

Nerd For Tech · [Follow publication](#)

C Interview Questions 03: Bitwise Operations / Sizes of Structs / Volatile

With hands-on examples

10 min read · Dec 16, 2023



Yu-Cheng (Morton) Kuo

[Follow](#)

Listen



Share

Continuing from the previous article.

C Interview Questions 02: Makefile, Scope and Lifetime, Call by Value, & Data Types

Outline

- (0) Warm-Up
- (1) Q01: Swap two variables without using a temporary variable
- (2) Q02: Use bitwise AND (&) for checking a number is odd or even
- (3) Q03: Use bitwise OR(|), XOR(^), AND(&) to toggle a specific bit. Use bitwise left-shift or right-shift for multiplication & division by power of 2
- (4) Q04: How to convert 0000 1011 into 0000 0010?
- (5) Q05: Use bitwise AND (&) for computing the lowest bit of a number (i.e., the rightmost 1)
- (6) Q06: How to clear the lowest set bit (the rightmost 1 of a binary) using bitwise operations?
- (7) Q07: Identify the sizes of the structs below
- (8) Q08: Explain the keyword volatile

(9) References

(0) Warm-Up

0-1

1. $A \wedge 0 = A$
2. $A \wedge A = 0$
3. $A \wedge B = B \wedge A$
4. $(A \wedge B) \wedge C = A \wedge (B \wedge C)$

0-2 2's complement: $x + x_bar = -1$

For an unsigned integer x , since $2 + 2_bar = -1$, $-2 = 2_bar + 1$, then

$2 = 0000\ 0010$

$2_bar = 1111\ 1101$

$-2 = 2_bar + 1 = 1111\ 1101 + 0000\ 0001 = 1111\ 1110$

(1) Q01: Swap two variables without using a temporary variable

```
// Method 01
a = a ^ b;
b = a ^ b; // Now b = original a
a = a ^ b; // Now a = original b

// Method 02
a = a + b;
b = a - b; // Now b = original a
a = a - b; // Now a = original b

// Method 03
a = a * b;
```

```
b = a / b; // Now b = original a
a = a / b; // Now a = original b
```

- Method 01 (XOR Swap): Safe from overflow and underflow.
- Method 02 (Addition and Subtraction Swap): Potential for overflow.
- Method 03 (Multiplication and Division Swap): High risk of overflow and the additional risk of division by zero. Underflow could be a concern with floating-point numbers.

So method one is always the best option.

(2) Q02: Use bitwise AND (&) for checking a number is odd or even

```
x = num & 1
```

- If $x = 0$, num is even.
- If $x = 1$, num is odd.

(3) Q03: Use bitwise OR(|), XOR(^), AND(&) to toggle a specific bit. Use bitwise left-shift or right-shift for multiplication & division by power of 2

```
num |= (1 << i); // Toggle a specific bit using bitwise OR
num ^= (1 << i); // Toggle the same specific bit using bitwise XOR
num &= ~(1 << i); // Toggle the same specific bit using bitwise AND with a not

num >> n; // Right-shift for division by 2^n
num << n; // Left-shift for multiplication by 2^n
```

(4) Q04: How to convert 0000 1011 into 0000 0010?

Solution 01: Right shift

```
0000 1011 >> 2 = 0000 0010
```

Solution 02: Bitwise AND

```
0000 1011 & 0000 0110 = 0000 0010
```

Solution 03: XOR

```
0000 1011 ^ 0000 1001 = 0000 0010
```

(5) Q05: Use bitwise AND (&) for computing the lowest bit of a number (i.e., the rightmost 1)

```
num & (-num)
```

It may seem weird at first glance, so let's see a toy example:

```
#include <stdio.h>

void printBinary(short int num) {
    short int bits = sizeof(num) * 8; // Number of bits in an integer (typic
```

```

    for (int i = bits - 1; i >= 0; i--) {
        short int bit = (num >> i) & 1; // Extract the i-th bit
        printf("%d", bit);
    }
    printf("\n");
}

int main() {
    short int binaryNum = 0b11010000;

    printf("num = %d \n", binaryNum);
    printf("Binary representation of num: ");
    printBinary(binaryNum);

    printf("\n-num = %d \n", -binaryNum);
    printf("Binary representation of num: ");
    printBinary(-binaryNum);

    printf("\nnum & (-num) = %d \n", binaryNum & (-binaryNum));
    printf("Binary representation of num & (-num): ");
    printBinary(binaryNum & (-binaryNum));

    printf("\n");
    return 0;
}

```

The output goes:

```

num = 208
Binary representation of num: 0000000011010000

-num = -208
Binary representation of num: 1111111100110000

num & (-num) = 16
Binary representation of num: 00000000000010000

```

(6) Q06: How to clear the lowest set bit (the rightmost 1 of a binary) using bitwise operations?

```
num & (num - 1)
```

Let's look at an example:

```
#include <stdio.h>

void printBinary(short int num) {
    short int bits = sizeof(num) * 8; // Number of bits in an integer (typical)
    for (int i = bits - 1; i >= 0; i--) {
        short int bit = (num >> i) & 1; // Extract the i-th bit
        printf("%d", bit);
    }
    printf("\n");
}

int main() {
    short int binaryNum = 0b11010000;

    // printf("num = %d \n", binaryNum);
    printf("Binary representation of num: ");
    printBinary(binaryNum);

    // printf("\nnum = %d \n", binaryNum - 1);
    printf("Binary representation of (num - 1): ");
    printBinary(binaryNum - 1);

    printf("\nnum & (num - 1) = %d \n", binaryNum & (-binaryNum));
    printf("Binary representation of num & (num - 1): ");
    printBinary(binaryNum & (binaryNum - 1));

    printf("\n");
    return 0;
}
```

The output:

```
Binary representation of num: 0000000011010000
Binary representation of (num - 1): 0000000011001111
```

```
num & (num - 1) = 16
```

```
Binary representation of num & (num - 1): 0000000011000000
```

Refer to [338. Counting Bits](#) on LeetCode for the application of this technique. The solution to this problem goes like:

```
class Solution {
public:
    vector<int> countBits(int n) {
        vector<int> ans(n + 1);
        // cf. vector<int> ans(n + 1, 5); | vector<int> ans{5, 6};
        for (int i = 1; i <= n; ++i) {
            ans[i] = ans[i & (i - 1)] + 1;
        }
        return ans;
    }
};
```

(7) Q07: Identify the sizes of the structs below

```
struct Example_01 {
    char a;    // 1 byte
    int b;     // 4 bytes (typically, depends on the system)
    char c;    // 1 byte
};

struct Example_02 {
    char a;    // 1 byte
    int b;     // 4 bytes (typically, depends on the system)
    char c;    // 1 byte
    float d;   // 4 bytes
};

struct Example_03 {
    char a;    // 1 byte
    int b;     // 4 bytes (typically, depends on the system)
    char c;    // 1 byte
    double d;  // 8 bytes
};
```

```
struct Example_04 {
    char a;      // 1 byte
    int b;       // 4 bytes (typically, depends on the system)
    char c;      // 1 byte
    long d;      // 4 bytes
};

struct Example_05 {
    char a;      // 1 byte
    int b;       // 4 bytes (typically, depends on the system)
    char c;      // 1 byte
    long long d; // 8 bytes
};
```

Alignment and Padding

Each data type in a struct has a preferred alignment, usually based on its size. The compiler inserts padding between members and sometimes at the end of the struct to ensure each member is aligned properly. The alignment of a struct as a whole is typically determined by its largest member.

Get Yu-Cheng (Morton) Kuo's stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email

Subscribe

Let's see the C code. Notice that Example_03, Example_05, Example_06 are quite different from others.

```
#include <stdio.h>

struct Example_01 {
    char a;      // 1 byte
    int b;       // 4 bytes (typically, depends on the system)
```



```

    char c;    // 1 byte
};

struct Example_02 {
    char a;    // 1 byte
    int b;     // 4 bytes (typically, depends on the system)
    char c;    // 1 byte
    float d;   // 4 bytes
};

struct Example_03 {
    char a;    // 1 byte
    int b;     // 4 bytes (typically, depends on the system)
    char c;    // 1 byte
    double d;  // 8 bytes
};

struct Example_04 {
    char a;    // 1 byte
    int b;     // 4 bytes (typically, depends on the system)
    char c;    // 1 byte
    long d;    // 4 bytes
};

struct Example_05 {
    char a;    // 1 byte
    int b;     // 4 bytes (typically, depends on the system)
    char c;    // 1 byte
    long long d; // 8 bytes
};

struct Example_06 {
    char a;    // 1 byte
    int b;     // 4 bytes (typically, depends on the system)
    char c;    // 1 byte
    short int d; // 2 bytes
};

int main() {
    printf("Size of struct Example_01: %zu bytes\n", sizeof(struct Example_01));
    printf("Size of struct Example_02: %zu bytes\n", sizeof(struct Example_02));
    printf("Size of struct Example_03: %zu bytes\n", sizeof(struct Example_03));
    printf("Size of struct Example_04: %zu bytes\n", sizeof(struct Example_04));
    printf("Size of struct Example_05: %zu bytes\n", sizeof(struct Example_05));
    printf("Size of struct Example_06: %zu bytes\n", sizeof(struct Example_06));
    return 0;
}

```

And the output:

```
Size of struct Example_01: 12 bytes
Size of struct Example_02: 16 bytes
Size of struct Example_03: 24 bytes
Size of struct Example_04: 16 bytes
Size of struct Example_05: 24 bytes
Size of struct Example_05: 12 bytes
```

Now, let's figure out why. It's due to the 4-byte/8-byte alignment requirement. We'll use memory layouts to explain this.

```
struct Example_01 {
    char a;      // 1 byte
    int b;       // 4 bytes (typically, depends on the system)
    char c;      // 1 byte
};
```

struct Example_01:

- char a; // 1 byte
- Padding: 3 bytes (to align int b)
- int b; // 4 bytes
- char c; // 1 byte
- Padding: 3 bytes (to align the struct to a multiple of 4 bytes, for array purposes)
- Total size: $1 + 3 + 4 + 1 + 3 = 12$ bytes

```
struct Example_02 {  
    char a;    // 1 byte  
    int b;     // 4 bytes (typically, depends on the system)  
    char c;    // 1 byte  
    float d;   // 4 bytes  
};
```

struct Example_02 (with float):

- char a; // 1 byte
- Padding: 3 bytes (to align int b)
- int b; // 4 bytes
- char c; // 1 byte
- Padding: 3 bytes (to align float d)
- float d; // 4 bytes
- Total size: $1 + 3 + 4 + 1 + 3 + 4 = 16$ bytes

```
struct Example_03 {  
    char a;    // 1 byte  
    int b;     // 4 bytes (typically, depends on the system)  
    char c;    // 1 byte  
    double d;  // 8 bytes  
};
```

struct Example_03 (with double):

- The total size would be: 1 (for `a`) + 3 (padding) + 4 (for `b`) + 1 (for `c`) + 7 (padding) + 8 (for `d`) = 24 bytes.

```
struct Example_04 {  
    char a;      // 1 byte  
    int b;       // 4 bytes (typically, depends on the system)  
    char c;      // 1 byte  
    long d;      // 4 bytes  
};
```

struct Example_04 (assuming `long` is 4 bytes):

- `char a; // 1 byte`
- Padding: 3 bytes (to align `int b`)
- `int b; // 4 bytes`
- `char c; // 1 byte`
- Padding: 3 bytes (to align `long d`)
- `long d; // 4 bytes`
- Total size: $1 + 3 + 4 + 1 + 3 + 4 = 16$ bytes

```
struct Example_05 {  
    char a;        // 1 byte  
    int b;         // 4 bytes (typically, depends on the system)  
    char c;        // 1 byte  
    long long d;   // 8 bytes  
};
```

struct Example_05:

- The total size would be: 1 (for `a`) + 3 (padding) + 4 (for `b`) + 1 (for `c`) + 7 (padding) + 8 (for `d`) = 24 bytes.

```
struct Example_06 {  
    char a;        // 1 byte  
    int b;         // 4 bytes (typically, depends on the system)  
    char c;        // 1 byte  
    short int d;   // 2 bytes  
};
```

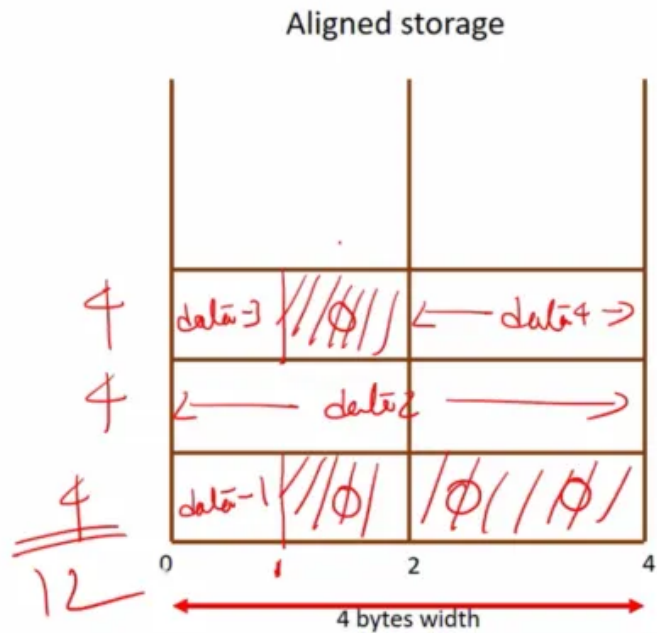
struct Example_06:

- The total size would be: 1 (for `a`) + 3 (padding) + 4 (for `b`) + 1 (for `c`) + 1 (padding) + 2 (for `d`) = 12 bytes.

Structure padding

```
struct data
{
    char data1 ;
    int data2 ;
    char data3 ;
    short data4 ;
};
```

sizeof(struct data) = 12 bytes



Retrieved from [Microcontroller Embedded C Programming Lecture 149 | Calculating structure size manually with and without padding](#)

(8) Q08: Explain the keyword volatile

In embedded systems, which often deal with I/O, interrupts, and real-time operating systems (RTOS), the `volatile` keyword is particularly important. A variable modified by `volatile` indicates that it may be updated unexpectedly. Therefore, it informs the compiler not to optimize the code involving this variable. Instead, each time the variable is accessed, it should read the latest value from the actual memory address, rather than reading the value from a register.

Common applications of `volatile` :

- (1) Hardware registers of devices (such as status registers)
- (2) Modifying global variables in multi-threaded environments
- (3) Modifying global variables that may be altered in an Interrupt Service Routine (ISR)

Combination of `const` and `volatile`:

```
extern const volatile unsigned int rt_clock;
```

This is a common declaration in RTOS kernels: `rt_clock` usually refers to the system clock, which is frequently updated by clock interrupts. Therefore, it is `volatile`. As such, when used, the compiler should fetch its value from memory each time. Typically, `rt_clock` has only one writer (the clock interrupt), and its use elsewhere is usually read-only. Hence, it is declared as `const`, indicating that this variable should not be modified here. Thus, `volatile` and `const` are not contradictory. In fact, it is meaningful for an object to possess both these attributes simultaneously.

Embedded systems programmers often deal with hardware, interrupts, RTOS, etc., and all these require the use of `volatile` variables.

Literally, the term `volatile` means prone to change. That is to say, during the execution of a program, some variables may be unexpectedly altered. To save time, the optimizer sometimes does not re-read the real value of this variable but reads a backup stored in a register. In this way, the real value of the variable might be “optimized” away by the optimizer, or in fashionable terms, it is “harmonized.”

Using this modifier is like telling the compiler not to be lazy and to honestly re-read the variable! Perhaps my explanation so far is too “colloquial,” so let me quote a “master’s” standard explanation:

The original intent of `volatile` is “prone to change.”

Since accessing registers is faster than accessing RAM, compilers generally optimize to reduce access to external RAM. However, this may result in reading stale data. When the value of a variable declared with `volatile` is required, the

system always re-reads the data from its memory location, even if the preceding instruction has just read from that location. Moreover, the read data is immediately saved.

To be precise, the optimizer must carefully re-read the value of this variable every time it is used, instead of using a backup saved in a register.

(9) References

1. 蔡文龍、何嘉益、張志成、張力元、歐志信、陳士傑（2021）。C & C++程式設計經典（第五版）。台北：碁峯資訊。
2. 劉邦鋒（2019）。由片語學習C程式設計（第二版）。台北：國立臺灣大學出版中心。
3. My interview experience [2023/11/30]
4. Microcontroller Embedded C Programming Lecture 149| Calculating structure size manually with and without padding [2022]
5. C/C++ — 常見 C 語言觀念題目總整理（適合考試和面試） [2017]
6. Classic C Interview Questions [2016]
7. [面試] 聯發科技 MTK (內含考題) [2011]
8. <https://github.com/doocs/leetcode/tree/main>

C

Bitwise Operations

Bit

Struct

Volatile



Follow

Published in Nerd For Tech

12.9K followers · Last published Aug 27, 2025

NFT is an Educational Media House. Our mission is to bring the invaluable knowledge and experiences of experts from all over the world to the novice. To know more about us, visit <https://www.nerdfortech.org/>.



Follow

Written by Yu-Cheng (Morton) Kuo

264 followers · 126 following

CS/ML blog with C++/C/Python. C++ Software Engineer. Email: morton.kuo.28@gmail.com

Responses (1)



Write a response

What are your thoughts?



lanjoyner

Mar 21, 2024



C is an old primitive system language full of traps and flaws.

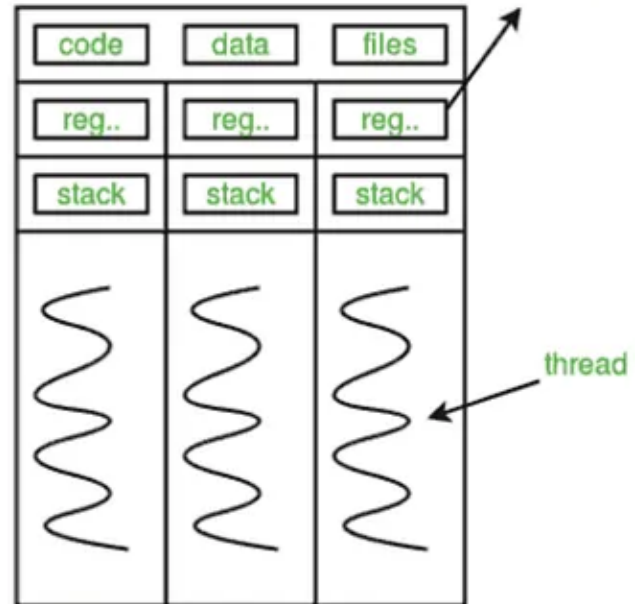
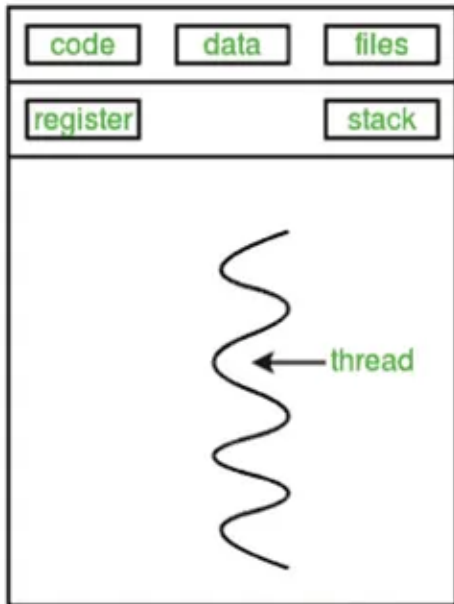
Anything using bitwise operations is tied to particular representations on machines. The knowledge of that should be in the compiler code generator.

Programmers dealing with this is... [more](#)



[Reply](#)



More from Yu-Cheng (Morton) Kuo and Nerd For Tech



 In Nerd For Tech by Yu-Cheng (Morton) Kuo

Hands-On Multithreading with C++ 01—Overview

From toy examples to a real-world instance

Dec 24, 2023  13  1





Top +30 Best GNOME Extensions | 2023 Updated

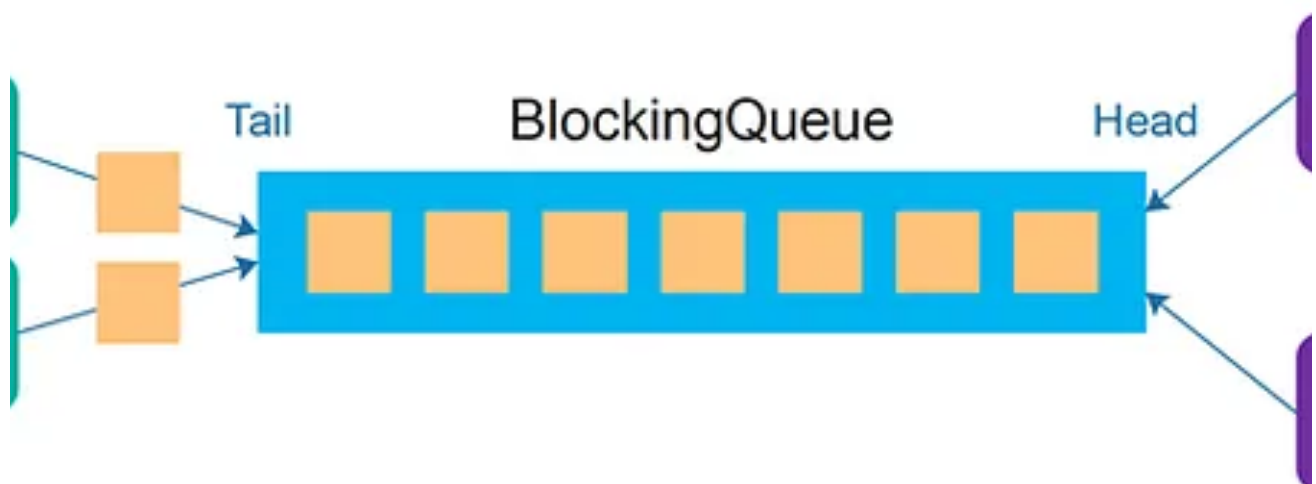
Use these +30 handpick gnome extensions for a better experience rich with features and power to customize your desktop — updated regularly

Apr 21, 2021  202  1



ads

Con





In Nerd For Tech by Yu-Cheng (Morton) Kuo

Hands-On Multithreading with C++ 04—Producer-Consumer Problem

Get to know the use of `std::unique_lock`, `std::mutex`, & `std::condition_variable`

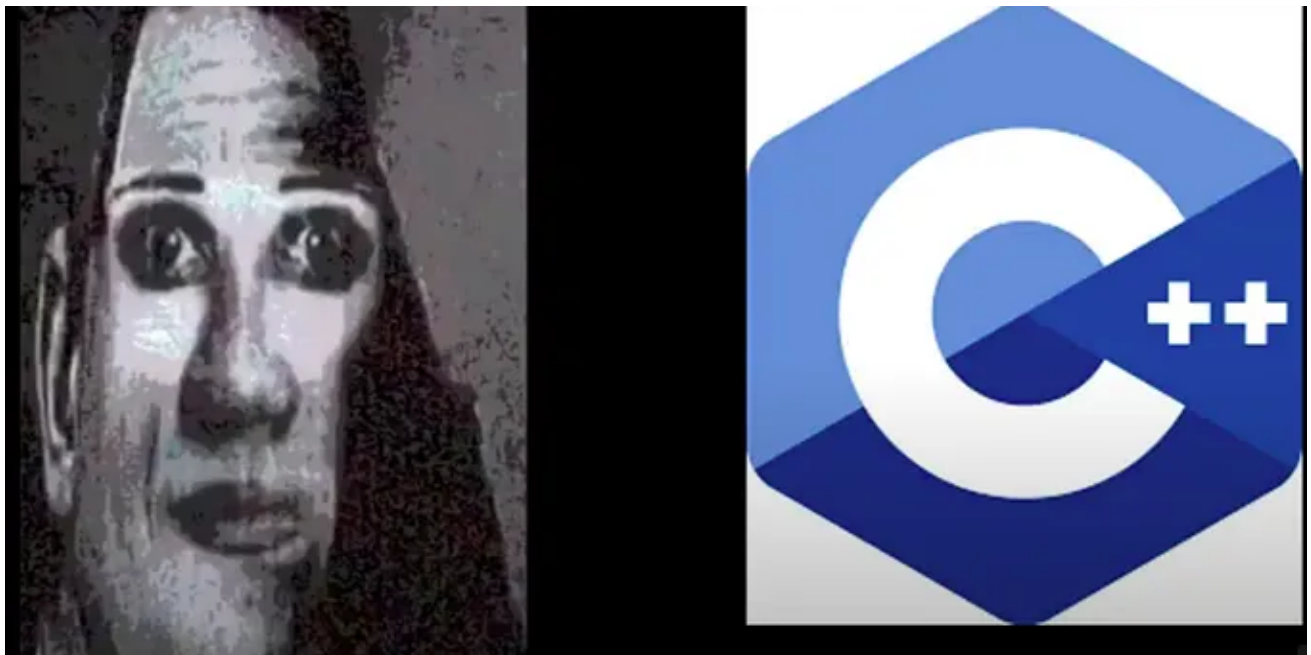
Jan 4, 2024 🖱 54

See all from Yu-Cheng (Morton) Kuo



See all from Nerd For Tech

Recommended from Medium



Francisco Zavala

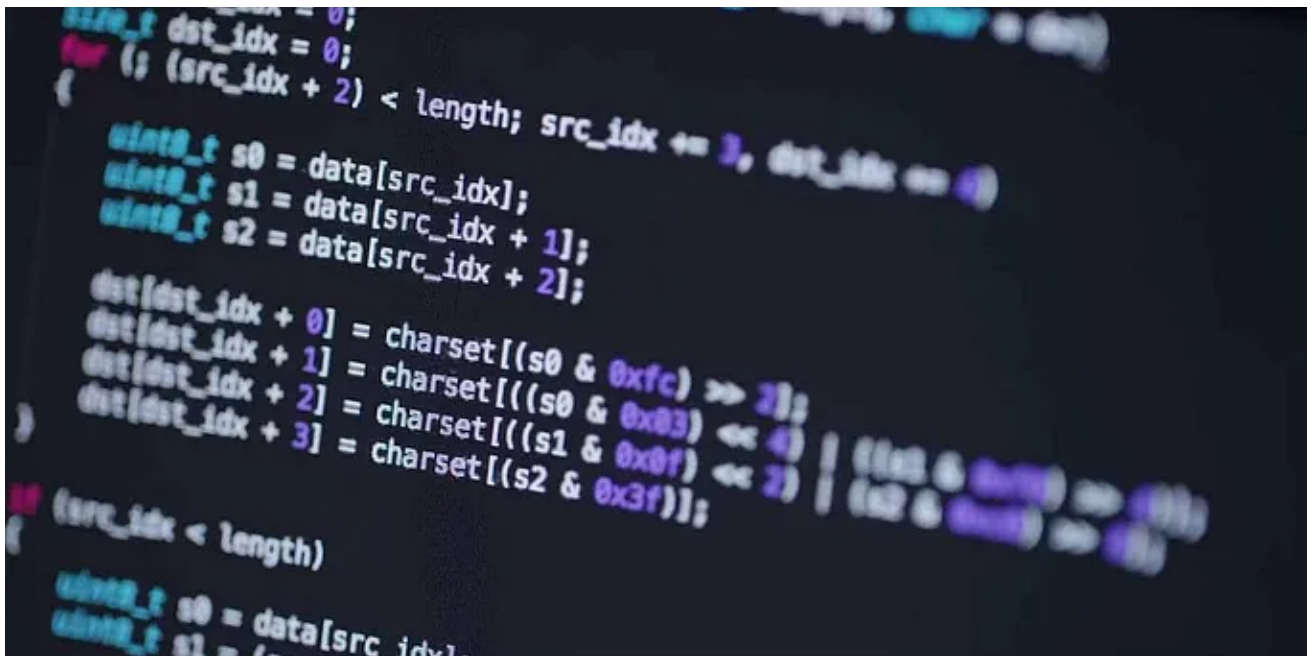
The Evolution of C++ Metaprogramming.


Explore the challenges of C++ metaprogramming and how languages like LISP and Ada shaped its evolution.



Apr 20 🖱 19



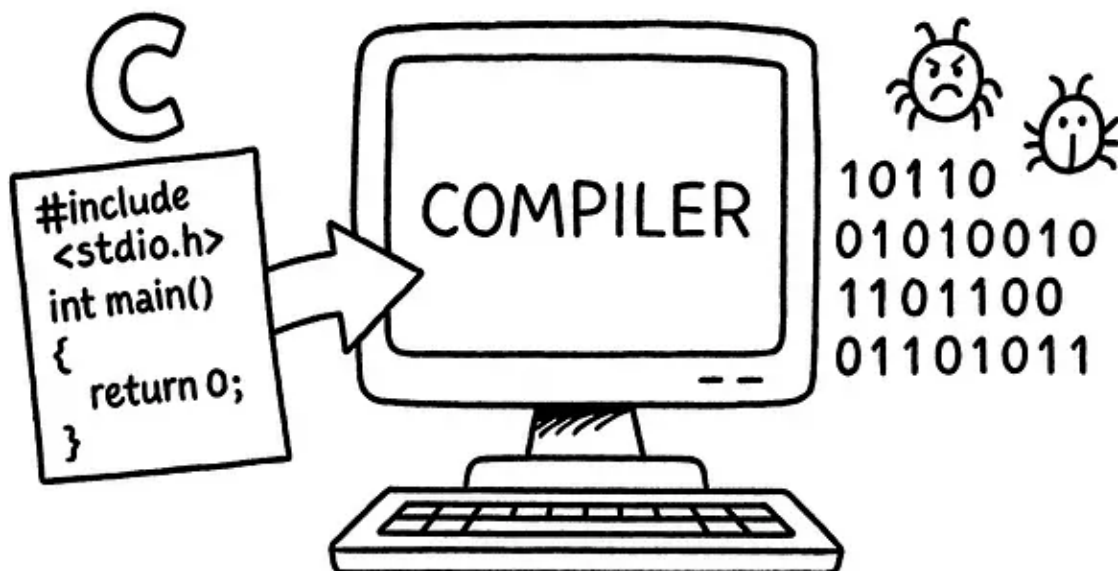


 Palak Thapar

C++ Interview questions- Part 4


Part 4 of series of interview questions that are frequently asked in C++ interviews

★ Oct 4 🖱 51



Can You Really Trust Your Compiler?

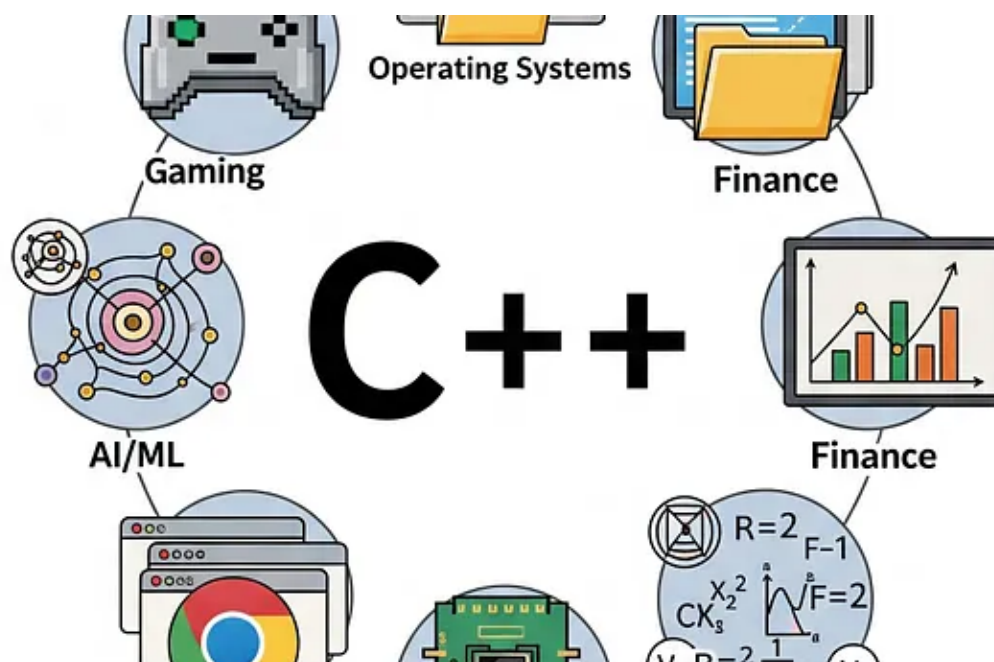
Metric	Old (virtual)	New (Ref)	Delta	Percentage
Time to publish	2760 ns	800 ns	-1960	-71 %
★ 👍 436 💬 16 Strategy onTrade	960 ns	330 ns	-630	-65 %
Time-to-end per tick	4600 ns	2450 ns	-2150	-47 %
Messages/sec	140k	167k	+27k	+19 %
Cache miss ratio	3.69 %	2.72 %	-0.97	-26 %
Cache per message	360k	280k	-80k	-22 %

 Evgenii

Polymorphism Without virtual in C++: Concepts, Traits, and Ref

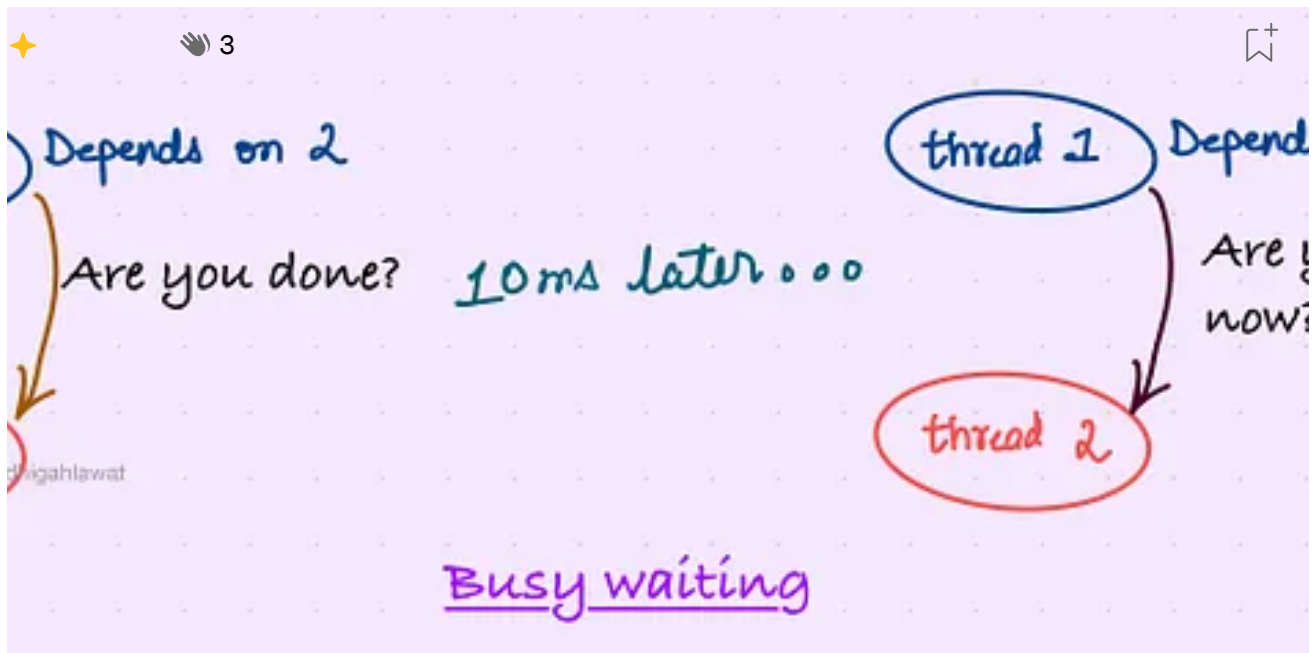
How polymorphism was reworked in the Flox C++ framework: replacing virtual with statically generated vtables using concepts.

Jul 9 👍 5 💬 2



Understanding RAI in C++ — The Heart of Resource Management

Let's talk about a cornerstone of modern C++ programming that makes your code safer, cleaner, and more robust: RAI (Resource Acquisition...



 In Code Like A Girl by Nidhi Gahlawat

Let Threads Sleep: A Simple Guide To Condition Variables in C++

When your threads need to talk, let them do it politely.

★ Jun 16 🖱 302 💬 5



See more recommendations