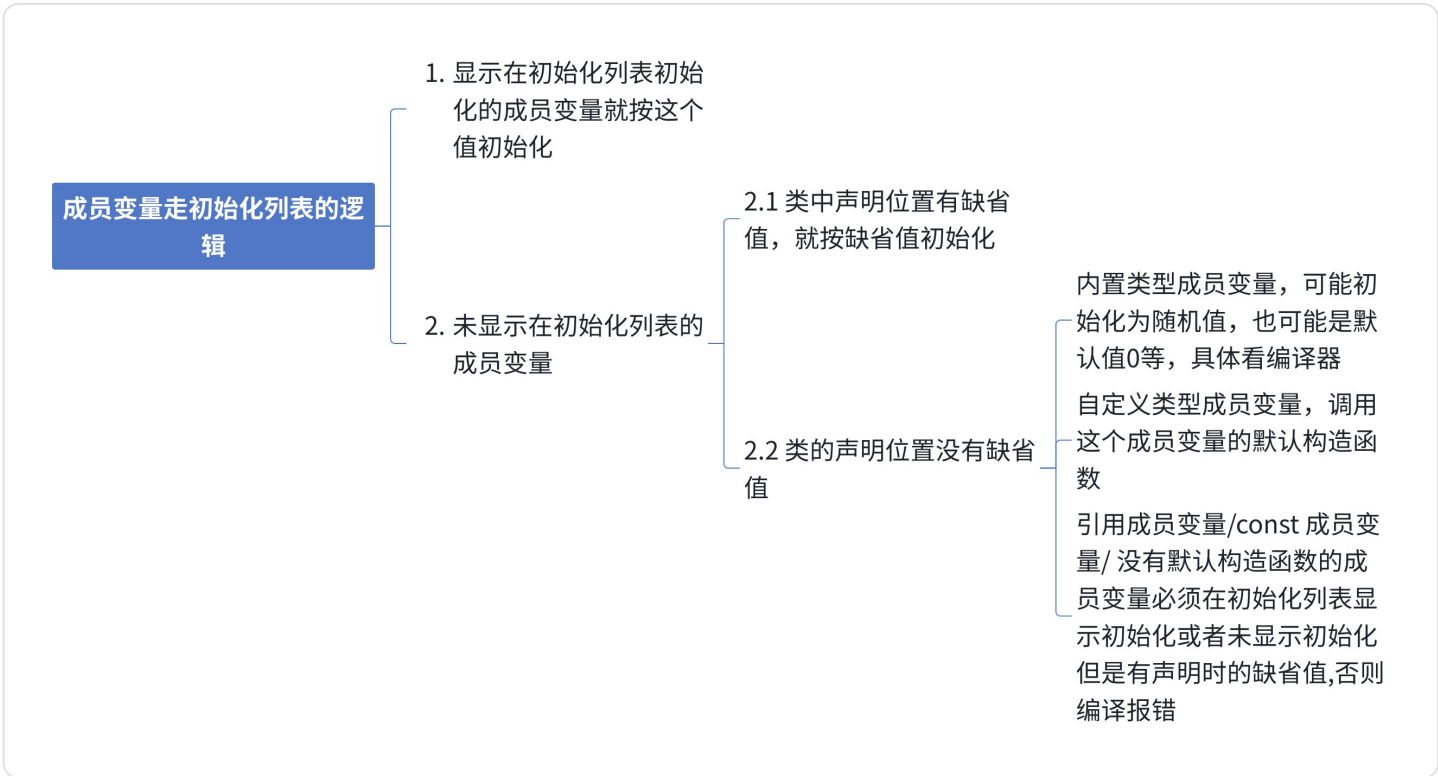
类和对象复习(下)

1. 再探构造函数

1. 构造函数初始化成员变量不仅可以在构造函数内进行初始化，还可以通过初始化列表。
2. 每个成员变量在初始化列表里只能出现一次。
3. 引用成员变量，const成员变量，没有默认构造的类类型变量，必须放在初始化列表位置进行初始化，否则会编译报错。
4. C++11中支持在成员变量初始化位置给缺省值，主要是给那些没有在初始化列表中的成员变量使用的。
5. 初始化列表中初始化顺序由成员变量声明顺序决定，但是最好以其声明顺序初始化，增强代码可读性。
6. 如果你没有给缺省值，对于没 有显示在初始化列表初始化的内置类型成员是否初始化取决于编译器，C++并没有规定。对于没有 显示在初始化列表初始化的自定义类型成员会调用这个成员类型的默认构造函数，如果没有默认构造会编译错误。

初始化列表总结：



下面程序的运行结果是:

代码块

```cpp

#include<iostream>
using namespace std;
class A
{
public:
    A(int a)
        :_a1(a)
        , _a2(_a1)
    {}

    void Print()
    {
        cout << _a1 << " " << _a2 << endl;
    }

private:
    int _a2 = 2;
    int _a1 = 2;
};

int main()
{
    A aa(1);
    aa.Print();
}

```

1和随机值,定义顺序不由初始化列表中决定，而由声明顺序决定。

2. 类型转换

· C++支持内置类型隐式类型转换为类类型对象，需要有相关内置类型为参数的构造函数。

· 构造函数前面加explicit就不再支持隐式类型转换。

· 类类型的对象之间也可以隐式转换，需要相应的构造函数支持。

代码块

```cpp
#include<iostream>
using namespace std;
class A
{
public:
    // 构造函数explicit就不再支持隐式类型转换
    // explicit A(int a1)
```

```
8       A(int a1)
9           :_a1(a1)
10      {
11      }
12
13      //explicit A(int a1, int a2)
14      A(int a1, int a2)
15          :_a1(a1)
16          , _a2(a2)
17      {
18      }
19
20      void Print()
21      {
22          cout << _a1 << " " << _a2 << endl;
23      }
24
25      int Get() const
26      {
27          return _a1 + _a2;
28      }
29
30  private:
31      int _a1 = 1;
32      int _a2 = 2;
33  };
34
35  class B
36  {
37  public:
38      B(const A& a)
39          :_b(a.Get())
40      {
41      }
42
43  private:
44      int _b = 0;
45  };
46
47  int main()
48  {
49      // 1构造一个A的临时对象，再用这个临时对象拷贝构造aa3
50      // 编译器遇到连续构造+拷贝构造->优化为直接构造
51      A aa1 = 1;
52      aa1.Print();
53      const A& aa2 = 1;
54
```

```
55        // C++11之后才支持多参数转化
56        A aa3 = { 2,2 };
57
58        // aa3隐式类型转换为b对象
59        // 原理跟上面类似
60        B b = aa3;
61        const B& rb = aa3;
62
63        return 0;
64    }
```

3. static成员

1. 用static修饰的成员变量，叫做静态成员变量，需要在类外初始化。
2. 静态成员变量存放在静态区，**为所有类对象共享**，不属于某个具体的对象。
3. **用static修饰的成员函数叫做静态成员函数，没有this指针。**
4. 静态成员函数可以访问其他静态成员，但是不能访问非静态的，因为没有this指针。
5. 非静态成员函数，可以访问任意的静态成员函数和变量。
6. 突破类域就可以访问静态成员，可以通过类名::静态成员 或者 对象.静态成员 来访问静态成员变量和静态成员函数。
7. 静态成员也受访问限定符的限制。
8. 静态成员变量不能在声明位置给缺省值初始化，因为缺省值是给构造函数初始化列表的，静态成员变量不属于某个类，不走构造函数初始化列表，而在类外初始化。

4. 友元

1. 友元分为：友元函数和友元类，在函数或类声明前加friend，并且把友元声明放在一个类中。
2. 友元函数可以在类的任意位置定义，不受访问限定符的限制。
3. 一个函数可以是多个类的友元函数。
4. 友元类中的成员函数都可以是另一个类的友元函数，都可以访问另一个类中的私有和保护成员。
5. 友元是单向性的。
6. 友元关系没有传递性，不符合交换律。
7. 友元破坏了封装性，不建议使用。

5. 内部类

1. 一个定义在另一个类内部的类。内部类是独立的，它只是多受到外部类的类域限制和访问限定符的限制，所以外部类定义的对象中不包含内部类。
2. 内部类默认为外部类的友元类。
3. 内部类本质也是一种封装，当A类与B类强耦合时，且A的作用就是服务于B时，那么可以考虑把A设计为B的内部类，如果放到private/protected中，那么A就是B的专属内部类，外部无法访问。

代码块

```
#include<iostream>
using namespace std;
class A
{
private:
    static int _k;
    int _h = 1;

public:
    class B // B默认就是A的友元
    {
    public:
        void foo(const A& a)
        {
            cout << _k << endl; //OK
            cout << a._h << endl; //OK
        }
        int _b1;
    };

};

int A::_k = 1;
int main()
{
    cout << sizeof(A) << endl;
    A::B b;
    A aa;
    b.foo(aa);
    return 0;
}
```

6. 匿名对象

1. 用类型(实参) 定义出来的对象叫做匿名对象，相比之前我们定义的类型对象名(实参) 定义出来的叫有名对象。
2. 匿名对象生命周期只在当前一行，一般临时定义一个对象当前用一下即可，就可以定义匿名对象。

代码块

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

class A
{
public:
    A(int a = 0)
        :_a(a)
    {
        cout << "A(int a)" << endl;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }

private:
    int _a;
};

class Solution
{
public:
    int Sum_Solution(int n)
    {
        //...
        return n;
    }
};

int main()
{
    A aa1;
    // 不能这么定义对象，因为编译器无法识别下面是一个函数声明，还是对象定义
    //A aa1();
    //
    // 但是我们可以这么定义匿名对象，匿名对象的特点不用取名字，
    // 但是他的生命周期只有这一行，我们可以看到下一行他就会自动调用析构函数
    A();
    A(1);

```

```
42      A aa2(2);
43      // 匿名对象在这样场景下就很好用，当然还有一些其他使用场景，这个我们以后遇到了再说
44      Solution().Sum_Solution(10);
45
46      return 0;
47  }
```

7. 对象拷贝时的编译器优化

1. 现代编译器会为了尽可能提高程序的效率，在不影响正确性的情况下会尽可能减少一些传参和传返回值的过程中可以省略的拷贝。
2. 如何优化C++标准并没有严格规定，各个编译器会根据情况自行处理。当前主流的相对新一点的编译器对于连续一个表达式步骤中的连续拷贝会进行合并优化，有些更新更"激进"的编译器还会进行跨行跨表达式的合并优化。
3. linux下可以将下面代码拷贝到test.cpp文件，编译时用 g++ test.cpp -fno-elide constructors 的方式关闭构造相关的优化。

代码块

```
1   #include<iostream>
2   using namespace std;
3   class A
4   {
5   public:
6       A(int a = 0)
7           :_a1(a)
8       {
9           cout << "A(int a)" << endl;
10      }
11
12      A(const A& aa)
13          :_a1(aa._a1)
14      {
15          cout << "A(const A& aa)" << endl;
16      }
17
18      A& operator=(const A& aa)
19      {
20          cout << "A& operator=(const A& aa)" << endl;
21          if (this != &aa)
22          {
23              _a1 = aa._a1;
24          }
25          return *this;
```

```
26      }
27
28      ~A()
29      {
30          cout << "~A()" << endl;
31      }
32
33  private:
34          int _a1 = 1;
35  };
36
37  void f1(A aa)
38  {
39  }
40
41  A f2()
42  {
43      A aa;
44      return aa;
45  }
46
47  int main()
48  {
49      // 传值传参
50      // 构造+拷贝构造
51      A aa1;
52      f1(aa1);
53      cout << endl;
54
55      // 隐式类型，连续构造+拷贝构造->优化为直接构造
56      f1(1);
57
58      // 一个表达式中，连续构造+拷贝构造->优化为一个构造
59      f1(A(2));
60      cout << endl;
61
62      cout << "********************************************" << endl;
63      // 传值返回
64      // 不优化的情况下传值返回，编译器会生成一个拷贝返回对象的临时对象作为函数调用表达
65      //式的返回值
66      //
67      // 无优化 （vs2019 debug）
68      // 一些编译器会优化得更厉害，将构造的局部对象和拷贝构造的临时对象优化为直接构造
69      //（vs2022 debug）
70      f2();
71      cout << endl;
72
```

```
73        // 返回时一个表达式中，连续拷贝构造+拷贝构造->优化一个拷贝构造 （vs2019 debug）
74        // 一些编译器会优化得更厉害，进行跨行合并优化，将构造的局部对象aa和拷贝的临时对象
75        //和接收返回值对象aa2优化为一个直接构造。（vs2022 debug）
76        A aa2 = f2();
77        cout << endl;
78
79        // 一个表达式中，开始构造，中间拷贝构造+赋值重载->无法优化（vs2019 debug）
80        // 一些编译器会优化得更厉害，进行跨行合并优化，将构造的局部对象aa和拷贝临时对象合
81        //并为一个直接构造（vs2022 debug）
82        aa1 = f2();
83        cout << endl;
84
85        return 0;
86    }
87
```