

Experiment No. 1

TITLE: To create, compile and run C program.

OBJECTIVE:

- To learn compile and run C program.
- To learn to use standard I/O header file.

THEORY:

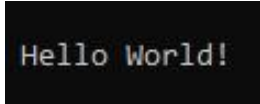
The C programming language is a procedural and general-purpose language that provides low-level access to system memory. A program written in C must be run through a C compiler to convert it into an executable that a computer can run.

PROGRAM:

Source Code

```
#include<stdio.h>
int main(){
    printf("\n Hello World! \n");
    return 0;
}
```

Output:



```
Hello World!
```

RESULTS AND DISCUSSION:

The experiment was successful in creating, compiling, and running a simple C program. The "Hello World!" program served as a basic introduction to the C programming language and its syntax.

CONCLUSION:

This laboratory exercise provided a hands-on experience in creating, compiling, and running a C program. Students gained practical knowledge of the basic in C programming and are now better equipped to undertake more complex programming tasks in the future.

Experiment No. 2

TITLE: To handle different data types used in C programming.

OBJECTIVE:

- To learn concept of data types in C programming language.
- To learn how to declare, initialize, and manipulate different data types in C.

THEORY:

In C programming, data types are declarations for variables. This determines the type and size of data associated with variables. Basic Data types of C with range are given below:

1. int (Integer):

- Size: 4 bytes
- Range: -2,147,483,648 to 2,147,483,647

2. char (Character):

- Size: 1 byte
- Range: -128 to 127 or 0 to 255 (depending on whether it is signed or unsigned)

3. float (Floating-point):

- Size: 4 bytes
- Range: 3.4e-38 to 3.4e+38 (6 decimal places precision)

4. double (Double precision floating-point):

- Size: 8 bytes
- Range: 1.7e-308 to 1.7e+308 (15 decimal places precision)

5. short (Short Integer):

- Size: 2 bytes
- Range: -32,768 to 32,767

6. long (Long Integer):

- Size: 4 bytes (on 32-bit systems) or 8 bytes (on 64-bit systems)
- Range: -2,147,483,648 to 2,147,483,647 (on 32-bit systems) or -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (on 64-bit systems)

7. long long (Long Long Integer):

- Size: 8 bytes
- Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

8. Bool (Boolean):

- Size: 1 byte
- Range: 0 to 1 (or false to true)

PROGRAM:

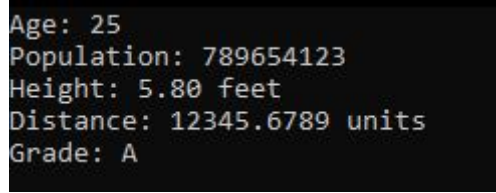
Source Code

```
#include <stdio.h>
int main() {
    //variable declaration & initialization
    int age = 25;
    long population = 789654123;
    float height = 5.8;
    double distance = 12345.6789;
    char grade = 'A';

    printf("Age: %d\n", age);
    printf("Population: %ld\n", population);
    printf("Height: %.2f feet\n", height);
    printf("Distance: %.4f units\n", distance);
    printf("Grade: %c\n", grade);

    return 0;
}
```

Output:

A screenshot of a terminal window showing the output of the C program. The text is displayed in a monospaced font on a dark background. The output consists of five lines, each corresponding to a printf statement in the source code: 'Age: 25', 'Population: 789654123', 'Height: 5.80 feet', 'Distance: 12345.6789 units', and 'Grade: A'.

```
Age: 25
Population: 789654123
Height: 5.80 feet
Distance: 12345.6789 units
Grade: A
```

RESULTS AND DISCUSSION:

The experiment successfully demonstrated the handling of different data types in C programming. Students were able to declare, initialize, and manipulate variables of integer, floating-point, and character types.

CONCLUSION:

This laboratory exercise provided practical experience in handling various data types in C programming. With the help of data type, different types of data can be stored and can reuse according to need.

Experiment No. 3

TITLE: To perform different formatted and unformatted I/O operations.

OBJECTIVE:

- To understand and practice different types of I/O operations in the C language.

THEORY:

In C programming, I/O operations involve reading input from a source (e.g., the user or a file) and writing output to a destination (e.g., the console or a file). Formatted and unformatted I/O operations differ in how the data is processed and presented. Formatted I/O uses format specifiers to control the conversion and representation of data, while unformatted I/O treats the data as a series of bytes. The header file for formatted or unformatted data is standard data type which is denoted by `<stdio.h>` at the top of the program.

- **Formatted I/O:** We use the formatted input and output functions in the C language for taking single or multiple inputs from the programmer/user at the console. These functions also allow a programmer to display single or multiple values, in the form of output, to the users present in the console.

For input: `scanf()` function is used.

For output: `printf()` function is used.

- **Unformatted I/O:** These functions are used exclusively for character data types or character arrays/strings. They are designed for reading single inputs from the user at the console and for displaying values on the console.

For input: `getchar()` and `gets()` functions are used.

For output: `putchar()` and `puts()` functions are used.

PROGRAM:

- **Formatted I/O**

```
#include <stdio.h>

int main() {
    int num1, num2;

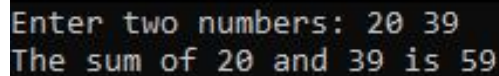
    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);
    printf("The sum of %d and %d is %d\n", num1,
num2,          num1 + num2);

    return 0;
}
```

Explanation

In the above code, we use `printf()` to display the values of `num1` & `num2`. The format specifier `%d` is used to display the integer value, and `%f` is used to display the floating-point value. We also use the precision specifier `'.2'` to display `num2` with two decimal places.

Output:



```
Enter two numbers: 20 39
The sum of 20 and 39 is 59
```

- **Unformatted I/O:**

```
#include <stdio.h>

int main() {
    char ch, str[20]; //Variable Declaration

    printf("Enter String:");
    gets(str);
    printf("Enter a character: ");
    ch = getchar();

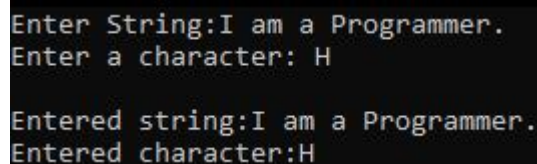
    printf("\nEntered string:");
    puts(str);
    printf("Entered character:");
    putchar(ch);

    return 0;
}
```

Explanation:

In the above code, we use `getchar()` to read a single character and `gets()` to read string from the user. The character entered by the user is stored in the variable `ch` and string in `str`. String variable is not available in C programming that why we use array `char` as string. For display the values, `putchar()` used for single character and `puts()` used for string.

Output:



```
Enter String:I am a Programmer.
Enter a character: H

Entered string:I am a Programmer.
Entered character:H
```

RESULTS AND DISCUSSION:

By performing the above experiments, we observed the following:

- Formatted input and output operations provide a convenient way to read and display data in a specific format. They are particularly useful when working with different data types.
- Unformatted input and output operations treat the data as a sequence of characters or bytes. They are useful when dealing with low-level operations or when data is stored in a raw format.

It is important to note that both formatted and unformatted I/O operations have their advantages and are suitable for different scenarios. The choice of which method to use depends on the specific requirements of the program.

CONCLUSION:

In this lab experiment, we explored and implemented different types of formatted and unformatted I/O operations in the C programming language. We successfully performed formatted input and output using `scanf()` and `printf()`, respectively. We also performed unformatted input using `getchar()` & `gets()` and unformatted output using `putchar()` & `puts()`. Through this experiment, we gained a better understanding of how these I/O operations work and their differences in terms of data processing and representation.

By learning the formatted and unformatted I/O operations, we can effectively handle input and output tasks in C programming, providing a more interactive and dynamic experience to the users of our programs.

Experiment No. 4

TITLE: To perform different operations using operators in C programming.

OBJECTIVE:

- To understand and practice various types of operations using operators in the C programming language.
- To learn arithmetic, relational, logical, bitwise and assignment operator.

THEORY:

Operators in C programming are symbols that perform specific operations on operands, which can be variables, constants, or expressions. Different types of operators serve different purposes, such as performing arithmetic calculations, comparing values, combining conditions, or manipulating bits.

1. Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations such as addition, subtraction, multiplication, division, and modulus.

2. Relational Operators

Relational operators are used to compare values and determine the relationship between them. They return a boolean value (1 or 0) indicating whether the comparison is true or false.

3. Logical Operators

Logical operators are used to combine conditions and perform logical operations. They return a boolean value (1 or 0) based on the truth values of the operands.

4. Bitwise Operators

Bitwise operators are used to manipulate individual bits of integer operands.

5. Assignment Operator

The assignment operator = is used to assign a value to a variable.

PROGRAM:

- **Arithmetic Operators**

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

```
int main() {  
    int a = 10, b = 3;  
    //initialization using assignment operator
```

```

    int sum = a + b;
    int difference = a - b;
    int product = a * b;
    int quotient = a / b;
    int remainder = a % b;

    printf("Sum: %d\n", sum);
    printf("Difference: %d\n", difference);
    printf("Product: %d\n", product);
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);

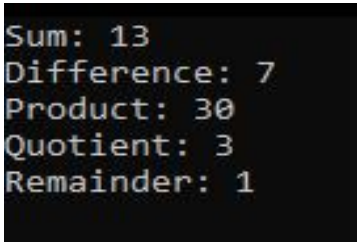
    return 0;
}

```

Explanation

In the above code, we perform arithmetic operations on the variables `a` and `b`. The `+` operator is used for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `%` for modulus (remainder). The results are stored in separate variables and displayed using `printf()`.

Output:



```

Sum: 13
Difference: 7
Product: 30
Quotient: 3
Remainder: 1

```

- **Relational Operators**

```

#include <stdio.h>
int main() {
    int a = 5, b = 10;

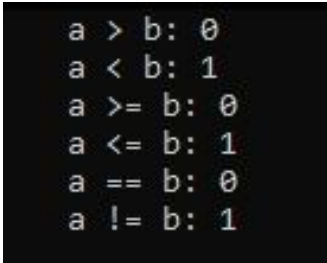
    printf("a > b: %d\n", a > b);
    printf("a < b: %d\n", a < b);
    printf("a >= b: %d\n", a >= b);
    printf("a <= b: %d\n", a <= b);
    printf("a == b: %d\n", a == b);
    printf("a != b: %d\n", a != b);
    return 0;
}

```


Explanation

In the above code, we compare the values of a and b using relational operators. The > operator checks if a is greater than b, < checks if a is less than b, >= checks if a is greater than or equal to b, <= checks if a is less than or equal to b, == checks if a is equal to b, and != checks if a is not equal to b. The results are displayed using printf().

Output:



```
a > b: 0
a < b: 1
a >= b: 0
a <= b: 1
a == b: 0
a != b: 1
```

- **Logical Operators**

```
#include <stdio.h>

int main() {
    int a = 5, b = 10, c = 15;

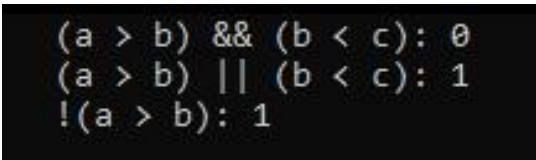
    printf("(a > b) && (b < c): %d\n", (a > b) && (b
< c));
    printf("(a > b) || (b < c): %d\n", (a > b) || (b
< c));
    printf("!(a > b): %d\n", !(a > b));

    return 0;
}
```

Explanation

In the above code, we combine conditions using logical operators. The && operator performs a logical AND operation, returning true if both conditions are true. The || operator performs a logical OR operation, returning true if at least one of the conditions is true. The ! operator performs a logical NOT operation, negating the truth value of the condition. The results are displayed using printf().

Output:



```
(a > b) && (b < c): 0
(a > b) || (b < c): 1
!(a > b): 1
```

- **Bitwise Operators**

```
#include <stdio.h>

int main() {
    unsigned int a = 5, b = 3;

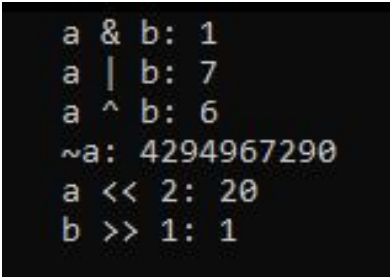
    printf("a & b: %u\n", a & b);
    printf("a | b: %u\n", a | b);
    printf("a ^ b: %u\n", a ^ b);
    printf("~a: %u\n", ~a);
    printf("a << 2: %u\n", a << 2);
    printf("b >> 1: %u\n", b >> 1);

    return 0;
}
```

Explanation

In the above code, we perform bitwise operations on the variables `a` and `b`. The `&` operator performs a bitwise AND operation, `|` performs a bitwise OR operation, `^` performs a bitwise XOR (exclusive OR) operation, `~` performs a bitwise NOT operation (inverting the bits), `<<` performs a left shift operation (shifting the bits to the left), and `>>` performs a right shift operation (shifting the bits to the right). The results are displayed using `printf()`.

Output:

A screenshot of a terminal window with a black background and light blue/green text. It displays the output of the C program: a & b: 1, a | b: 7, a ^ b: 6, ~a: 4294967290, a << 2: 20, and b >> 1: 1.

```
a & b: 1
a | b: 7
a ^ b: 6
~a: 4294967290
a << 2: 20
b >> 1: 1
```

RESULTS AND DISCUSSION

By performing the above experiments, we observed the following:

- Arithmetic operators allow us to perform mathematical calculations and manipulate numerical values.
- Relational operators enable us to compare values and determine relationships between them.
- Logical operators allow us to combine conditions and perform logical operations.
- Bitwise operators offer the ability to manipulate individual bits of integer operands.
- The assignment operator is used to assign a value to a variable.

Understanding and utilizing different operators in C programming is crucial for performing various operations and implementing complex algorithms.

CONCLUSION

In this lab experiment, we explored and implemented various types of operations using operators in the C programming language. We successfully performed arithmetic calculations using arithmetic operators, compared values using relational operators, combined conditions using logical operators, manipulated bits using bitwise operators, and assigned values using the assignment operator. Through this experiment, we gained a better understanding of how operators work and their significance in programming.

After learning operators, we can perform a wide range of operations in C programming, making our code more efficient, expressive, and powerful.