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

浅复制与深复制

浅复制是指在复制对象时，只复制对象的值，而不复制对象的指针，这样会导致两个指针指向同一个对象，当其中一个指针释放对象时，另一个指针就会指向一个已经释放的对象，这就是悬空指针。

深复制是指在复制对象时，不仅复制对象的值，还复制对象的指针，这样就不会出现悬空指针的情况。 **预置的运算符重载做的是浅复制，所以可以通过深复制重载使指针指向不同的地方**

```
class String{
    char *ptr;
public:
    String():ptr(new char[1]){ptr[0]='\0';}
    String(const char *s){
        int len=strlen(s);
        ptr=new char[len+1];
        strcpy(ptr,s);
    }
    const char *c_str(){return ptr;}
    friend String &operator=(const char* s){
        if(this==&s)
            return *this;//这是为了防止String s=s
        delete[] ptr;
        int len=strlen(s);
        ptr=new char[len+1];
        strcpy(ptr,s);
        return *this;
    }
```

```
    ~String(){delete[] ptr;}
};
```

注意：在上述条件下，优先进行重载的等于，而不是进行隐式转换+等于

友元的好处

在重载运算符时，很多时候我们都选择使用友元重载，而不是将重载写入成员函数，这是因为对于1+c的格式，无法使用成员函数重载，而只能使用友元重载。

```
class complex{
public:
    int real,imag;
    complex(int r):real(r){}
    friend complex operator+(int n,const complex &c){
        return complex(n+c.real);
    }
};
complex c(1);
1+c;//调用operator+(int n,const complex &c)
```

类型转换重载

类型转换重载是指在c++中，我们可以重载类型转换运算符，使得我们可以自定义类型转换的过程，其语法为：

```
operator type();
```

即：

```
class complex{
public:
    int real,imag;
    complex(int r):real(r){}
    operator int(){return real;}
};
complex c(1);
int n=c;//调用operator int()函数
```

流插入/提取运算符重载

cout的本质是在iostream中定义的ostream类的对象 所以我们可以对<<和>>进行重载

```
class complex{
public:
    int real,imag;
    complex(int r):real(r){}
    friend ostream &operator<<(ostream &out,const complex &c){
        out<<c.real;
        return out;
    }
};
complex c(1);
cout<<c;
```

自增减运算符重载

c++为了区分i++和++i，前者被称为后置运算符，后者被称为前置运算符。 重载后置运算符的语法为：

```
type &operator++(type2);
type &operator--(type2);
```

重载前置运算符的语法为：

```
type operator++(type2,int);
type operator--(type2,int);
```

其中int是一个占位符，用来区分前置和后置运算符。

函数调用运算符重载

函数调用运算符()也可以被重载，其语法为：

```
type operator()(type2);
```

例如：

```
class complex{
public:
    int real,imag;
    complex(int r):real(r){}
    int operator()(int n){
        return real+n;
    }
};
complex c(1);
c(2);//返回3
```

继承和多态

派生类成员的可见性

对于公开继承的派生类，派生类可以访问基类的public成员和protected成员，但是不能访问基类的private成员。

```
class base{
    int privatea;
public:
    int publica;
};
class derived:public base{
public:
    void func(){
        publica=1;//正确
        privatea=1;//错误
    }
};
```

对于私有继承的派生类，外部不可以通过派生类访问任何从基类继承的成员。

```
class base{
    int privatea;
public:
    int publica;
};
class derived:private base{
public:
    void func(){
        publica=1;//正确
        privatea=1;//错误
    }
};
int main(){
    derived d;
    d.publica=1;//错误
}
```

基类的私有成员对派生类不可见有些不够灵活，所以需要一种折中的机制，即将基类的成员定义为protected
protected的基类成员可以被以下函数访问：

- 基类的成员函数
- 基类的友元函数
- 派生类的成员函数

- 派生类的友元函数

三种类型的继承总结一下，可以得到以下表格：

基类成员在派生类的权限 \ 继承方式	公有继承	私有继承	保护继承
公有成员	公有	私有	保护
私有成员	派生类成员不可访问	派生类成员不可访问	派生类成员不可访问
保护成员	保护	私有	保护

派生类到基类

首先指出：**在派生类的虚函数后加override关键字表示派生类的虚函数显式覆盖了基类的虚函数，和explicit起相同效果，显式的一定是更好的 final则是禁止覆盖的关键字**

派生类到基类的隐式转换

一个公开继承的派生类对象可以隐式转换到基类

- 该对象不包括派生类定义的成员
- 从派生类构造基类时会调用基类的复制构造函数
- 将派生类赋值到基类时会调用基类的复制赋值重载

```
class base{};
class derived:public base{};
void func(base b);
int main(){
    derived d;
    func(d);//从派生类构造基类
    base b=d;//派生类赋值到基类
    b=d;//基类形参，派生类实参
    f(d);//
}
```

基类的引用到派生类

若派生类公开继承自基类，则基类的引用可以绑定到派生类对象 但是即使绑定到了派生类，也只能访问基类的成员，不能访问派生类的成员

```
class base{
public:
    int basemem;
};
class derived:public base{
```

```
public:
    int derivedmem;
};
int main(){
    derived d;
    base &b{d};
    b.basemem=1;//正确
    b.derivedmem=1;//错误
}
```

当通过基类引用调用基类和派生类中重名的虚函数时，引用基类对象则调用基类虚函数，引用派生类对象则调用派生类虚函数

基类的指针到派生类

若派生类公开继承自基类，派生类对象的指针可以直接赋值给基类的指针 即使基类指针指向的是派生类对象，也不能通过基类指针访问基类没有但是派生类有的成员

```
class base{
public:
    int basemem;
};
class derived:public base{
public:
    int derivedmem;
};
int main(){
    derived d;
    base *b=&d;
    b->basemem=1;//正确
    b->derivedmem=1;//错误
}
```

通过强制类型转换，也可以将基类的指针转换为派生类的指针：

```
derived *d=(derived*)b;
```

虚函数的访问权限则是根据指针类型来判断的

运行时类型识别

引入头文件

```
#include <typeinfo>
```

对于相同表达式的类型或相同类型，typeid运算符可以得到相同的值：

```
#include <iostream>
#include <typeinfo>
using namespace std;
class base{
public:
    virtual ~b(){}
};
class d1:public b{};
class d2:public b{};
int main(){
    B* b=new d1;
    if(typeid(*b)==typeid(d1))
        cout<<"b has type d1";
    if(typeid(*b)==typeid(d2))
        cout<<"b has type d2";
    delete b;
}
```

四种类型转换

危险的const_cast和reinterpret_cast这里就不过多介绍了

static_cast

静态的类型转换，基本上安全

- 算术类型转换
- 自定义类型转换
- **派生类指针和基类指针间的相互转换**

但是static_cast在向下转型：基类指针到派生类指针时会有风险，在长数到段数转型时也有溢出的风险

dynamic_cast

动态类型转换，在保证多态类型向下转型绝对安全

```
Base* ptrOK=new derived{};
Base* ptrBad=new base{};
dynamic_cast<Derived*>(ptrOK);//正确
dynamic_cast<Derived*>(ptrBad);//正确，但得到空指针
```

在if条件判断语句中可以声明变量，如果变量声明为假，即变赋值右侧转换为bool值为false，则不进行if中的语句。

```
if(auto ptrDog=dynamic_cast<const Dog*>(pAnimal))
    ptrDog->bark();
```

如上式，我们可以利用这个特性做到运行时类型识别，但是它的性能要低于typeid

模板

非类型模板参数

非类型模板参数是指模板参数不是类型，而是一个值，其语法为：

```
template<int N>
class array{
    int data[N];
public:
    int size(){return N;}
};
array<10> a;
```

在编译器角度看，array<10>和array<20>是两个不同的类 使用非类型模板参数可以提高程序的执行效率

模板的特化与实例化

模板的特化是指对于某些特殊的类型，我们可以对模板进行特殊的定义，其语法为：

```
template<typename T>
class complex{
public:
    T real,imag;
    complex(T r):real(r){}
};
template<>
class complex<double>{
public:
    double real,imag;
    complex(double r):real(r){}
};
```

在上述代码中，我们对于double类型的模板进行了特化，即对于double类型的模板，我们可以使用更加高效的实现。

在类模板派生时，需要对基类实例化，即：

```
template<typename T>
class base{};
template<typename T>
class derived:public base<T>{};
```

标准模板库

容器

顺序容器

顺序容器是指容器中的元素是按照一定的顺序排列的，包括vector、list、deque、array、forward_list

- vector：动态数组，支持随机访问，支持尾部插入和删除，不支持中间插入和删除
- list：双向链表，支持双向访问，支持任意位置插入和删除，**不支持随机存取**
- deque：双端队列，支持随机访问，支持头尾插入和删除

关联容器

关联容器是指容器中的元素是按照一定的规则排列的，包括set、map、multiset、multimap

- set：集合，元素唯一，按照一定的规则排列
- map：映射，元素唯一，按照一定的规则排列，每个元素包含一个键值对
- multiset：集合，元素不唯一，按照一定的规则排列
- unordered_set：集合，元素唯一，按照一定的规则排列，使用哈希表实现 unordered_set是一种无序关联容器，插入和检索只需要常数时间，但是不支持有序性操作，如lower_bound、upper_bound等

容器适配器

容器适配器是指容器的接口和实现分离，包括stack、queue、priority_queue

上述容器会有一些共有的成员函数：

- begin()：返回指向容器第一个元素的迭代器
- end()：返回指向容器最后一个元素的下一个位置的迭代器
- rbegin()：返回指向容器最后一个元素的迭代器
- rend()：返回指向容器第一个元素的前一个位置的迭代器
- erase()：删除指定位置的元素
- clear()：删除容器中的所有元素
- empty()：判断容器是否为空
- size()：返回容器中元素的个数
- swap()：交换两个容器的元素

顺序容器还会有一些常用的成员函数：

- push_back()：在容器尾部插入元素
- pop_back()：删除容器尾部的元素
- front()：返回容器头部的元素的引用
- back()：返回容器尾部的元素的引用

迭代器

迭代器是一种类似指针的对象，可以用来遍历容器中的元素，迭代器的类型取决于容器的类型，迭代器的类型可以分为以下几种：

- 输入迭代器：只读，只能单步向前移动（*p可以获取数据）
- 输出迭代器：只写，只能单步向前移动（*p可以修改数据）
- 前向迭代器：可读写，只能单步向前移动（可以p++）
- 双向迭代器：可读写，可以双向移动（可以p--）
- 随机访问迭代器：可读写，可以双向移动，可以随机访问（可以p+=i，比较大小，获取下标p[i]）
- 连续迭代器：可读写，可以双向移动，可以随机访问，可以进行指针运算（数据连续存储，可以使用p-q来获得地址）

std::copy

std::copy是一个泛型算法，可以将一个容器的元素复制到另一个容器中，其语法为：

```
copy(begin,end,begin2);
```

其中begin和end是第一个容器的起始位置和结束位置，begin2是第二个容器的起始位置，第二个容器的大小必须大于等于第一个容器的大小。

back_inserter()是一个函数模板，可以将元素插入到容器的尾部，其语法为：

```
copy(begin,end,back_inserter(container));
```

std::copy_if

std::copy_if比std::copy多了一个参数，最后一个参数是可调用对象，如果作用在当前参数上返回true，则将当前参数复制到第二个容器中，其语法为：

```
copy_if(begin,end,begin2,func);
```

函数对象

函数对象有独立于调用的初始化过程，使用lambda表达式可以方便的定义函数对象，其语法为：

```
template<typename T>
void someAsyncTask(T callback){
    return;
    int result=/*answer*/;
    callback(result);
}
int main(){
    someAsyncTask([&](int result){
        cout<<"The answer is "<<result<<endl;
    });
}
```

视图

视图是一种容器适配器，可以将容器的部分元素作为容器使用，其语法为：

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ranges>
int main(){
    std::vector<int> v{1,2,3,4,5,6,7,8,9,10};
    std::ranges::reverse_view v2{v};
    for(auto i:v2)
        std::cout<<i<<' ';
}
```

上述reverse_view是反转视图，试图提供了一种不修改原容器的方法来访问容器的元素，这种方法可以提高程序的效率。

当然，还有其他视图：

- filter_view：筛选视图

```
std::vector<int> v{1,2,3,4,5,6,7,8,9,10};
std::ranges::filter_view v2{v,[](int i){return i%2==0;}};
for(auto i:v2)
    std::cout<<i<<' ';
```

- transform_view：转换视图

```
std::vector<int> v{1,2,3,4,5,6,7,8,9,10};
std::ranges::transform_view v2{v,[](int i){return i*2;}};
for(auto i:v2)
    std::cout<<i<<' ';
```

- take_view：早退视图

```
std::vector<int> v{1,2,3,4,5,6,7,8,9,10};
std::ranges::take_view v2{v,5};
for(auto i:v2)
    std::cout<<i<<' ';
```

习题精选