1. 立方体案例
2. 点和圆关系案例
   1. 圆内的属性里有个其它的自定义数据类型 Point
   2. 三种关系判断
   3. 分文件编写
      1. .h中写类的成员函数声明
      2. .cpp中写成员函数实现
3. 对象的初始化和清理
   1. 构造函数
      1. 没有返回值 没有void，类名相同，可以发生重载，可以有参数
      2. 构造函数由编译器自动调用，而不是手动，而且只会调用一次
   2. 析构函数
      1. 没有返回，没有void ，函数名称： ~类名，不可以发生重载，不可以有参数
   3. 系统会默认调用 构造函数和析构函数，而且只会调用一次
   4. 如果程序员没有提供构造和析构，系统会默认提供，空实现
4. 构造函数的分类及调用
   1. 按照参数分类
      1. 无参构造（默认构造） 有参构造
   2. 按照类型分类
      1. 普通构造函数 拷贝构造函数
   3. 无参构造写法 和调用
      1. Person p1 ; 注意不能写 Person p1() ，因为编译器认为这个是函数声明
   4. 有参构造写法和调用
      1. Person p2(10) 或者 Person p2 = Person(10)
      2. Person(10) 匿名对象 ，执行当前行后就会释放这个对象
   5. 拷贝构造函数
      1. Person( const Person & p )
      2. Perons p1( p2) 或者 Person p1 = Person(p2)
      3. 不能用拷贝构造函数初始化匿名对象
         1. 如果写成 Person (p1) 这种写法等价于 Person p1
         2. 写到右值可以做拷贝构造函数Person p8 = p7; // 相当于 Person p8 = Person(p7);
   6. Person P = 100 隐式类型转换 相当于调用 Person p = Person(100)

|  |
| --- |
| void test01()  {  //构造函数调用方式  //括号法调用  //Person p1(1); //有参  //p1.m\_Age = 10;  //Person p2(p1); //拷贝  //cout << "p2的年龄" << p2.m\_Age << endl;  //Person p3; //默认构造函数 不要加() Person p3(); //编译器认为这行是函数的声明  //显示法调用  //Person p4 = Person(100);  //Person p5 = Person(p4);、  //Person(100); //叫匿名对象 ,匿名对象特点，如果编译器发现了对象是匿名的，那么在这行代码结束后就释放这个对象  //不能用拷贝构造函数 初始化匿名对象  //Person p6 = Person(p5); //如果写成左值，编译器认为你写成 Person p5; 对象的声明，如果写成右值，那么可以  Person p7 = 100; //相当于调用了 Person p7 = Person(100) ，隐式类型转换  Person p8 = p7; // 相当于 Person p8 = Person(p7);  } |

1. 拷贝构造函数调用时机
   1. 1、用已经创建好的对象来初始化新的对象
   2. 2、以值传递的方式给函数参数传值
   3. 3、以值方式返回局部对象
   4. release 默认下会做优化
2. 构造函数的调用规则  
   系统默认给一个类提供 3个函数 默认构造 、 拷贝构造 、 析构函数
   1. 如果提供了有参的构造，那么系统就不会提供默认的构造了，但是会提供拷贝构造
   2. 如果提供了拷贝构造函数，那么系统就不会提供其他的构造函数了
3. 深拷贝与浅拷贝
   1. 系统默认提供的拷贝构造 会进行简单的值拷贝
   2. 如果属性里有指向堆区空间的数据，那么简单的浅拷贝会导致重复释放内存的异常
   3. 解决上述问题，需要我们自己提供拷贝构造函数，进行深拷贝

|  |
| --- |
| class Person  {  public:  Person()  {}  //初始化属性  Person(char \* name,int age)  {  m\_Name = (char\*)malloc(strlen(name) + 1);  strcpy(m\_Name, name);  m\_age = age;  }  //拷贝构造 系统会提供默认拷贝构造，而且是简单的值拷贝  //自己提供拷贝构造，原因简单的浅拷贝会释放堆区空间两次，导致挂掉  //深拷贝  Person(const Person&p)  {  m\_age = p.m\_age;  m\_Name = (char\*)malloc(strlen(p.m\_Name) + 1);  strcpy(m\_Name, p.m\_Name);  }  ~Person()  {  cout << "析构函数调用" << endl;  if (m\_Name!=NULL)  {  free(m\_Name);  m\_Name = NULL;  }  }  //姓名  char \* m\_Name;  //年龄  int m\_age;  }; |

1. 初始化列表语法
   1. 在构造函数后面 + : 属性(值、参数), 属性（值、参数）…

|  |
| --- |
| Person() :m\_A(10), m\_B(20), m\_C(30)  {}  //利用初始化列表 初始化数据  // 构造函数后面 + : 属性（参数）, 属性（参数）...  Person(int a, int b, int c) : m\_A(a), m\_B(b), m\_C(c)  {}  int m\_A;  int m\_B;  int m\_C; |

1. 类对象作为成员的案例
   1. 当类对象作为类的成员时候，构造顺序是先构造类对象的构造，然后构造自己，
   2. 析构顺序与构造相反
2. explicit关键字
   1. 作用：防止构造函数中的隐式类型转换

|  |
| --- |
| class MyString  {  public:  MyString(const char \* str)  {  //  }  explicit MyString(int a)  {  mSize = a;  }  char\* mStr;  int mSize;  };  void test01()  {  MyString str = "abc";  MyString str2(10);  //MyString str3 = 10; //做什么用图？ str2字符串为 "10" 字符串的长度10  //隐式类型转换 Mystring str3 = Mystring (10);  // explicit关键字 ，防止隐式类型转换  } |

1. new 运算符 和 delete运算符
   1. Person \* p = new Person 会返回一个Person指针
   2. 默认调用构造函数，开辟空间，返回不是void\* ，不需要强制转换
   3. delete释放

|  |
| --- |
| void test01()  {  //Person p1; 栈区开辟  Person \* p2 = new Person; //堆区开辟  //所有new出来的对象 都会返回该类型的指针  //malloc 返回 void\* 还要强转  //malloc会调用构造吗？ 不会  // new会调用构造  // new 是运算符 ； malloc是函数  //释放 堆区空间  // delete也是运算符 配合 new用 malloc 配合 free用  delete p2;  } |

* 1. new 对象 用void\* 去接受，释放不了对象

|  |
| --- |
| void test02()  {  void \*p = new Person(10);  //当用void\* 接受new出来的指针 ，会出现释放的问题  delete p;  //无法释放p ，所以避免这种写法  } |

* 1. new出来的是数组 ，如何释放？ delete [] …
  2. new出来的是数组，肯定会调用默认构造

|  |
| --- |
| void test03()  {  //通过new开辟数组 一定会调用默认构造函数,所以一定要提供默认构造  Person \* pArray = new Person[10];  //Person pArray2[2] = { Person(1), Person(2) }; //在栈上开辟数组，可以指定有参构造  //释放数组 delete []  delete [] pArray;  } |