Shastri 5th Semester

Computer Science

Unit: 3rd

Advanced features of 'C' Language

C language is a powerful, general-purpose programming language that has been widely used for over four decades. Some of the advanced features of C language include:

- **Pointers:** C language supports pointers, which are variables that hold the memory address of another variable. Pointers allow for powerful and efficient memory management and are used in many advanced features of C language, such as dynamic memory allocation and function pointers.
- **Dynamic memory allocation:** C language supports the allocation of memory at runtime using functions such as **malloc()** and **calloc()**. This feature allows for the creation of variable-size data structures, such as linked lists and trees.
- **Bit manipulation:** C language provides bitwise operators that allow for the manipulation of individual bits in a variable. This feature can be used for efficient low-level programming and for working with binary data.
- **Preprocessor directives:** C language has a powerful preprocessor that can be used to include header files, define macros, and perform conditional compilation. This feature allows for more flexible and modular code organization.
- **Function pointers:** C language supports function pointers, which are pointers that hold the memory address of a function. Function pointers can be used to create callback functions and to implement function polymorphism.
- **Structs and Unions:** C language provides structs, which allow for the creation of user-defined data types, and unions, which allow for the use of the same memory location for multiple data types.
- **File handling:** C language provides a set of functions for performing basic file operations such as opening, reading, writing, and closing files.

- **Multi-threading:** C language provides a set of functions to create and manage multiple threads of execution in a program, which allows for concurrent processing and can improve the performance of certain types of applications.
- **Standard Template Library (STL):** C++ which is an extension of C language provides STL which is a collection of template classes and functions that implement common data structures and algorithms.

These features make C language a versatile and powerful language that can be used to create a wide range of applications, from low-level system software to high-performance applications.

C language Sorting

Sorting is the process of arranging a collection of elements in a specific order. In C language, sorting an array of elements can be achieved using different algorithms, such as:

- **Bubble sort:** this algorithm repeatedly iterates through the array, comparing adjacent elements and swapping them if they are in the wrong order.
- **selection sort:** this algorithm divides the array into two parts, a sorted part and an unsorted part. In each iteration, the smallest element in the unsorted part is found and moved to the end of the sorted part.
- **insertion sort:** this algorithm starts with an empty sorted part and repeatedly selects an element from the unsorted part and inserts it into the correct position in the sorted part.
- **merge sort:** this algorithm divides the array into two equal parts, sorts each part separately, and then merges the two sorted parts together.
- **quick sort:** this algorithm selects a pivot element from the array and partition the other elements into two groups, one group with elements smaller than the pivot and one group with elements larger than the pivot.

These are some of the most common sorting algorithms used in C language, each one with its own set of advantages and disadvantages. Some sorting algorithms are efficient for small arrays, while others are efficient for large arrays. Some sorting algorithms have a simple implementation, while others are more complex.

It's worth noting that most of the time, the complexity of a sorting algorithm is measured in terms of "n", which represents the number of elements in the array. The most common complexities are O(n^2) and O(nlogn)

Algorithms for Sorting Methods in C Language:

BUBBLE SORT:

Example

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

In this algorithm, we use nested loops to iterate through the array. In each iteration of the outer loop, the inner loop compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the array is sorted.

SELECTION SORT:

Example

```
void selectionSort(int arr[], int n) {
   for (int i = 0; i < n-1; i++) {
```

```
        int minIndex = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

In this algorithm, we use nested loops to iterate through the array. In each iteration of the outer loop, the inner loop finds the index of the smallest element in the unsorted part of the array. Then, it swaps the element at the current index with the smallest element. This process is repeated until the array is sorted.

INSERTION SORT:

Example

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

```
        arr[j+1] = key;
    }
}
```

In this algorithm, we use a nested loop to iterate through the array. In each iteration of the outer loop, the inner loop starts at the current index and compares the element with the previous element. If the element is smaller, it is shifted to the left. This process is repeated until the element is in its correct position in the sorted part of the array.

C LANGUAGE: SEARCHING METHODS

Searching is the process of finding an element within a collection of elements. In C language, searching an array of elements can be achieved using different algorithms, such as:

Linear search: This algorithm iterates through the array sequentially and compares each element with the target element. It returns the index of the first occurrence of the target element or -1 if the element is not found.

Example

```
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Binary search: This algorithm is used on a sorted array and it repeatedly divides the search interval in half. It compares the middle element of the search interval with the target element, if the target is smaller than the middle element, the search continues in the lower half of the array, if the target is larger than the middle element,

the search continues in the upper half of the array. This process is repeated until the target element is found or the search interval is empty.

Example

```
int binarySearch(int arr[], int n, int target) {
    int left = 0;
    int right = n-1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Jump search: This algorithm is used on a sorted array, it works by first jumping over a certain number of elements, then checking each element in the next block of elements one by one. The number of elements jumped over each time is called the jump size. This process is repeated until the target element is found or the search interval is empty.

Example

```
int jumpSearch(int arr[], int n, int target) {
    int jumpSize = sqrt(n);
```

```
int left = 0;
int right = jumpSize;
while (right < n && arr[right] < target) {
   left = right;
   right += jumpSize;
}
for (int i = left; i <= right && i < n; i++) {
   if (arr[i] == target) {
      return i;
   }
}
```

The top-down approach of problem-solving

The top-down approach of problem-solving is a method of breaking down a complex problem into smaller and simpler sub-problems. The approach starts with understanding the overall problem and then dividing it into smaller, manageable sub-problems that can be solved independently.

The steps involved in the top-down approach of problem-solving are:

1. Understand the problem: The first step is to understand the problem statement and the requirements of the problem.
2. Break the problem down: Once the problem is understood, it is broken down into smaller sub-problems. The sub-problems should be independent and should be solvable with the knowledge and resources available.
3. Solve the sub-problems: The next step is to solve the sub-problems one by one, starting with the simplest ones.
4. Combine the solutions: The solutions of the sub-problems are combined to form a solution for the overall problem.
5. Test and verify: The final solution is tested and verified to ensure that it meets the requirements and solves the problem.

This approach allows the problem to be broken down into smaller, manageable pieces, making it easier to understand, solve, and implement. It also allows for a step-by-step development of the solution, making it easier to track progress and identify errors.

It's worth noting that, this approach is also known as the "divide and conquer" approach, because it involves breaking down the problem into smaller sub-problems, solving them independently and then combining the solutions. It's also worth noting that, the top-down approach is used in many programming languages, such as C, C++, JAVA, and Python, to design the program structure, algorithms, and functions.

Modular programming and functions

Modular programming is a software design technique that involves breaking a large, complex program into smaller, independent, and reusable modules or components. Each module contains a specific set of functions that perform a specific task or set of tasks.

Functions are a fundamental building block in modular programming. A function is a self-contained block of code that performs a specific task. Functions in C language have a specific syntax, which includes the function name, a list of parameters, and a block of statements that are executed when the function is called.

Functions have several advantages in modular programmings, such as:

- Code reusability: Functions can be reused in multiple parts of the program, reducing the amount of code that needs to be written and maintained.
- Code organization: Functions make it easier to organize and structure the code, making it more readable and maintainable.
- Code testing and debugging: Functions can be tested and debugged independently, making it easier to identify and fix errors in the program.
- Abstraction: Functions hide the implementation details and provide a clear interface for interacting with the module, this allows for more flexibility in the design and implementation of the program.

Modular programming and functions work together to make the code more manageable and maintainable. Functions make it easy to divide the code into smaller, more manageable pieces, and modular programming makes it easy to

organize and reuse those pieces. This makes it easier to understand the program, identify errors, and make changes as needed.

Standard Library of C functions

The standard library of C functions is a collection of pre-written functions that are included with the C programming language. These functions are provided by the C standard library and are available to all C programs. The standard library functions perform a wide variety of tasks, such as input and output, memory allocation, string manipulation, and math operations.

Some of the most commonly used functions in the C standard library include:

- Input and output functions: scanf() and printf() for reading and writing to the console, fopen() and fclose() for working with files.
- Memory management functions: malloc() and calloc() for dynamic memory allocation, free() for deallocating memory.
- String manipulation functions: strlen() for finding the length of a string, strcpy() for copying a string, strcmp() for comparing two strings.
- Math functions: sqrt() for finding the square root of a number, sin() and cos() for trigonometric operations, pow() for raising a number to a power.
- Time and date functions: time() and ctime() for working with time and date values.
- Type conversion functions: atoi() for converting a string to an integer, itoa() for converting an integer to a string.

These functions are a part of the C standard library and they are defined in the standard headers such as <stdio.h>, <stdlib.h>, <string.h>, <math.h> and <time.h> . The C standard library provides a wide range of functions that can be used to perform a variety of tasks, making it easier to write efficient and effective C programs.

PROTOTYPE OF A FUNCTION

A function prototype is a declaration of a function that specifies the function's return type, name, and parameter list. In C language, function prototypes are used to inform

the compiler about the function's interface, including the number and types of parameters it expects and the type of value it returns.

The basic syntax of a function prototype in C language is:

Example

return_type function_name(parameter_type parameter_name, parameter_type parameter_name, ...);

For example, the prototype of a function that takes two integers and returns their sum would be:

Example

int add(int a, int b);

Function prototypes are typically placed in a header file, which can be included in multiple source files. This allows the compiler to check the function calls against the prototypes and ensure that the correct number and types of arguments are passed to the function.

In C language, a function prototype is not mandatory but it's a good practice to use a function prototype. It allows the compiler to check the function calls against the prototypes and ensures that the correct number and types of arguments are passed to the function, also it allows the compiler to check for type mismatch and other errors before the program is executed.

Formal parameter list, Return Type, Function call

A formal parameter list is a list of parameters that a function expects to receive when it is called. It is specified in the function prototype or definition. The formal parameter list includes the type and name of each parameter, and it is used by the compiler to check that the function is called with the correct number and types of arguments.

The return type of a function is the type of value that the function returns when it is called. It is specified in the function prototype or definition. The return type can be any valid C data type, such as int, float, char, or void. If the function does not return a value, the return type is void.

A function call is the act of invoking a function and passing it the necessary arguments. In C language, a function is called by specifying the function name followed by a set of parentheses enclosing the arguments. The number and types of arguments must match the formal parameter list of the function.

For example, given the following function prototype:

Example

```
int add(int a, int b);
```

A call to this function would look like this:

Example

```
int result = add(3, 5);
```

In this example, the function is called with the arguments 3 and 5, which match the formal parameter list of the function (two integers). The function returns an integer, which is stored in the variable 'result'.

Function Block Structure

The function block structure is the structure of a function in C language, which consists of several elements:

1. **Function header:** This includes the return type, function name, and formal parameter list. The return type is the data type of the value that the function returns. The function name is a unique identifier that is used to call the function. The formal parameter list specifies the types and names of the arguments that the function expects to receive when it is called.
2. **Local variable declarations:** These are variables that are declared within the function and can only be accessed within the function. They are used to store temporary values and intermediate results.
3. **Function body:** This is the main block of statements that are executed when the function is called. It contains the code that performs the specific task of the function.
4. **Return statement:** This is an optional statement that is used to return a value from the function. If the return type of the function is void, the return statement is not used.

Here is an example of a function block structure in C:

Example

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

This function takes two integers as input and returns their sum. The function header specifies the return type (int), function name (add), and a formal parameter list (two integers, 'a' and 'b'). The local variable 'result' is declared inside the function, the function body contains the code that performs the addition, and the return statement returns the value of 'result' as the output of the function.

It's worth noting that, the function block structure is a fundamental concept in C language, it allows to structure of the code in a more readable and maintainable way. Functions are used to divide the code into smaller, more manageable pieces, and the function block structure makes it easy to organize and reuse those pieces.

Passing arguments to a Function argument.

In C language, arguments can be passed to a function in two ways: by value and by reference.

Pass by value: In this method, the actual value of the argument is passed to the function. The function receives a copy of the argument, so any changes made to the argument within the function do not affect the original value of the argument. In C, all basic data types such as int, float, and char are passed by value.

Example

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

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

Pass by reference: In this method, the memory address of the argument is passed to the function. The function receives a pointer to the argument, so any changes made to the argument within the function affect the original value of the argument. In C, this is achieved by passing a pointer to the variable.

Example

```
void increment(int *x) {
   (*x)++;
}
int main() {
   int a = 5;
   increment(&a);
   printf("%d", a); // prints 6
}
```

Passing by reference allows a function to modify the value of an argument, passing by value does not, it's a way to obtain a side effect.

The choice between passing by value and passing by reference depends on the desired behavior of the function. If the function is supposed to modify the value of an argument, passing by reference is the best option, otherwise, passing by value is the best option.

Q: What is a function in C?

A: In C, a function is a block of code that performs a specific task. A function can take zero or more inputs (known as parameters) and can return zero or one output (known as the return value). Functions allow you to reuse code, making your program more organized and easier to read and maintain.

For example, the following code defines a function called "add" that takes two integers as parameters and returns the sum of them:

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

Q: How can we call a function in C?

A: To call a function in C, you need to use its name followed by a pair of parentheses, and pass any necessary parameters inside the parentheses.

For example, the following code calls the "add" function and assigns the return value to a variable called "result":

```
int result = add(3, 4);
```

You can also call a function without assigning its return value to a variable, for example:

```
add(3,4);
```

It's worth noting that a function can be called multiple times with different parameters, and it will execute the code block inside the function every time it's called.

Q: What is the difference between a function prototype and a function definition?

A function prototype is a declaration of a function that provides the compiler with information about the function's name, return type, and the number and types of parameters. A function definition, on the other hand, includes the function prototype and the actual code that gets executed when the function is called.

For example, the following code defines a function prototype for the "add" function:

```
int add(int a, int b);
```

And the following code defines the function definition:

Example

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

function prototypes should be placed in a header file and included in the source code files that call the function, this will allow the compiler to check for any errors related to the function call before the execution.

Q: What is a conditional statement in C?

A: In C, a conditional statement is used to control the flow of a program based on a certain condition. The most common conditional statements are if-else and switch-case statements.

For example, the following code uses an if-else statement to check if a variable called "x" is greater than 10. If it is, the code inside the if block is executed. If it isn't, the code inside the else block is executed:

Example

```
if (x > 10) {
   printf("x is greater than 10\n");
} else {
   printf("x is less than or equal to 10\n");
}
```

1. What is the difference between bubble sort and insertion sort?
2. How do you implement a linear search in C?
3. What is the time complexity of quick sort?
4. How do you sort an array of integers using selection sort in C?
5. What is the advantage of using binary search over linear search?

6. How do you implement a merge sort algorithm in C?
7. What is the difference between stable and unstable sorting algorithms?
8. How do you implement a heap sort algorithm in C?
9. What is the time complexity of insertion sort?
10. How do you sort an array of strings using bubble sort in C?
11. What is the difference between quicksort and heapsort?
12. How do you implement a radix sort algorithm in C?
13. What is the time complexity of selection sort?
14. How do you sort an array of structures using insertion sort in C?
15. What is the advantage of using quick sort over merge sort?
16. How do you implement a counting sort algorithm in C?
17. What is the time complexity of bubble sort?
18. How do you sort an array of doubles using quicksort in C?
19. What is the difference between internal and external sorting algorithms?
20. How do you implement a shell sort algorithm in C?
21. What is the purpose of a function in C?
22. How do you define a function in C?
23. What is the difference between a function declaration and a function definition in C?
24. How do you call a function in C?
25. What is the use of the return statement in a function in C?
26. How do you pass arguments to a function in C?
27. What is the difference between call by value and call by reference in C?
28. How do you return multiple values from a function in C?
29. What is the purpose of the main function in C?
30. How do you use function pointers in C?

31. What is the difference between a library function and a user-defined function in C?
32. How do you use recursion in C functions?
33. What is the difference between a static function and a non-static function in C?
34. How do you use variable-length arguments in a function in C?
35. What is the purpose of the void keyword in a function in C?
36. How do you use the inline keyword for a function in C?
37. What is the difference between a global function and a local function in C?
38. How do you use the const keyword in a function in C?
39. What is the purpose of the extern keyword in a function in C?
40. How do you use the register keyword for a function in C?