c++面向对象编程：一些说明

> 写在前面 已经很久没有写代码了，包括自己高中的时候学的一些小伎俩基本也忘干净了，现在对于计算机课来说还是很紧张的。所以打算再写一篇文档，以复刻ai基础考前抱佛脚熟读讲义之便。 唉
>
> 哦哦，不过这个markdown转换成网页或者pdf应该会相对的好看一些，毕竟它不需要什么图片和数学公式，哪天我一定要学一学比较专业的前端，感觉自己对这个还挺有兴趣的

面向对象编程的基本结构

常量与只读变量

常量

常量指的是在**编译**期间就可以确定的值，且整个运行期间常量的值不会发生变化。 在c++代码中，我们可以使用constexpr来表示常量，常量拥有可以在编译时就确定的特性，所以常量有着非常奇妙的应用：

▶ 变参模板：包展开error

```
#include <iostream>
#include <string>
template<typename T, typename... Us>
void f(T t,Us... us);
template<typename... Ts>
void print(Ts... ts);
template<typename T, typename... Us>
```

```
void f(T t, Us... us) {
    if (sizeof...(us) == 0){
        std::cout << t << std::endl;
        return;
    }
    else{
        std::cout << t;
        print(us...);
    }
    return;
}
template<typename... Ts>
void print(Ts... ts) {
    if (sizeof...(ts) == 0){
        std::cout<<std::endl;
        return;
    }
    else{
        f(ts...);
    }
}
int main() {
    std::string n("nullptr");
    print(false);
    print();
    print("hello", " world");
    print("do not ", "dereference ", n);
    print(1, '+', 1LL, "=", 2.0);
}
```

上述代码是不是看着很对？两个模板函数相互调用，可以实现递归输出全部args中的元素，但是它却无法编译，究其原因是因为编译器在编译时看不到程序中的if，所以编译器在预处理时会好奇print()函数是什么，但是编译时程序并没有给出print()的定义，所以会报编译错误。 我们只需把程序中两个if类似的改为：

▶ 变参模板：包展开accept

```
void print(Ts... ts) {
    if constexpr(sizeof...(ts) == 0){
        std::cout<<std::endl;
        return;
    }
    else{
        f(ts...);
    }
}
```

而constexpr if是给编译器看的if，这样程序就可以正确运行了。

const和引用

只读和引用结合，可以产生四种不同的变量：

```
T x;
const T x;
T& x;
const T& x;
```

其中，T&、T、const T都可以用来初始化const T &类型的引用，但是const T类型的只读变量和const T&类型的引用不能用来初始化T&类型的引用（除非进行强制的类型转换），即：

```
int x = 1;
const int& y = x;//正确
int& z = x;//正确
int& w = y;//错误
```

即等式左边的读写能力一定要小于等于等式右边的读写能力。 这很好理解，因为如果等式左边的读写能力更强，就因为可以通过左边的引用修改右边一个较弱的引用，而这显然是有违正义的。

const与指针

定义指针是可以在指针前加入const，意为：指向只读变量的指针

- 不可通过该指针修改其指向的内容
- 变量不需要真的是只读变量，而编译器会认为只读指针指向的变量是只读的

```
int n,m;
const int *ptr=&n;
*p=5;//编译错误
n=4;//正确
p=&m;//正确，指针的指向可以变化，只是不能通过指针修改内容
```

相同的，c++也可以定义只读的指针变量：

- 该指针初始化后不可修改指针内容，不允许指针指向其他对象

```
int a=1;
int *const ptr=&a;
*a=2;//正确
```

总结：**只读变量的指针不允许指针修改指向对象的内容，只读的指针变量不允许指针更改指向的对象**

函数的高级特性

缺省函数

缺省函数指在c++中，我们可以给函数最右边连续的任意个参数指定缺省值，调用函数时若不在相应的位置写参数，则调用默认参数：

```
void func1(int x1,int x2=1,int x3=2);
void func2(int x1=1,int x2,int x3=2);//错误
```

函数重载的复杂情况

函数重载时，**只有参数列表不同的两个同名函数才会被认为是重载的函数，参数列表不同包括参数个数不同和参数类型不同**，如果两个同名函数只是返回值不同，则不构成重载。 函数重载还具有以下几个复杂情况：

- T类型参数和const T类型是相同的类型
- T*类型参数和T* const类型是相同的类型
- **T&类型参数和const T&类型是不同的类型**
- **T*类型参数和const T*类型是不同的类型**

也就是说，同一个函数可以通过引用是否有const、指针是否有const进行重载，这可以解决题目中一些重载的问题

函数指针

函数指针即可以写一个函数，使函数的参数也是表示过程的函数：

```
int (*ptr)(int,int);//开头的int是函数指针返回的类型
int max(int,int);
(*ptr)=&max;
(*ptr)(3,5);
```

可以通过函数指针的解地址运算符来调用其所指向的函数

lambda表达式

lambda表达式是c++11中引入的新特性，可以用来表示一个匿名函数，其语法为：

```
[capture](parameters)->return-type{body}
```

其中，capture是捕获列表，parameters是参数列表，return-type是返回值类型，body是函数体。 即：

```
int sum(int n){
    return [n](int x){return x+n;};
}
```

函数体内可以访问全局变量，但是不能访问局部变量，如果想访问局部变量，可以在捕获列表中加入该变量。捕获列表有几种形式：

- [a,b]：表示以值的形式捕获a和b
- [=]：表示以值的形式捕获所有局部变量
- [&a,&b]：表示以引用的形式捕获a和b
- [&]：表示以引用的形式捕获所有局部变量

当然，我们也可以类似的定义返回lambda的lambda：

```
auto add=[](int x){
    return [=](int y){
        return x+y;
    };
};
auto add5=add(5);
add5(3);//8
```

命名空间

命名空间是c++中用来解决命名冲突的一种机制，其语法为：

```
using 别名=类型;
using Func=int(int,int);
Func* ptr=max;
```

面向对象编程

类型转换

c++可以进行显式类型转换和隐式类型转换：

```
class complex{
public:
    int real,imag;
    complex(int r):real(r){}
};
complex c1(1);
complex c2=3;//隐式类型转换,调用构造函数
complex c3=complex(4);//显式类型转换
```

使用explicit关键字可以阻止隐式类型转换，但是仍然可以使用显式类型转换

静态成员

静态成员是指在类中声明的static成员，静态成员不属于任何对象，而是属于整个类，静态成员可以通过类名访问，也可以通过对象访问，但是不能通过对象访问私有的静态成员。

```
class complex{
public:
    static int count;
    complex(int r):real(r){count++;}
    ~complex(){count--;}
};
int complex::count=0;
```

其中的count就是一个静态成员。

oop中的只读

在c++中，我们可以添加只读对象、只读成员函数与只读成员变量，只读对象的值在构造时就确定，且在整个运行期间不会发生变化，**只读成员函数不会修改对象的值（即不修改任何只读成员变量）**，只读成员变量在构造时就确定，且在整个运行期间不会发生变化。

```
class complex{
public:
    int real,imag;
    complex(int r):real(r){}
    int getreal()const{return real;}
    void setreal(int r){real=r;}
};
```

注意：只读对象只能使用构造函数、析构函数、只读成员函数，不可以修改成员变量，除非使用mutable关键字修饰。

```
class test{
    mutable int n;
    bool flag;
public:
    bool getdata()const{
        n++;//只读成员函数修改了mutable的成员变量
        return flag;
    }
};
```

运算符重载

继承和多态

模板