6 Lecture - CS410

Important Subjective

1. What are bitwise operators, and how are they different from logical operators?

Answer: Bitwise operators perform operations on individual bits of data, while logical operators operate on Boolean values (true or false). Bitwise operators include AND, OR, XOR, and shift operators, whereas logical operators are represented by && (AND), || (OR), and ! (NOT).

2. Explain the purpose of the bitwise AND operator (&) and how it can be used to check if a specific bit is set in a number.

Answer: The bitwise AND operator (&) is used to perform a bitwise AND operation on two integers. To check if a specific bit is set in a number 'num', you can use the expression (num & (1 << bit_position)). If the result is non-zero, then the bit at the given 'bit_position' is set; otherwise, it is not set.

3. How can you set a specific bit in an integer variable 'num' using a bitwise OR operation?

Answer: To set a specific bit at position 'bit_position' in 'num', you can use the expression (num |= (1 << bit_position)). This sets the bit at 'bit_position' to 1 without affecting other bits in the 'num' variable.

4. Describe the purpose of the bitwise XOR operator (^) and give an example of how it can be used to toggle a bit.

Answer: The bitwise XOR operator (^) performs a bitwise exclusive OR operation on two integers. It returns 1 for each position where the corresponding bits in the operands differ. To toggle a bit at position 'bit_position' in 'num', you can use the expression (num ^= (1 << bit_position)).

5. What are macros in C/C++, and how do they enhance code readability and reusability?

Answer: Macros are preprocessor directives that allow defining constants, functions, or code snippets that are replaced before compilation. They enhance code readability by introducing meaningful names for

constants and reducing magic numbers. Macros also facilitate code reusability by providing a way to encapsulate complex operations into a single macro, which can be used multiple times in the code.

6. How can you check if a macro is defined or not using preprocessor directives?

Answer: You can use the #ifdef preprocessor directive to check if a macro is defined or not. For example:

```c
#ifdef MACRO_NAME
    // Code to be executed if the macro is defined
#else
    // Code to be executed if the macro is not defined
#endif
```

7. Explain the significance of the bitwise left shift operator (<<) and how it can be used for multiplication by powers of 2.

Answer: The bitwise left shift operator (<<) shifts the bits of an integer to the left. It effectively multiplies the number by 2 raised to the power of the specified shift count. For example, 'num << n' is equivalent to 'num * 2^n', which is useful for fast multiplication and division by powers of 2.

8. How do you use macros to create a generic swap function for any data type in C/C++?

Answer: You can define a macro for a generic swap function as follows:

```c
#define SWAP(x, y) do { typeof(x) temp = x; x = y; y = temp; } while(0)
```

This macro uses the C/C++ "typeof" extension to determine the data type of variables 'x' and 'y' at compile time and performs the swap accordingly.

9. Discuss the benefits and drawbacks of using macros in C/C++.

Answer: Benefits:

- Macros enhance code readability by giving meaningful names to constants and reducing magic numbers.

- They enable code reusability by encapsulating complex operations into a single macro.

- Macros are preprocessed, so they incur no runtime overhead.

- Macros can perform conditional compilation, allowing for feature customization.

Drawbacks:

- Macros lack type safety, and errors may not be caught until compile time.

- Macros can lead to unexpected behavior when used improperly or with complex expressions.

- Debugging macros can be challenging, as they do not appear in the call stack during runtime errors.

10. When should you prefer bitwise operators over arithmetic operators in C/C++ programming?

Answer: Bitwise operators are preferred in scenarios where operations need to be performed at the bit level, such as:

- Manipulating individual bits in a bitfield or hardware register.

- Implementing bit flags or bitmasks for configuration or status checking.

- Efficiently packing multiple boolean values into a single variable to save memory.

- Performing fast multiplication or division by powers of 2 using bitwise shift operators.