INTERVIEW QUESTIONS

Keywords

Keywords are reserved words in a programming language **that have special meanings and purposes**. They are **reserved by the language** and **cannot be used for naming variables, functions, or other identifiers**.

Magic Numbers

Values that appear out of nowhere in program code are known as magic numbers. These may be mathematical constants, standard rates or default values.
We avoid magic numbers by identifying them with symbolic names and using those names throughout the code - We set their value in either of the two ways:

1. Using an **unmodifiable variable**

These are variables declared with the **'const' keyword**. Once a variable is declared as const, its value cannot be changed after initialization. Attempting to modify a const variable will result in a compilation error.

2. Using a **macro directive**

A macro is NOT a variable but is used for substitution at compile-time

Flag

Flags are variables that **determine whether an iteration continues or stops**. A flag is either true or false. Flag helps ensure that no paths cross one another. By introducing a flag, we avoid jump and multiple exit, obtain a flow chart where no path crosses any other and hence improved design.

A flag variable can **enforce the single entry/exit principle** for a loop by initially setting the flag to a specific value, modifying the flag within the loop based on a condition, and then checking the flag's value after the loop to ensure that the loop executed as expected. This approach ensures that the loop has a clear entry point before it starts and a clear exit point after it finishes, allowing for controlled and predictable loop behaviour.

```
#include <stdio.h>

int main(void)
{
   int i, value;
   int done = 0;  // flag
   int total = 0; // accumulator

   for (i = 0; i < 10 && done == 0; i++)
   {
      printf("Enter integer (0 to stop) ");
      scanf("%d", &value);

      if (value == 0)
      {
```

```
            done = 1;
        }
        else
        {
            total += value;
        }
    }

    printf("Total = %d\n", total);

    return 0;
}
```

Compilation Process

The code written in the C language (also known as **source code** - this is the input or program instructions for the compiler) is converted into **machine-level equivalents (or Binary Code**, this is the output of the compiler) an **exe file** is generated by the compiler The OS loads the Binary Code (given by the compiler) into the RAM and starts executing the code based on the inputs given by the user and generates the output for it.

Comments

Comments in programming languages are **non-executable portions of code** used to **annotate** or **document the code** for humans reading it. They are **ignored by the compiler** or interpreter when the code is executed.

Auto Sizing Vs Explicit sizing of the arrays

Auto sizing of the arrays refers to **initialising an array** with 1 or more elements while **declaring it without specifying its size.** The array will automatically store space in the memory according to the number of variables we initialise it with.
For example:
int arr[] = {2,4,5};
Here, we do not mention the size of the array but we initialise it with 3 elements, which will define an array of size 3 (12 bytes in this case).

Explicit sizing of the arrays refer to **defining the size of the array while declaring it.** After we explicitly define the size of the array, we cannot change the size of the array later on during the program's execution.
For example:
int arr[5];
Here, we are explicitly defining the size of the array as 5.

It is better to work with currency as integers rather than floating point. Why?

Working with currency as integers is often preferred over floating point numbers primarily due to **precision**, and **rounding issues.**

Integers are whole numbers and so they do not have any inherent precision issues associated with floating-point numbers. Floating point numbers are limited in their precision due to the way they are stored in binary, which lead to small rounding errors in calculations. By working with integers we can eliminate rounding errors.

Style Guidelines

1. **Readability and Maintainability:** Adhering to style guidelines improves code readability and maintainability. Clear and consistent code is easier to understand, reducing the chances of introducing bugs during development and making it simpler to update or modify code in the future.
2. **Consistency:** Style guidelines ensure that the codebase maintains a consistent and uniform appearance, making it easier for multiple developers to work on the same project. Consistency simplifies code review, debugging, and maintenance, as programmers can quickly understand and follow established conventions.

Advantages

1. A well written code is easy to read and maintain.
2. The coding style is consistent and clear throughout.
3. It is easy to modify or upgrade code.
4. It becomes easy to spot bugs, find errors and debug the code

Absence of Style Guidelines

1. **Inconsistent Code -** Developers write code in their preferred style leading to lack of consistency within the codebase making it more difficult among the team mates to understand and work with each other's code, slowing down development and increasing the likelihood of errors.
2. **Code Maintenance Challenges** - Without guidelines, maintaining and updating the codebase becomes more challenging, as there is no standard to follow.
3. **Quality and Debugging Issues:** Inconsistent code may introduce quality issues and make debugging more challenging. Developers might overlook errors or spend extra time debugging due to code that deviates from expected practices.

Initialization of Variables - It ensures that the variable has defined initial value before they are used. This helps prevent unexpected behaviour and bugs due to uninitialized variables.
If a variable is not initialised it may contain arbitrary or garbage values left in memory, leading to unpredictable results, making code more challenging to debug and maintain.

Type - A type is a rule that defines how to store values in memory and which operations are admissible on those values. It defines the number of bytes available for storing values and hence the range of possible values.

What is the significance of the number preceding the (.) in the display of the floating point numbers? Provide explanations with reference to %3.5 lf and %0.5lf formatting?

The number preceding the (.) in the format specifier for displaying floating-point numbers controls the **minimum width of the field** that the number is printed in.

In this case,

1. **%3.5lf** - The number before the . (in this case, 3) specifies the minimum width of the field. It means that the floating-point number will be printed in a field at least 3 characters wide. The 5 after the (.) specifies the number of digits to display after the decimal point (the precision). If the number has more than 3 characters (including the integer part, the decimal point, and the fractional part), it will be displayed as is. If it's shorter, it will be padded with leading spaces to meet the minimum width requirement. For example, if you have the number 12.34500 and use %3.5lf, it will be printed as "12.34500" because it already has more than 3 characters.

0. **%0.5lf -** The number before the . (in this case, 0) specifies the minimum width of the field. It means that there is no minimum width requirement, and the output will expand to accommodate the number. The 5 after the . specifies the number of digits to display after the decimal point (5 in this case). If there are less than 5 digits after the (.) the compiler will fill the remaining places with 0's in most of the cases.

In C programming, casting is used to explicitly convert a value from one data type to another. This is done to ensure that the data is interpreted and processed correctly by the program. Casting can be useful when you need to perform operations on variables of different data types or when you want to store a value in a different type of variable.

```
#include <stdio.h>
int main()
{
double num1 = 3.14; int num2; // Attempt to assign a double to an int num2 = num1; // This would generate a warning // Casting the double to num2 = (int)num1;
printf("num1 (double): %f\n", num1); printf("num2 (int): %d\n", num2); return 0;
}
```

A structured program is a type of computer program designed with clear modules and organized control structures, making the code more readable and maintainable. It emphasizes a top-down design approach, breaking the program into smaller, manageable parts. This approach helps reduce errors and makes it easier to understand and modify the code.

Coupling

Coupling describes the degree of interrelatedness of a module with other modules. Low coupling in the context of a function call refers to minimising the dependencies between the calling function and the called function, promoting modular and independent design by limiting the knowledge each function has about the internal workings of the other.

Modular Design

Modular design Involves breaking down a complex task into smaller, independent blocks (modules) that can be developed and tested separately. These modules can then be easily combined to create a larger, well-organised program, making it easier to understand, maintain, and update.

Const Qualifier Used For A Function

The const qualifier used for a function parameter indicates that the function promises not to modify the value of that parameter within the function, providing a clear contract and preventing unintended modifications to the argument. This enhances code clarity, readability, and helps catch accidental modifications during compilation.

Pass By Reference

Passing by address allows functions to directly access and modify the original data, avoiding the overhead of creating a duplicate copy of the entire variable. This approach is particularly beneficial for large data structures, as it reduces memory consumption and improves performance. Additionally, passing by address is essential when you want a function to modify the original value, as changes made to a copy in pass by value would not affect the original variable outside the function.