C++基础学习笔记

2020年10月8日 13:45

new运算符

在堆中新建一个整型数据 Int * p = new int(10);

释放该数据：delete p；

在堆中新建一个长度为10的整型数组 int * arr = new int[10];

释放该数组：delete[] arr;

内存四区：

代码区：存放函数体的二进制代码，由操作系统管理

全局区：存放全局变量，静态变量以及常量

栈区：由编译器分配释放，存放函数的参数值和局部变量

堆区：由程序猿分配释放，若程序猿不释放，代码结束由操作系统释放

引用：给变量其别名

语法：数据类型 &别名 = 原名；

注意事项：1.引用必须初始化	2.引用初始化后，不可以改变

引用做函数参数

1.值传递	2.地址传递	3.引用传递

Void swap(int a ,int b){
Int temp = a;
a = b;
b = temp;
}
 	
Void main(){
swap(a, b);
}
	Void swap(int * a ,int * b){
	Int temp = *a;
	*a = *b;
	*b = temp;
	}
	
	Void main(){
	swap(&a, &b);
	}
		Void swap(int &a ,int &b){
		Int temp = a;
		a = b;
		b = temp;
		}
		
		Void main(){
		swap(a, b);
		}

main中a,b无改变	main中a.b改变	main中a，b改变

引用做函数返回值

1.不要返回局部变量的引用

2.函数的调用可以作为左值

1的案例	2的案例

Int& test (){
Int a = 10;
Return a;
}
Void main(){
Int &ref = test();
cout << ref << endl;
//第一次结果正确，因为编译器做了保留
cout << ref<< endl;
//第二次结果错误，因为a的内存已经释放
}
	Int& test (){
	Int a = 10;
	Return a;
	}
	Void main(){
	Int &ref = test();
	Test() = 1000;
	cout <<ref<< endl;//此时ref为1000
	}

引用的本质是指针常量

Int &ref = a;//自动转换为 int * const ref = &a;

ref = 20；//内部发现ref是引用，自动哥帮我们转换为 *ref = 20;

函数的重载

1.必须在同一个作用域下

2.函数名称相同

3.函数参数类型不同，或者个数不同，或者顺序不同

封装的访问权限

公共权限 public	成员 类内可以访问 类外可以访问
保护权限 protected	成员 类内可以访问 类外不可以访问 子类可以父类访问
私有权限 private	成员 类内可以访问 类外不可以访问 子类不可以访问私有

struct与class区别：struct默认是公共权限，class默认是私有权限

对象的初始化和清理

1.构造函数 进行初始化操作　类名() {}

没有返回值 不用谢void

函数名与类名相同

构造函数可以有参数，可以发生重载

创建对象的时候，构造函数会自动调用，而且只调用一次

2.析构函数 进行清理操作,释放内存　~类名() {}

没有返回值 不用写void

函数名与类名相同，在名称钱加~

析构函数不可以有参数，不可以发生重载

对象在销毁前 会自动调用析构函数 且只会调用一次

构造函数的分类和调用

按参数分为：有参构造和无参构造

按类型分为：普通构造和拷贝构造

```
拷贝构造函数
Person (const Person &p)
{
}
```

三种调用方式：括号法，显式法，隐式转换法

1.括号法

```
Person p1; //默认构造
Person p2(10);//有参构造
Person p3(p2);//拷贝构造
```

注意：调用默认构造函数，不要加()，因为编译器会误判为函数声明

2.显式法

```
Person p1;
Person p2 = Person(10);
Person p3 = Person(p2);
```

注意：不要利用拷贝构造函数 初始化匿名对象，编译器会认为 Person (p3) 等价 Person p3，编译器对象会重定义

3.隐式法

```
Person p4 = 10;
Person p5 = p4;//多个参数时：Person p6 = (1,2,3,p3);
```

深拷贝和浅拷贝

浅拷贝：简单的赋值拷贝操作　问题：浅拷贝带来的问题是堆区的内存重复释放

```
Person(const Person &p)
{
    m_Age = p.m_Age;//
    m_Height = p.mHeight;// 这是一个指针类型的数据，指向堆区
}
```

深拷贝：在堆区重新申请空间，进行拷贝操作

```
Person(const Person &p)
{
    m_Age = p.m_Age;//
    m_Height = new int(*p.m_Height);
}
```

初始化列表

作用：用来初始化属性

语法：构造函数(): 属性1(值1),属性2(值2)...{}

```
Person() :  m_A(10),  m_B(20), m_C(30)
{
}
```

或者

```
Person(int a, int b, int c) :  m_A(a), m_B(b), m_C(c)
{
}
```

静态成员

静态成员变量

所有对象共享一份数据

在编译阶段分配内存

类内声明，类外初始化

静态成员函数

所有对象共享一个函数
静态成员函数只能访问静态成员变量
都可以通过对象、类名进行访问

类的内存中只有非静态变量，其他的属性和函数分开存储

this指针指向被调用的成员函数所属的对象
this指针式隐含每一个非静态成员函数内的一种指针
作用：解决名称冲突；返回对象本身用*this；

const修饰成员函数
常函数：
成员函数后加const后我们称这个函数为常函数
常函数内不可以修改成员属性
成员属性声明时加关键字mutable后，在常函数中依然可以修改
常对象
声明对象前加const称该对象为常对象
常对象只能调用常函数

友元
全局函数作为友元，可以访问类的私有内容
friend 函数（）；
类作为友元，可以访问类内的私有内容
friend class 类名（）；
成员函数作为友元，可以访问类内的私有内容
friend 类名::函数名（）；

运算符重载
加号运算符重载
作用：实现两个自定义类型数据相加的运算
1.成员函数重载+号

```
Class Person{
 Person& operator+(Person &p)
  {
      Person temp;
      temp.m_A = this->m_A + p.m_A;
      temp.m_B = this->m_B + p.m_B;
      Return temp;
  }
 int m_A;
 int m_B;
}
Person p3 = p1 + p2;
```

本质:Person p3 = p1.operator+(p2);
2.全局函数重载+号

```
Person& operator+(Person &p1,Person &p2)
{
      Person temp;
      temp.m_A = p1.m_A + p2.m_A;
      temp.m_B = p1.m_B + p2.m_B;
      return temp;
}
Person p3 = p1 + p2;
```

本质:Person p3 = operator+(p1,p2);
左移运算符重载
因为成员函数不能实现cout >> p的写法，所以只能用全局函数重载写法

```
Ostream& operator<<(ostream &cout,Person &p)
{
      Cout << "m_A" << p.m_A << "m_B" << p.m_B;
      Return cout;
}
```

递增运算符重载
重载前置++运算符，一定要返回引用

```
Myinteger& operator++()
{
      m_Num++;
      Return *this;
```

```
}
重载后置++运算符,一定要返回值
Myinteger operator++(int) //括号中插入int表示后缀
{
    MyInteger temp = *this;
    m_Num++;
    Return temp;
}
```

赋值运算符的重载

```
Person& operator=(Person& p){
    //应该判断是否有属性在堆区，如果有就先释放干净，然后再深拷贝
    If (m_Age != NULL )
    {
        Delete m_Age;
        m_Age = null;
    }
    m_Age = new int(* p.m_Age);
    Return *this;
}
```

关系运算符重载 == , != , >,<

```
Bool operator==(Person& p)
{
    If (m_Name == p.m_Name)&&(m_Age == p.m_Age)){
    Return true}
    Else return false;
}
```

函数调用运算符重载 ()的重载，也可以称为仿函数，没有固定写法，非常灵活

```
Class Myprint {
    Void operator()(string text)
    {
        cout << text << endl;
    }
}
```

继承

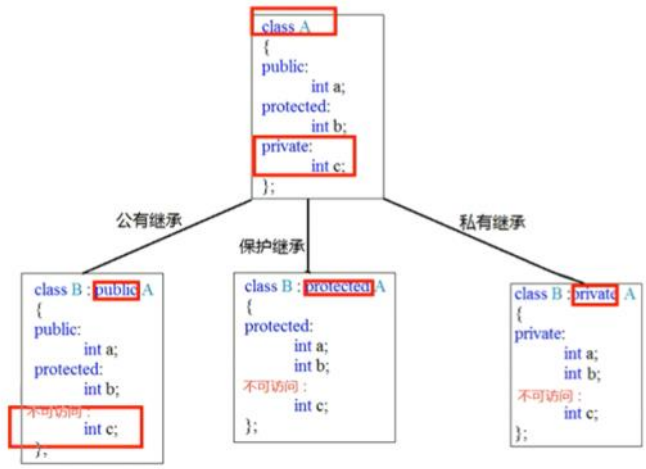
写法

```
Class BasePage{}

Class java : public BasePage{

}
```

继承方式：公共、保护、私有



父类中所有非静态成员属性都会被子类继承下去，父类中私有成员确实被继承了只是访问不到

利用开发人员命令提示工具查看对象模型，跳转到文件路径下 使用 cl /d1 reportSingleClassLayout类名 文件名

子类继承父类构造析构顺序：构造父类、构造子类、析构子类、析构父类

子类同名成员属性、函数访问：直接访问为访问子类的(s.m_A) ，访问父类需要加作用域(s.Base::m_A)

注意：当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数

继承同名静态成员处理方式：同上，访问子类的直接访问，访问父类需要加作用域

使用对象访问s.m_A和s.Base::m_A，成员函数一样	使用类名访问Son::m_A和Son::Base::m_A，成员函数一样

多继承语法

语法：class 子类： 继承方式 父类1，继承方式 父类2...

多继承引发父类中有同名成员出现，需要加作用域区分。实际开发不建议用多继承

菱形继承，两个父类拥有相同数据，需要加作用域区分。如果同一数据只需保留一份，使用虚继承，父类成为虚基类

```
Class Animal{
Public:
    Int m_Age;
}
Class Sheep : virtual public Animal {};
Class Tuo : virtual public Animal {};
Class Sheeptuo : publi sheep,public tuo{}
此时SheepTuo的m_Age只有一份，其原理为继承了指针
```

多态

多态分为两类

静态多态：函数重载 和 运算符重载属于静态多态，复用函数名

动态多态：派生类和虚函数实现运行时多态

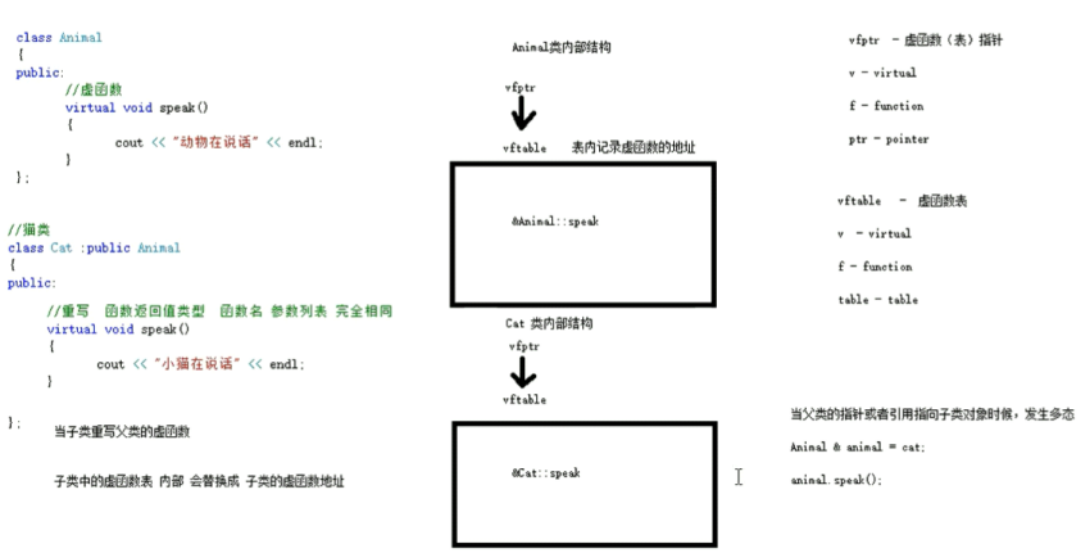
静态多态和动态多态区别

静态多态的函数地址早绑定 - 编译阶段确定函数地址

动态多态的函数地址晚绑定 - 运行阶段确定函数地址

Class Animal {
Public:
Void speak(){
输出animal说话
}
}
Class cat : public animal {
Public
Void speak(){
输出猫在说话
}
}
Void test(Animal &animal){
Animal.speak();
}
	Class Animal {
	Public:
	Void virtual speak(){
	输出animal说话
	}
	}
	Class cat : public animal {
	Public
	Void speak(){
	输出猫在说话
	}
	}
	Void test(Animal &animal){
	Animal.speak();
	}

输出animal在说话 这是地址早绑定，编译时绑定	输出猫在说话 这是地址晚绑定，在运行时绑定
动态多态满足条件 1.有继承关系 2.子类重写父类的虚函数，函数返回值类型，函数名，参数列表完全相同	动态多态的使用 父类的指针或者引用执行子类对象



纯虚函数和抽象类

纯虚函数语法：virtual 返回值类型 函数名（参数列表） = 0;
当类中有了纯虚函数，这个类也称为抽象类
抽象类特点：
无法实例化对象
子类必须重写抽象类中的纯虚函数，否则也属于抽象类

虚析构和纯虚析构
多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码，出现内存泄漏
解决方法：将父类中的析构函数改为虚析构或者纯虚析构
虚析构和纯虚析构共性：
都可以解决父类指针释放子类对象
都需要有具体的函数实现
虚析构和纯虚析构区别
如果是纯虚析构，该类属于抽象类，无法实例化对象
虚析构语法：virtual ~类名(){}
纯虚析构语法：virtual ~类名() = 0;
类名::~类名(){}

文件操作
C++中对文件操作需要包含头文件fstream
文件类型分为两种
文本文件和二进制文件
操作文件的三大类
1.ofstream：写操作
2.ifstream：读操作
3.fstream：读写操作
文本文件
写文件
1.包含头文件
#include<fstream>
2.创建流对象
Ofstream ofs;
3.打开文件
ofs.open
4.写数据
Ofs << "写入数据";
5.关闭文件
Ofs.close();

文件打开方式	解释
los::in	为读文件而打开文件
los:out	为写文件而打开文件
los:ate	初始位置：文件尾
los::app	追加方式写文件
los::trunc	如果文件存在先删除，再创建
los::binary	二进制方式

注意：文件打开方式可以配合使用，利用|操作符
例如：用二进制方式写文件 ios::binary | ios::out

读文件
1.包含头文件 #include <fstream>
2.创建刘对象 ifstream ifs;
3.打开文件并判断是否打开成功
Ifs.open("路径"，打开方式);
4.读数据
四种方式

方式一	方式二	方式三	方式四

char buf[1024] = { 0 };
while (ifs >> buf)
{
cout << buf << endl;
	char buf[1024] = { 0 };
	while (ifs.getline(buf, sizeof(buf))) {
	cout << buf << endl;
	}
		string buf;
		while (getline(ifs, buf)) {
		cout << buf << endl;
		}
			char c;
			while ((c = ifs.get()) != EOF) //eof = end of file
			{
			cout << c;

}			}

5.关闭文件

Ifs.close();

二进制方式写文件

流程一样，写入操作如下

```
Person p = {"张三", 18};
Ofs.write((const char *)&p, sizeof(Person));
```