Name: Vaibhav Gorakh Lonkar

PRN: 21410027

Batch: EN2

RTOS Experiment No.7

Title: Review of C Language.

1) Data Types

1. What is the need of Data types?
→ Data types are required to define the kind of data a variable can store (e.g., integers, characters, floating-point numbers), how much memory it will occupy, and the range of values it can hold. This ensures efficient memory usage and error prevention during compilation and execution.

2. What is the size of character data type (in bytes)?
→ The size of a char data type is 1 byte.

3. What is the range of values the character data type can hold (mention signed and unsigned values)?
→
- Signed char: -128 to 127
- Unsigned char: 0 to 255

4. When to use signed char and when to use unsigned char?
→ Use signed char when you need to store both negative and positive values within the range of -128 to 127. Use unsigned char when you need only positive values (0 to 255), typically for representing characters or small integers.

5. What is the size of integer data type?
→ The size of an int data type is 4 bytes.

6. What is the range of values the integer data type can hold (mention signed and unsigned values)?
→
- Signed int: -2,147,483,648 to 2,147,483,647
- Unsigned int: 0 to 4,294,967,295

7. What is the purpose of integer data type?
→ The int data type is used to store whole numbers (without decimals), typically used for counting, indexing, or arithmetic operations that require integer precision.

8. What is the size of float data type (in bytes)?
→ The size of a float data type is 4 bytes.

9. What is the range of values the float data type can hold (mention signed and unsigned values)?
→ The range of a float for signed values is approximately ±3.4E−38 to ±3.4E+38. The concept of unsigned does not apply to float because floating-point numbers can be both positive and negative.

10. What is the purpose of float data type?
→ The float data type is used for storing decimal (floating-point) numbers with single precision, useful when precision is not critical but memory usage is.

11. What is the size of long data type (in bytes)?
→ The size of a long data type is 8 bytes.

12. What is the range of values the long data type can hold (mention signed and unsigned values)?
→ □ Signed long: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
□ Unsigned long: 0 to 18,446,744,073,709,551,615

13. What is the purpose of long data type?
→ The long data type is used when larger integer values are needed beyond the standard int range, such as in large computations or file handling.

14. Why is it better to explicitly declare signed or unsigned (for long data type)?
→ Explicitly declaring signed or unsigned avoids ambiguity and potential bugs, ensuring that variables behave as expected, especially when working with large numbers or memory-constrained environments.

15. What is the size of double data type (in bytes)?
→ The size of a double data type is 8 bytes.

16. What is the range of values the double data type can hold (mention signed and unsigned values)?
→ The range of a double is approximately ±1.7E−308 to ±1.7E+308. Like float, there is no unsigned version of double since it represents both positive and negative values.

17. What is the purpose of double data type?
→ The double data type is used for storing decimal numbers with double precision, offering higher accuracy and range than float, useful in scientific calculations.

18. What is the meaning of void?
→ Void represents "no type" or "no value." It is commonly used to indicate that a function does not return a value.

19. Can we declare a variable of void type? Why?
→ No, you cannot declare a variable of void type because void means "no value." It is used in functions to indicate no return type, but it cannot hold data.

20. What is the size of void data type?
→ Void has no size, as it represents the absence of data.

21. What is the purpose of void?
→ The purpose of void is to define functions that do not return any value or to specify pointers that do not refer to a specific data type, such as void*.

2) Pointers in C:

1. What is a pointer?
→ A pointer is a variable that stores the memory address of another variable. Instead of holding a data value directly, a pointer "points" to the location where the data is stored in memory.

2. What are the advantages of pointers?
→ **Efficient Memory Use**: Allows direct manipulation of memory and dynamic memory allocation (e.g., with malloc or free).
 Pass by Reference: Functions can modify variables outside their scope by passing the address instead of a copy of the variable.
 Dynamic Data Structures: Pointers enable the creation of data structures like linked lists, trees, and graphs.
 Efficient Arrays and Strings: Arrays and strings can be handled more efficiently using pointers.

3. How to declare a pointer? Why do we need to specify the data type while declaring it?
→ A pointer is declared by specifying the data type it points to, followed by an asterisk (*) before the pointer variable name. For example:
int *ptr; // Pointer to an integer
The data type is necessary to determine the size of the data it points to, how the pointer will interpret the memory it references, and to perform pointer arithmetic correctly.

4. Write a program to list the use of pointers. And mention the possible outcomes.
→
```
 #include <stdio.h>
int main() {
   int x = 10;
   int *ptr = &x;  // Pointer to x
   printf("Value of x: %d\n", x);
   printf("Address of x: %p\n", &x);
   printf("Value of ptr (address of x): %p\n", ptr);
   printf("Value pointed to by ptr: %d\n", *ptr); // Dereferencing the pointer
   *ptr = 20; // Modify the value of x using the pointer
   printf("New value of x after modification: %d\n", x);
   return 0;
}
```

Possible Outcomes:

- Prints the value of x, its memory address, and the value at the address pointed to by ptr.
- Modifies the value of x through the pointer, updating it to 20.

5. What is a void pointer?

→ A void pointer (or generic pointer) is a pointer that can hold the address of any data type. It is declared as void*, and before dereferencing, it must be typecast to the correct data type. Example:

```
void *ptr;
int x = 5;
ptr = &x;  // Store address of an int in void pointer
```

6. What is a null pointer?

→ A null pointer is a pointer that points to nothing or is explicitly assigned the address NULL. It is used to indicate that the pointer is not currently pointing to any valid memory location.
Example:

```
int *ptr  NULL;
```

3) Structure in C:

1. What is a structure?

→ A structure in C is a user-defined data type that allows grouping different types of data together under a single name. It is used to store different types of data (such as int, char, float) as a single entity.

2. Write advantages of structures?

→ **Grouped Data**: Structures allow grouping different data types under one name, making the code more organized and readable.

 Data Management: Easier management of related data (e.g., employee records, student details) by using a single structure rather than individual variables.

 Efficient Memory Use: Structures can store large sets of heterogeneous data without wasting memory.

 Supports Complex Data Types: Useful for creating complex data types like linked lists, trees, and graphs by combining different data types.

 Modular Code: Helps in modular code development where related data can be grouped, aiding in code maintenance and scalability.

3. How to declare a structure? Why do we need to specify the data type while declaring it?

→ A structure is declared using the struct keyword, followed by the structure name and a block containing the data types and members of the structure. Specifying the data type for each member is necessary to define the type of data each member will store.

```
struct Employee {
   int id;
   char name[50];
   float salary;
```

};
In this declaration, the structure Employee contains three members: id (an integer), name (a character array), and salary (a float). The data types are needed to allocate memory and define the type of each member.

4. Write a program to list the use of structure. And mention the possible outcomes.
→

```
 #include <stdio.h>
// Structure declaration
struct Employee {
    int id;
    char name[50];
    float salary;
};
int main() {
    // Creating a structure variable
    struct Employee emp1;
    // Assigning values to structure members
    emp1.id = 101;
    sprintf(emp1.name, "John Doe"); // Using sprintf to assign string
    emp1.salary = 50000.00;

    // Accessing and printing structure members
    printf("Employee ID: %d\n", emp1.id);
    printf("Employee Name: %s\n", emp1.name);
    printf("Employee Salary: %.2f\n", emp1.salary);
    return 0;
}
```

Possible Outcomes:

- The program will create an Employee structure and assign values to the id, name, and salary members.

Employee ID: 101
Employee Name: John Doe
Employee Salary: 50000.00

4) Writing portable C CODE:

1. Illustrate portability with 1 example.
→ Portability refers to the ability of software to run on different platforms or operating systems with minimal modification. A good example of portability is the use of the C programming language across various platforms (Windows, Linux, macOS).
Example:
Consider a simple "Hello World" program written in C:

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

```
int main() {
   printf("Hello, World!\n");
   return 0;
}
```

This program is portable because it can be compiled and run on multiple platforms without changes to the source code, as long as a C compiler is available. The same program can be compiled with GCC on Linux or Windows using the same syntax.

2. Write the guidelines to be followed while writing portable code.

→ **Avoid Platform-Specific Code**: Avoid using functions or features that are specific to a particular operating system (e.g., system calls, file handling specific to one OS).

 Use Standard Libraries: Stick to standard libraries (like C Standard Library or C++ Standard Library) instead of platform-dependent libraries.

 Handle Endianness: Ensure your code works for both big-endian and little-endian systems, especially when working with byte streams.

 Use Preprocessor Directives for Platform-Specific Code: When platform-specific code is unavoidable, use preprocessor directives to conditionally compile code for different platforms.

 Avoid Compiler-Specific Features: Stick to standard language features instead of relying on extensions or specific features provided by one compiler.

 Data Types: Use fixed-width integer types (e.g., int32_t, uint16_t) from <stdint.h> instead of regular types (int, short), which may vary in size across platforms.

 Be Mindful of File System Differences: File system handling can differ between platforms (e.g., different path formats in Windows and Linux), so use platform-independent methods for file I/O.

 Adopt Cross-Platform Build Tools: Use build systems like CMake or Meson, which help in generating platform-specific build scripts, making your project easier to compile across different platforms.

3. Whether the RTOS code should be portable? Why?

→ Yes, **RTOS code should be portable**, and here's why:

- **Scalability Across Platforms**: An RTOS may be required to run on various embedded systems with different microcontrollers, hardware architectures, or peripherals. Making the RTOS code portable allows it to be used across different hardware platforms without significant rework.
- **Code Reuse**: Portability ensures that the same RTOS code can be reused in different projects, reducing development time and costs.
- **Long-Term Maintenance**: Portable code is easier to maintain and update, especially when hardware or platform changes occur.
- **Vendor Independence**: Portability helps avoid being locked into a particular hardware or software vendor, giving developers the flexibility to switch platforms without rewriting large portions of the code.

In RTOS development, abstraction layers (like hardware abstraction layers or platform-specific drivers) can be used to separate platform-dependent code, improving portability across different embedded systems.

Review of Keil Software:

Title of Task: LED blinking in the waveform.

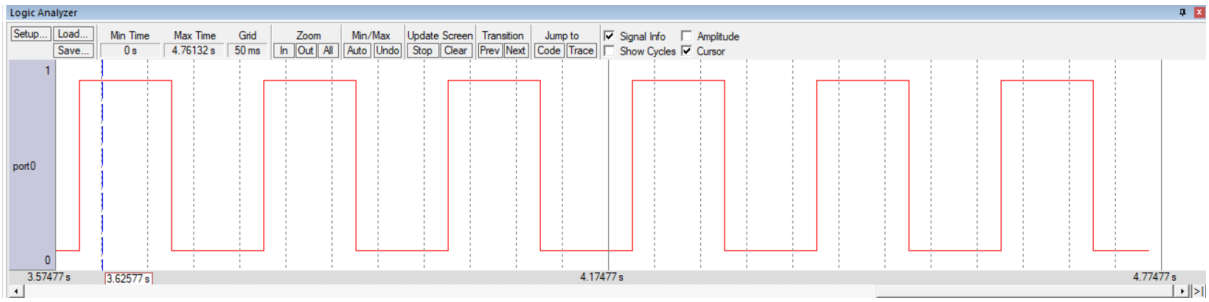
Code:

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

#define TaskStkLengh 64  //Define the Task0 stack length
OS_STK TaskStk0 [TaskStkLengh];  //Define the Task stack
void Task0(void *pdata);

void Task0 (void *pdata){
 pdata = pdata;
 while(1){
  LED_on(0);
  OSTimeDly(10);
  LED_off(0);
  OSTimeDly(10);
 }
}
int main (void) {
 LED_init();
 TargetInit();
 OSInit ();
 OSTaskCreate (Task0,(void *)0, &TaskStk0[TaskStkLengh - 1], 6);
          OSStart();
  return 0;
}
```

Result:


Logic Analyzer
Setup...
Load...
Save...
Min Time
0 s
Max Time
4.76132 s
Grid
50 ms
Zoom
In
Out
All
Min/Max
Auto
Undo
Update Screen
Stop
Clear
Transition
Prev
Next
Jump to
Code
Trace
Signal Info
Show Cycles
Amplitude
Cursor
1
port0
0
3.57477 s
3.62577 s
4.17477 s
4.77477 s