1.什么是C风格转换?

- 转换的含义是通过改变一个变量的类型为别的类型从而改变该变量的表示方式。为了类型转换一个简单对象为另一个对象你会使用传统的类型转换操作符。比如，为了转换一个类型为double的浮点数的指针到整型：

```
int i;
double d;
i = (int) d;
//或者:
i = int (d);
```

- 对于具有标准定义转换的简单类型而言工作的很好。然而，这样的转换符也能不分皂白的应用于类（class）和类的指针。ANSI-C++标准定义了四个新的转换符：'reinterpret_cast', 'static_cast', 'dynamic_cast' 和 'const_cast'，目的在于控制类(class)之间的类型转换。

2.什么是static_cast, dynamic_cast 以及 reinterpret_cast？区别是什么？为什么要注意？

```
reinterpret_cast<new_type>(expression)
dynamic_cast<new_type>(expression)
static_cast<new_type>(expression)
const_cast<new_type>(expression)
```

四种关键字:

- const_cast, 常量性转除;
- dynamic_cast, 向下安全转型;
- reinterpret_cast, 重新解释转型;
- static_cast, 静态转型;

2.1 const_cast, 常量性转除:

主要对变量的**常量性(const)进行操作, 移除变量的常量性, 即可以被非常量指向和引用**

```
#include <iostream>
/*常量性移除指针详解*/
struct S {
    S() : value(0) {}
    int value;
};

void CastConst (void)
{
    const S s;
    std::cout << "s.value = " << s.value << std::endl;
    //S* ps = &s; //error, 指向常量
    S* ps = const_cast<S*>(&s);
    ps->value = 1;
    std::cout << "s.value = " << s.value << std::endl;
    //S& rs = s; //error, 引用常量
    S& rs = const_cast<S&>(s);
    rs.value = 2;
    std::cout << "s.value = " << s.value << std::endl;
    //常量性转除:
    //s.value = 0
    //s.value = 1
    //s.value = 2
```

2.2 dynamic_cast, 向下安全转型:

- 主要应用于**继承体系**, 可以由 "**指向派生类的基类部分的指针**", 转换"**指向派生类**"或"**指向兄弟类**";

- static_cast**只能**转换为"指向派生类";

```
/*安全向下转型*/

struct B /*基类B*/ {
    virtual void f() { std::cout << "Base::f" << std::endl; }
    void thisf() { std::cout << "Base::thisf" << std::endl;}
    virtual ~B() {}
};

struct B2 /*基类B2*/ {
    virtual void g() { std::cout << "Base2::g" << std::endl; }
    void thisg() { std::cout << "Base2::thisg" << std::endl;}
    virtual ~B2() {}
};

struct D : public B, public B2 /*派生类D*/ {
    virtual void f() { std::cout << "Derived::f" << std::endl; }
    virtual void g() { std::cout << "Derived::g" << std::endl; }
    virtual ~D() {}
};

void CastDynamic (void)
{
     B* pB_D = new D;
     pB_D->f();
     //pD->g(); //error, 只包含B部分

     D *pD_D = dynamic_cast<D*>(pB_D); //转换为派生类
     pD_D->g();
     B2* pB2_D = dynamic_cast<B2*>(pB_D); //转换为兄弟类
     pB2_D->g();

     D *pD_Ds = static_cast<D*>(pB_D);//转换为派生类
     pD_Ds->g();
     //B2* pB2_Ds = static_cast<B2*>(pB_D); //error, 不能转换为兄弟类
}
    //安全向下转型:
    //Derived::f
    //Derived::g
    //Derived::g
    //Derived::g
```

2.3 reinterpret_cast, 重新解释转型:

- **主要是对2进制数据进行重新解释(re-interpret),不改变格式, 而static_cast会改变格式进行解释**;
- 如由**派生类转换基类**, 则重新解释转换, 不改变地址, 静态转换改变地址;

```
/*重新解释转型*/

struct rA { int m_a; };
struct rB { int m_b; };
struct rC : public rA, public rB {};

void CastReinterpret (void)
{
    int *i= new int;
    *i = 10;
    std::cout << "*i = " << *i << std::endl;
    std::cout << "i = " << i << std::endl;
    double *d=reinterpret_cast<double*> (i);
    std::cout << "*d = " << *d << std::endl;
    std::cout << "d = " << d << std::endl;

    rC c;
    std::cout << "&c = " << &c << std::endl
            << "reinterpret_cast<rB*>(&c) = " <<reinterpret_cast<rB*>(&c) <<
std::endl
            << "static_cast <rB*>(&c) = " << static_cast <rB*>(&c) << std::endl
            << "reinterpret_cast<rA*>(&c) = " <<reinterpret_cast<rA*>(&c) <<
std::endl
            << "static_cast <rA*>(&c) = " << static_cast <rA*>(&c) << std::endl
            << std::endl;

    //重新解释转型:
    //*i = 10
    //i = 0x471718
    //*d = 2.55917e-307
    //d = 0x471718
    //&c = 0x22feb0
    //reinterpret_cast<rB*>(&c) = 0x22feb0
    //static_cast <rB*>(&c) = 0x22feb4
    //reinterpret_cast<rA*>(&c) = 0x22feb0
    //static_cast <rA*>(&c) = 0x22feb0
```

2.4 static_cast, 静态转型:

- 主要是数据类型的转换, 还可以用于继承;

参考资料： C++/面试 - 四种类型转换(cast)的关键字 详解 及 代码

3.手写

```
#include<iostream>
#include<cassert>
using namespace std;
```

```
void *memcpy2(void *memTo, const void *memFrom, size_t size)
{
    assert((memTo != NULL) && (memFrom != NULL));
    char *tempFrom = (char*)memFrom;           //保存memFrom首地址
    char *tempTo = (char*)memTo;
    while (size-- > 0)
    {
        *tempTo++ = *tempFrom++;

    }
    return memTo;
}

int main()
{
    int x;
    char strSrc[] = "Hello World ! Fighting!";
    char strDest[20];
    cout << "please enter x" << endl;
    cin >> x;
    memcpy2(strDest, strSrc, x);
    strDest[x] = '\0';
    cout << "strDest:" << strDest << endl;
    return 0;
}
```

4. 代码题-网易k子序列

给定一个长度为n的整型数组nums和一个数值k,返回第k小的子序列和。 一个子序列是指非空且不间断的子数组.子序列和 则指子序列的和.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    int k;
    cin >> n >> k;
    vector<int> num;
    vector<int> sum;
    int sum_num = 0;
    for (int i = 0; i < n; i++)
    {
        int temp;
        cin >> temp;
        sum_num += temp;
        num.push_back(temp);
        sum.push_back(sum_num);
    }
```

```
    vector<int> ans;
    for (int i = 0; i < n; i++)
    {
        ans.push_back(sum[i]);
        for (int j = i + 1; j < n; j++)
        {
            ans.push_back(sum[j] - sum[i]);
        }
    }
    sort(ans.begin(), ans.end());
    cout << ans[k - 1] << endl;
    return 0;
}
```

输入: nums = [2,1,3], k = 4 输出: 3

5.面向对象的三大特性是什么

封装、继承、多态

6.如何解决哈希冲突

6.1哈希表简介

1. key(关键字)
2. f(key)哈希函数
3. hash冲突：f(key)相同

> 1）开放定址法： 这种方法也称再散列法，其基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。这种方法有一个通用的再散列函数形式： Hi=（H（key）+di）% m i=1，2，…，n 其中H（key）为哈希函数，m 为表长，di称为增量序列。增量序列的取值方式不同，相应的再散列方式也不同。主要有以下三种：
>
> 1. 线性探测再散列 dii=1，2，3，…，m-1 这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。
> 2. 二次探测再散列 di=1^2，-1^2，2^2，-2^2，…，k^2，-k^2 (k<=m/2) 这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。
> 3. 伪随机探测再散列 di=伪随机数序列。 具体实现时，应建立一个伪随机数发生器，（如i=(i+p) % m），并给定一个随机数做起点。 例子
> 4. 例如，已知哈希表长度m=11，哈希函数为：H（key）= key % 11，则H（47）=3，H（26）=4，H（60）=5，假设下一个关键字为69，则H（69）=3，与47冲突。 如果用线性探测再散列处理冲突，下一个哈希地址为H1=（3 + 1）% 11 = 4，仍然冲突，再找下一个哈希地址为H2=（3 + 2）% 11 = 5，还是冲突，继续找下一个哈希地址为H3=（3 + 3）% 11 = 6，此时不再冲突，将69填入5号单元。
> 5. 如果用二次探测再散列处理冲突，下一个哈希地址为H1=（3 + 1^2）% 11 = 4，仍然冲突，再找下一个哈希地址为H2=（3 - 1^2）% 11 = 2，此时不再冲突，将69填入2号单元。

> 6. 如果用伪随机探测再散列处理冲突，且伪随机数序列为：2，5，9，.....，则下一个哈希地址为H1=（3 + 2）% 11 = 5，仍然冲突，再找下一个哈希地址为H2=（3 + 5）% 11 = 8，此时不再冲突，将69填入8号单元。

> 2）再哈希法 这种方法是同时构造多个不同的哈希函数： Hi=RH1（key） i=1，2，…，k 当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

> 3）链地址法 这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

> 4）建立公共溢出区 这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

6.2优缺点

6.2.1开放散列表（open hashing）/拉链法（针对桶链结构）

> 1）优点：

①对于记录总数频繁可变的情况，处理的比较好（也就是避免了动态调整的开销）
②由于记录存储在结点中，而结点是动态分配，不会造成内存的浪费，所以尤其适合那种记录本身尺寸（size）很大的情况，因为此时指针的开销可以忽略不计了
③删除记录时，比较方便，直接通过指针操作即可\

> 2）缺点：

①存储的记录是随机分布在内存中的，这样在查询记录时，相比结构紧凑的数据类型（比如数组），哈希表的**跳转访问会带来额外的时间开销**

②如果所有的 key-value 对是可以提前预知，并之后不会发生变化时（即不允许插入和删除），可以人为创建一个不会产生冲突的完美哈希函数（perfect hash function），此时封闭散列的性能将远高于开放散列

③由于使用指针，记录**不容易进行序列化（serialize）操作**

6.2.2封闭散列（closed hashing）/ 开放定址法

> 1）优点：

①记录更容易进行序列化（serialize）操作

②如果记录总数可以预知，可以创建完美哈希函数，此时处理数据的效率是非常高的

> 2）缺点：

①存储记录的数目不能超过桶数组的长度，如果超过就需要扩容，而扩容会导致某次操作的时间成本飙升，这在实时或者交互式应用中可能会是一个严重的缺陷

②使用探测序列，有可能其**计算的时间成本过高**，导致哈希表的处理性能降低

③由于记录是存放在桶数组中的，而桶数组必然存在空槽，所以当记录本身尺寸（size）很大并且记录总数规模很大时，**空槽占用的空间会导致明显的内存浪费**

④删除记录时，比较麻烦。比如需要删除记录a，记录b是在a之后插入桶数组的，但是和记录a有冲突，是通过探测序列再次跳转找到的地址，所以如果直接删除a，a的位置变为空槽，而空槽是查询记录失败的终止条件，这样会导致记录b在a的位置重新插入数据前不可见，所以不能直接删除a，而是设置删除标记。这就需要额外的空间和操作。

6.5总结

> 1）拉链法

1. (优点)经常的数据变化，数据的增加删除处理起来非常方便
2. (优点)内存浪费少
3. (缺点)因为指针内存不连续，跳转访问时存在时间浪费
4. (缺点)如果提前知道不会有哈希冲突，用封闭散列性能会高于开放散列

> 2）封闭散列

1. (优点)更容易序列化
2. (优点)如果提前知道记录，可以创建完美哈希函数，此时数据处理效率高
3. (缺点)增加删除处理非常麻烦，扩容，删除都存在一定的问题
4. (缺点)探测序列的计算成本有可能非常高
5. (缺点)记录内存空槽导致明显的内存浪费

6.6总结

> 1）开放定址法：

1. 线性探测再散列
2. 二次探测再散列
3. 伪随机数再散列

> 2）再哈希法 3）拉链法 4）建立公共溢出区

参考资料：Hash冲突的四种解决办法

7.const修饰的变量和类成员函数与普通的变量，函数有什么区别

> 在类中，如果你不希望某些数据被修改，可以使用const关键字加以限定。const 可以用来修饰成员变量和成员函数。

1. const成员变量

> **const成员变量的用法和普通const变量的用法相似**，只需要在声明时加上const关键字。初始化const成员变量只有一种方法，就是通过构造函数的**初始化列表**。

2. const成员函数（常成员函数）

> **const成员函数可以使用类中的所有成员变量，但是不能修改它们的值，这种措施主要还是为了保护数据而设置的。const成员函数也称为常成员函数。** 我们通常将 get 函数设置为常成员函数。读取成员变量的函数的名字通常以get开头，后跟成员变量的名字，所以通常将它们称为get函数。 常成员函数需要

> 在声明和定义的时候在函数头部的结尾加上const关键字。 常成员函数需要在声明和定义的时候在函数头部的结尾加上 const 关键字，请看下面的例子：

```
class Student{
public:
    Student(char *name, int age, float score);
    void show();
    //声明常成员函数
    char *getname() const;
    int getage() const;
    float getscore() const;
private:
    char *m_name;
    int m_age;
    float m_score;
};

Student::Student(char *name, int age, float score): m_name(name), m_age(age),
m_score(score){ }
void Student::show(){
    cout<<m_name<<"的年龄是"<<m_age<<", 成绩是"<<m_score<<endl;
}
//定义常成员函数
char * Student::getname() const{
    return m_name;
}
int Student::getage() const{
    return m_age;
}
float Student::getscore() const{
    return m_score;
}
```

> 需要强调的是，必须在成员函数的声明和定义处同时加上const关键字。char *getname() const和char *getname()是两个不同的函数原型，如果只在一个地方加const会导致声明和定义处的函数原型冲突。

3. 最后再来区分一下 const 的位置：

> **函数开头的const用来修饰函数的返回值**，表示返回值是const类型，也就是不能被修改，例如`const char * getname()`。 **函数头部的结尾加上const表示常成员函数**，这种函数只能读取成员变量的值，而不能修改成员变量的值，例如`char * getname() const`。

8.

9.

10.

2. 了解过STL的哪些容器?

3. vector怎么实现元素的扩充?

4. vector删除元素能不能直接用迭代器++?

5. 了解过C++11吗?

6. 智能指针怎么解决循环引用?

7. 了解过右值引用吗？讲一讲

1、C++的多态 3、知道哪些排序方法，描述一下快排，哪些排序是稳定的，插入排序和冒泡排序哪个更好 4、struct 和 class的区别 5、哈希表，冲突处理方法，一个长度很长的字符串如何计算它的键 6、数组和链表的区别，数组满了如何扩容，数组移动时元素中的指针怎么处理，数组移动是深拷贝还是浅拷贝 7、静态全局变量和类中的静态变量的区别，他们存储的位置；全局变量和局部变量分别是在什么时候分配的 8、

问了我的项目。

C++： 1.讲一讲虚函数 2.虚函数是怎么实现的多态的 3.智能指针 4.shared_ptr的计数器为0将指向的变量释放后，指向这个变量的weak_ptr怎么删除 5.静态变量和局部变量的不同，储存在哪

网络： 1.三次握手、四次挥手 2.握手挥手过程中信号传输失败应该怎么处理 3.失败重传的时间间隔（1，2，4，8.。。。达到限制后自动断开连接）

操作系统： 1.讲一讲虚拟内存 2.内存虚拟内存之间是怎么调度的 3.页地址是怎么查询到物理地址的 4.页表到实际物理地址的实现设计什么数据结构（我答了个map的映射关系，不知道对不对）

算法数据结构： 1.列举有什么排序 2.快排的实现过程 3.快排复杂度、最坏情况复杂度以及在什么情况下达到最坏复杂度 4.插入排序的复杂度，什么情况下比快排好

https://leetcode-cn.com/circle/discuss/j0axtu/