CHAPTER 5

High-Quality Code

Interview candidates will want to write code of high quality during interviews, but many of them cannot meet their interviewers' expectations. This chapter provides detailed instructions on how to write clean, complete, and robust code. While solving a set of sample coding interview problems, we will illustrate how functional, boundary, and negative test cases, as well as defensive programming, are used to produce complete and robust code.

Clearness

Interviewers make their hiring decisions based on candidates' code. Code clearness has an important impact on the decision. As shown in Figure 5-1, handwriting, layout, and naming are three factors of code clearness important in interviews.

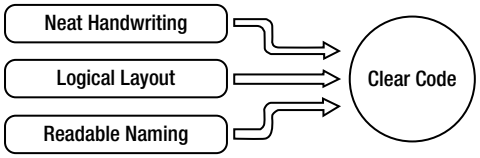

Figure 5-1. *Three factors of clear code: handwriting, layout, and naming*

First, clear code is written neatly. Engineers write code with keyboards for their daily development, but they have to write code on paper or whiteboards in most interviews. It is a good idea to practice writing code on paper during interview preparation and, if you have a whiteboard, a little practice can make a big difference. Many candidates' handwriting is illegible when they feel rushed, especially when they find problems in their own code and have to make some changes. The number of lines of code for interview problems is usually less than 50, so it does not take too much time to write. In other words, candidates should take their time, within reason, while writing code. The key factor is to form clear solutions and express them with neat and readable code.

Second, clear layout shows logical structure. Programmers usually write code in an integrated development environment, such as Visual Studio or Eclipse, which facilitates layout greatly. Candidates should pay attention to layout when such tools are unavailable in interviews. Candidates shouldn't write too large on a whiteboard or they may not have room to finish their code and then have to worry about cramming it in somewhere, interrupting their thinking about the code itself. Similarly, indentation level increases when there are many loops and conditional checks. Complex code without clear layout makes an interviewer confused if indentation does not show the logical structures, or curly brackets do not appear in pairs.

H. He, *Coding Interviews*

Last but not least, names in clear code are easy to read and understand. Many novices like the shortest names. They name variables as `i`, `j`, `k`, and name functions as `f`, `g`, `h`. Since such names do not show the meaning and purpose of variables and functions, they are very difficult to understand when a function gets long. It is recommended that you use word concatenations for names. For example, if a function takes a root node of a binary tree as a parameter, the parameter can be declared as `BinaryTreeNode* pRoot` in C/C++. Readable variables and function names make it simpler for interviewers to understand candidates' code. Otherwise, interviewers have to guess. For instance, is a variable `m` the maximum or minimum in an array?

■ **Tip** In order to improve code readability, it is a good practice to name variables and functions (methods) with English words or word concatenations.

Completeness

The ability to write complete code is a core competence of software developers, and it is also a key factor impacting interview results. Code is complete when it fulfills basic functional requirements, performs well on boundary inputs, and handles invalid input gracefully.

Test Cases for Completeness

In order to prevent bugs from occurring during interviews, it is a good practice for candidates to figure out a complete set of unit test cases before they begin to write code. After finishing coding, the candidate can test the code with the test cases in mind, which include functional test cases, boundary test cases, and negative test cases (Figure 5-2). If all cases pass, the candidate should feel confident in handing his or her code to the interviewer.

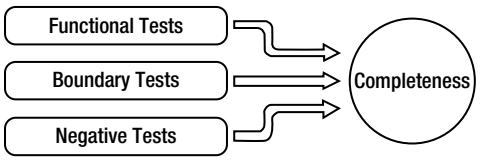

Figure 5-2. *Three types of tests to guarantee complete code*

The first type of test case is about functionality because candidates' code has to fulfill basic functional requirements. For example, if a candidate is asked to convert a string to a number, he or she may test with input strings “123”, “0”, and “-123”. The function is expected to be able to convert strings for positive and negative numbers as well as zero.

Candidates have to break through the limit of conventional thinking while designing functional test cases. Sometimes functional requirements are ignored because of conventional thinking. For example, many candidates think the coding interview problem to print from 1 to the maximal numbers with n digits is quite easy—or is it? The maximum with three digits is 999, and the maximum with four digits is 9999. These maximal values are not difficult to calculate. However, the `int` or `long` type cannot accommodate numbers with n digits when n is large. If it is required that the program is able to handle

arbitrarily large *n*, some data structures for big numbers, such as strings or arrays, may be utilized to solve this problem.

Boundary values should also be covered during unit tests. It is not rare to utilize iteration and recursion in code. What are the boundary values to end loops or recursion? Candidates may design some test cases based on these boundary values. Take the problem to convert a string to a number as an example again. Candidates should make sure their code can convert the maximal and minimal integers as expected.

Invalid inputs should also be considered while designing test cases. Cases about negative tests should be included to make sure that the program handles errors gracefully when given invalid inputs. When developing a function to convert a string to a number, strings with non-digit characters such as "1a2b3c" should be taken into consideration.

The points above are presented to consider all possible inputs according to the current requirements. In software engineering, what never changes is the fact that the requirements always change. If candidates take potential requirement changes into consideration to reduce risks in the future, they demonstrate their understanding of scalability and maintainability. Scalability and maintainability will be discussed in detail later for the question "*Partition Numbers in Arrays*" (Question 45).

Strategies to Handle Errors

When given inputs are invalid, they cause errors and the normal execution flow in a function cannot continue. It is necessary to handle errors in the function and pass error messages to the function caller. There are three common strategies to pass error messages to the caller of a function.

The first strategy is to use the return value. Many Windows APIs return 0 when they execute smoothly and return non-zero values when they fail. Microsoft defines different return values for different types of errors, so API callers have the opportunity to know the reason for failure according to the return values. The disadvantage is that the calculation result cannot be assigned to variables or passed to other functions as a parameter directly.

The second strategy is to pass calculation results via return values and set a global variable when an error occurs. Callers may assign results to variables or pass results to other functions directly with this strategy. When errors occur in many Windows APIs, they update a global variable that is accessible by the function `GetLastError`. It leaves in risks if a caller forgets to check the global variable and fails to handle the corresponding errors.

The third strategy is to throw exceptions when errors occur. Different types of exceptions are defined for different types of errors, so callers are able to handle the specific error according to the type of a caught exception. Additionally, code structure looks clear because the exception mechanism partitions a function into a `try` block for normal execution, a `catch` block to handle errors, and a `finally` block to make sure resources are cleaned up whether an exception happens or not. Some programming languages, such as C, do not support the exception mechanism. The execution flow is interrupted when an exception is thrown, and it has a negative impact on performance.

Every error-handling mechanism has its own pros and cons (Table 5-1). Which one is suitable for interviews? It depends on interviewers' requirements. When candidates are asked to solve a certain coding problem, they should figure out all possible invalid inputs and discuss how to handle errors with their interviewers.

Table 5-1. Pros and Cons of Three Strategies to Handle Errors via Return Values, Global Variables, and Exceptions

	Advantages	Disadvantages
Return Values	Consistent with system APIs	Inconvenient to use results
Global Variables	Convenient to use results	Possible for callers to forget to check the variable
Exceptions	Clear code structure isolating blocks for normal execution and error handling	Some languages don't support exceptions, and exceptions impact performance negatively

Power of Integers

■ **Question 40** Please implement the function `double Power(double base, int exponent)` to raise a number `base` to power of `exponent`. You are not allowed to call library functions, and it is not necessary to take care of overflow issues.

As we know, there is a `pow` function in the C/C++ library for power calculation. You will try to simulate the functionality of `pow` here. Similarly, there are many interview questions simulating library functions, especially functions about numeric calculations and string operations. In order to solve such problems, candidates should not only be familiar with the usages of library functions, but also understand the functional implementation inside of them.

Incomplete

It looks like a simple problem because it does not require taking care of overflow issues. Some candidates finish writing code in a minute, as shown in Listing 5-1.

Listing 5-1. *C++ Code to Calculate Power (Version 1)*

```
double Power(double baseNumber, int exponent) {
    double result = 1.0;

    for(int i = 1; i <= exponent; ++i)
        result *= baseNumber;

    return result;
}
```

A fast solution is not necessarily a good one. The code in Listing 5-1 underestimates the complexity of the problem because it only handles cases where an exponent is positive. What happens when it is negative? Or zero? Incomplete solutions are not acceptable to almost all interviewers.

Complete but Inefficient

When the exponent is negative, it calculates power with absolute value of the exponent first and then gets the reciprocal of the power result. It raises an error when it tries to get the reciprocal of zero. Therefore, the scenario with a negative base and zero as the exponent should be handled specially. How do you handle the error in such scenarios? As discussed before, there are three choices available. Candidates can describe the pros and cons of each choice to the interviewers and select a solution based on the requirements. The solution in Listing 5-2 sets the global variable `errno` when an error occurs, which is defined in the standard C library.

Listing 5-2. *C++ Code to Calculate Power (Version 2)*

```
double Power(double base, int exponent) {
    errno = 0;

    if(Equal(base, 0.0) && exponent < 0) {
        errno = EDOM;
        return 0.0;
    }

    unsigned int absExponent = (unsigned int)(exponent);
    if(exponent < 0)
        absExponent = (unsigned int)(-exponent);

    double result = PowerWithUnsignedExponent(base, absExponent);
    if(exponent < 0)
        result = 1.0 / result;

    return result;
}

double PowerWithUnsignedExponent(double base, unsigned int exponent) {
    double result = 1.0;

    for(int i = 1; i <= exponent; ++i)
        result *= base;

    return result;
}

bool Equal(double num1, double num2) {
    if((num1 - num2 > -0.0000001)
        && (num1 - num2 < 0.0000001))
        return true;
    else
        return false;
}
```

A corner case is worthy of attention: it is undefined in mathematics when both base and exponent are zero, so the corresponding power result is acceptable if it is one or zero. Candidates should explain this to interviewers to show that they have considered such a boundary value.

A function `Equal` is defined to check whether the base is equal to zero. Comparing float numbers with the == operator can be problematic because of the precision issue of the floating-point representation. When the difference between two float numbers is in a narrow range, these two numbers are equal to each other.

■ **Tip** Checking the equality of decimals (including float and double numbers) with the == operator is error-prone. Two decimals are treated as equal when the difference between them is in a narrow range, such as a range between 0 and 0.0000001.

The solution is already comprehensive and it is acceptable to many interviewers. However, if an interviewer is very passionate about performance, he or she may give you some hints that the efficiency of the function `PowerWithUnsignedExponent` can be improved.

Complete and Efficient

When the parameters `baseNumber` and `exponent` of the function `PowerWithUnsignedExponent` are n and 32 respectively, it requires 31 multiplications to get n^{32}. However, it only requires five multiplications with another strategy: it calculates n^2 first, and then calculates $n^4 = n^2 \times n^2$, and similarly calculates n^8, n^{16}, and finally n^{32}.

That is to say, a^n can be calculated with the following equation efficiently:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} & n \text{ is odd} \end{cases}$$

It is easy to implement such an equation recursively, so the function `PowerWithUnsignedExponent` can be updated, as shown in Listing 5-3.

Listing 5-3. *C++ Code to Calculate Power (Version 3)*

```
double PowerWithUnsignedExponent(double base, unsigned int exponent) {
    if(exponent == 0)
        return 1;
    if(exponent == 1)
        return base;

    double result = PowerWithUnsignedExponent(base, exponent >> 1);
    result *= result;
    if((exponent & 0x1) == 1)
        result *= base;

    return result;
}
```

Some details are worthy of attention. This program replaces a division operation (/) with a right-shift (>>), and replaces a modulo operation (%) with a bitwise AND (&). Right-shift and bitwise AND operations are more efficient than arithmetic division and modulo operations.

Source Code:

`040_Power.cpp`

Test Cases:

- The exponent is positive/negative/zero
- The base is positive/negative/zero

Big Numbers as Strings

■ **Question 41** Given a number *n*, please print all numbers from 1 to the maximum *n* digit number in order. For example, if the input is 3, it prints 1, 2, 3, …, 999, which is the maximum number with three digits.

Underestimating Complexity

This problem looks simple. An intuitive solution is to get the maximum number with *n* digits, which is 10^n-1, and print numbers from 1 one by one. Many candidates can create the code in Listing 5-4 in a very short time.

Listing 5-4. *C++ Code to Print Numbers (Version 1)*

```
void Print1ToMaxOfNDigits_1(int n) {
    int number = 1;
    int i = 0;
    while(i++ < n)
        number *= 10;

    for(i = 1; i < number; ++i)
        printf("%d\t", i);
}
```

If candidates scrutinize the requirement carefully, they may find that the range of *n* is unspecified. When the input *n* is quite large, it causes overflow on the variable `number`, even if it is declared as `long long`. Therefore, it is necessary to handle big numbers.

■ **Tip** If the range of input numbers is not specified explicitly in an interview question, it might be necessary to handle big numbers. Candidates can ask for more clarification on the requirement. Utilizing strings or arrays is a simple solution to handling big numbers.

Simulating Increment on a String

There are many methods available to express big numbers. The most common method, also the simplest one, is to utilize arrays or strings to express big numbers. The following solution is based on strings.

When a number is expressed as a string, each character in the string is in the range '0' to '9'. A string with length n+1 is created to accommodate numbers with n digits at most (there is a '\0' and the end of string). When the number of digits is less than n, some leading characters are set to '0'. For example, when the input n is 3, the number 98 is expressed as the string "098".

First, all characters in the string are set as '0', and then a number in the string is increased by 1 and printed in each step of an iteration. There are two tasks to be finished: simulating addition on a string and printing a number in a string. Therefore, the skeleton function to solve this problem can be implemented as shown in Listing 5-5.

Listing 5-5. *C++ Code to Print Numbers (Version 2)*

```
void Print1ToMaxOfNDigits_2(int n) {
    if(n <= 0)
        return;

    char *number = new char[n + 1];
    memset(number, '0', n);
    number[n] = '\0';

    while(!Increment(number, n))
        PrintNumber(number);

    delete []number;
}
```

This solution increases a number in a string by one in the function `Increment` and prints the number in the function `PrintNumber`. Let's consider the implementation details of these two functions.

The return type of the function `Increment` is `bool`. It returns `true` when it reaches the maximum number with n digits to stop the `while` loop in its caller. Otherwise, it returns `false`. How do you check if it reaches the maximum number? A straightforward method is to compare the string with "999…99"(n 9s) with the library function `strcmp`. It is done if they are equal. It costs O(n) time to compare two strings with length n.

There is a more efficient solution available for the `Increment` function. It carries the first digit (with index 0) when it adds 1 to the number "999…99", but it does not carry the first digit for other numbers. Therefore, it reaches the maximal number with n digits when it carries the first digit during an increment. The sample code of `Increment` is shown in Listing 5-6.

Listing 5-6. C++ Code to Increase on a String

```
bool Increment(char* number, int length) {
    bool isOverflow = false;
    int carry = 0;

    for(int i = length - 1; i >= 0; i --) {
        int sum = number[i] - '0' + carry;
        if(i == length - 1)
            sum ++;

        if(sum >= 10) {
            if(i == 0)
                isOverflow = true;
            else {
                sum -= 10;
                carry = 1;
                number[i] = '0' + sum;
            }
        }
        else {
            number[i] = '0' + sum;
            break;
        }
    }

    return isOverflow;
}
```

It only takes O(1) time to check whether a number is increased to "999…99" in the function `Increment`, which is faster than utilizing the function `strcmp`.

Now let's consider how to print the number in a string. The library function `printf` is a handy tool to print a string, but it is not the best choice for this scenario. As mentioned earlier, the leading characters are 0s when the number of digits is less than *n*. The leading 0s in a number usually are not printed. Therefore, the function `PrintNumber` skips the leading 0s, as shown in Listing 5-7.

Listing 5-7. *C++ Code to Increase on a String*

```
void PrintNumber(char* number) {
    char* pChar = number;
    while(*pChar == '0')
        ++pChar;

    if(*pChar != '\0')
        printf("%s\t", pChar);
}
```

Simulating Permutation

The permutations of *n* digits ranging from 0 to 9 compose all numbers between 1 and 10^n-1. It is easy to implement the permutation recursively. Each digit is set as a value between 0 and 9, and then the next digit is set. It exits the recursion when all digits are set, as shown in Listing 5-8.

Listing 5-8. *C++ Code to Print Numbers (Version 3)*

```
void Print1ToMaxOfNDigits_3(int n) {
    if(n <= 0)
        return;

    char* number = new char[n + 1];
    number[n] = '\0';
    Print1ToMaxOfNDigitsCore(number, n, -1);

    delete[] number;
}

void Print1ToMaxOfNDigitsCore(char* number, int length, int index) {
    if(index == length - 1) {
        PrintNumber(number);
        return;
    }

    for(int i = 0; i < 10; ++i) {
        number[index + 1] = i + '0';
        Print1ToMaxOfNDigitsCore(number, length, index + 1);
    }
}
```

The function `PrintNumber` is the same as before, which skips the leading 0s and prints a number in a string.

More discussion about permutation is available in the section *Permutation and Combination.*

Source Code:

`041_Print1ToMaxOfNDigits.cpp`

Test Cases:

- Functional Cases (2, 3, 4, …)
- Boundary Cases (1, 0, -1)

■ **Question 42** Please design a function to add two arbitrary positive integers.

The problem requires you to add two arbitrary positive numbers, of which ranges are not specified. Therefore, you have to handle huge numbers beyond the range of the type `int` and even `long long`. Inspired by the solution of the preceding problem, strings are employed to express huge numbers.

What is the signature of the function? Your intuition might suggest that you pass the addition result in the return value, and the signature is defined as `char* add(char* num1, char* num2)`. It returns a `NULL` pointer for invalid inputs (including strings with non-digital characters and `NULL` pointers for numbers). Otherwise, it returns a string containing the addition result.

The problem with this solution is that the `add` function allocates a string containing the addition result on the heap, and it expects its caller to release the memory. However, it is highly possible that the caller will forget to release memory because the memory is allocated by others. Usually, a function that allocates memory is responsible for release.

Therefore, the signature of the function can be defined as `int add(char* num1, char* num2, char* sum)`. It returns a non-zero value such as -1 when given invalid inputs; otherwise, it returns 0. It takes a pointer to a string for the addition result as the last parameter, which is passed from the caller. Therefore, the caller is responsible for allocating sufficient memory to accommodate the result and to release it afterward.

After the signature is figured out, it is time to implement the addition functionality. The sample code is shown in Listing 5-9.

Listing 5-9. *C Code to Add Two Numbers in Strings*

```
/* It returns -1 if the input invalid, otherwise returns 0. */
int add(char* num1, char* num2, char* sum) {
    int index1, index2, indexSum;
    int sumDigit, carry, digit1, digit2;

    if(checkInvalidInput(num1, num2, sum) != 0)
        return -1;

    reverse(num1);
    reverse(num2);

    index1 = index2 = indexSum = 0;
    carry = 0;
    while(num1[index1] != '\0' || num2[index2] != '\0') {
         digit1 = (num1[index1] == '\0') ? 0 : num1[index1] - '0';
         digit2 = (num2[index2] == '\0') ? 0 : num2[index2] - '0';

         sumDigit = digit1 + digit2 + carry;
         carry = (sumDigit >= 10) ? 1 : 0;
         sumDigit = (sumDigit >= 10) ? sumDigit - 10 : sumDigit;

         sum[indexSum++] = sumDigit + '0';

         if(num1[index1] != '\0')
            ++index1;
         if(num2[index2] != '\0')
            ++index2;
    }

    if(carry != 0)
```

```
        sum[indexSum++] = carry + '0';

    sum[indexSum] = '\0';

    reverse(sum);
}
```

Usually, addition begins from the last digits (also the least significant digits). However, it reverses numbers at first in the previous code and then adds numbers from the first digits. The reason is about simplicity. If it begins to add from the last digit, the index of the last digit in the sum is unknown before addition completes. For example, the sum of two numbers with two digits 33 and 44 is 77, which also has two digits. However, the sum of two other numbers with two digits 88 and 99 is 187, which has three digits. The index of the least significant digit is different even if it adds two numbers with two digits in both cases. When numbers are reversed, it begins to add from the first digits and set the first digit in the sum, no matter how many digits the sum will have finally.

Strings with non-digital characters are treated as invalid inputs. Additionally, `NULL` pointers to numbers should also be excluded from inputs in order to avoid execution crashes. The helper function `checkInvalidInput` in Listing 5-10 is defined to validate inputs.

Listing 5-10. *C Code to Validate Inputs*

```
int checkInvalidInput(char* num1, char* num2, char* sum) {
    int length1, length2, i;

    if(num1 == NULL || num2 == NULL || sum == NULL)
        return -1;

    length1 = strlen(num1);
    for(i = 0; i < length1; ++i) {
        if(num1[i] < '0' || num1[i] > '9')
            return -1;
    }

    length2 = strlen(num2);
    for(i = 0; i < length2; ++i) {
        if(num2[i] < '0' || num2[i] > '9')
            return -1;
    }

    return 0;
}
```

The use of the following `reverse` function in Listing 5-11 reverses a given string.

Listing 5-11. *C Code to Validate Inputs*

```
void reverse(char* str) {
    int i, length;
    char temp;
```

```
    length = strlen(str);
    for(i = 0; i < length / 2; ++i) {
        temp = str[i];
        str[i] = str[length - 1 - i];
        str[length - 1 - i] = temp;
    }
}
```

Source Code:

```
042_AddNumericStrings.c
```

Test Cases:

- Add numbers with/without a carry
- One or two numbers are very large
- One or two numbers are invalid inputs (with non-numeric characters, or NULL pointers)

Delete Nodes from a List

■ **Question 43** How to delete a given node from a single-linked list in O(1) time?

The most straightforward way to delete a node from a single-linked list is to scan all nodes in it and to find the target node to be deleted. It updates the next pointer in the preceding node and deletes the target.

Suppose node *i* in the list in Figure 5-3(a) is going to be deleted. It scans the list from its head node *a*. When it reaches node *h* (the preceding node of the target node *i*), the next pointer is updated to reference to node *j* (the next node of *i*). After reconnecting the link, node *i* is deleted. Nodes in the list remain connected, as shown in Figure 5-3(b). It costs O(*n*) time on a list with *n* nodes because it has to scan the list.

The reason it has to scan the list from its head is that it needs to get the node preceding the target. There is not a pointer to the preceding node in a single-linked list, so it has to traverse the list.

Is it absolutely necessary to get the preceding node of the target to be removed? The answer is no. It is easy to get the next node of the target in a single-linked list. If the target node data is replaced with the data in the next node and then the next node is deleted, it is equivalent to deleting the target node. Only the data in the target node gets removed, and the data in the next node is stored.

Let's take the list in Figure 5-3(a) as an example again. The node *i* is the target node to be removed. The new solution copies the data in node *j*, pastes it into node *i*, and then sets the next node of node *j* as the next node of node *i*. When node *j* is deleted, it looks like node *i* is deleted, as shown in Figure 5-3(c).

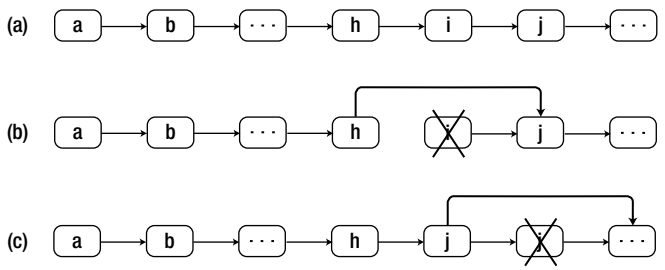

Figure 5-3. *Two methods to remove a node from a list. (a) A sample list. (b) It scans the list from its head node. When it reaches node h, the preceding node of the target node i, it updates the next pointer in h to node j, which is the next node of node i. And then node i is deleted. (c) It replaces the data in node i with the data in node j and then deletes node j.*

Notice that the tail node of the list does not have a next node. How do you delete it? We have to traverse the whole list, get the preceding node of the tail, and then delete the tail.

The last scenario is worthy of consideration when there is only one node in the list. When the only node is deleted, the pointer for the head node should be set as a `NULL` pointer, as shown in Listing 5-12.

Listing 5-12. *C++ Code to Delete a Node from a List*

```
void DeleteNode(ListNode** pListHead, ListNode* pToBeDeleted) {
    if(!pListHead || !pToBeDeleted)
        return;

    // The node to be deleted is not the tail of the list.
    if(pToBeDeleted->m_pNext != NULL) {
        ListNode* pNext = pToBeDeleted->m_pNext;
        pToBeDeleted->m_nValue = pNext->m_nValue;
        pToBeDeleted->m_pNext = pNext->m_pNext;

        delete pNext;
        pNext = NULL;
    }
    // The list has only one note. Delete the only node.
    else if(*pListHead == pToBeDeleted) {
        delete pToBeDeleted;
        pToBeDeleted = NULL;
        *pListHead = NULL;
    }
    // Delete the tail node of a list with multiple nodes
    else {
        ListNode* pNode = *pListHead;
        while(pNode->m_pNext != pToBeDeleted) {
            pNode = pNode->m_pNext;
        }

        pNode->m_pNext = NULL;
        delete pToBeDeleted;
```

```
            pToBeDeleted = NULL;
        }
}
```

Now let's analyze the time efficiency of this solution. It only costs O(1) time to copy the data from the next node and delete the next node for the n-1 nodes, which are not at the end of the list. It costs O(n) time to delete the tail node because it has to scan the whole list. Therefore, the time efficiency on average is $[(n\text{-}1)\times O(1)+O(n)]/n$, and so is O(1).

Source Code:

`043_DeleteNodeInList.cpp`

Test Cases:

- Functional Cases (Delete the head/tail node from a list or delete a node inside a list)
- Boundary Cases (Delete the only node in a list)
- Cases about Robustness (The pointer to the head node is `NULL`, or the pointer to the target node is `NULL`)

■ **Question 44** Given a sorted linked list, please delete all duplicated numbers and leave only distinct numbers from the original list. For example, when the duplicated numbers in the list in Figure 5-4(a) are removed, it becomes the list in Figure 5-4(b).

The first step to solve this problem is to figure out the signature of the required function. The head of the original list will be deleted if it is duplicated in the following nodes. Therefore, the function should be declared as `void deleteDuplication(ListNode** pHead)` rather than `void deleteDuplication(ListNode* pHead)`.

We traverse the list beginning from the head of the list. When the value in the current visited node (`pNode` in the node below) is the same as the value in the next node, the current visited node and its duplicate will be deleted. In order to make sure the list is still connected, the node prior to the current visited node (`pPreNode` in the code) is linked to the node with value greater than the current visited value. It should be guaranteed that `pPreNode` always points to the last node having a distinct value in the visited list so far.

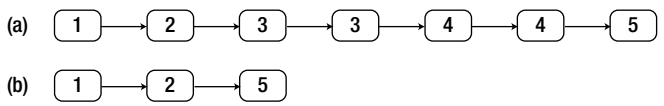

Figure 5-4. *Remove duplicate numbers in a list. (a) A list with seven elements. (b) Three elements are left when duplications are removed.*

Take the sample list as an example. When it visits the first node with value 3, `pPreNode` points to the node with value 2. Since the next node also has value 3, both nodes with value 3 should be deleted, and

then the node pointed to by `pPreNode` is linked to the first node with value 4. Since the node with value 4 also has a duplicate, the node pointed by `pPreNode` is linked to the node with value 5 after nodes with value 4 are deleted. (See Listing 5-13.)

Listing 5-13. *C++ Code to Delete Duplicated Nodes*

```
void deleteDuplication(ListNode** pHead) {
    if(pHead == NULL || *pHead == NULL)
        return;

    ListNode* pPreNode = NULL;
    ListNode* pNode = *pHead;
    while(pNode != NULL) {
        ListNode *pNext = pNode->m_pNext;
        bool needDelete = false;
        if(pNext != NULL && pNext->m_nValue == pNode->m_nValue)
            needDelete = true;

        if(!needDelete) {
            pPreNode = pNode;
            pNode = pNode->m_pNext;
        }
        else {
            int value = pNode->m_nValue;
            ListNode* pToBeDeleted = pNode;
            while(pToBeDeleted != NULL && pToBeDeleted->m_nValue == value) {
                pNext = pToBeDeleted->m_pNext;

                delete pToBeDeleted;
                pToBeDeleted = NULL;

                pToBeDeleted = pNext;
            }

            if(pPreNode == NULL)
                *pHead = pNext;
            else
                pPreNode->m_pNext = pNext;
            pNode = pNext;
        }
    }
}
```

Source Code:

`044_DeleteDuplicationInList.cpp`

Test Cases:

- Functional Cases (Groups of duplicated nodes are at the head/tail of a list, or groups of duplicated nodes are inside a list, or groups of duplicated nodes are continuous)

- Boundary Cases (All nodes in a list are duplicated, either in a duplication group or in multiple duplication groups)
- Cases Involving Robustness (The pointer to the head node is `NULL`)

Partition Numbers in Arrays

■ **Question 45** Please reorder an input array to place all odds before evens. For example, after numbers in the array {1, 2, 3, 4, 5} are reordered, it looks like {1, 5, 3, 4, 2}. The result is not unique, and there are multiple arrays meeting the reordering criteria.

The brute-force solution is to scan the array sequentially and relocate every even number. Move all numbers behind the even number backward and then place it at the end of array. Since it has to shift O(n) numbers for an even number, the overall time complexity is O(n^2).

Workable but Not Scalable

The problem asks you to place all odd numbers before even numbers. Therefore, we have to swap all pairs of numbers in which an even number is before an odd one.

Two pointers are defined to find such pairs. The first pointer P_1 is initialized at the first number, which moves forward, and the second one P_2 is initialized at the last number, which moves backward. When the number pointed to by P_1 is even and the number pointed by P_2 is odd, we have to swap these two numbers.

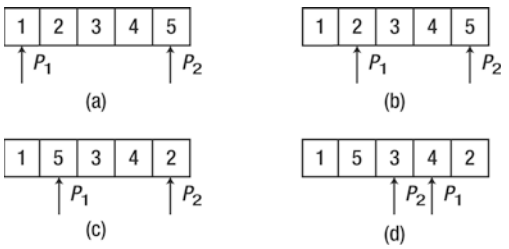

Figure 5-5. *The process to place all odd numbers before even numbers in an array {1, 2, 3, 4, 5}. (a)* P_1 *points to the first number, and* P_2 *points to the last one in the array. (b)* P_1 *moves forward until it reaches the first even number, 2. Since* P_2 *already points to an odd number, it does not move. (c) Numbers pointed to by* P_1 *and* P_2 *are swapped. (d)* P_1 *moves forward until it reaches an even number, 4, and* P_2 *moves backward until it reaches an odd number, 3. All odd numbers have been moved before even numbers because* P_2 *is before* P_1.

Let's take the sample array {1, 2, 3, 4, 5} as an example to simulate the process to reorder numbers step-by-step. During the initialization phase, P_1 points to the first number in the array, and P_2 points to the last one (Figure 5-5(a)). The pointer P_1 moves forward until it reaches an even number 2, but P_2 does

not move because it already points to an odd number 5 (Figure 5-5(b)). A pair of numbers in which an even number is before an odd one has been found. The state of the array is shown in Figure 5-5(c) after swapping.

It continues to move P_1 forward until reaching another even number, 4, and moves P_2 backward until reaching another odd number, 3 (Figure 5-5(d)). Since P_2 is now before P_1, all odd numbers have been placed before even numbers in the array {1, 5, 3, 4, 2}.

We are able to implement this solution in Java with the step-by-step analysis above, as shown in Listing 5-14.

Listing 5-14. *Java Code to Reorder Numbers in an Array (Version 1)*

```
void reorderOddEven_solution1(int nums[]) {
    int begin = 0;
    int end = nums.length - 1;
    while(begin < end) {
        // Move begin forward until it meets an even number
        while(begin < end && (nums[begin] & 0x1) != 0)
            begin++;

        // Move end backward until it meets an odd number
        while(begin < end && (nums[end] & 0x1) == 0)
            end--;

        if(begin < end) {
            int temp = nums[begin];
            nums[begin] = nums[end];
            nums[end] = temp;
        }
    }
}
```

Scalable to Handle Similar Scenarios

The solution in Listing 5-13 is acceptable for a graduate or a junior engineer. However, an interviewer may follow up with more questions if the candidate applies for a senior position as demonstrated in the following dialogue:

Interviewer: Let's change the requirement to place all negative numbers before non-negative ones. How do you do it?

Candidate: It's easy. Just define a method similar to before and modify the conditions in the second and third `while` loop.

Interviewer: Let's change the requirement again. We are going to partition an array into two parts where primes are placed before others.

Candidate: Define another new method of which the conditions …

Interviewer: (Interrupting the candidate's words): Are there any better choices?

The candidate should get the interviewer's point: what he or she expects is a scalable solution to a series of similar problems.

Let's take a close look at those two new problems. In order to solve them, it is only necessary to modify conditions in the second and third `while` loop and leave the overall code structure untouched. Therefore, we can decouple the method into two parts: one is the criterion to place an element and the other is about how to partition an array into two sub-arrays. Let's define an interface `Criterion` and a method `reorder` corresponding to these two purposes, as shown in Listing 5-15.

Listing 5-15. *Java Code to Decouple Partitioning Criterion and Process*

```
public interface Criterion{
    boolean check(int num);
}

void reorder(int nums[], Criterion criterion) {
    int begin = 0;
    int end = nums.length - 1;
    while(begin < end) {
        while(begin < end && !criterion.check(nums[begin]))
            begin++;

        while(begin < end && criterion.check(nums[end]))
            end--;

        if(begin < end) {
            int temp = nums[begin];
            nums[begin] = nums[end];
            nums[end] = temp;
        }
    }
}
```

In order to place odd numbers before even ones, a specific criterion `isEven` is defined, as shown in Listing 5-16.

Listing 5-16. *Java Code to Reorder Numbers in an Array (Version 2)*

```
void reorderOddEven_solution2(int nums[]) {
    Criterion isEven = new Criterion() {
        public boolean check(int num) {
            if((num & 0x1) == 0)
                return true;

            return false;
        }
    };

    reorder(nums, isEven);
}
```

If you must reorder numbers with other criteria, it is only necessary to implement new classes from the interface `Criterion`, but keep the method `reorder` untouched. Therefore, our decoupling strategy minimizes modification and potential risks to fulfill new requirements.

Source Code:

`045_ReorderNumbers.java`

Test Cases:

- Functional Cases (Including cases where all odd numbers are before even numbers in the original array, or all even numbers are before odd numbers, or numbers do not appear in any particular order)
- Boundary Cases (There is only an odd/even number in the array)

■ **Question 46** Given an array and a value, how do you implement a function to remove all instances of that value in place and return the new length of the array? It is not required that you keep the order of the remaining numbers. It doesn't matter what you leave beyond the new length.
For example, if the input array is {4, 3, 2, 1, 2, 3, 6}, the resulting array after removing value 2 contains the numbers {4, 3, 1, 3, 6}, and the new length of the remaining array is 5.

Move Numbers for O(*n*) Times

The most straightforward solution for this problem is to scan the whole array from the beginning to the end. When a target number is scanned, it is skipped and numbers behind it are moved backward, as shown in Listing 5-17.

Listing 5-17. *Java Code to Remove Numbers in an Array (Version 1)*

```
int remove_solution1(int numbers[], int n) {
    int i = 0;
    for (int j = 0; j < numbers.length; j++) {
        if (numbers[j] != n)
            numbers[i++] = numbers[j];
    }

    return i;
}
```

Obviously, the time complexity of this solution is O(*n*). It is also noticeable that it has to move numbers for O(*n*) times even if the target number occurs only once in the original array because there are O(*n*) numbers behind it. Is it possible to reduce the times needed to move numbers? Let's explore alternatives.

Move Numbers for O(*k*) Times

Notice that it is not required to keep the order for the remaining numbers, and the problem does not care about what numbers are left except the new length. Therefore, this problem is essentially the same as the previous one. We partition the array into two parts: the left part contains all non-target values, and the right part contains target values to be removed.

Two pointers are defined to solve this problem. The first pointer (denoted as P_1) moves forward until it reaches a number equal to the target value, which is initialized at the beginning of the array. The other pointer (denoted as P_2) moves backward until it reaches a number different from the target value, which is initialized at the end of the array. Two numbers pointed to by P_1 and P_2 are swapped. We repeat the moving and swapping operations until all target numbers are scanned.

The sample code is shown in Listing 5-18.

Listing 5-18. *Java Code to Remove Numbers in an Array (Version 2)*

```
int remove_solution2(int numbers[], int n) {
    int p1 = 0;
    while(p1 < numbers.length && numbers[p1] != n)
        ++p1;

   int p2 = numbers.length - 1;
    while(p1 < p2){
        while(p1 < numbers.length && numbers[p1] != n)
            ++p1;
        while(p2 > 0 && numbers[p2] == n)
            --p2;

        if(p1 < p2){
            numbers[p1] = numbers[p2];
            numbers[p2] = n;
        }
    }

    return p1;
}
```

Since it is only necessary to scan the whole array once and it costs O(1) time to swap a target value, the overall time complexity is O(*n*) for an array with *n* numbers. Additionally, it swaps a pair of numbers only when a target number to be removed is scanned. If there are *k* target numbers in the array, it swaps O(*k*) times. Therefore, it is more efficient in terms of data moving.

Source Code:

`046_RemoveNumbers.java`

Test Cases:

- Functional Cases (There is a single target number to be removed in the array, there are multiple target numbers in the array, or there are no target numbers in the array)

- Boundary Cases (There is only one number in the array, which is/is not the target number to be removed; all numbers in the array are to be removed; the first/last number is/is not the target number)

Robustness

Robust applications are able to check whether inputs are valid and handle invalid inputs appropriately. If applications are not robust, they might behave in unpredictable ways or crash when unexpected things happen, such as cases when users input incorrect user names, files to be opened do not exist, or servers cannot be connected. User experiences with such applications are almost surely disasters.

It is necessary for candidates to pay careful attention to code robustness. Defensive programming is an effective method of improving robustness, which predicts where it is possible to have problems, and handles problems gracefully. For example, it prompts users to check the file name when the file to be opened does not exist, or it tries a backup server when the server is inaccessible. The behavior of a robust application is predictable.

The most simple and effective defensive programming technique during interviews is to check whether inputs are valid at the entry of each function. When a function has a pointer argument, what is the expected behavior when the pointer is `NULL`? When a function takes a string as an argument, what is the expected behavior when the string is empty? If all problems have been considered in advance, it is a strong demonstration of defensive programming habits and a candidate's ability to develop robust applications.

Not all problems related to robustness are as simple as validating inputs at function entries. It is better to ask questions in the pattern of "how to handle it when … is not …". For instance, it assumes that there are more than k nodes in a list for the interview problem "k^{th} Node from End" (Question 47). Candidates should ask themselves what the expected behavior is when the number of nodes is less than k. This kind of question helps to find potential problems and handle them in advance.

k^{th} Node from End

■ **Question 47** Please implement a function to get the k^{th} node from tail of a single linked list. For example, if a list has six nodes whose values are 1, 2, 3, 4, 5, 6 in order from head, the third node from tail contains the value 4.

The program can only move forward in a single-linked list from the head node, but it cannot move backward from the tail. Supposing a list has n nodes. The k^{th} node from the tail is the $(n\text{-}k\text{+}1)^{th}$ node from the head. It reaches the target node if it moves $n\text{-}k\text{+}1$ steps along links to the next nodes. How do you find the total number of nodes n in a list? Just traverse the whole list and count.

As a result, it gets the k^{th} node from the tail after it traverses the list twice: it traverses the whole list to get the number of nodes n in the list, and then it moves along the links between nodes for $n\text{-}k\text{+}1$ steps.

This solution works. However, usually interviewers prefer a solution that traverses the list only once.

In order to solve this problem with only one traversal, two pointers are employed. The first pointer (P_1) moves k-1 steps from the head of the list. The second pointer (P_2) begins to move from the head and

P_1 continues to move. Since the distance between these two pointers keeps at k-1, the second pointer reaches the k^{th} node from the tail when the first one reaches the tail of the list.

Figure 5-6 simulates the process of finding the third node from the end of a list with six nodes. The pointer P_1 moves 2 steps (2=3-1) in the list (Figure 5-6(a)). The pointer P_2 is initialized at the head of the list (Figure 5-6(b)). Then these two pointers move at the same speed. When P_1 reaches the tail of the list, P_2 arrives at the third node from the end (Figure 5-6(c)).

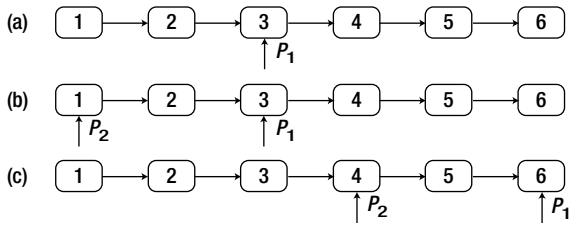

Figure 5-6. *The process to get the third node from the tail of a list with six nodes. (a) Pointer* P_1 *moves two steps on the list. (b) The pointer* P_2 *is initialized at the head of the list. (c) Both* P_1 *and* P_2 *move ahead until* P_1 *reaches the tail of the list. The node pointed to by* P_2 *is the third node from the tail.*

Some candidates implement a solution with two pointers quickly, as shown in Listing 5-19.

Listing 5-19. *C++ Code to Get the* k^{th} *Node from End (Version 1)*

```
ListNode* FindKthToTail(ListNode* pListHead, unsigned int k) {
    ListNode *pAhead = pListHead;
    ListNode *pBehind = NULL;

    for(unsigned int i = 0; i < k - 1; ++ i) {
        pAhead = pAhead->m_pNext;
    }

    pBehind = pListHead;
    while(pAhead->m_pNext != NULL){
        pAhead = pAhead->m_pNext;
        pBehind = pBehind->m_pNext;
    }

    return pBehind;
}
```

Many candidates read web pages about the solution with two pointers during interview preparation. When they meet such a problem in interviews, they feel lucky and write code similar to the code in Listing 5-19 quickly. Unfortunately, they may receive a rejection letter rather than an offer. Is it unfair? It is a fair result because the code has three potential risks for crash:

- The head of the list `pListHead` is a `NULL` pointer. An application crashes when it tries to access the `NULL` address in memory.

- The number of nodes in the list with head `pListHead` is less than *k*. It crashes because it tries to move *k*-1 steps in the `for` loop.
- The input parameter *k* is 0. Because *k* is an unsigned integer, the value of *k*-1 is 4294967295 (unsigned 0xFFFFFFFF) rather than -1. It iterates in the `for` loop many more times than expected.

There are too many risks of a crash in such a simple piece of code, so it is reasonable for the writer of this code to be rejected.

■ **Tip** Robustness is worthy of attention in interviews. Rejection is quite likely if there are many potential risks of crash in code.

Let's fix the listed problems one by one:

- If the pointer of the head node is `NULL`, the whole list is empty, so naturally the k^{th} node from the tail is also `NULL`.
- An `if` statement can be inserted to check whether it reaches a `NULL` address when it tries to move *k* steps on the list. When the total number of nodes in the list is less than *k*, it returns `NULL`.
- It counts from 1 here, and the first node from the tail is the tail node itself. It is an invalid input if *k* is 0 because it is meaningless to find the zeroth node from the tail. It returns `NULL` too in such a case.

The revised version of the code is shown in Listing 5-20.

Listing 5-20. *C++ Code to Get the* kth *Node from End (Version 2)*

```
ListNode* FindKthToTail(ListNode* pListHead, unsigned int k) {
    if(pListHead == NULL || k == 0)
        return NULL;

    ListNode *pAhead = pListHead;
    ListNode *pBehind = NULL;

    for(unsigned int i = 0; i < k - 1; ++ i) {
        if(pAhead->m_pNext != NULL)
            pAhead = pAhead->m_pNext;
        else {
            return NULL;
        }
    }

    pBehind = pListHead;
    while(pAhead->m_pNext != NULL) {
        pAhead = pAhead->m_pNext;
        pBehind = pBehind->m_pNext;
```

```
    }

    return pBehind;
}
```

Source Code:

`047_KthNodeFromEnd.cpp`

Test Cases:

- Functional Cases (The k^{th} node is the head/tail of a list or inside a list)
- Cases for Robustness (The pointer to the list head is `NULL`; the number of a list is less than k; k is 0)

Reverse a List

■ **Question 48** Please implement a function to reverse a list.

Lots of pointer operations are involved in solving problems related to linked lists. Interviewers know that many candidates are prone to making mistakes on pointer operations, so they like problems related to linked lists to qualify candidates' programming capabilities. Candidates should analyze and design carefully before they begin to write code. It is much better to write robust code with comprehensive analysis than to write code quickly with many bugs.

The direction of pointers should be reversed in order to reverse a linked list. Let's utilize figures to analyze visually the complex steps to reverse pointers. As shown in the list in Figure 5-7(a), node *h*, *i*, and *j* are three adjacent nodes. Supposing pointers to all nodes prior to *h* have been reversed after some operations and all next pointers are linked to their preceding nodes. The next step is to reverse the next pointer in node *i*. The status of the list is shown in Figure 5-7(b).

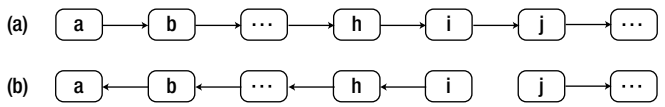

Figure 5-7. *A list gets broken when it is reversed. (a) A list. (b) All next pointers in nodes prior to the node* i *are reversed to reference their preceding nodes. The link between the nodes* i *and* j *gets disconnected.*

Notice that the next pointer in node *i* references its preceding node *h*, so the list is broken and node *j* is inaccessible (Figure 5-7(b)). Node *j* should be saved before the next pointer of node *i* is adjusted in order to prevent the list from becoming broken.

When the next pointer in node *i* is reversed, we need access to node *h* since the pointer is adjusted to reference node *h*. It also needs to access node *j*; otherwise, the list will be broken. Therefore, three pointers should be declared in our code that point to the current visited node, its preceding node, and its next node respectively.

Lastly, the head node of the reversed list should be returned. Obviously, the head in the reversed list should be the tail of the original list. Which pointer is the tail? It should be the node whose next pointer is `NULL`.

With comprehensive analysis above, we are ready to write code, which is shown in Listing 5-21.

Listing 5-21. *C++ Code to Reverse a List*

```
ListNode* ReverseList(ListNode* pHead) {
    ListNode* pReversedHead = NULL;
    ListNode* pNode = pHead;
    ListNode* pPrev = NULL;
    while(pNode != NULL) {
        ListNode* pNext = pNode->m_pNext;

        if(pNext == NULL)
            pReversedHead = pNode;

        pNode->m_pNext = pPrev;

        pPrev = pNode;
        pNode = pNext;
    }

    return pReversedHead;
}
```

The common issues remaining in many candidates' code are in three categories:

- The program crashes when the pointer of the head node is `NULL` or there is only one node in the list.
- The reversed list is broken.
- The returned node is the head node of the original list rather than the head node of the reversed list.

How do you make sure there are no problems remaining in code during interviews? A good practice is to have comprehensive test cases. After finishing writing code, candidates can test their own code with the prepared cases. The code can be handed to interviewers only after all unit tests are passed. Actually, interviewers have their own test cases to verify candidates' code. If a candidate's test covers all test cases prepared by his or her interviewer, it is highly possible to pass this round of interviews.

Source Code:

`048_ReverseList.cpp`

Test Cases:

- Functional Cases (The number of nodes in a list is even/odd; there is only one node in a list)
- Cases for Robustness (The pointer to the list head is `NULL`)

■ **Question 49** How do you design an algorithm to reverse every *k* nodes in a list? A list is divided into several groups, and each group has *k* nodes except the last group, where the number of nodes may be less than *k*. Please reverse the nodes in each group and connect all groups together.

For example, when groups with three nodes are reversed in the list of Figure 5-8(a), it becomes the list in Figure 5-8(b).

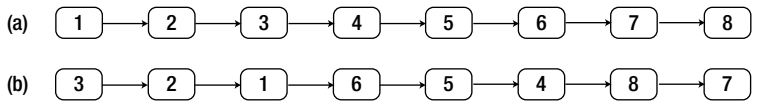

Figure 5-8. *Reverse every group with three nodes in a list with eight nodes.*

Three steps are necessary to reverse a list group by group. First, the process finds a group with *k* nodes (the number of nodes in the last group may be less than *k*). Then it reverses pointers to nodes inside the group. Finally, it connects reversed groups together. Therefore, the overall code structure looks like the function `Reverse`, as shown in Listing 5-22.

Listing 5-22. *C++ Code to Reverse Nodes in a List in Groups*

```
ListNode* Reverse(ListNode* pHead, unsigned int k) {
    if(pHead == NULL || k <= 1)
        return pHead;

    ListNode* pReversedHead = NULL;
    ListNode* pNode1 = pHead;
    ListNode* pPrev = NULL;
    while(pNode1 != NULL) {
        // find k nodes within a group
        ListNode* pNode2 = pNode1;
        ListNode* pNext = NULL;
        for(unsigned int i = 1; pNode2->m_pNext != NULL && i < k; ++i)
            pNode2 = pNode2->m_pNext;

        pNext = pNode2->m_pNext;

        // reverse nodes within a group
        ReverseGroup(pNode1, pNode2);

        // connect groups together
        if(pReversedHead == NULL)
            pReversedHead = pNode2;

        if(pPrev != NULL)
            pPrev->m_pNext = pNode2;
        pPrev = pNode1;
```

```
        pNode1 = pNext;
    }

    return pReversedHead;
}
```

The process required to reverse nodes inside a group is similar to the process discussed for the preceding problem, so it will not be analyzed step-by-step. The function `ReverseGroup` is used to reverse a group between two nodes, as shown in Listing 5-23.

Listing 5-23. *C++ Code to Reverse Nodes in a Group*

```
void ReverseGroup(ListNode* pNode1, ListNode* pNode2) {
    ListNode* pNode = pNode1;
    ListNode* pPrev = NULL;
    while(pNode != pNode2) {
        ListNode* pNext = pNode->m_pNext;
        pNode->m_pNext = pPrev;

        pPrev = pNode;
        pNode = pNext;
    }

    pNode->m_pNext = pPrev;
}
```

Source Code:

`049_ReverseListInGroups.cpp`

Test Cases:

- Functional Cases (The number of nodes in the last group of a list is *k* or less than *k*; the number of nodes in a list is less than or equal to *k*)
- Cases for Robustness (The pointer to the list head is `NULL`; *k* is 0 or negative)

Substructures in Trees

■ **Question 50** Given two binary trees, please check whether one is a substructure of the other. For example, the tree *b* on the right side of Figure 5-9 is a substructure of the tree *a* on the left side.

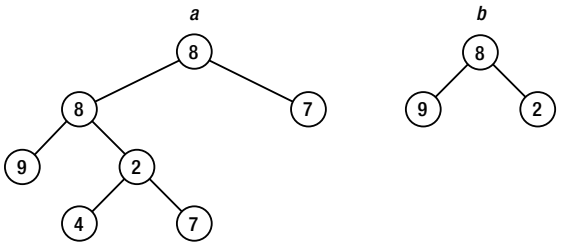

Figure 5-9. *Two binary trees of which tree* b *on the right side is a substructure of tree* a *on the left side.*

Similar to lists, pointer operations on trees are also quite complicated, so it is usually not easy to solve coding interview problems about trees. Candidates have to be very careful on pointer operations; otherwise, it is highly possible to leave crash risks.

Let's return to the problem itself. It can be solved in two steps. The first step is to find a node *r* in tree *a* whose value is the same as the value in the root node of tree *b*, and the second step is to check whether the subtree rooted at the node *r* has the same structure as tree *b*.

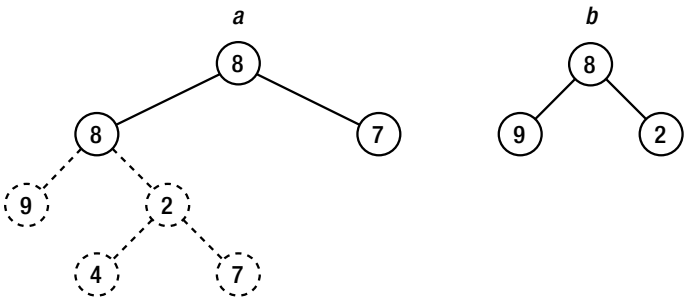

Figure 5-10. *Two root nodes in trees* a *and* b *have the same value, but their children are different.*

Take the two sample trees as an example. Our solution traverses tree *a* and finds a node with a value of 8 (the value in the root node of tree *b*). Since the value in the root of tree *a* is 8, the solution checks whether the subtree under the root node in tree *a* has the same structure as tree *b*. As shown in Figure 5-10, their structures are different.

It continues to traverse on tree *a*, and another node with value 8 is found in the second level. It checks the structure of the subtree again, and the left and right children have the same values as the children nodes in tree *b* (Figure 5-11). Therefore, a subtree in tree *a* with the same structure as tree *b* has been found, so tree *b* is a substructure of tree *a*.

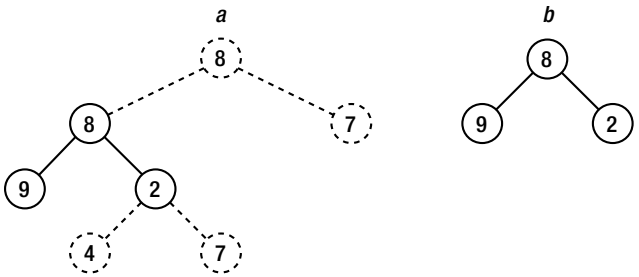

Figure 5-11. *The structure under the second node with value 8 in tree* a *is the same as the structure of tree* b.

The first step to find a node in tree *a* with the same value in the root node of tree *b* is actually a traversal process. There are different traversal algorithms available, and the following code is based on the pre-order traversal algorithm. The following function `HasSubtree` solves this problem recursively in Listing 5-24 because it is simpler to implement traversal algorithms with recursion.

Listing 5-24. *C++ Code Check Whether a Tree Is a Subtree of Another*

```
bool HasSubtree(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2) {
    bool result = false;

    if(pRoot1 != NULL && pRoot2 != NULL) {
        if(pRoot1->m_nValue == pRoot2->m_nValue)
            result = DoesTree1HaveTree2(pRoot1, pRoot2);
        if(!result)
            result = HasSubtree(pRoot1->m_pLeft, pRoot2);
        if(!result)
            result = HasSubtree(pRoot1->m_pRight, pRoot2);
    }

    return result;
}
```

In the function `HasSubtree`, it moves on to check whether a subtree in tree *a* has the same structure as tree *b* with the function `DoesTree1HaveTree2`, as shown in Listing 5-25.

Listing 5-25. *C++ Code Check Whether a Tree Is a Subtree of Another*

```
bool DoesTree1HaveTree2(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2) {
    if(pRoot2 == NULL)
        return true;

    if(pRoot1 == NULL)
        return false;

    if(pRoot1->m_nValue != pRoot2->m_nValue)
        return false;
```

```
    return DoesTree1HaveTree2(pRoot1->m_pLeft, pRoot2->m_pLeft) &&
        DoesTree1HaveTree2(pRoot1->m_pRight, pRoot2->m_pRight);
}
```

Notice that there are many places to check whether a pointer is `NULL`. Candidates have to be careful when writing code to traverse trees. They should ask themselves whether it is possible for a pointer to be `NULL` in every statement with pointers and how to handle it when the pointer is `NULL`. Programs are prone to crash when some `NULL` pointers are not handled appropriately.

■ **Tip** It is important to ask two questions when writing code with pointer operations: Is it possible for the pointer to be `NULL`? How do you handle it when the pointer is `NULL`?

Source Code:

`050_SubtreeInTree.cpp`

Test Cases:

- Functional Cases (A binary tree is/is not a subtree of another)
- Cases for Robustness (The pointers to the head of one or two binary trees are `NULL`; some special binary trees where nodes do not have left/right subtrees)

Summary

This chapter has discussed how to improve code quality in interviews with clearness, completeness, and robustness.

Candidates write code on paper or whiteboards in most interviews. It improves code readability if code is written neatly with a clear layout and indentation, and variables and functions are named reasonably.

It is a good practice to figure out all possible inputs before writing code. Code is complete only when it fulfills functional requirements as well as when it handles boundary values and errors appropriately.

Defensive programming is a good habit to improve code robustness. Candidates should check input validity at the entry of each function and handle invalid inputs gracefully.