of how to check whether a number is an Armstrong number or not using recursion in C:

#include <stdio.h>

#include <math.h>

int isArmstrong(int n) {

int num = n;

int sum = 0;

int digits = 0;

while (num > 0) {

digits++;

num /= 10;

}

num = n;

while (num > 0) {

int digit = num % 10;

sum += pow(digit, digits);

num /= 10;

}

return sum == n;

}

int main() {

int n = 153;

if (isArmstrong(n)) {

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

} else {

printf("%d is not an Armstrong number.\n", n);

}

return 0;

}

In this example, the isArmstrong() function is called recursively to check whether a number is an Armstrong number or not. The function first calculates the number of digits in the input number. It then calculates the sum of the digits raised to the power of the number of digits. If the sum is equal to the input number, the function returns 1, indicating that the number is an Armstrong number. Otherwise, it returns 0.

To check whether a number is an Armstrong number, you can call the isArmstrong() function with the input number as an argument. If the function returns 1, the number is an Armstrong number. Otherwise, it is not.

Differentiate between structure and union.

[In C programming, **structures** and **unions** are two user-defined data types that can be used to store data of different types as a single unit](https://www.geeksforgeeks.org/structure-vs-union-in-c/) [1](https://www.geeksforgeeks.org/structure-vs-union-in-c/)[2](https://www.upgrad.com/tutorials/software-engineering/software-key-tutorial/difference-between-structure-and-union/).

A **structure** is a collection of variables of different data types that are grouped together under a single name. Each variable in the structure is called a member. We can access the members of a structure using the . operator. Here’s an example of how to declare a structure in C:

struct person {

char name[50];

int age;

float height;

};

In this example, we define a structure called person that has three members: name, age, and height. We can create variables of this structure type and access its members using the . operator.

A **union** is a special data type that allows storing different data types in the same memory location. Unlike structures, unions can only store one value at a time. We can access the members of a union using the . operator. Here’s an example of how to declare a union in C:

union data {

int i;

float f;

char str[20];

};

In this example, we define a union called data that has three members: i, f, and str. Since a union can only store one value at a time, the size of the union is determined by the size of its largest member. We can create variables of this union type and access its members using the . operator.

The main difference between a structure and a union is that a structure allocates memory for all its members, whereas a union allocates memory for only one member at a time. [This makes unions more memory-efficient than structures when we need to store only one value at a time 1](https://www.geeksforgeeks.org/structure-vs-union-in-c/)[2](https://www.upgrad.com/tutorials/software-engineering/software-key-tutorial/difference-between-structure-and-union/).

WAP to find whether a string is palindrome or not.

Here’s an example of how to check whether a string is a **palindrome** or not using recursion in C:

#include <stdio.h>

#include <string.h>

int isPalindrome(char \*str, int start, int end) {

if (start >= end) {

return 1;

} else if (str[start] != str[end]) {

return 0;

} else {

return isPalindrome(str, start + 1, end - 1);

}

}

int main() {

char str[100];

printf("Enter a string: ");

scanf("%s", str);

int len = strlen(str);

if (isPalindrome(str, 0, len - 1)) {

printf("%s is a palindrome.\n", str);

} else {

printf("%s is not a palindrome.\n", str);

}

return 0;

}

In this example, the isPalindrome() function is called recursively to check whether a string is a palindrome or not. The function takes three arguments: the string str, the starting index start, and the ending index end. If the starting index is greater than or equal to the ending index, the function returns 1, indicating that the string is a palindrome. If the characters at the starting and ending indices are not equal, the function returns 0, indicating that the string is not a palindrome. Otherwise, it calls itself with the starting index incremented by 1 and the ending index decremented by 1.

To check whether a string is a palindrome, you can call the isPalindrome() function with the input string and its starting and ending indices as arguments. If the function returns 1, the string is a palindrome. Otherwise, it is not.

WAP to display reverse of a number.

Here’s an example of how to display the **reverse of a number** using recursion in C:

#include <stdio.h>

int reverse(int n) {

if (n == 0) {

return 0;

} else {

printf("%d", n % 10);

return reverse(n / 10);

}

}

int main() {

int n = 12345;

printf("The reverse of %d is ", n);

reverse(n);

printf("\n");

return 0;

}

In this example, the reverse() function is called recursively to display the reverse of a number. If the input number is 0, the function returns 0. Otherwise, it prints the last digit of the input number and calls itself with the input number divided by 10. The output of this program is:

The reverse of 12345 is 54321

What are header files? What are its types? WAP to implement a header file in a program.

[In C programming, a **header file** is a file that contains function declarations, data type definitions, and macro definitions that can be shared between several source files 1](https://www.geeksforgeeks.org/header-files-in-c-cpp-and-its-uses/)[2](https://www.geeksforgeeks.org/header-files-in-c-c-with-examples/).

[There are two types of header files in C: **standard header files** and **user-defined header files** 2](https://www.geeksforgeeks.org/header-files-in-c-c-with-examples/).

**Standard header files** contain the libraries defined in the ISO standard of the C programming language. They are stored in the default directory of the compiler and are present in all the C compilers from any vendor. Some commonly used standard header files in C are:

* stdio.h: used for input and output operations using functions like scanf() and printf().
* stdlib.h: used for performing general-purpose functions like memory allocation, process control, etc.
* string.h: used for performing various functionalities related to string manipulation like strlen(), strcmp(), strcpy(), etc.
* math.h: used for performing mathematical operations like sqrt(), log2(), pow(), etc.

**User-defined header files** are files that are defined by the user and can be imported using the #include directive. They can be created by writing C/C++ code and saving it with the .h extension. Here’s an example of how to create and use a user-defined header file in C:

// sum.h

int sum(int a, int b);

// sum.c

#include "sum.h"

int sum(int a, int b) {

return a + b;

}

// main.c

#include <stdio.h>

#include "sum.h"

int main() {

int a = 5, b = 10;

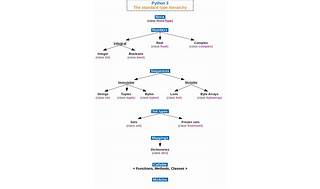
printf("The sum of %d and %d is %d\n", a, b, sum(a, b));

return 0;

}

In this example, we define a function sum() in the file sum.c and declare its prototype in the file sum.h. We then include the header file sum.h in the file main.c and call the sum() function to calculate the sum of two numbers.

What are data types? Explain its types with ranges.





Explore

[In C programming, a **data type** is a classification of data that specifies the type of operations that can be performed on it and the amount of memory required to store it](https://www.geeksforgeeks.org/data-types-in-c/) [1](https://www.geeksforgeeks.org/data-types-in-c/).

[There are two types of data types in C: **primitive data types** and **derived data types**](https://www.geeksforgeeks.org/data-types-in-c/) [1](https://www.geeksforgeeks.org/data-types-in-c/)[2](https://stackoverflow.com/questions/944224/definition-of-range-of-a-data-type).

**Primitive data types** are the most basic data types in C. They are used to represent simple values such as integers, floating-point numbers, and characters. The following table shows the primitive data types in C, along with their size, range, and format specifier:

| **Data Type** | **Size (bytes)** | **Range** | **Format Specifier** |
| --- | --- | --- | --- |
| char | 1 | -128 to 127 or 0 to 255 | %c |
| int | 4 | -2,147,483,648 to 2,147,483,647 | %d |
| short | 2 | -32,768 to 32,767 | %hd |
| long | 4 | -2,147,483,648 to 2,147,483,647 | %ld |
| float | 4 | 1.2E-38 to 3.4E+38 | %f |
| double | 8 | 2.3E-308 to 1.7E+308 | %lf |
| long double | 16 | 3.4E-4932 to 1.1E+4932 | %Lf |

**Derived data types** are data types that are derived from the primitive data types. They include arrays, structures, and pointers.

Explain the types of errors with example.

In programming, there are several types of errors that can occur. Here are some of the most common types of errors with examples:

1. **Syntax errors**: These are errors that occur when the code violates the rules of the programming language. For example, forgetting to close a bracket or semicolon can cause a syntax error. Here’s an example of a syntax error in C:

#include <stdio.h>

int main() {

printf("Hello, world!")

return 0;

}

In this example, the semicolon is missing at the end of the printf() statement, which causes a syntax error.

1. **Runtime errors**: These are errors that occur during the execution of the program. For example, dividing by zero or accessing an array out of bounds can cause a runtime error. Here’s an example of a runtime error in C:

#include <stdio.h>

int main() {

int a = 5, b = 0;

int c = a / b;

printf("The result is %d\n", c);

return 0;

}

In this example, the program tries to divide a by b, which is zero. This causes a runtime error.

1. **Logic errors**: These are errors that occur when the program does not produce the expected output due to incorrect logic. For example, using the wrong formula to calculate a value can cause a logic error. Here’s an example of a logic error in C:

#include <stdio.h>

int main() {

int a = 5, b = 10;

int c = a + b \* 2;

printf("The result is %d\n", c);

return 0;

}

In this example, the program tries to calculate a plus b times 2. However, due to operator precedence rules, b is multiplied by 2 first, and then added to a. This causes a logic error.

What are features of an algorithm? Differentiate between algorithm and flowchart



Explore

An **algorithm** is a set of instructions that describe how to solve a problem or perform a task. The following are some of the features of an algorithm:

1. **Finiteness**: An algorithm must terminate after a finite number of steps.
2. **Definiteness**: Each step of an algorithm must be precisely defined and unambiguous.
3. **Input**: An algorithm must have zero or more inputs.
4. **Output**: An algorithm must have one or more outputs.
5. **Effectiveness**: Each step of an algorithm must be simple enough to be carried out by a computer.
6. **Generality**: An algorithm must be applicable to a wide range of inputs.

A **flowchart** is a graphical representation of an algorithm. It uses symbols and arrows to depict the flow of information and processing. The process of drawing a flowchart for an algorithm is known as “flowcharting”. Here are some differences between an algorithm and a flowchart:

1. **Definition**: An algorithm is a set of instructions that describe how to solve a problem or perform a task, while a flowchart is a graphical representation of an algorithm.
2. **Representation**: An algorithm can be represented in natural language, pseudocode, or programming languages, while a flowchart uses standardized symbols and arrows to depict control flow.
3. **Complexity**: An algorithm can be complex to understand, while a flowchart is easy to understand.
4. **Debugging**: An algorithm is easy to debug, while a flowchart is hard to debug.
5. **Construction**: An algorithm is difficult to construct, while a flowchart is simple to construct.
6. **Rules**: An algorithm does not follow any rules, while a flowchart follows rules to be constructed.
7. **Pseudo-code**: An algorithm is the pseudo-code for the program, while a flowchart is just a graphical representation of that logic.

what are control flow statements? Explain in brief.

In programming, **control flow statements** are used to control the order in which statements are executed in a program. They allow us to make decisions, repeat blocks of code, and break out of loops. Here are some examples of control flow statements in C:

1. **Conditional statements**: These statements allow us to execute different blocks of code depending on whether a given condition is true or false. The most common conditional statement is the if statement, which executes a block of code if a condition is true. Here’s an example:

if (x > 0) {

printf("x is positive\n");

} else {

printf("x is non-positive\n");

}

1. **Loop statements**: These statements allow us to repeat a block of code multiple times. The most common loop statements are the for loop, which repeats a block of code a fixed number of times, and the while loop, which repeats a block of code as long as a condition is true. Here’s an example of a for loop:

for (int i = 0; i < 10; i++) {

printf("%d\n", i);

}

1. **Jump statements**: These statements allow us to break out of loops or skip over blocks of code. The most common jump statements are the break statement, which breaks out of a loop, and the continue statement, which skips over the rest of a loop iteration. Here’s an example of a break statement:

for (int i = 0; i < 10; i++) {

if (i == 5) {

break;

}

printf("%d\n", i);

}

In this example, the loop will terminate when i is equal to 5.

WAP to display grades of students using switch case. Input- Marks of 5 subjects

Here’s an example of how to display the grades of students using **switch case** in C:

#include <stdio.h>

int main() {

int marks[5];

int total = 0;

float average;

char grade;

printf("Enter the marks of 5 subjects:\n");

for (int i = 0; i < 5; i++) {

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

total += marks[i];

}

average = (float) total / 5.0;

switch ((int) average / 10) {

case 10:

case 9:

grade = 'A';

break;

case 8:

grade = 'B';

break;

case 7:

grade = 'C';

break;

case 6:

grade = 'D';

break;

case 5:

case 4:

grade = 'E';

break;

default:

grade = 'F';

break;

}

printf("The grade of the student is %c\n", grade);

return 0;

}

In this example, the program prompts the user to enter the marks of 5 subjects. It then calculates the total marks and the average marks. Finally, it uses a switch case statement to determine the grade of the student based on the average marks.

The grading system used in this example is:

| **Score** | **Grade** |
| --- | --- |
| >= 90 | A |
| 80-89 | B |
| 70-79 | C |
| 60-69 | D |
| 50-59 | E |
| < 50 | F |

What are pre-processor directives? What are its types? WAP to find area of a rectangle using Macros.

In C programming, **preprocessor directives** are special instructions that are processed by the preprocessor before the actual compilation of the program begins. [They are used to define macros, include header files, and conditionally compile code based on certain conditions](https://www.geeksforgeeks.org/cc-preprocessors/) [1](https://www.geeksforgeeks.org/cc-preprocessors/).

There are several types of preprocessor directives in C, including:

1. **Macro substitution directives**: These directives are used to define and undefine macros. The #define directive is used to define a macro, while the #undef directive is used to undefine a macro.
2. **File inclusion directives**: These directives are used to include header files in the source code program. The #include directive is used to include a file in the source code program.
3. **Conditional compilation directives**: These directives are used to include or exclude sections of code based on certain conditions. The #ifdef directive is used to include a section of code if a certain macro is defined by #define, while the #ifndef directive is used to include a section of code if a certain macro is not defined by #define. The #if directive is used to check for a specified condition, while the #else directive is used to include alternate code that executes when #if fails. The #endif directive is used to mark the end of #if, #ifdef, and #ifndef.

Here’s an example of how to find the area of a rectangle using **macros** in C:

#include <stdio.h>

#define AREA(l, b) (l \* b)

int main() {

int l = 10, b = 5;

int area = AREA(l, b);

printf("The area of the rectangle is %d\n", area);

return 0;

}

In this example, the #define directive is used to define a macro called AREA that calculates the area of a rectangle. The macro takes two arguments, l and b, which represent the length and breadth of the rectangle, respectively. The macro definition is (l \* b), which multiplies the length and breadth to calculate the area of the rectangle. The AREA macro is then used to calculate the area of a rectangle with length 10 and breadth 5. The output of this program is:

The area of the rectangle is 50

how do you declare and initialize an array? WAP to implement 2D matrix multiplication

To declare and initialize an array in C, you can use the following syntax:

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

In this example, we declare an integer array arr with 5 elements and initialize it with the values {1, 2, 3, 4, 5}.

Here’s an example of how to implement **2D matrix multiplication** in C:

#include <stdio.h>

int main() {

int a[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

int b[3][3] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};

int c[3][3] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

c[i][j] += a[i][k] \* b[k][j];

}

}

}

printf("The product of the two matrices is:\n");

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

printf("%d ", c[i][j]);

}

printf("\n");

}

return 0;

}

In this example, we declare and initialize two 3x3 matrices a and b. We also declare a third 3x3 matrix c to store the product of the two matrices. We then use three nested loops to calculate the product of the two matrices and store the result in c. Finally, we print the product of the two matrices.

What is a string? Explain String Manipulation operations with example

In C programming, a **string** is an array of characters that is terminated by a null character (\0). Strings are used to represent text in a program. Here are some examples of string manipulation operations in C:

1. **String length**: To find the length of a string, you can use the strlen() function. Here’s an example:

#include <stdio.h>

#include <string.h>

int main() {

char str[] = "Hello, world!";

int len = strlen(str);

printf("The length of the string is %d\n", len);

return 0;

}

1. **String copy**: To copy one string to another, you can use the strcpy() function. Here’s an example:

#include <stdio.h>

#include <string.h>

int main() {

char str1[] = "Hello, world!";

char str2[20];

strcpy(str2, str1);

printf("The copied string is %s\n", str2);

return 0;

}

1. **String concatenation**: To concatenate two strings, you can use the strcat() function. Here’s an example:

#include <stdio.h>

#include <string.h>

int main() {

char str1[] = "Hello, ";

char str2[] = "world!";

strcat(str1, str2);

printf("The concatenated string is %s\n", str1);

return 0;

}

1. **String comparison**: To compare two strings, you can use the strcmp() function. Here’s an example:

#include <stdio.h>

#include <string.h>

int main() {

char str1[] = "Hello, world!";

char str2[] = "Hello, world!";

if (strcmp(str1, str2) == 0) {

printf("The strings are equal\n");

} else {

printf("The strings are not equal\n");

}

return 0;

}

Explain call by value and call by reference with the help of example.

In C programming, there are two ways to pass arguments to a function: **call by value** and **call by reference**.

In **call by value**, a copy of the value of the actual parameter is passed to the function. Any changes made to the parameter inside the function do not reflect outside of it. Here’s an example:

#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 example, the swap() function is called with two integer arguments a and b. However, since the arguments are passed by value, the function receives copies of the values of a and b. Any changes made to x and y inside the function do not affect the values of a and b outside the function.

In **call by reference**, the address of the actual parameter is passed to the function as the formal parameter. Both the actual and formal parameters refer to the same memory location. Any changes made inside the function are actually reflected in the actual parameters of the caller. Here’s an example:

#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 example, the swap() function is called with two integer pointers &a and &b. The function receives the addresses of a and b as arguments. Any changes made to \*x and \*y inside the function affect the values of a and b outside the function.

Explain Dynamic Memory Allocation. WAP to allocate space for an int array of size 10.

In C programming, **dynamic memory allocation** is a process of allocating memory during the runtime of a program. [It allows you to allocate memory space as per your program’s requirements, use it for storing data, and release it when it is no longer needed](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/) [1](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/).

There are several library functions provided by C defined under the stdlib.h header file to facilitate dynamic memory allocation in C programming. The most commonly used functions are malloc(), calloc(), realloc(), and free().

Here’s an example of how to allocate space for an int array of size 10 using malloc():

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*arr = (int \*) malloc(10 \* sizeof(int));

if (arr == NULL) {

printf("Memory allocation failed\n");

exit(0);

}

for (int i = 0; i < 10; i++) {

arr[i] = i + 1;

}

printf("The elements of the array are: ");

for (int i = 0; i < 10; i++) {

printf("%d ", arr[i]);

}

free(arr);

return 0;

}

In this example, we use the malloc() function to allocate memory space for an int array of size 10. The malloc() function returns a pointer of type void which is cast into a pointer of type int. If the memory allocation fails, the function returns a NULL pointer. We then use a for loop to initialize the array with values from 1 to 10. Finally, we print the elements of the array and free the allocated memory using the free() function.

Explain the concept of structures. WAP to create a structure of Employees and display their details.

[In C programming, a **structure** is a user-defined data type that groups together variables of different data types under a single name](https://www.geeksforgeeks.org/structures-c/) [1](https://www.geeksforgeeks.org/structures-c/).

Here’s an example of how to create a structure of Employees and display their details:

#include <stdio.h>

struct Employee {

char name[50];

int age;

float salary;

};

int main() {

struct Employee emp;

printf("Enter employee name: ");

scanf("%s", emp.name);

printf("Enter employee age: ");

scanf("%d", &emp.age);

printf("Enter employee salary: ");

scanf("%f", &emp.salary);

printf("Employee details:\n");

printf("Name: %s\n", emp.name);

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

printf("Salary: %.2f\n", emp.salary);

return 0;

}

In this example, we define a structure called Employee that has three members: name, age, and salary. We then declare a variable emp of type Employee. We use the scanf() function to read the values of name, age, and salary from the user. Finally, we print the details of the employee using the printf() function.

what are the various modes of a file? WAP to read and display the content of file on screen.

In C programming, there are several modes in which a file can be opened. The most commonly used modes are:

* r: Read mode. Opens a file for reading. The file must exist.
* w: Write mode. Opens a file for writing. If the file exists, it is truncated. If the file does not exist, it is created.
* a: Append mode. Opens a file for appending. The file pointer is at the end of the file. If the file does not exist, it is created.
* r+: Read and write mode. Opens a file for both reading and writing. The file must exist.
* w+: Write and read mode. Opens a file for both writing and reading. If the file exists, it is truncated. If the file does not exist, it is created.
* a+: Append and read mode. Opens a file for both appending and reading. The file pointer is at the end of the file. If the file does not exist, it is created.

Here’s an example of how to read and display the content of a file on screen:

#include <stdio.h>

int main() {

FILE \*fp;

char ch;

fp = fopen("file.txt", "r");

if (fp == NULL) {

printf("Error opening file\n");

return 1;

}

printf("The contents of the file are:\n");

while ((ch = fgetc(fp)) != EOF) {

putchar(ch);

}

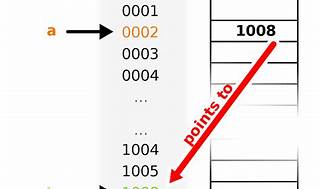
fclose(fp);

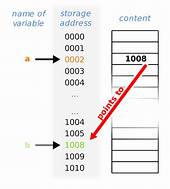
return 0;

}

In this example, we use the fopen() function to open a file called file.txt in read mode. We then use a while loop to read each character from the file using the fgetc() function and display it on the screen using the putchar() function. Finally, we close the file using the fclose() function.

Explain pointers, its types and chain of pointers with the help of program(s).





Explore

In C programming, a **pointer** is a variable that stores the memory address of another variable. Pointers are used to manipulate data stored in memory and to create complex data structures such as linked lists, trees, and graphs.

There are several types of pointers in C, including:

1. **Null pointers**: These are pointers that do not point to any memory location. They are represented by the NULL macro.
2. **Void pointers**: These are pointers that can point to any data type. They are represented by the void \* data type.
3. **Function pointers**: These are pointers that point to functions. They are used to implement callbacks and function pointers.
4. **Array pointers**: These are pointers that point to the first element of an array.
5. **Pointer to pointer**: These are pointers that point to another pointer. They are used to implement complex data structures such as linked lists and trees.

Here’s an example of how to use a chain of pointers in C:

#include <stdio.h>

int main() {

int var = 10;

int \*ptr1;

int \*\*ptr2;

int \*\*\*ptr3;

ptr1 = &var;

ptr2 = &ptr1;

ptr3 = &ptr2;

printf("Value of var = %d\n", var);

printf("Value of var using level-1 pointer = %d\n", \*ptr1);

printf("Value of var using level-2 pointer = %d\n", \*\*ptr2);

printf("Value of var using level-3 pointer = %d\n", \*\*\*ptr3);

return 0;

}

In this example, we declare three pointers ptr1, ptr2, and ptr3. We then initialize them to point to the memory location of the variable var. We use the \* and \*\* operators to dereference the pointers and access the value of var.

WAP to find Fibonacci series upto n numbers.

Here’s an example of how to find the **Fibonacci series** up to n numbers in C:

#include <stdio.h>

int main() {

int n, t1 = 0, t2 = 1, nextTerm;

printf("Enter the number of terms: ");

scanf("%d", &n);

printf("Fibonacci Series: ");

for (int i = 1; i <= n; ++i) {

printf("%d, ", t1);

nextTerm = t1 + t2;

t1 = t2;

t2 = nextTerm;

}

return 0;

}

In this example, we use a for loop to print the Fibonacci series up to n numbers. We initialize two variables t1 and t2 to 0 and 1, respectively. We then use a for loop to calculate the next term in the series by adding t1 and t2. We then update the values of t1 and t2 to the next two terms in the series.

Explain opertors in detail

In C programming, **operators** are symbols that represent operations to be performed on one or more operands. They are the basic components of the C programming. Operators in C can be defined as the symbol that helps us to perform some specific mathematical, relational, bitwise, conditional, or logical computations on values and variables. The values and variables used with operators are called operands. So we can say that the operators are the symbols that perform operations on operands.

C language provides a wide range of operators that can be classified into 6 types based on their functionality:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators
7. **Arithmetic Operators** are used to perform arithmetic/mathematical operations on operands. There are 9 arithmetic operators in C language:

| **Symbol** | **Operator** | **Description** | **Syntax** |
| --- | --- | --- | --- |
| + | Plus | Adds two numeric values. | a + b |
| - | Minus | Subtracts right operand from left operand. | a - b |
| \* | Multiply | Multiply two numeric values. | a \* b |
| / | Divide | Divide two numeric values. | a / b |
| % | Modulus | Returns the remainder after diving the left operand with the right operand. | a % b |
| + | Unary Plus | Used to specify the positive values. | +a |
| - | Unary Minus | Flips the sign of the value. | -a |
| ++ | Increment | Increases the value of the operand by 1. | a++ |
| – | Decrement | Decreases the value of the operand by 1. | a– |

1. **Relational Operators** are used for the comparison of the two operands. All these operators are binary operators that return true or false values as the result of comparison. These are a total of 6 relational operators in C:

| **Symbol** | **Operator** | **Description** | **Syntax** |
| --- | --- | --- | --- |
| < | Less than | Returns true if the left operand is less than the right operand. Else false. | a < b |
| > | Greater than | Returns true if the left operand is greater than the right operand. Else false. | a > b |
| <= | Less than or equal to | Returns true if the left operand is less than or equal to the right operand. Else false. | a <= b |
| >= | Greater than or equal to | Returns true if the left operand is greater than or equal to right operand. Else false. | a >= b |
| == | Equal to | Returns true if both the operands are equal. | a == b |
| != | Not equal to | Returns true if both the operands are NOT equal. | a != b |

1. **Logical Operators** are used to perform logical operations on two or more operands. There are three logical operators in C language:

| **Symbol** | **Operator** | **Description** | **Syntax** |
| --- | --- | --- | --- |
| && | Logical AND | Returns true if both the operands are true. Else false. | a && b |
| || | Logical OR | Returns true if either of the operands is true. Else false. | a || b |
| ! | Logical NOT | Returns true if the operand is false. Else false. | !a |

1. **Bitwise Operators** are used to perform bitwise operations on two operands. There are six bitwise operators in C language:

| **Symbol** | **Operator** | **Description** | **Syntax** |
| --- | --- | --- | --- |
| & | Bitwise AND | Performs bitwise AND operation on two operands. | a & b |
| | | Bitwise OR | Performs bitwise OR operation on two operands. | a | b |
| ^ | Bitwise XOR | Performs bitwise XOR operation on two operands. | a ^ b |
| ~ | Bitwise NOT | Performs bitwise NOT operation on an operand. | ~a |
| << | Left Shift | Shifts the bits of the left operand to the left by the number of positions specified by the right operand. | a << b |
| >> | Right Shift | Shifts the bits of the left operand to the right by the number of positions specified by the right operand. | a >> b |

1. **Assignment Operators** are used to assign specific values to variables. There are 12 assignment operators in C language:

| **Symbol** | **Operator** | **Description** | **Syntax** |
| --- | --- | --- | --- |
| = | Simple Assignment | Assign |  |

What are branch statements? Explain all.

In programming, **branch statements** are used to change the flow of execution of a program. They allow the program to jump to a different part of the code based on certain conditions. There are several types of branch statements in C programming:

1. **if statement**: The if statement is used to execute a block of code if a certain condition is true. Here’s an example:

if (x > 0) {

printf("x is positive\n");

}

1. **if-else statement**: The if-else statement is used to execute one block of code if a certain condition is true, and another block of code if the condition is false. Here’s an example:

if (x > 0) {

printf("x is positive\n");

} else {

printf("x is non-positive\n");

}

1. **nested if-else statement**: The nested if-else statement is used to execute a block of code if a certain condition is true, and another block of code if the condition is false. Here’s an example:

if (x > 0) {

if (x < 10) {

printf("x is between 0 and 10\n");

} else {

printf("x is greater than or equal to 10\n");

}

} else {

printf("x is non-positive\n");

}

1. **switch statement**: The switch statement is used to execute a block of code based on the value of a variable. Here’s an example:

switch (x) {

case 1:

printf("x is 1\n");

break;

case 2:

printf("x is 2\n");

break;

default:

printf("x is neither 1 nor 2\n");

break;

}

1. **break statement**: The break statement is used to exit a loop or switch statement. Here’s an example:

for (int i = 0; i < 10; i++) {

if (i == 5) {

break;

}

printf("%d\n", i);

}

1. **continue statement**: The continue statement is used to skip the current iteration of a loop. Here’s an example:

for (int i = 0; i < 10; i++) {

if (i == 5) {

continue;

}

printf("%d\n", i);

}

Differentiate between:

A.Structure and union

* A structure is a user-defined data type that groups together variables of different data types under a single name.
* A union is a special data type that allows storing different data types in the same memory location.
* Structures allocate separate memory locations for each member, while unions share a single memory location among multiple members.

B.While and do while

* The while loop checks the condition before executing the statement(s), whereas the do-while loop executes the statement(s) at least once before checking the condition.
* The while loop is entry-controlled, while the do-while loop is exit-controlled

C.Source code object code

* Source code is the human-readable version of a program, while object code is the machine-readable version.
* Source code is easier to read, edit, and debug, while object code is more secure and efficient for execution.

D.calloc and malloc

* malloc() is a function that creates one block of memory of a fixed size. It does not initialize the allocated memory.
* calloc() is a function that assigns a specified number of blocks of memory to a single variable. It initializes every byte in the allocated memory to 0

E.algorithm and flowchart

* An algorithm is a set of rules/instructions that step-by-step define how a work is to be executed in order to get the expected results.
* A flowchart is a diagrammatic representation of an algorithm. It makes use of symbols that are connected among them to indicate the flow of information and processing

F.constant and variable

* A constant is a value that cannot be changed during the execution of a program.
* A variable is a value that can be changed during the execution of a program

G.local and global variable

* A local variable is a variable that is declared inside a function or block and can only be accessed within that function or block.
* A global variable is a variable that is declared outside of any function or block and can be accessed from anywhere in the program