

Username: Pralay Patoria **Book:** Coding Interviews: Questions, Analysis & Solutions. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.


Divergent Thinking Skills

Divergent thinking is a thought process to generate creative ideas in a short period of time by looking for new opportunities and ways to get things done. Instead of taking obvious steps and walking in a straight line from a problem to the solution, we try to see different aspects of the situation and use unusual points of view.

Interviewers pay a lot of attention to candidates' divergent thinking skills because divergent thinking skills demonstrate creativity and passion to explore new solutions. Sometimes interviewers intentionally disallow candidates from taking the traditional solutions. What they expect is for candidates to think from creative perspectives. For example, some interviewers ask candidates to add, subtract, multiply, and divide without using $+$, $-$, \times , and \div operations. They expect candidates to jump outside the boundary of arithmetic calculation and find solutions with bit operations.

Divergent thinking skills also demonstrate the breadth and depth of knowledge. Candidates are able to explore solutions from various points of view only when they have deep understanding of various domains. For instance, there is a popular interview problem that requires calculating $1+2+\dots+n$ without using loops, multiplication and division, keywords including `if`, `switch`, and `case`, as well as the conditional operator. If a candidate has a broad and deep understanding of C++, he or she could solve it with constructors, virtual functions, function pointers, and template specialization.

Calculating $1+2+\dots+n$

 **Question 99** How do you calculate $1+2+\dots+n$ without multiplication, division, key words including `for`, `while`, `if`, `else`, `switch`, and `case`, as well as a conditional operator (`A?B:C`)?

As we know, $1+2+\dots+n = n(n+1)/2$. However, this equation cannot be utilized because multiplication and division are disallowed. $1+2+\dots+n$ is also usually calculated iteratively or recursively. Since `for` and `while` have been disallowed, loops cannot be utilized. If we are going to calculate recursively, we have to use `if` or the conditional operator to determine when to end the recursion. However, neither are they allowed.

Based on Constructors

The purpose of a loop is to repeat execution n times. Actually, we can repeat execution without `for` and `while` statements. For example, a type is defined first and its constructor will be invoked n times if n instances are created. If the code to accumulate is inserted into the constructor, $1+2+\dots+n$ is calculated, as shown in [Listing 8-26](#).

Listing 8-26. C++ Code to Calculate $1+2+\dots+n$ (Version 1)

```
class Temp {
public:
    Temp() { ++N; Sum += N; }

    static void Reset() { N = 0; Sum = 0; }
    static unsigned int GetSum() { return Sum; }

private:
    static unsigned int N;
    static unsigned int Sum;
};

unsigned int Temp::N = 0;
unsigned int Temp::Sum = 0;

unsigned int Sum_Solution1(unsigned int n) {
    Temp::Reset();

    Temp *a = new Temp[n];

    delete []a;

    a = NULL;

    return Temp::GetSum();
}
```

Based on Virtual Functions

The difficulty with utilizing recursion without `if` and the conditional operator is that we cannot determine when to stop. If it is difficult to end recursion inside a function, how about defining two functions? The first function takes the calculation, and the second one acts as a terminator. We have to choose one of them at every step during execution. That is to say, the terminator is selected when n becomes 0; otherwise, the first function is selected for all non-zero n .

Therefore, we have another solution, as shown in [Listing 8-27](#).

Listing 8-27. C++ Code to Calculate $1+2+\dots+n$ (Version 2)

```
class A;

A* Array[2];

class A {
public:
    virtual unsigned int Sum (unsigned int n) {
        return 0;
    }
};

class B: public A {
public:
    virtual unsigned int Sum (unsigned int n) {
        return Array[!n]->Sum(n-1) + n;
    }
};

int Sum_Solution2(int n) {
    A a;
    B b;
    Array[0] = &a;
    Array[1] = &b;

    int value = Array[1]->Sum(n);

    return value;
}
```

It makes choices based on virtual functions `A::Sum` and `B::Sum` in the code above. When `n` is not zero, the result of `!n` is 1 (`true`), so it invokes `B::Sum` to accumulate; otherwise, it tells `A::Sum` to stop when `n` is zero.

Based on Function Pointers

Virtual functions are not available in C, so the preceding solution does not work for C programmers. Fortunately, we could simulate virtual functions with function pointers, as shown in [Listing 8-28](#).

Listing 8-28. C Code to Calculate $1+2+\dots+n$ (Version 3)

```
typedef unsigned int (*fun)(unsigned int);

unsigned int Solution3_Teminator(unsigned int n) {
    return 0;
}

unsigned int Sum_Solution3(unsigned int n) {
    static fun f[2] = {Solution3_Teminator, Sum_Solution3};
    return n + f[!n](n - 1);
}
```

The function `Sum_Solution3` calls itself recursively till `n` is decreased to 0 because `!!0` is 0 (`false`).

Based on Templates

We could also utilize compilers to calculate with the code in [Listing 8-29](#).

Listing 8-29. C++ Code to Calculate $1+2+\dots+n$ (Version 4)

```
template <unsigned int n> struct Sum_Solution4 {
    enum Value { N = Sum_Solution4<n - 1>::N + n };
};
```

```
template <N> struct Sum_Solution4<1> {
    enum Value { N = 1};
};
```

The value of `Sum_Solution4<100>::N` is the result of $1+2+\dots+100$. When the C++ compiler sees `Sum_Solution4<100>`, it generates code for the class template `Sum_Solution4` with 100 as the template argument. Note that the type `Sum_Solution4<100>` depends on the type `Sum_Solution4<99>` because in the code above `Sum_Solution4<100>::N=Sum_Solution4<99>::N+100`. The compiler generates code for `Sum_Solution4<99>`, which depends on `Sum_Solution4<98>`. The recursive process stops at the type `Sum_Solution4<1>` because it is explicitly defined.

The process to calculate `Sum_Solution4<100>::N` for $1+2+\dots+100$ is in the compiling time, so the input n should be a constant value. Additionally, C++ compilers have constraints on the depth of recursive compiling, so n cannot be a large value.


Source Code:

099_Accumulate.cpp

Test Cases:

- Normal Test Cases (Input 5 and 10 to calculate $1+2+\dots+5$ and $1+2+\dots+10$)
- Boundary Test Cases (Input 1)

Implementation of +, -, *, and /

 **Question 100** How do you implement a function to add two integers without utilization of arithmetic +, -, *, and / operators?

First of all, let's analyze how we add decimals with three steps. Take $5+17=22$ as an example. The first step is to add digits without carries, and we get 12. (The digits of the least significant bits are 5 and 7, and the sum of them is 2 without a carry. The tens digits are 0 and 1, and the sum of them is 1.) The second step is for carries. There is a 1 carry for the least significant digits, and the actual value for the carry is 10 ($1 \times 10 = 10$ in decimal). The last step is to add the sums of the two steps together, and we get $12+10=22$.

What can we calculate without arithmetic operators? It seems that we do not have other choices except bit operations. Bit operations are based on binary numbers. Let's have a try at adding binary numbers with three steps similar to the process above.

The binary representations of 5 and 17 are 101 and 10001 respectively. We get 10100 without carries in the first step. (They are two 1 digits at the least significant digits. In binary $1+1=10$, and the sum is 0 without carries.) Carries are recorded in the second step. There is a carry 1 at the least significant digits to add 101 and 10001, and the actual value of the carry is 10 ($1 \times 10 = 10$ in binary). When 10100 and 10 are added, the sum is 10110, and it is 22 in decimal. Therefore, binary numbers can also be added in three steps.

Let's move on now to replace the steps with binary operations. The first step is to add digits without carries. The results are 0 when adding 0 and 0, as well as adding 1 and 1. The results are 1 when adding 0 and 1, as well as 1 and 0. Note that these results are the same as bitwise XOR operations. The bitwise XOR results of 0 and 0, as well as 1 and 1, are 0, while the XOR results of 0 and 1, as well as 1 and 0, are 1.

The second step is for carries. There are no carries while adding 0 and 0, 0 and 1, as well as 1 and 0. There is a carry only when 1 and 1 are added. These results are the same as bitwise AND operations. Additionally, we have to shift carries to the left for one bit to get the actual carry value.

The third step is to add results of the first two steps. The adding operations can be replaced with bit operations again. These two steps above are repeated until there are carries.

It is time to write code after we have clear picture of how to simulate addition with bit operation. The sample code is shown in [Listing 8-30](#).

Listing 8-30. Java Code to Add

```
int add(int num1, int num2) {
    int sum, carry;

    do {
        sum = num1 ^ num2;
        carry = (num1 & num2) << 1;
        num1 = sum;
        num2 = carry;
    } while (num2 != 0);


    return num1;
}
```

Source Code:

100_103_ArithmeticOperations.java

Test Cases:

- Add positive numbers, negative numbers, and 0

 **Question 101** How do you implement a function for the subtraction operation without utilization of arithmetic +, -, *, and / operators?

Subtraction can be implemented with addition because $a-b=a+(-b)$. Additionally, $-b$ can be gotten with bit operations because $-b=\sim b+1$. We have simulated addition with bit operations, so there are no problems in simulating subtraction with bit operations, as shown in [Listing 8-31](#).

Listing 8-31. Java Code to Subtract

```
int subtract(int num1, int num2) {
    num2 = add(~num2, 1);
    return add(num1, num2);
}
```

Source Code:

100_103_ArithmeticOperations.java

Test Cases:

- Two numbers for subtraction are positive numbers, negative numbers, or 0

Question 102 How do you implement a function for the multiplication operation without utilization of arithmetic $+$, $-$, $*$, and $/$ operators?

As we know, $a \times n$ ($n \geq 0$) is the same as $a+a+\dots+a$ (for n times). Therefore, we could invoke the `add` method above in a loop to implement multiplication. If we are going to multiply a number with n , we have to invoke `add` for n times. Is it possible to reduce the times that are necessary to be invoked?

Let's take $a \times 6$ as an example. The number 6 is 110 in binary, which is 2 (10 in binary) plus 4 (100 in binary), so $a \times 6 = a \times 2 + a \times 4$. Note that $a \times 2$ and $a \times 4$ can be calculated based on left-shift operations, which are $a \ll 1$ and $a \ll 2$, respectively.

Therefore, $a \times n$ can be implemented with a left-shift and additions. Since there are $O(\log n)$ 1 bits in the binary representation of n , the new solution invokes the `add` method for $O(\log n)$ times, as implemented in [Listing 8-32](#).

Listing 8-32. Java Code to Multiply

```
int multiply(int num1, int num2) {
    boolean minus = false;

    if ((num1 < 0 && num2 > 0) || (num1 > 0 && num2 < 0))
        minus = true;

    if (num1 < 0)
        num1 = add(~num1, 1);

    if (num2 < 0)
        num2 = add(~num2, 1);

    int result = 0;
    while (num1 > 0) {
        if ((num1 & 0x1) != 0) {
            result = add(result, num2);
        }

        num2 = num2 << 1;
        num1 = num1 >> 1;
    }

    if (minus)
        result = add(~result, 1);

    return result;
}
```

Source Code:

100_103_ArithmeticOperations.java

Test Cases:

- The two numbers for multiplication are positive numbers, negative numbers, or 0

Question 103 How do you implement a function to divide an integer by another without utilization of arithmetic +, -, *, and / operators?

For two positive integers, if $a/b=n$, $a-b \times n \geq 0$ and $a-b \times (n+1) < 0$. Therefore, the division can be implemented with the subtract method in a loop. It invokes the method subtract for $O(n)$ times when the result is n .

There is a more efficient solution available. If the result of a/b is n , and m bits are 1 in the binary representation of n (the n_1, n_2, \dots, n_m bit counting from the right end),

$$n = \sum_{i=1}^m 2^{n_i-1}$$

and

$$a - b \times \sum_{i=1}^m 2^{n_i-1} \geq 0$$

Similar to before,

$$b \times 2^{n_t-1}$$

can be implemented with left-shift operation. Therefore, the division can be implemented with the code shown in [Listing 8-33](#).

Listing 8-33. Java Code to Divide

```
int divide(int num1, int num2) {
    if (num2 == 0)
        throw new ArithmeticException("num2 is zero.");

    boolean minus = false;
    if ((num1 < 0 && num2 > 0) || (num1 > 0 && num2 < 0))
        minus = true;

    if (num1 < 0)
        num1 = add(~num1, 1);
    if (num2 < 0)
        num2 = add(~num2, 1);

    int result = 0;
    for (int i = 0; i < 32; i=add(i, 1)) {
        result = result << 1;
        if ((num1 >> (31 - i)) >= num2) {
            num1 = subtract(num1, num2 << (31 - i));
            result = add(result, 1);
        }
    }

    if (minus)
        result = add(~result, 1);

    return result;
}
```

Source Code:

100_103_ArithmeticOperations.java

Test Cases:

- The two numbers for multiplication are positive numbers, negative numbers, or 0

Final/Sealed Classes in C++

Question 104 Please design a class in C++ that cannot be inherited.

In C# there is a keyword `sealed`, indicating a class that cannot be a parent class of other classes. In Java, there is a similar keyword `final`. There are no such keywords in C++, so we have to implement a mechanism for `sealed` or `final` classes.

Based on Private Constructors

As we know, a constructor of a C++ class invokes the constructor of its parent class, and a destructor invokes its parent class's destructor. If a constructor or destructor is declared as a private function, it cannot be invoked outside the class where it is defined. Therefore, a class whose constructors and destructor are private cannot have children classes. If we are going to inherit it, a compiling error is raised.

How do you create and release instances of a class whose constructors and destructor are private? We may utilize public static member functions, as shown in [Listing 8-34](#).

Listing 8-34. C++ Code for Sealed/Final Classes (Version 1)

```
class SealedClass1 {
public:
    static SealedClass1* GetInstance() {
        return new SealedClass1();
    }

    static void DeleteInstance( SealedClass1* pInstance) {
        delete pInstance;
    }

private:
    SealedClass1() {}
    ~SealedClass1() {}
};
```

It is a bit inconvenient to use these classes because there are some differences from normal classes. For example, we can only create instances of a `SealedClass1` on the heap, but we cannot create instances on the stack.

Based on Private Constructors

It is a little bit tricky to define a `sealed` / `final` class whose instances can be created on the stack. [Listing 8-35](#) contains the sample code.

Listing 8-35. C++ Code for Sealed/Final Classes (Version 2)

```
template <typename T> class MakeSealed {
    friend T;

private:
    MakeSealed() {}
    ~MakeSealed() {}
};

class SealedClass2 : virtual public MakeSealed<SealedClass2> {
public:
    SealedClass2() {}
    ~SealedClass2() {}
};
```

We can create instances of the class `SealedClass2` on both the heap and stack space, so it is more convenient to use than `SealedClass1`.

Even though the constructor and destructor of `MakeSealed<SealedClass2>` are defined as private functions, they can be invoked by the class `SealedClass2` because `SealedClass2` is a friend class of `MakeSealed<SealedClass2>`.

The compiler raises an error when we are trying to derive a new class from `SealedClass2`, such as the class `Try` in [Listing 8-36](#).

Listing 8-36. C++ Code to Derive a Class from SealedClass2

```
class Try2 : public SealedClass2 {
public:
    Try2() {}
};
```

```
~Try2() {}  
};
```

Since the class `SealedClass2` is virtually inherited from `MakeSealed<SealedClass2>`, the constructor of `Try` skips `SealedClass2` and invokes the constructor of `MakeSealed<SealedClass2>`. The class `Try` is not a friend of `MakeSealed<SealedClass2>`, so it raises a compiling error to invoke the constructor of `MakeSealed<SealedClass2>` from `Try`.

Therefore, we cannot inherit children classes from `SealedClass2`, and it is a `sealed / final` class.

The source code for `SealedClass2` can be compiled smoothly in Visual Studio, but it cannot be compiled in GCC. Currently, the template argument cannot be a friend type in GCC. Therefore, the second type has problems from the perspective of portability.

Source Code:

```
104_SealedClass.cpp
```

Array Construction

Question 105 Given an array $A[0, 1, \dots, n-1]$, please construct an array $B[0, 1, \dots, n-1]$ in which $B[i]=A[0]\times A[1]\times \dots \times A[i-1]\times A[i+1]\times \dots \times A[n-1]$. No division should be involved to solve this problem

$$\prod_{j=0}^{n-1} A[j] / A[i]$$

If there are no limitations to utilize the division operation, $B[i]$ can be calculated by $\prod_{j=0}^{n-1} A[j] / A[i]$. Be careful when $A[i]$ is zero.

It is not allowed to use division here, so we have to explore alternatives. An intuitive solution to calculate $B[i]$ is to multiply $n-1$ numbers, so the time complexity to construct the array B is $O(n^2)$. There are more efficient solutions available.

$B[i]$ is $A[0]\times A[1]\times \dots \times A[i-1]\times A[i+1]\times \dots \times A[n-1]$, which is the multiplication result of two sequences $A[0]\times A[1]\times \dots \times A[i-1]$ and $A[i+1]\times \dots \times A[n-2]\times A[n-1]$. Therefore, the array B can be visualized as a matrix, as shown in Figure 8-4. The multiplication result of numbers in the i^{th} row is $B[i]$.

Let's define $C[i]=A[0]\times A[1]\times \dots \times A[i-1]$ and $D[i]=A[i+1]\times \dots \times A[n-2]\times A[n-1]$, so $B[i]=C[i]\times D[i]$. Note that $C[i]=C[i-1]\times A[i-1]$, and $D[i]=D[i+1]\times A[i+1]$. Therefore, $C[i]$ can be calculated in a top down order, and $D[i]$ can be calculated in bottom up order. After $C[i]$ and $D[i]$ are calculated, they are multiplied for $B[i]$.

| | | | | | | |
|-----------|-------|-------|-------|-----------|-----------|-----------|
| B_0 | 1 | A_1 | A_2 | ... | A_{n-2} | A_{n-1} |
| B_1 | A_0 | 1 | A_2 | ... | A_{n-2} | A_{n-1} |
| B_2 | A_0 | A_1 | 1 | ... | A_{n-2} | A_{n-1} |
| ... | A_0 | A_1 | ... | 1 | A_{n-2} | A_{n-1} |
| B_{n-2} | A_0 | A_1 | ... | A_{n-3} | 1 | A_{n-1} |
| B_{n-1} | A_0 | A_1 | ... | A_{n-3} | A_{n-2} | 1 |

Figure 8-4. The constructed array B is visualized as a matrix.

This solution can be implemented in Java with the code in Listing 8-37.

Listing 8-37. Java Code to Construct an Array

```
void multiply(double array1[], double array2[]){  
  
    if(array1.length == array2.length && array1.length > 0){  
  
        array2[0] = 1;  
  
        for(int i = 1; i < array1.length; ++i){  
  
            array2[i] = array2[i - 1] * array1[i - 1];  
  
        }  
  
    }  
  
}
```

```
int temp = 1;

for(int i = array1.length - 2; i >= 0; --i){

    temp *= array1[i + 1];

    array2[i] *= temp;

}

}
```

The time complexity of this solution is $O(n)$ obviously, and it is more efficient than the intuitive solution above.

Source Code:

105_ConstructArray.java

Test Cases:

- Normal Test Cases (There are positive numbers, negative numbers, and 0 in the array *A*)
- Special Test Cases (There are no, only one, or more 0s in the array *A*)