第5章 面向对象

In [1]:
```
#include <iostream>
#include <cstdio>
using namespace std;
```

注意，由于这里是交互式环境，成员变量的定义必须写在类里面，不能拿出来写，否则报错

构造函数和析构函数
构造函数负责对象的初始化，析构函数负责对象消亡时释放动态申请的内存空间，
由于动态申请的内存空间不在函数栈当中，因此需要析构函数手动释放，这个析构函数
会在函数退栈时自动调用函数栈中所有对象的析构函数，另外，普通成员变量不需要
析构函数释放空间，因为这些变量就在函数栈中，会随着函数退栈自动释放。

In [2]:
```
class myClass{
    public:
        int *data;
        int size;
        myClass(int s) {
            size = s;
            data = new int[size];
            cout << sizeof(size) << endl << sizeof(data) << endl << sizeof(this) << endl;
        }
        ~myClass() {delete [] data;}
};
```

In [3]:
```
myClass c1 = myClass(10);  // data是一个指针，只占4个字节，size占4个字节
```

```
4
8
8
```

上面data指针和size整数存放在函数栈中，退栈时自动释放，但是data所指向的内存在其他地方，
这部分内存空间在退栈时会自动调用上面的析构函数释放空间，如果没有上面的析构函数，在整个
程序运行期间，那个地方的存储空间会被一直占用。

对象的初始化
除了上面显示调用构造函数，还有下面两个方法

In [4]:
```
myClass c1 = 10;
```

```
4
8
8
```

In [5]:
```
myClass c1(10);
```

```
4
8
8
```

其实这3种形式也适用于普通类型变量

In [6]:
```
int a = int(1);
cout << a << endl;
int b = 2;
cout << b << endl;
int c(3);
cout << c << endl;
```

```
1
2
3
```

成员变量的作用域和访问
成员变量的作用域就是类的定义的那个大花括号中，里面的函数随便访问，
如果想通过对象访问，使用 对象.成员变量 的形式，但是要求这个
成员变量必须在public中，在private中定义的成员变量不能通过点访问

In [7]:
```
cout << c1.size;
```

```
10
```

关于this指针
this指针其实就相当于python里面的self，就是这个实例自己，
早期c++程序没有自己的编译器，都是先转化成c程序，再进行编译，
this指针就是转换成c程序的时候用到的

定义一个对象，包含成员变量和成员函数

In [8]:
```
class Redminote12{
    public:
        float price;
        void setPrice(float a){price = a;}
}
```

In [9]:
```
Redminote12 note1;
```

In [10]:
```
note1.setPrice(899);
```

In [11]:
```
cout << note1.price;
```

```
899
```

将上面程序转换为c程序
注意this是c++关键字，下面我用this_替代

In [12]:
```
struct redminote12{
    float price;
};
```

In [13]:
```
void setPrice(redminote12 *this_,float a){
    this_ -> price = a;
}
```

In [14]:
```
redminote12 note2;
```

In [15]:
```
setPrice(&note2, 899)
```

In [16]:
```
cout << note2.price;
```

```
899
```

也可以将上面程序转换为Python程序

```
class Redminote12:
    def __init__(self):
        self.price = None
    def setPrice(self,price):
        self.price = price
```

运算符重载
注意，Xeus-Cling目前为止(2024.03.01)，运算符重载必须放在类中定义，
拆开写会引发编译器报错"function definition is not allowed here"
其他普通成员函数可以在类外面定义

In [17]:
```
class Complex{
    public:
        float real,imag;
        Complex(float r=0, float i=0){real = r; imag = i;}
        Complex operator+(Complex &c2){
            return Complex(real+c2.real,imag+c2.imag);
        }
};
```

In [18]:
```
Complex c1 = Complex(2,3);
Complex c2 = Complex(4,6);
```

In [19]:
```
Complex c3;  // 注意，上面构造函数有默认参数，所以这里初始化为默认参数，
             // 如果构造函数没有默认参数，那么必须提供参数进行初始化
```

In [20]:
```
c3 = c1 + c2;
```

In [21]:
```
cout << c3.real << " " << c3.imag;
```

```
6 9
```

In [22]:
```
c3 = c1.operator+(c2);              // 这里也可以使用一般成员函数的调用方法
cout << c3.real << " " << c3.imag;
```

```
6 9
```

c++对结构体的补充
在c++中，对标准c中的结构体进行了一些扩充，使得结构体中也能定义成员变量

In [23]:
```
struct Complex2{
    float real,imag;
    Complex2(float r=0, float i=0){real = r; imag = i;}
```

```
    void func(){cout << "hello world";}
};
```

In [24]:
```
Complex2 c4 = Complex2(3,9);
c4.func();
```

```
hello world
```

继承与派生

有3种继承方式，这里主要讲的是公有继承的继承规则

假定A是基类，B继承了A，即B是派生类，那么对于A的成员变量和成员函数：
private：复制到B中的special当中，只能调用父类的成员变量进行访问；
protected：可以看成直接拷贝进B的protected中，内部成员函数随便访问，对外不可见；如果变量名重复，加个前缀A::再复制过来；
public：相当于直接拷贝过来,对内对外均可以访问，对外访问需要用对象点的形式访问；如果变量名重复，加个前缀A::再复制过来；

注意：
1.上面的special是我自定义的，其实不存在，只是为了形象说明加的
2.不管什么类型，都会继承，都占用空间

In [25]:
```
class A{
    private:
        int a;
    protected:
        int c;
    public:
        int d;
        A(int s){ a = s;}
};
```

In [26]:
```
class B: public A{
    public:
        int d;
        B(int x, int y, int z, int k):A(x){        // 调用父类的构造函数初始化父类的私有变量
            c = z;                                 // A的保护变量可以直接访问
            d = k;                                 // A的公有变量也可以直接访问
            A::d = 100;
        }
};
```

In [27]:
```
auto x = B(1,2,3,4);
```

In [28]:
```
sizeof(x)
```

Out[28]:
```
16
```

逐个分析上面A中变量的继承： 1.对于私有变量a，复制到B中的special，只能调用A的构造函数进行初始化；
另外，由于构造函数在public中，因此会直接复制到B的public中，因此可以直接调用；
2.调用父类的构造函数时，必须在构造函数初始化列表的地方使用，不能在函数体中使用；
3.对于保护变量c，直接复制到B的保护变量中；
4.对于公有变量d，由于B已经有d了，所以变成A::d再复制到public中；
5.对于公有函数A()，直接复制过来

由于A中3个成员变量复制过来，加上B的一个变量，总共4个int型变量，占16个字节。

派生类数据视图：
special：把私有变量拷贝进来，只允许父类成员函数访问
private：不改变
protected：父类protect直接拷贝进来（遇见同名，加个前缀）
public： 父类public变量直接拷贝进来（遇见同名，加个前缀）

两阶段命名查找

上述就是普通类的继承，但是如果涉及到模板类的继承，在gcc编译器中，派生类访问基类的protected和public中的成员变量和成员函数时，必须使用this指针，因为编译器使用了两阶段命名查找：
第一阶段：仅处理独立的变量，也就是那些不依赖于模板参数T的变量，此时编译器会自动忽略带模板参数T的变量，以及用this指针指向的变量，先处理没有这些标志的变量，如果这个变量是从父类继承的，并且没有添加this指针，编译器会找不到这个变量的声明；
第二阶段：实例化阶段，此时模板参数变成了具体类型，此时就可以处理刚刚忽略的变量
参考链接：
https://blog.csdn.net/mkxding/article/details/119485034
https://blog.csdn.net/gettogetto/article/details/52955741

一个更方便的方法：
使用using在子类public中声明父类中需要用到的成员变量和成员函数即可，
而不必每次都是用A::x 或者 this -> x 这种形式

In [29]:
```
template <class T>
class A{
    protected:
        T b;
    public:
        T c;
        A(T x, T y){ b = x; c = y;}
};
```

In [30]:
```
template <class T>
class B: public A<T>{
    public:
        using A<T>::b;
        using A<T>::c;
        B(T x, T y):A<T>(x,y){}  // 构造函数必须重写

        // 注意，下面直接使用b和c，不会编译报错，就是因为上面使用了using，否则编译会报错
        void show(){
            cout << "b = " << b << endl;
            cout << "c = " << c << endl;
        }
};
```

In [31]:
```
auto b = B<int>(1,2);
b.show()
```

```
b = 1
c = 2
```

In [32]:
```
// 实例化之后，实例的属性就是按照前面说的copy的标准，继承下来的属性和自己的属性没什么不同
cout << b.b << b.c;
```

```
12
```

__虚方法__

如果在成员函数声明的前面加上关键字virtual，后面加个 = 0，则这个方法就是纯虚方法，需要派生类重写此方法，注意，此时父类的这个函数将会被覆盖，而不是前面说的加个类前缀改个名copy下来

In [ ]: