Table of Contents

Introduction

编程语言

定义一个空类型，里面没有任何成员变量和成员函数，对该类型求sizeof，得到的结果是1

空类型的实例不包含任何信息，本来求sizeof应该是0，但是声明实例的时候，它必须在内存中占有一定的空间，否则无法使用这些实例．至于占多少内存，由编译器决定；

如果该类型中添加一个构造函数和析构函数，结果仍然是1，调用构造函数和析构函数只需要知道函数的地址即可，而这些函数的地址只与类型相关，而与类型的实例无关，编译器也不会因为两个函数在实例内添加任何额外的信息．

如果把析构函数标记为虚函数，C++编译器一旦发现一个类型中有虚函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向虚函数表的指针，32为操作系统中，一个指针占4字节，64位操作系统中，一个指针占8字节。

代码：

1. sizeof与空类型

```
#include<iostream>

using namespace std;

class A {};
class B {};

class C {
    public:
        void func1();
        virtual void func2();

    private:
        static int n;
        int m;
};

class D : public A {
    virtual void fun() = 0;
};

class E : public B, public D{};

int main() {
    cout << "sizeof(A):" << sizeof(A) << endl;
    cout << "sizeof(B):" << sizeof(B) << endl;
    cout << "sizeof(C):" << sizeof(C) << endl;
    cout << "sizeof(D):" << sizeof(D) << endl;
    cout << "sizeof(E):" << sizeof(E) << endl;
    return 0;
}

// 输出
// sizeof(A):1
// sizeof(B):1
// sizeof(C):16
// sizeof(D):8
// sizeof(E):8
```

数组中的重复数字

1. 找出数组中重复的数字

> 在一个长度为n的数组里的所有数字都在0~n-1的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道这个数字重复了几次。请找出数组中任意一个重复的数字。
>
> 例如，如果输入长度为7的数组{2, 3, 1, 0, 2, 5, 3}，那么对应的重复数字是2或者3.

简单解法，先排序，然后找重复数字

利用哈希表来解决问题。从头到尾按顺序扫描数组的每一个数字，每扫描到一个数字的时候，都可以用O(1)的时间来判断哈希表里是否已经包含了该数字。如果哈希表里还没有这个数字，就把它加入到哈希表。如果哈希表里存在该数字，就找到了一个重复数字。这个算法的时间复杂度为O(n)，但是它提高时间效率是以一个大小为O(n)的哈希表为代价的。

数组中的数字是0~n-1范围，如果数组中没有重复数字，那么当数组排序之后的数字i将出现在下标为i的位置；由于数组中有重复的数字，有些位置可能存在多个数字，同事有些位置可能没有数字。

从头到尾依次扫描这个数组中的每一个数字，当扫描到下标为i的数字时，首先比较这个数组(m)是不是等于i。如果是则接着扫描下一个数字；如果不是，则再拿它和第m个数字进行比较。如果它和第m个数字相等，就找到了一个重复的数字；如果他和第m个数字不相等，就把第i个数字和第m个数字交换，把m放到属于它的位置。接下来再重复这个比较、交换的过程，直到发现一个重复的数字。

```
#include <iostream>

using namespace std;

bool duplicate(int numbers[], int length, int*
duplication)
{

    // 判断长度和指针是否为不正常
    if (numbers == nullptr || length <= 0) {
        return false;
    }

    for (int i = 0; i < length; i++) {
        if (numbers[i] < 0 || numbers[i] > length
- 1) {
            return false;
        }
    }

    for (int i = 0; i < length; i++) {
        while (numbers[i] != i) {
            if (numbers[i] == numbers[numbers[i]])
{
                *duplication = numbers[i];
                return true;
            }
            // swap
            int temp = numbers[i];
            numbers[i] = numbers[temp];
            numbers[temp] = temp;
        }
    }
    return false;
}

int main(int argc, char* argv[])
{
    // test
    int numbers1[7] = { 2, 3, 1, 0, 2, 5, 3 };
    int duplication1 = -1;
    bool result1 = duplicate(numbers1, 7,
&duplication1);
```

```
    cout << "result: " << result1 << "
duplication: " << duplication1 << endl;

    return 0;
}
```

2. 不修改数组找出重复数字

> 在一个长度为n+1的数组里的所有数字都在1~n的范围内，所以数组中至少有一个数字是重复的，请找出数组中任意一个重复的数字，但不能修改输入的数组。
>
> 例如，如果输入的长度为8的数组{2,3,5,4,3,2,6,7}，那么对应的重复数字为2或者3

简单：创建一个长度为n+1的辅助数组，然后逐一把元数组的每一个复制到辅助数组。如果原来数组中被复制的数字是m，则把她复制到辅助数组中下标为m的卫视，这样容易发现哪个数字是重复的。由于需要创建一个数组，该方案需要O(n)的辅助空间。

假如没有重复的数字，那么1~n的范围内只有n个数字。由于数组里包含超过n个数字，所以一定包含了重复的数字。

把1~n的数字从中间的数字m分为两个部分，前一半为1~m，后一半为m+1~n。然后在数组里统计区间数字出现的次数，如果1~m的数字的数目超过m，那么这一半的区间里一定包含了重复数字；否则，另一半m+1~n里一定包含重复数字。继续把包含重复数字的区间一分为2，直到找到这个数字为止。

```
#include<iostream>

using namespace std;

int coutRange(const int* numbers, int length, int
start, int end) {
    if(numbers == nullptr) {
        return 0;
    }
    int count = 0;
    for(int i = 0; i < length; i ++) {
        if(numbers[i] >= start && numbers[i] <=
end) {
            count ++;
        }
    }
    return count;
}

int getDeplication(const int* numbers, int length)
{
    if(numbers == nullptr || length <= 0) {
        return -1;
    }

    int start = 1;
    int end = length - 1;
    while(end >= start) {
        int middle = ((end - start) >> 1) + start;
        int count = coutRange(numbers, length,
start, middle);
        if(end == start) {
            if(count > 1) {
                return start;
            } else {
                break;
            }
        }
        if(count > (middle - start + 1)) {
            end = middle;
        } else {
            start = middle + 1;
        }
```

```
    }
    return -1;
}

int main(int argc, char *argv[])
{
    int a[8] = {2,3,5,4,3,2,6,7};
    int result = getDeplication(a, 8);
    cout << result << endl;
    return 0;
}
```

二维数组中的查找

> 题目：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。
>
> 例如下面的二维数组就是每行、每列都递增排序。如果在这个数组中查找数字 7，则返回 true；如果查找数字 5，由于数组不含有该数字，则返回 false。
>
> 1 2 8 9
> 2 4 9 12
> 4 7 10 13
> 6 8 11 15

首先选取数组右上角的数字。如果该数字等于要查找的数字，则查找过程结束；如果该数字大于要查找的数字，则剔除这个数字所在的列；如果该数字小于要查找的数字，则剔除这个数字所在的行。这样每一步都可以缩小范围，直到找到要查找的数字，或者查找的范围为空。

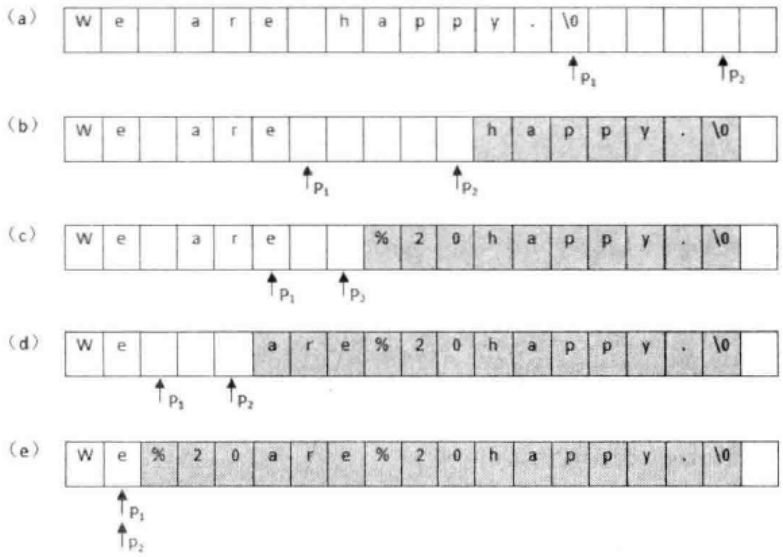
```
#include<iostream>

using namespace std;

// 传入二维数组
bool find1(int matrix[][4], int rows, int columns,
int number) {
    bool result = false;

    if(matrix != nullptr) {
        int row = 0;
        int column = columns - 1;
        while(row < rows && column >= 0) {
            if(matrix[row][column] == number) {
                result = true;
                break;
            } else {
                if(matrix[row][column] > number) {
                    column --;
                } else {
                    row ++;
                }
            }
        }
    }

    return result;
}

// 传入一维数组
bool find2(int* matrix, int rows , int columns,
int number) {
    bool result = false;

    if(matrix != nullptr) {
        int row = 0;
        int column = columns - 1;
        while(row < rows && column >= 0) {
            if(matrix[row * column + column] ==
number) {
                result = true;
                break;
            } else {
```

```
                if(matrix[row * column + column] >
number) {
                    column --;
                } else {
                    row ++;
                }
            }
        }
    }
    return result;
}

int main(int argc, char *argv[])
{
    int a[4][4] = {{1, 2, 8, 9},{2, 4, 9, 12}, {4,
7, 10, 13}, {6, 8, 11, 15}};
    // test1
    bool result1 = find1(a, 4, 4, 7);
    cout << "test1 find 7 : " << result1 << endl;
    bool result2 = find1(a, 4, 4, 5);
    cout << "test1 find 5 : " << result2 << endl;
    // test2
    bool result3 = find1(a, 4, 4, 7);
    cout << "test2 find 7 : " << result3 << endl;
    bool result4 = find1(a, 4, 4, 5);
    cout << "test2 find 5 : " << result4 << endl;



    return 0;
}
```

替换空格

> 题目：请实现一个函数，把字符串中的每个空格替换成"%20"。例如，输入“We are happy.”，则输出“We%20are%20happy.”。

两个指针，一个指向原始字符串末尾，另一个指向替换后的字符传末尾，然后从后往前逐一替换。



```
#include<iostream>

using namespace std;

void replaceSpaces(char* str, int length) {
    if(str == nullptr || length <= 0) {
        return;
    }

    // 真实长度
    int realLength = 0;
    // 空格长度
    int blankLength = 0;

    // 统计字符
    int i = 0;
    while(str[i] != '\0') {
        realLength ++;
        if(str[i] == ' ') {
            blankLength ++;
        }

        i ++;
    }

    int newLength = realLength + (2 +
blankLength);
    if(newLength > length) {
        return;
    }

    int indexOfRealLength = realLength;
    int indexOfNewLength = newLength;

    while(indexOfRealLength >= 0 &&
indexOfNewLength > indexOfRealLength) {
        if(str[indexOfRealLength] == ' ') {
            str[indexOfNewLength --] = '0';
            str[indexOfNewLength --] = '2';
            str[indexOfNewLength --] = '%';
        } else {
            str[indexOfNewLength --] =
str[indexOfRealLength];
```

```
            }
            indexOfRealLength --;
        }
    }

    int main(int argc, char *argv[])
    {
        const int length = 100;
        char a[length] = "We are happy.";
        char b[length] = "    ";
        replaceSpaces(a, length);
        replaceSpaces(b, length);
        cout << a << endl;
        cout << b << endl;
        return 0;
    }
```

从头到尾打印链表

题目：输入一个链表的头节点，从尾到头反过来打印出每个节点的值。链表节点定义如下：

```
struct ListNode
{
    int         m_nKey;
    ListNode* m_pNext;
};
```

1. 栈实现
2. 递归实现

```
#include<iostream>
#include<stack>

using namespace std;

struct ListNode {
    int m_nKey;
    ListNode* m_pNext;
};

ListNode* createListNode(int value) {
    ListNode* pNode = new ListNode();
    pNode->m_nKey = value;
    pNode->m_pNext = nullptr;
    return pNode;
}

ListNode* connectListNode(ListNode* node1,
ListNode* node2) {
    if(node1 == nullptr) {
        cout << "connectListNode error" << endl;
        exit(1);
    }
    node1->m_pNext = node2;
    return node1;
}

void destroyList(ListNode* pHead) {
    ListNode* pNode = pHead;
    if(pNode != nullptr) {
        pHead = pHead->m_pNext;
        delete pNode;
        pNode = pHead;
    }
}

void printListReversingly_Iteratively(ListNode*
pHead) {
    stack<ListNode*> nodes;
    ListNode* pNode = pHead;
    while(pNode != nullptr) {
        nodes.push(pNode);
        pNode = pNode->m_pNext;
```

```
    }

    while(!nodes.empty()) {
        pNode = nodes.top();
        cout << pNode->m_nKey << endl;
        nodes.pop();
    }
}

void printListReversingly_Recursively(ListNode*
pHead) {
    if(pHead != nullptr) {
        if(pHead->m_pNext != nullptr) {

printListReversingly_Recursively(pHead->m_pNext);
        }
        cout << pHead->m_nKey << endl;
    }
}

int main(int argc, char *argv[])
{
    ListNode* node1 = createListNode(1);
    ListNode* node2 = createListNode(2);
    ListNode* node3 = createListNode(3);
    ListNode* node4 = createListNode(4);
    ListNode* node5 = createListNode(5);
    ListNode* node6 = createListNode(6);

    connectListNode(node1, node2);
    connectListNode(node2, node3);
    connectListNode(node3, node4);
    connectListNode(node4, node5);
    connectListNode(node5, node6);

    cout << "------------test Iteratively---------
---" << endl;
    printListReversingly_Iteratively(node1);

    cout << "------------test Recursively---------
---" << endl;
    printListReversingly_Recursively(node1);

    destroyList(node1);
```

```
    return 0;
}
```

重建二叉树

题目：输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如，输入前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}，则重建如图 2.6 所示的二叉树并输出它的头节点。二叉树节点的定义如下：

```
struct BinaryTreeNode
{
    int                    m_nValue;
    BinaryTreeNode*        m_pLeft;
    BinaryTreeNode*        m_pRight;
};
```