SDM College of Engineering and Technology

Dhavalagiri, Dharwad-580 002. Karnataka State. India.

Email: principal@sdmcet.ac.in, cse.sdmcet@gmail.com

Ph: 0836-2447465/ 2448327 Fax: 0836-2464638 Website: sdmcet.ac.in

Department

of

COMPUTER SCIENCE AND ENGINEERING

CTA REPORT

[22UHUC500 - Software Engineering and Project Management]

Odd Semester: Sep 2024 - Jan 2025

Course Teacher: Dr. U.P.Kulkarni

2024- 2025

Submitted by

By

Samiksha Pattan

2SD22CS084

5th Semester B division

Table of Contents

Termwork-1:

Problem Statement:

write a c program to show that c programming language support only call by value.

Theory:

In **call by value**, when a function is called, the values of the actual parameters (arguments) are copied into the formal parameters of the function. Since only a copy is passed, changes made to the formal parameters inside the function do not affect the actual parameters. C uses this mechanism for passing arguments to functions.

Design:

1. Define a function that accepts two integer parameters and tries to modify them.
2. Pass values to this function from the `main()` function.
3. Print the values of the arguments before and after calling the function in the `main()` function to show that changes inside the function do not affect the original arguments.
4. Provide a version that modifies variables through pointers to show how reference-like behavior can be simulated in C using pointers.

Program:

```
#include <stdio.h>
// Function to demonstrate call by value
void modifyValues(int a, int b) {
   // Modifying the values of a and b
   a = 20;
   b = 40;
   printf("Inside modifyValues function: a = %d, b = %d\n", a, b);
}
int main() {
   int x = 10, y = 30;
   // Printing values before function call
   printf("Before function call: x = %d, y = %d\n", x, y);
   // Calling the function with x and y as arguments
   modifyValues(x, y);

   // Printing values after function call
   printf("After function call: x = %d, y = %d\n", x, y);
   return 0;
}
```

Sample input and output:

```
Before function call: x = 10, y = 30
Inside modifyValues function: a = 20, b = 40
After function call: x = 10, y = 30
```

References:

1) **Kernighan, B. W., & Ritchie, D. M. (1988).** *The C Programming Language* (2nd ed.). Prentice Hall.
2) **Graham, S. L., et al. (1989).** *Programming Languages: Concepts and Constructs.* Prentice Hall.

Termwork-2:

Problem Statement:

Study the concept "usability",Prepare a report on usability of at least two uis of major software product you have seen

Theory:

Usability Definition and Importance

Usability refers to the quality of the user experience when interacting with a software application. It encompasses various aspects, including:

- Learnability: The ease with which new users can accomplish basic tasks.
- Efficiency: The speed at which users can perform tasks once they have learned the system.
- Memorability: The ability of users to remember how to use the system after a period of non-use.
- Error Rate: The frequency of errors encountered by users and how easily they can recover from them.
- Satisfaction: The overall enjoyment and satisfaction users derive from using the application.

Usability is a critical factor in determining how effectively users can engage with software applications. In the context of productivity tools like Microsoft Word and entertainment platforms like Spotify, usability impacts user satisfaction, efficiency, and overall performance. Despite their differing purposes—document creation in Microsoft Word and music streaming in

Spotify—both applications must provide an intuitive user interface (UI) to enhance user experience.

Design:

Microsoft Word

Overview: Microsoft Word is a widely-used word processing application that allows users to create, edit, and format text documents. Its UI features a ribbon interface, contextual menus, and a variety of toolbars.

- Design Features:
  - Ribbon Interface: Provides quick access to frequently used tools such as font formatting, alignment, and styles.
  - Contextual Tabs: Offer additional options based on the user’s current task (e.g., inserting images or tables).
  - Templates: Pre-designed layouts help users create documents quickly.
  - Accessibility Features: Options for screen readers and keyboard shortcuts enhance usability for all users.
- Usability Strengths:
  - Comprehensive Features: Offers extensive formatting and editing options.
  - Familiarity: Many users are already familiar with Word, reducing the learning curve.
  - Customization: Users can customize the ribbon to access frequently used commands.
- Usability Challenges:
  - Complexity: The abundance of features can overwhelm new users.
  - Hidden Functions: Some advanced features may not be immediately visible, requiring additional navigation.

Spotify

Overview: Spotify is a music streaming platform that enables users to listen to music, create playlists, and discover new artists. Its UI is designed to be simple and user-friendly.

- Design Features:
  - Navigation Bar: Provides quick access to home, search, and library features.
  - Playlist Management: Easy-to-use interface for creating and managing playlists.
  - Discovery Features: Personalized playlists and recommendations based on user listening habits.
  - Cross-Platform Accessibility: Users can access Spotify on various devices (desktop, mobile, tablet) seamlessly.
- Usability Strengths:
  - Intuitive Design: Simple navigation makes it easy for new users to get started.
  - Personalization: Custom playlists and recommendations enhance user engagement.
  - Visual Appeal: Clean layout and attractive visuals improve user experience.
- Usability Challenges:
  - Limited Offline Features: Users need a premium subscription for offline listening.
  - Search Functionality: While the search feature is generally effective, users sometimes struggle to find specific songs or artists quickly.

Sample input and output:

Microsoft Word

Task: Create a formatted report with a title, headings, and bullet points.

Sample Input:

1. Open Microsoft Word.
2. Type "Quarterly Report" and set the font to 24 pt, bold.
3. Insert a heading "Introduction" and format it as a Heading 1 style.
4. Add bullet points under "Key Findings":
    - Finding 1
    - Finding 2
    - Finding 3

Expected Output:

- A well-formatted document titled "Quarterly Report" with an organized heading and bullet points.

Spotify

Task: Create a new playlist and add songs.

Sample Input:

1. Open Spotify.
2. Click on "Your Library."
3. Select "Create Playlist."
4. Name the playlist "Chill Vibes."
5. Search for songs like "Blinding Lights," "Levitating," and "Watermelon Sugar."
6. Add the searched songs to the "Chill Vibes" playlist.

Expected Output:

- A new playlist named "Chill Vibes" containing the selected songs, accessible from the user's library.

References:

1) Nielsen, J. (1994). **Usability Engineering**. Morgan Kaufmann.
2) Norman, D. A. (2013). **The Design of Everyday Things**. MIT Press.

Termwork-3:

Problem Statement:

list all the features of programming language and write programs to show how they help to write robust code

Theory:

C is a powerful systems programming language known for its versatility, efficiency, and performance. Some of its key features include:

1. Low-Level Access to Memory
2. Portability
3. Rich Set of Operators
4. Modularity through Functions
5. Structured Programming
6. Efficient Use of Pointers
7. Static and Dynamic Memory Allocation
8. Preprocessor Directives
9. Standard Libraries
10. Recursion

Design:

- **Low-Level Access to Memory**

  C allows direct manipulation of memory using pointers, giving the programmer fine-grained control over how memory is managed.

  Example: Pointer Manipulation to Access Memory

```
#include <stdio.h>

int main() {

   int a = 10;

   int *ptr = &a;  // Pointer to the memory location of a

   printf("Value of a: %d\n", a);

   printf("Address of a: %p\n", ptr);

   printf("Value at the address ptr is pointing to: %d\n", *ptr);

   return 0;

}
```

Robustness: Low-level memory access provides flexibility and power to optimize performance, but it requires careful management to avoid errors like segmentation faults.

- **Portability**

C is highly portable, meaning programs written in C can be run on different platforms with minimal modification.

Example: Simple Portable Program

```
#include <stdio.h>

int main() {

   printf("Hello, World!\n");

   return 0;

}
```

Robustness: The portability of C allows code to run on various operating systems and hardware architectures, ensuring compatibility and flexibility.

- **Rich Set of Operators**

C provides a wide range of operators (arithmetic, relational, logical, bitwise, etc.) that allow for efficient expression of algorithms.

Example: Use of Bitwise Operators

```
#include <stdio.h>

int main() {

   int x = 5; // Binary: 101

   int y = 9; // Binary: 1001

   printf("x & y = %d\n", x & y); // AND

   printf("x | y = %d\n", x | y); // OR

   printf("x ^ y = %d\n", x ^ y); // XOR

   printf("~x = %d\n", ~x);       // NOT

   printf("y << 1 = %d\n", y << 1); // Left Shift

   printf("y >> 1 = %d\n", y >> 1); // Right Shift

   return 0;

}
```

Robustness: Rich operators enable efficient and expressive code, particularly in low-level operations such as bit manipulation, which is essential in system programming.

- **Modularity through Functions**

C encourages modularity by allowing code to be divided into functions, promoting code reuse and easier debugging.

Example: Modular Code with Functions

```
#include <stdio.h>

int add(int a, int b) {

   return a + b;

}

int main() {

   int result = add(5, 10);

   printf("Sum: %d\n", result);

   return 0;

}
```

Robustness: Functions break down the program into manageable units, making it easier to debug, test, and maintain, thus improving code robustness.

- **Structured Programming**

C is a structured programming language that supports clear, hierarchical structuring of code using loops, conditionals, and blocks.

Example: Structured Programming Using Loops and Conditionals.

```
#include <stdio.h>

int main() {

   int i;

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

      if (i % 2 == 0)

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

      else

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

   }
```

```
    return 0;

}
```

Robustness: Structured programming in C ensures that the code is organized and easy to follow, leading to fewer logical errors and easier maintenance.

- **Efficient Use of Pointers**

Pointers provide powerful ways to work with memory and data structures like arrays, structs, and dynamic memory.

Example: Passing Pointers to Functions

```
#include <stdio.h>

void swap(int *a, int *b) {

    int temp = *a;

    *a = *b;

    *b = temp;

}

int main() {

    int x = 5, y = 10;

    printf("Before swap: x = %d, y = %d\n", x, y);

    swap(&x, &y);

    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;

}
```

Robustness: Pointers allow direct access and modification of memory, enhancing performance for certain operations like passing large data structures to functions without copying them.

- **Static and Dynamic Memory Allocation**

C provides both static and dynamic memory management, giving flexibility in how memory is allocated and deallocated.

Example: Dynamic Memory Allocation using `malloc` and `free`

```
#include <stdio.h>

#include <stdlib.h>

int main() {
    int *arr;
    int size, i;
    printf("Enter number of elements: ");
    scanf("%d", &size);
    arr = (int *)malloc(size * sizeof(int));  // Dynamically allocate memory
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    printf("Enter %d elements:\n", size);
    for (i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    printf("You entered: ");
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    free(arr);  // Deallocate memory
    return 0;
}
```

Robustness: Dynamic memory allocation allows for flexible memory management, ensuring efficient memory usage for varying needs, while avoiding issues like memory leaks by properly freeing allocated memory.

- **Preprocessor Directives**

C preprocessor directives allow for conditional compilation, file inclusion, and macro expansion, providing flexibility in managing code.

Example: Use of Preprocessor Directives

```
#include <stdio.h>
#define PI 3.14
```

```
int main() {
   printf("Value of PI: %.2f\n", PI);
   return 0;
}
```

Robustness: Preprocessor directives allow for customizable and efficient code, reducing redundancy and enabling conditional compilation.

- **Standard Libraries**

C provides a vast set of standard libraries (e.g., `<stdio.h>`, `<stdlib.h>`, `<string.h>`) that offer reusable functions for input/output, memory management, string manipulation, and more.

Example: Use of `string.h` Library for String Operation.

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[20] = "Hello";
   char str2[20] = "World";
   strcat(str1, str2);  // Concatenates str2 to str1
   printf("Concatenated String: %s\n", str1);
   return 0;
}
```

Robustness: Standard libraries provide well-tested and efficient functionality, reducing the need for reinventing the wheel and minimizing bugs.

- **Recursion**

C supports recursion, which allows a function to call itself, simplifying complex problems like factorial calculation, tree traversal, and more.

Example: Recursive Function for Calculating Factorial

```
#include <stdio.h>
int factorial(int n) {
   if (n == 0) return 1;
   return n * factorial(n - 1);
}

int main() {

   int num = 5;

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

   return 0;

}
```

Robustness: Recursion helps simplify complex tasks, enabling more readable and maintainable code, particularly for algorithms involving divide-and-conquer approach.

References:

1) Herbert Schildt, Java-The Complete Reference, 9th Edition, Tata McGraw Hill, 2014.

2) Grady Booch, Object-Oriented Analysis and Design with Applications, 3rd Edition, Pearson Education, 2007.

Termwork-4:

Problem Statement:

Study the "assertations" in c Language and its importance in writing reliable code.study Poix standard and write a C program under unix to show use of posix standard in writing portable code

Theory:

Assertions in C:

- An assertion is a statement in the code that checks whether a given condition is true. If the condition evaluates to false, the program will terminate, typically with an error message.
- Assertions are used primarily during development and debugging to catch programming errors early. They help ensure that the program operates under expected conditions.
- The standard assertion mechanism in C is provided by the `<assert.h>` header file, which includes the `assert()` macro.

Importance of Assertions:

1. Error Detection: Assertions help catch bugs by verifying assumptions made by the programmer, allowing for earlier detection of logical errors.
2. Documentation: Assertions serve as documentation for the expected behavior of the code, making it clearer for others (or the same developer at a later time) what conditions should hold.
3. Debugging Aid: When an assertion fails, it provides a clear indication of where the problem lies, simplifying debugging.
4. Performance: While assertions are typically disabled in production code (by defining `NDEBUG`), they can be a powerful tool for ensuring code reliability during development.

POSIX Standard:

- POSIX (Portable Operating System Interface) is a family of standards specified by the IEEE for maintaining compatibility between operating systems. It defines a standard operating system interface, ensuring that software can run on different systems with minimal changes.
- POSIX compliance is essential for writing portable code that can operate across various UNIX-like operating systems.

Design:

Objective: Create a C program that uses assertions to validate input and demonstrates POSIX compliance through the use of standard POSIX functions for file operations.

Approach:

- Implement a simple program that reads integers from a file and calculates their sum.
- Use assertions to ensure that the file is opened correctly and that valid integers are read.
- Utilize POSIX functions for file handling to ensure portability across UNIX-like systems.

Program:

```
#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <fcntl.h>

#include <unistd.h>

#define MAX_LINE_LENGTH 100

// Function to read integers from a file and calculate their sum

int sum_from_file(const char *filename) {

   int sum = 0;

   char line[MAX_LINE_LENGTH];

   FILE *file;

   // Open the file in read mode

   file = fopen(filename, "r");

   assert(file != NULL); // Assert that file opened successfully

   // Read integers from the file and calculate the sum

   while (fgets(line, sizeof(line), file) != NULL) {

      int num = atoi(line); // Convert string to integer

      sum += num;
```

```
    }
    // Close the file
    fclose(file);
    return sum;
}
int main() {
    const char *filename = "numbers.txt";
    // Calculate the sum of numbers from the file
    int total_sum = sum_from_file(filename);
    // Output the result
    printf("The sum of numbers in the file %s is: %d\n", filename, total_sum);
    return EXIT_SUCCESS;
}
```

Sample input and output:

Sample Input (File: `numbers.txt`):

```
10  20  30  40
```

Sample Output:

```
The sum of numbers in the file numbers.txt is: 100
```

Additional Input and Output (Error Case)

Input (File Does Not Exist):Attempting to run the program with a non-existent file would produce the following output:

```
Assertion failed: file != NULL, file sum.c, line 13

Aborted (core dumped)
```

References:

1) **ISO/IEC (2011)**. *ISO/IEC 9899:2011: Information technology – Programming languages – C.*
2) **Kernighan, B. W., & Ritchie, D. M. (1988)**. *The C Programming Language* (2nd ed.). Prentice Hall.