CUC 春季学期C++整理6.0

一.封闭类

一个类的成员变量如果是另一个类的对象，就称之为“成员对象”。包含成员对象的类叫封闭类（enclosed class）。
当封闭类的对象生成并初始化时，它包含的成员对象也需要被初始化，这就会引发成员对象构造函数的调用。如何让编译器知道，**成员对象到底是用哪个构造函数初始化**的呢？这可以通过在定义封闭类的构造函数时，**添加初始化列表的方式解决。**

在构造函数中添加初始化列表的写法如下：

```
类名::构造函数名(参数表): 成员变量1(参数表), 成员变量2(参数表), ...
{
    ...
}
```

`:`和`{`之间的部分就是初始化列表。**初始化列表中的成员变量既可以是成员对象，也可以是基本类型的成员变量。**对于成员对象，初始化列表的“参数表”中**存放的是构造函数的参数**（它指明了该成员对象如何初始化）。对于基本类型成员变量，“参数表”中就是一个初始值。

“参数表”中的参数可以是任何有定义的表达式，该表达式中可以包括变量甚至函数调用等，只要表达式中的标识符都是有定义的即可。例如：

```
#include <iostream>
using namespace std;
class CTyre //轮胎类
{
    private:
        int radius; //半径
        int width; //宽度
    public:
        CTyre(int r, int w) : radius(r), width(w) { }
};
class CEngine //引擎类
{

};
class CCar { //汽车类
    private:
        int price; //价格
        CTyre tyre;//该成员变量就是另一个类的对象
        CEngine engine;
    public:
        CCar(int p, int tr, int tw);
};
CCar::CCar(int p, int tr, int tw) : price(p), tyre(tr, tw)
{

};
int main()
{
    CCar car(20000, 17, 225);
```

```
30      return 0;
31  }
```

注： 第 9 行的构造函数添加了初始化列表，将 radius 初始化成 r，width 初始化成 w。这种写法比在函数体内用 r 和 w 对 radius 和 width 进行赋值的风格更好。建议对成员变量的初始化都使用这种写法。

`CCar` 是一个封闭类，有两个成员对象： `tyre` 和`engine`。在编译第 27 行时，编译器需要知道 car 对象中的 tyre 和 engine 成员对象该如何初始化。

编评器已经知道这里的 car 对象是用上面的`CCar(int p, int tr, int tw)`构造函数初始化的，那么 tyre 和 engine 该如何初始化，就要看第 22 行`CCar(int p,int tr,int tw)`后面的初始化列表了。该初始化列表表明，tyre 应以`tr`和`tw`作为参数调用`CTyre(intr, hit w)`构造函数初始化，但是并没有说明 engine 该如何处理。在这种情况下，编译器就认为 engine 应该用`CEngine`类的无参构造函数初始化。而`CEngine`类确实有一个编译器自动生成的默认无参构造函数，因此，整个 car 对象的初始化问题就都解决了。

总之，生成封闭类对象的语句一定要让编译器能够弄明白其成员对象是如何初始化的，否则就会编译错误。

在上面的程序中，如果`CCar`类的构造函数没有初始化列表，那么第 27 行就会编译出错，因为编译器不知道该如何初始化`car.tyre`对象，**因为`CTyre`类没有无参构造函数**，而编译器又找不到用来初始化`car.tyre`对象的参数。

封闭类对象生成时，**先执行所有成员对象的构造函数，然后才执行封闭类自己的构造函数。成员对象构造函数的执行次序和成员对象在类定义中的次序一致，与它们在构造函数初始化列表中出现的次序无关。(这很重要！！！)**

当封闭类对象消亡时，**先执行封闭类的析构函数，然后再执行成员对象的析构函数，成员对象析构函数的执行次序和构造函数的执行次序相反，即先构造的后析构**，这是 C++ 处理此类次序问题的一般规律。

再来举个例子：

```
1   #include<iostream>
2   using namespace std;
3   class CTyre {
4       public:
5           CTyre() { cout << "CTyre constructor" << endl; }
6           ~CTyre() { cout << "CTyre destructor" << endl; }
7   };
8   class CEngine {
9       public:
10          CEngine() { cout << "CEngine constructor" << endl; }
11          ~CEngine() { cout << "CEngine destructor" << endl; }
12  };
13  class CCar {
14      private:
15          CEngine engine;
16          CTyre tyre;
17      public:
18          CCar() { cout << "CCar constructor" << endl; }
19          ~CCar() { cout << "CCar destructor" << endl; }
20  };
21  int main()
22  {
23      CCar car;
24      return 0;
```

```
25  }
```

运行结果：

```
CEngine constructor
CTyre constructor
CCar constructor
CCar destructor
CTyre destructor
CEngine destructor
```

~~有兴趣可以去看：[https://blog.csdn.net/dxpqxb/article/details/102650836?ops_request_misc=%257B%2522request%255Fid%2522%253A%252215868444081972524761249 2%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=158684440819725247612492&biz_id=0&utm_source=distribute.pc_search_result.none-task-blog-soetl_so_first_rank_v2_rank_v25-4]~~

二.常对象

可以在定义对象时加关键字`const`，指定对象为常对象。**常对象必须有初值**

```
Time const t1(12,23,24);
```

这样，在t1的生命周期中，对象t1的所有数据成员的值都不能被修改。

定义常对象的一般形式：

`类名 const 对象名{（实参表）}`

也可以把`const`写在最左边，二者等价，在定义常对象时必须同时对他初始化，之后不能改变。

说明：

- **常对象只能调用它的常成员函数，不能调用普通成员函数**（系统自动调用的析构函数和构造函数除外！）如果一定要引用常对象中的数据成员，只需要将该成员函数声明为`const`即可
- 常函数成员可以访问常对象中的数据成员，但不允许修改它的值。

三.常对象成员

我们可以将对象的成员声明为`const`，包括常数据成员和常成元函数。

1.常数据成员

其用法和一般变量类似，用关键字`const`来声明数据成员。常数据成员的值无法改变。但需要注意：**只能通过构造函数的初始化列表对常数据成员进行初始化，任何其他函数都不能对常数成员赋值。**

比如在类体中定义了常数据成员`hour`

```
Time :: Time (int h)
{
    hour = h;
}
```

这样的通过赋值初始化的方法是错误的！

```
Time :: Time (int h) : hour(h){}
```

这样是对的。

2.常函数成员

如果我们将成员函数声明为常函数成员，**则只能引用本类中的数据成员但不能修改它**

声明常函数成员的一般形式：

`类型名 函数名（参数表） const`

`const`是函数类型的一部分，在声明和定义函数时都要加，调用时不必。常成员函数可以引用非常数据成员，也可以调用常数据成员。具体关系有下表：

数据成员	非`const`的普通成员函数	`const`成员函数
非`const`的普通数据成员	可以引用，也可以修改值	可以引用，不能改变值
`const`数据成员	可以引用但不能改变值	可以引用，不能改变值
`const`对象	不允许	可以引用，不能改变值

四.友元函数

1.友元函数：

如果我们在本类的其他地方定义了一个函数（这个函数可以是不属于任何类的非成员函数，也可以是其他类的成员函数），在类体中用`friend`对其进行声明，此函数就称为本类的**友元函数。友元函数可以访问这个类中的私有成员。**

```
#include<bits/stdc++.h>
using namespace std;
class Time
{
    public:
        friend void display(Time &);
        Time(int,int,int);
    private:
        int hour;
        int minute;
        int sec;
};
Time :: Time(int h,int m,int s)
{
    hour = h;
    minute = m;
    sec = s;
}
void display(Time &t)
{
    cout << t.hour << ":" << t.minute << ":" << t.sec << endl; //由于我们声明了display是类的友元函数所以display可以访问到类中的私有成员但是必须加上对象名，因为display不含this指针
```

```
}
int main()
{
    Time t1(10,13,56);
    display(t1);
    return 0;
}
```

输出结果：

```
10：13：56
```

2.友元成员函数：

`friend`函数不仅可以是一般函数，也可以是另一个类中的成员函数。

```
#include<bits/stdc++.h>
using namespace std;
class Time
{
    public:
        void display(Date &);
        Time(int,int,int);
    private:
        int hour;
        int minute;
        int sec;
};
class Date
{
    public:
        Date(int ,int ,int );
        friend void Time :: display(Date &);
    private:
        int month;
        int day;
        int year;
};
Time :: Time(int h,int m,int s)
{
    hour = h;
    minute = m;
    sec = s;
}
void Time :: display(Date &d)
{
    cout << d.hour << ":" << d.minute << ":" << d.sec << endl;
    cout << hour << '/' << minute << '/' << sec << endl;
}
Date :: Date(int m,int d,int y)
{
    month = m;
    day = d;
    year = y;
}
int main()
```

```
{
    Time t1(10,13,56);
    Date d1(12,25,2004);
    t1.display(d1);//调用t1中的display函数，实参是Date类的对象d1
    return 0;
}
```

运行结果：

```
10：13：56
12/25/2004
```

注意：

- **函数名`display`的前面要加上其所在对象名。**
- **`display`成员函数的实参是Date类对象d1，否则就无法访问d1中的数据。**
- **在`Time::display`函数中引用Date类私有数据时必须加上对象名。**

五.友元类

声明友元类的一般形式：

`friend 类名`

说明：

- **友元的关系是单向的，而不是双向的。**
- **友元的关系不能传递**

~~小黄鸭Debug保命~~


trulia