c++复习

指针与引用的区别

1. 指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已
2. 可以有const指针，但是没有const引用
3. 指针可以有多级，但是引用只能是一级
4. 指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化
5. 指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了
6. "sizeof引用"得到的是所指向的变量(对象)的大小，而"sizeof指针"得到的是指针本身的大小
7. 指针和引用的自增(++)运算意义不一样，一个是内存地址++，一个是值++
8.

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

void test(int *p)
{
    int a=1;
    p=&a;
    cout<<p<<" "<<*p<<endl;
}

int main(void)
{
    int *p=NULL;
    test(p);
    if(p==NULL)
    cout<<"指针p为NULL"<<endl;
    system("pause");
    return 0;
}

// output:0x22ff44 1
// 说明*p作为形参是新创造出来的
```

9.

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

void test(int *&p)
{
    int a=1;
    p=&a;
    cout<<p<<" "<<*p<<endl;
}

int main(void)
{
    int *p=NULL;
```

```
    test(p);
    if(p!=NULL)
    cout<<"指针p不为NULL"<<endl;
    system("pause");
    return 0;
}

// output:0x22ff44 1
//          指针p不为NULL
// 这个是指针引用同时改的例子
```

C++中public、protected、private的区别

1. **访问范围**：private: 只能由该类中的函数、其友元函数访问,不能被任何其他访问，该类的对象也不能访问。protected: 可以被该类中的函数、子类的函数、以及其友元函数访问,但不能被该类的对象访问。public: 可以被该类中的函数、子类的函数、其友元函数访问,也可以由该类的对象访问
2. **类的继承后方法属性变化**：使用private继承,父类的所有方法在子类中变为private。使用protected继承,父类的protected和public方法在子类中变为protected,private方法不变。使用public继承,父类中的方法属性不发生改变

const关键字

1. C++ const 允许指定一个语义约束，编译器会强制实施这个约束，允许程序员告诉编译器某值是保持不变的。如果在编程中确实有某个值保持不变，就应该明确使用const，这样可以获得编译器的帮助
2.
```
#include<iostream>

using namespace std;

void Cpf(int *const a) //指针常量?
{
    cout<<*a<<" ";
    *a = 9;
}

int main(void)
{
    int a = 8;
    Cpf(&a);
    cout<<a; // a 为 9
    system("pause");
    return 0;
}
```

3.
```
#include<iostream>

using namespace std;

class Test
{
public:
    Test(){}
    Test(int _m):_cm(_m){}
```

```
    int get_cm()const //即使没有改变对象的值，编译器也认为函数会改变对象的值，所以必须加const
    {
        return _cm;
    }

private:
    int _cm;
};



void Cmf(const Test& _tt) //常引用
{
    cout<<_tt.get_cm();
}

int main(void)
{
    Test t(8);
    Cmf(t);
    system("pause");
    return 0;
}
```

4. const 关键字不能与 static 关键字同时使用，因为 static 关键字修饰静态成员函数，静态成员函数不含有 this 指针，即不能实例化，const 成员函数必须具体到某一实例

static关键字

1. **静态成员**：在类的所有对象中是共享的。如果不存在其他的初始化语句，在创建第一个对象时，所有的静态数据都会被初始化为零。我们不能把静态成员的初始化放置在类的定义中，但是可以在类的外部通过使用范围解析运算符 :: 来重新声明静态变量从而对它进行初始化
2. **静态成员函数**：没有this指针，只能访问类中静态成员和静态成员函数，普通成员函数有 this 指针，可以访问类中的任意成员
3.

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // 构造函数定义
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // 每次创建对象时增加 1
         objectCount++;
      }
      double Volume()
      {
```

```
21          return length * breadth * height;
22       }
23       static int getCount()
24       {
25          return objectCount;
26       }
27    private:
28       double length;     // 长度
29       double breadth;    // 宽度
30       double height;     // 高度
31 };
32 
33 // 初始化类 Box 的静态成员，固定格式
34 int Box::objectCount = 0;
35 
36 int main(void)
37 {
38 
39    // 在创建对象之前输出对象的总数
40    cout << "Inital Stage Count: " << Box::getCount() << endl;
41 
42    Box Box1(3.3, 1.2, 1.5);    // 声明 box1
43    Box Box2(8.5, 6.0, 2.0);    // 声明 box2
44 
45    // 在创建对象之后输出对象的总数
46    cout << "Final Stage Count: " << Box::getCount() << endl;
47 
48    return 0;
49 }
```

friend关键字

1. 友元是单向的
2. 因为友元函数没有this指针，则参数要有三种情况： 要访问非static成员时，需要对象做参数；要访问static成员或全局变量时，则不需要对象做参数；如果做参数的对象是全局对象，则不需要对象做参数.可以直接调用友元函数，不需要通过对象或指针

new与malloc区别

1. new操作符从**自由存储区（free store）**上为对象动态分配内存空间，而malloc函数从**堆**上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。
2. new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合**类型安全**性的操作符。而malloc内存分配成功则是返回void * ，需要通过强制类型转换将void*指针转换成我们需要的类型。
3. new内存分配失败时，会抛出bac_alloc异常，它**不会返回NULL**；malloc分配内存失败时返回NULL。

```
try
{
    int *a = new int();
}
catch (bad_alloc)
{
    ...
}
```

4. 使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算，而malloc则需要显式地指出所需内存的尺寸。
5. 使用new操作符来分配对象内存时会经历三个步骤：
    - 第一步：调用operator new 函数（对于数组是operator new[]）分配一块足够大的，**原始**的，未命名的内存空间以便存储特定类型的对象。
    - 第二步：编译器运行相应的**构造函数**以构造对象，并为其传入初值。
    - 第三部：对象构造完成后，返回一个指向该对象的指针。

    使用delete操作符来释放对象内存时会经历两个步骤：
    - 第一步：调用对象的析构函数。
    - 第二步：编译器调用operator delete(或operator delete[])函数释放内存空间。

    总的来说，new/delete会调用对象的构造函数/析构函数以完成对象的构造/析构。而malloc则不会。
6. C++提供了new[]与delete[]来专门处理数组类型，使用new[]分配的内存必须使用delete[]进行释放，至于malloc，它并知道你在这块内存上要放的数组还是啥别的东西

```
A * ptr = new A[10];//分配10个A对象
delete [] ptr;
int * ptr = (int *) malloc( sizeof(int)* 10 );//分配一个10个int元素的数组
```

7. operator new /operator delete的实现可以基于malloc，而malloc的实现不可以去调用new
8. opeartor new /operator delete可以被重载。标准库是定义了operator new函数和operator delete函数的8个重载版本
9. 使用malloc分配的内存后，如果在使用过程中发现内存不足，可以使用**realloc**函数进行内存重新分配实现内存的扩充。realloc先判断当前的指针所指内存是否有足够的连续空间，如果有，原地扩大可分配的内存地址，并且返回原来的地址指针；如果空间不够，先按照新指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存区域，而后释放原来的内存区域。new没有这样直观的配套设施来扩充内存。
10. 在operator new抛出异常以反映一个未获得满足的需求之前，它会先调用一个用户指定的错误处理函数，这就是**new-handler**。new_handler是一个指针类型，指向了一个没有参数没有返回值的函数,即为错误处理函数。为了指定错误处理函数，客户需要调用set_new_handler，这是一个声明于的一个标准库函数，set_new_handler的参数为new_handler指针，指向了operator new 无法分配足够内存时该调用的函数。其返回值也是个指针，指向set_new_handler被调用前正在执行（但马上就要发生替换）的那个new_handler函数。对于malloc，客户并不能够去编程决定内存不足以分配时要干什么事，只能看着malloc返回NULL。

11.

特征	new/delete	malloc/free
分配内存的位置	自由存储区	堆
内存分配成功的返回值	完整类型指针	void*
内存分配失败的返回值	默认抛出异常	返回NULL
分配内存的大小	由编译器根据类型计算得出	必须显式指定字节数
处理数组	有处理数组的new版本new[]	需要用户计算数组的大小后进行内存分配
已分配内存的扩充	无法直观地处理	使用realloc简单完成
是否相互调用	可以，看具体的operator new/delete实现	不可调用new
分配内存时内存不足	客户能够指定处理函数或重新制定分配器	无法通过用户代码进行处理
函数重载	允许	不允许
构造函数与析构函数	调用	不调用

虚基类

1. 虚基类是用关键字virtual声明继承的父类，即便该基类在多条链路上被一个子类继承，但是该子类中只包含一个该虚基类的备份，虚基类主要用来解决继承中的二义性问题，这就是是虚基类的作用所在
2. 每个子类的构造函数中必须显示的调用该虚基类的构造函数，不管该虚基类是不是直接的父类。
3. 虚基类的构造函数的调用早于其他非虚基类的构造函数的调用
4.

```
#include <iostream>
using namespace std;
class CBase
{
public:
    int a;
public:
    CBase(int na)
    {
        a=na;
        cout<<"CBase constructor! "<<endl;
    }


    ~CBase(){cout<<"CBase deconstructor! "<<endl;}
};


```

```
//派生类1(声明CBase为虚基类)
class CDerive1:virtual public CBase
{
public:
    CDerive1(int na):CBase(na)
    {
        cout<<"CDerive1 constructor! "<<endl;
    }


    ~CDerive1(){cout<<"CDerive1 deconstructor! "<<endl;}


    int GetA(){return a;}
};


//派生类2(声明CBase为虚基类)
class CDerive2:virtual public CBase
{
public:
    CDerive2(int na):CBase(na)
    {
        cout<<"CDerive2 constructor! "<<endl;
    }
    ~CDerive2(){cout<<"CDerive2 deconstructor! "<<endl;}
    int GetA(){return a;}
};


//子派生类
class CDerive12:public CDerive1,public CDerive2
{
public:
    CDerive12(int na1,int na2,int
na3):CDerive1(na1),CDerive2(na2),CBase(na3)
    {
        cout<<"CDerive12 constructor! "<<endl;
    }
    ~CDerive12(){cout<<"CDerive12 deconstructor! "<<endl;}
};
void main()
{
    CDerive12 obj(100,200,300);
    cout<<" CDerive12:a = "<<obj.a<<endl;
    //得到从CDerive1继承的值
    cout<<" from CDerive1 : a = "<<obj.CDerive1::GetA()<<endl;
    //得到从CDerive2继承的值
    cout<<" from CDerive2 : a = "<<obj.CDerive2::GetA()<<endl;


}
```

多态

1. **多态**按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

2.
```cpp
#include <iostream>
using namespace std;

class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
// 程序的主函数
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // 多态性的体现
   // 存储矩形的地址
   shape = &rec;
   // 调用矩形的求面积函数 area
   shape->area();

   // 存储三角形的地址
   shape = &tri;
   // 调用三角形的求面积函数 area
   shape->area();
```

```
54
55      return 0;
56  }
```

虚函数与纯虚函数

1. **虚函数** 是在基类中使用关键字 **virtual** 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为**动态链接**，或**后期绑定**。
2. 您可能想要在基类中定义虚函数，以便在派生类中重新定义该函数更好地适用于对象，但是您在基类中又不能对虚函数给出有意义的实现，这个时候就会用到**纯虚函数**
3.
```
1   class Shape {
2      protected:
3         int width, height;
4      public:
5         Shape( int a=0, int b=0)
6         {
7            width = a;
8            height = b;
9         }
10        // pure virtual function
11        virtual int area() = 0;
12  };
```

4. 有纯虚函数的类称为**抽象类**，不能建立抽象类对象，可将该类的构造函数说明为保护的访问控制权限。
5. **虚析构函数**是为了避免内存泄露，而且是当子类中会有指针成员变量时才会使用得到的。也就说虚析构函数使得在删除指向子类对象的基类指针时可以调用子类的析构函数达到释放子类中堆内存的目的，而防止内存泄露的
6.
```
1   #include<iostream>
2   using namespace std;
3
4   class ClxBase
5   {
6       public:
7           ClxBase() {};
8           virtual ~ClxBase() { cout<<"delete ClxBase"<<endl; };
9
10          virtual void DoSomething() { cout << "Do something in class
    ClxBase!" << endl;  };
11
12  };
13
14  class ClxDerived : public ClxBase
15  {
16      public:
17          ClxDerived() {};
18          ~ClxDerived() { cout << "Output from the destructor of
    class ClxDerived!" << endl;  };
19
20          void DoSomething() { cout << "Do something in class
    ClxDerived!" << endl;  };
21
```

```
};

int main(int argc, char const* argv[])
{
     ClxBase *pTest = new ClxDerived;
     pTest->DoSomething();
     delete pTest;
     return 0;
}
```

运算符重载

1.
```
//
// Created by zhang on 2020/9/1.
//

#ifndef NEWFAT_COMPLEX_H
#define NEWFAT_COMPLEX_H


class Complex {
private:
    double real;
    double imag;
public:
    Complex(double r = 0, double i = 0);

    Complex operator+(Complex c2);

    Complex operator-(Complex c2);

    double getReal() const;

    double getImag() const;

    Complex operator++();

    Complex operator++(int);

    Complex operator=(Complex c2);
};


#endif //NEWFAT_COMPLEX_H
```

2.
```
//
// Created by zhang on 2020/9/1.
//

#include "Complex.h"

Complex::Complex(double r, double i) : real(r), imag(i) {}

Complex Complex::operator+(Complex c2) {
    real += c2.real;
    imag += c2.imag;
```

```
    return {real, imag};
}

Complex Complex::operator-(Complex c2) {
    real -= c2.real;
    imag -= c2.imag;
    return {real, imag};
}

double Complex::getReal() const {
    return real;
}

double Complex::getImag() const {
    return imag;
}

Complex Complex::operator++() {
    double temp_real;
    temp_real = ++real;
    return {temp_real, imag};
}

Complex Complex::operator++(int) {
    double temp_real;
    temp_real = real++;
    return {temp_real, imag};
}

Complex Complex::operator=(Complex c2) {
    real = c2.real;
    imag = c2.imag;
    return {real, imag};
}
```

3.

```
#include<iostream>
#include<cmath>
#include <iomanip>
#include "Complex.h"

using namespace std;

int main() {
    double weight, length, division;
    cin >> weight >> length;
    division = weight / pow(length, 2);
    cout << fixed << setprecision(1) << division << endl;
    if (division > 25) cout << "PANG" << endl;
    else cout << "Hai Xing" << endl;
    Complex a;
    Complex b(1.1, 2.2);
    Complex c(2.2, 3.3);
    cout << b.getReal() << " " << b.getImag() << endl;
    cout << c.getReal() << " " << c.getImag() << endl;
    cout << a.getReal() << " " << a.getImag() << endl;
    a = c + b;
    cout << a.getReal() << " " << a.getImag() << endl;
```

```
    Complex d;
    d = a++;
    cout << d.getReal() << " " << d.getImag() << endl;
    d = ++a;
    cout << d.getReal() << " " << d.getImag() << endl;
    b = d;
    cout << b.getReal() << " " << b.getImag() << endl;
    return 0;
}
```

4.