21. Merge Two Sorted Lists

C++ solution:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {

        if (!l1)
        {
            return l2;
        }

        if (!l2)
        {
            return l1;
        }

        ListNode* head = nullptr;

        if (l2->val < l1->val)
        {
            head = l2;
            l2 = l2->next;
        }
        else
        {
            head = l1;
            l1 = l1->next;
        }

        ListNode* current = head;
        while (l1 && l2)
        {
            if (l2->val < l1->val)
            {
                current->next = l2;
                current = l2;
                l2 = l2->next;
            }
            else
            {
                current->next = l1;
                current = l1;
                l1 = l1->next;
            }
        }

        if (!l1)
        {
            current->next = l2;
        }
        else if (!l2)
        {
            current->next = l1;
        }

        return head;
    }
};
```

[22. Generate Parentheses](#)

C++ solution:

```
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> results;

        if (n == 0)
        {
            return results;
        }

        if (n == 1)
        {
            results.insert(results.end(), "()");
            return results;
        }

        set<string> unique_results;
        for (string s : generateParenthesis(n - 1))
        {
            unique_results.insert("()" + s);
            unique_results.insert(s + "()");
            int n = s.length();

            for (int i = 0; i < n - 1; ++i)
            {
                unique_results.insert(s.substr(0, i + 1) + "()" + s.substr(i + 1));
            }
        }

        copy(unique_results.begin(), unique_results.end(), back_inserter(results));

        return results;
    }
};
```

23. Merge k Sorted Lists

C++ solution:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
```

```
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {

        if (!l1)
        {
            return l2;
        }

        if (!l2)
        {
            return l1;
        }

        ListNode* head = nullptr;

        if (l2->val < l1->val)
        {
            head = l2;
            l2 = l2->next;
        }
        else
        {
            head = l1;
            l1 = l1->next;
        }

        ListNode* current = head;
        while (l1 && l2)
        {
            if (l2->val < l1->val)
            {
                current->next = l2;
                current = l2;
                l2 = l2->next;
            }
            else
            {
                current->next = l1;
                current = l1;
                l1 = l1->next;
            }
        }

        if (!l1)
        {
            current->next = l2;
        }
        else if (!l2)
        {
            current->next = l1;
        }

        return head;
    }

    ListNode* mergeKLists(vector<ListNode*>& lists) {
        ListNode* result = nullptr;
        for (ListNode* l : lists)
        {
            result = mergeTwoLists(result, l);
        }
        return result;
    }
};
```

24. Swap Nodes in Pairs

C++ solution:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {

        if ((!head) || (!head->next))
        {
            return head;
        }

        ListNode* result = head->next;

        ListNode* before = new ListNode(0);
        before->next = head;
        ListNode* current = head;
        ListNode* after = nullptr;

        while (current && current->next)
        {
            after = current->next->next;
            before->next = current->next;
            current->next->next = current;
            current->next = after;

            before = current;
            current = current->next;
        }

        return result;
    }
};
```

25. Reverse Nodes in k-Group

C++ solution:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {

        if (k < 2)
        {
            return head;
        }

        int count = 0;
        ListNode* before = new ListNode(0);
        before->next = head;
        ListNode* current = head;
        ListNode* after = new ListNode(0);
        ListNode* counter = head;
        ListNode* temp = new ListNode(0);
        ListNode* result = head;

        while (true)
        {
            while (counter)
            {
                ++count;
                if (count % k == 0)
                {
                    break;
                }
                counter = counter->next;
            }

            if (!counter)
            {
                if (count < k)
                {
                    return head;
                }
                break;
            }

            if (count == k)
            {
                result = counter;
            }

            for (int i = 1; i < k; ++i)
            {
                temp = before->next;
                after = current->next->next;
                before->next = current->next;
                before->next->next = temp;
                current->next = after;
            }

            counter = after;
            before = current;
            current = counter;
        }

        return result;
    }
};
```

26. Remove Duplicates from Sorted Array

C++ solution:

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {

        int count = 0, length = nums.size();

        for (int i = 0; i < length; ++i)
        {
            if (i == 0 || nums[i] != nums[i - 1])
            {
                nums[count] = nums[i];
                ++count;
            }
        }

        return count;
    }
};
```

27. Remove Element

C++ solution:

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {

        int count = 0, length = nums.size();

        for (int i = 0; i < length; ++i)
        {
            if (nums[i] != val)
            {
                nums[count] = nums[i];
                ++count;
            }
        }

        return count;
    }
};
```

28. Implement strStr()

C++ solution:

```
class Solution {
public:
    int strStr(string haystack, string needle) {

        if (needle.empty())
        {
            return 0;
        }

        int nh = haystack.length();
        int nn = needle.length();

        if (nh < nn)
        {
            return -1;
        }

        for (int i = 0; i <= nh - nn; ++i)
        {
            int j = 0;
            for (; j < nn; ++j)
            {
                if (haystack[i + j] != needle[j])
                {
                    break;
                }
            }

            if (j == nn)
            {
                return i;
            }
        }

        return -1;
    }
};
```

29. Divide Two Integers

C++ solution:

```
class Solution {
public:
    int divide(int dividend, int divisor) {

        if (dividend == INT_MIN && divisor == -1)
        {
            return INT_MAX;
        }

        long _dividend = static_cast<long>(dividend);
        long _divisor = static_cast<long>(divisor);

        int sign1 = (dividend >> 31) & 1;
        int sign2 = (divisor >> 31) & 1;
        int sign = sign1 ^ sign2;

        if (sign1 == 1)
        {
            _dividend = ~_dividend + 1;
        }

        if (sign2 == 1)
        {
            _divisor = ~_divisor + 1;
        }

        long result = 0, temp = 0, k = 0;

        while (_dividend >= _divisor)
        {
            temp = _divisor;
            k = 1;
            while ((temp << 1) <= _dividend)
            {
                k <<= 1;
                temp <<= 1;
            }

            _dividend -= temp;
            result += k;
        }

        if (sign == 0)
        {
            return static_cast<int>(result);
        }
        else
        {
            return static_cast<int>(-result);
        }
    }
};
```

30. Substring with Concatenation of All Words

C++ solution:

```
class Solution {
public:
    vector<int> findSubstring(string s, vector<string>& words) {
        int length_s = s.length();
        int number_words = words.size();
        vector<int> result;
        if (number_words == 0)
        {
            return result;
        }

        unordered_map<string, int> string_count;
        for(string word : words)
        {
            if (string_count.find(word) == string_count.end())
            {
                string_count.insert(pair<string, int>(word, 1));
            }
            else
            {
                ++string_count[word];
            }
        }

        int number_unique_word = string_count.size();
        int length_word = words[0].length();
        int end = length_s - length_word * number_words;

        if (end < 0)
        {
            return result;
        }

        bool visited[length_s];
        for (int i = 0; i < end + 1; ++i)
        {
            visited[i] = false;
        }

        for (int i = 0; i <= end; ++i)
        {
            if (visited[i])
            {
                continue;
            }
            unordered_map<string, int> found_count;
            int current = i;
            int last = i + length_word * number_words;
            while (current < last)
            {
                string _word = s.substr(current, length_word);
                unordered_map<string, int>::iterator iter = string_count.find(_word);

                if (iter == string_count.end())
                {
                    int j = current;
                    while (j > i)
                    {
                        visited[j] = true;
                        j -= length_word;
                    }
                    break;
                }

                if (found_count.find(iter->first) == found_count.end())
                {
                    found_count.insert(pair<string, int>(iter->first, 1));
                }
                else
                {
                    ++found_count[_word];
                }

                if (found_count[_word] > string_count[_word])
                {
                    break;
                }

                current += length_word;
            }

            if (current == last)
            {
                result.insert(result.end(), i);
                int j = last, begin = i;
                while (j <= length_s - length_word)
                {
                    visited[begin + length_word] = true;
                    unordered_map<string, int>::iterator it = found_count.find(s.subst
r(j, length_word));
                    if (it== found_count.end())
                    {
                        int k = j;
                        while (k > begin + length_word)
                        {
                            visited[k] = true;
                            k -= length_word;
                        }
                        break;
                    }
                    int count = found_count[it->first];
                    ++(it->second);
                    --(found_count[s.substr(begin, length_word)]);
                    if (it->second == count)
                    {
                        result.insert(result.end(), begin + length_word);
                        begin += length_word;
                        j += length_word;
                    }
                    else
                    {
                        break;
                    }
                }
            }
        }

        return result;
    }
};
```