04.类和对象(下)

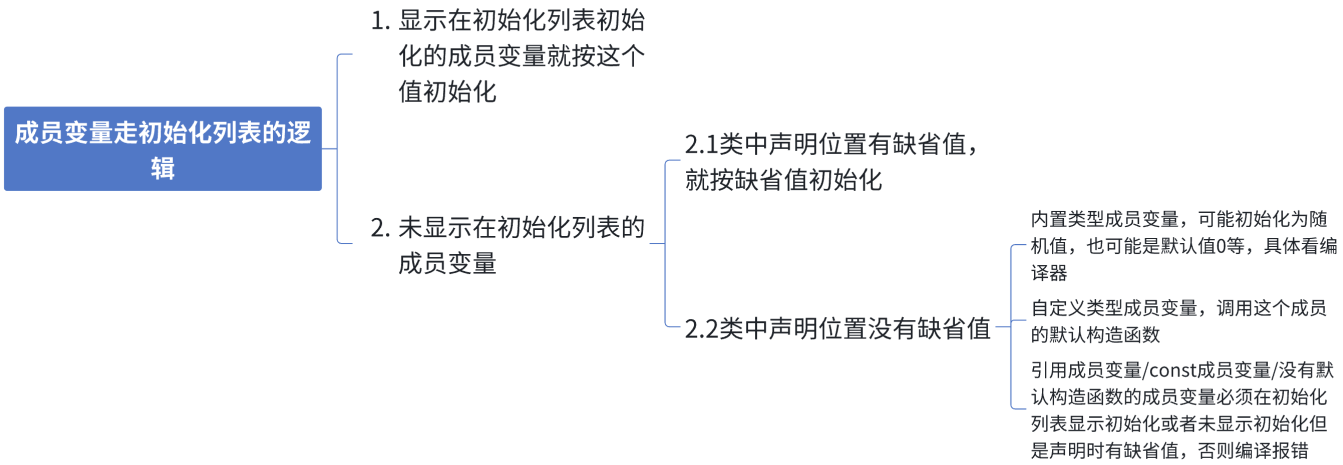
1. 再探构造函数

- 之前我们实现构造函数时，初始化成员变量主要使用函数体内赋值，构造函数初始化还有一种方式，就是初始化列表，初始化列表的使用方式是以一个冒号开始，接着是一个以逗号分隔的数据成员列表，每个"成员变量"后面跟一个放在括号中的初始值或表达式。
- 每个成员变量在初始化列表中只能出现一次，语法理解上初始化列表可以认为是每个成员变量定义初始化的地方。
- 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。
- C++11支持在成员变量声明的位置给缺省值，这个缺省值主要是给没有显示在初始化列表初始化的成员使用的。
- 尽量使用初始化列表初始化，因为那些你不在初始化列表初始化的成员也会走初始化列表，如果这个成员在声明位置给了缺省值，初始化列表会用这个缺省值初始化。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。
- 初始化列表中按照成员变量在类中声明顺序进行初始化，跟成员在初始化列表出现的的先后顺序无关。建议声明顺序和初始化列表顺序保持一致。

初始化列表总结：

无论是否显示写初始化列表，每个构造函数都有初始化列表；

无论是否在初始化列表显示初始化成员变量，每个成员变量都要走初始化列表初始化；




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

class Time
{
public:
    Time(int hour)
        :_hour(hour)
    {
        cout << "Time()" << endl;
    }
private:
    int _hour;
};

class Date
{
public:
    Date(int& x, int year = 1, int month = 1, int day = 1)
        :_year(year)
        ,_month(month)
        ,_day(day)
        ,_t(12)
        ,_ref(x)
        ,_n(1)
    {
        // error C2512: “Time”: 没有合适的默认构造函数可用
        // error C2530 : “Date::_ref” : 必须初始化引用
        // error C2789 : “Date::_n” : 必须初始化常量限定类型的对象
    }

    void Print() const
```

```
    {
        cout << _year << "-" << _month << "-" << _day << endl;
    }
private:
    int _year;
    int _month;
    int _day;

    Time _t;           // 没有默认构造
    int& _ref;           // 引用
    const int _n; // const
};

int main()
{
    int i = 0;
    Date d1(i);
    d1.Print();

    return 0;
}
```

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

class Time
{
public:
    Time(int hour)
        :_hour(hour)
    {
        cout << "Time()" << endl;
    }
private:
    int _hour;
};

class Date
{
public:
    Date()
        :_month(2)
    {
        cout << "Date()" << endl;
    }
```

```
24
25      void Print() const
26      {
27          cout << _year << "-" << _month << "-" << _day << endl;
28      }
29  private:
30      // 注意这里不是初始化，这里给的是缺省值，这个缺省值是给初始化列表的
31      // 如果初始化列表没有显示初始化，默认就会用这个缺省值初始化
32      int _year = 1;
33      int _month = 1;
34      int _day;
35
36      Time _t = 1;
37      const int _n = 1;
38      int* _ptr = (int*)malloc(12);
39  };
40
41  int main()
42  {
43      Date d1;
44      d1.Print();
45
46      return 0;
47  }
```

下面程序的运行结果是什么（）

A. 输出 1 1

B. 输出 2 2

C. 编译报错

D. 输出 1 随机值

E. 输出 1 2

F. 输出 2 1

```
1  #include<iostream>
2  using namespace std;
3
4  class A
5  {
6  public:
7      A(int a)
8          :_a1(a)
9          , _a2(_a1)
```

```
    {}

    void Print() {
        cout << _a1 << " " << _a2 << endl;
    }
private:
    int _a2 = 2;
    int _a1 = 2;
};

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

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。

```
#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)
        A(int a1)
                :_a1(a1)
        {}

        //explicit A(int a1, int a2)
        A(int a1, int a2)
                :_a1(a1)
                , _a2(a2)
        {}

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

```
23
24          int Get() const
25          {
26                  return _a1 + _a2;
27          }
28  private:
29          int _a1 = 1;
30          int _a2 = 2;
31  };
32
33  class B
34  {
35  public:
36          B(const A& a)
37                  :_b(a.Get())
38          {}
39
40  private:
41          int _b = 0;
42  };
43
44  int main()
45  {
46          // 1构造一个A的临时对象，再用这个临时对象拷贝构造aa3
47          // 编译器遇到连续构造+拷贝构造->优化为直接构造
48          A aa1 = 1;
49          aa1.Print();
50
51          const A& aa2 = 1;
52
53          // C++11之后才支持多参数转化
54          A aa3 = { 2,2 };
55
56          // aa3隐式类型转换为b对象
57          // 原理跟上面类似
58          B b = aa3;
59          const B& rb = aa3;
60
61          return 0;
62  }
```


3. static成员

- 用static修饰的成员变量，称之为静态成员变量，静态成员变量一定要在类外进行初始化。
- 静态成员变量为所有类对象所共享，不属于某个具体的对象，不存在对象中，存放在静态区。

- 用static修饰的成员函数，称之为静态成员函数，静态成员函数没有this指针。
- 静态成员函数中可以访问其他的静态成员，但是不能访问非静态的，因为没有this指针。
- 非静态的成员函数，可以访问任意的静态成员变量和静态成员函数。
- 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。
- 静态成员也是类的成员，受public、protected、private 访问限定符的限制。
- 静态成员变量不能在声明位置给缺省值初始化，因为缺省值是个构造函数初始化列表的，静态成员变量不属于某个对象，不走构造函数初始化列表。

```cpp
// 实现一个类，计算程序中创建出了多少个类对象？
#include<iostream>
using namespace std;

class A
{
public:
    A()
    {
        ++_scount;
    }

    A(const A& t)
    {
        ++_scount;
    }

    ~A()
    {
        --_scount;
    }

    static int GetACount()
    {
        return _scount;
    }
private:
    // 类里面声明
    static int _scount;
};

// 类外面初始化
int A::_scount = 0;

```

```cpp
int main()
{
    cout << A::GetACount() << endl;
    A a1, a2;
    A a3(a1);
    cout << A::GetACount() << endl;
    cout << a1.GetACount() << endl;

    // 编译报错: error C2248: “A::_scount”: 无法访问 private 成员(在“A”类中声明)
    //cout << A::_scount << endl;

    return 0;
}
```

求1+2+3+...+n_牛客题霸_牛客网

```cpp
class Sum
{
public:
    Sum()
    {
        _ret += _i;
        ++_i;
    }

   static int GetRet()
   {
        return _ret;
   }
private:
    static int _i;
    static int _ret;
};

int Sum::_i = 1;
int Sum::_ret = 0;

class Solution {
public:
    int Sum_Solution(int n) {
        // 变长数组
        Sum arr[n];
        return Sum::GetRet();
    }
};
```

4. 友元

- 友元提供了一种突破类访问限定符封装的方式，友元分为：友元函数和友元类，在函数声明或者类声明的前面加friend，并且把友元声明放到一个类的里面。
- 外部友元函数可访问类的私有和保护成员，友元函数仅仅是一种声明，他不是类的成员函数。
- 友元函数可以在类定义的任何地方声明，不受类访问限定符限制。
- 一个函数可以是多个类的友元函数。
- 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。
- 友元类的关系是单向的，不具有交换性，比如A类是B类的友元，但是B类不是A类的友元。
- 友元类关系不能传递，如果A是B的友元， B是C的友元，但是A不是C的友元。
- 有时提供了便利。但是友元会增加耦合度，破坏了封装，所以友元不宜多用。

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

// 前置声明，都则A的友元函数声明编译器不认识B
class B;

class A
{
    // 友元声明
    friend void func(const A& aa, const B& bb);
private:
    int _a1 = 1;
    int _a2 = 2;
};

class B
{
    // 友元声明
    friend void func(const A& aa, const B& bb);
private:
    int _b1 = 3;
    int _b2 = 4;
};

void func(const A& aa, const B& bb)
{
    cout << aa._a1 << endl;
    cout << bb._b1 << endl;
}
```

```
30
31 int main()
32 {
33     A aa;
34     B bb;
35     func(aa, bb);
36
37     return 0;
38 }
```

```
1  #include<iostream>
2  using namespace std;
3
4
5  class A
6  {
7      // 友元声明
8      friend class B;
9  private:
10     int _a1 = 1;
11     int _a2 = 2;
12 };
13
14 class B
15 {
16 public:
17     void func1(const A& aa)
18     {
19         cout << aa._a1 << endl;
20         cout << _b1 << endl;
21     }
22
23     void func2(const A& aa)
24     {
25         cout << aa._a2 << endl;
26         cout << _b2 << endl;
27     }
28 private:
29     int _b1 = 3;
30     int _b2 = 4;
31 };
32
33
34 int main()
35 {
```

```
    A aa;
    B bb;
    bb.func1(aa);
    bb.func1(aa);

    return 0;
}
```

5. 内部类

- 如果一个类定义在另一个类的内部，这个内部类就叫做内部类。内部类是一个独立的类，跟定义在全局相比，他只是受外部类类域限制和访问限定符限制，所以外部类定义的对象中不包含内部类。
- 内部类默认是外部类的友元类。
- 内部类本质也是一种封装，当A类跟B类紧密关联，A类实现出来主要就是给B类使用，那么可以考虑把A类设计为B的内部类，如果放到private/protected位置，那么A类就是B类的专属内部类，其他地方都用不了。

```
#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;
```

```
28
29          A::B b;
30
31          A aa;
32          b.foo(aa);
33
34          return 0;
35  }
```

求1+2+3+...+n_牛客题霸_牛客网

```
1
2  class Solution {
3      // 内部类
4      class Sum
5      {
6      public:
7          Sum()
8          {
9              _ret += _i;
10             ++_i;
11         }
12     };
13
14     static int _i;
15     static int _ret;
16 public:
17     int Sum_Solution(int n) {
18         // 变长数组
19         Sum arr[n];
20         return _ret;
21     }
22 };
23
24 int Solution::_i = 1;
25 int Solution::_ret = 0;
```


6. 匿名对象

- 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
- 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。

```cpp
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);

	A aa2(2);

	// 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
	Solution().Sum_Solution(10);

	return 0;
}
```


7. 对象拷贝时的编译器优化

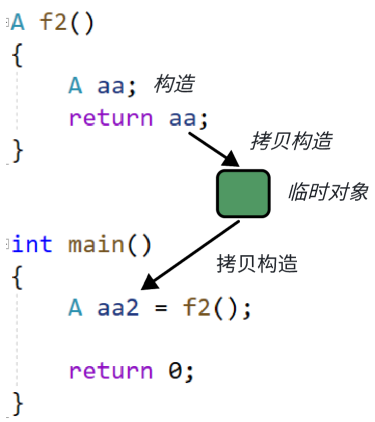
- 现代编译器会为了尽可能提高程序的效率，在不影响正确性的情况下会尽可能减少一些传参和传返回值的过程中可以省略的拷贝。
- 如何优化C++标准并没有严格规定，各个编译器会根据情况自行处理。当前主流的相对新一点的编译器对于连续一个表达式步骤中的连续拷贝会进行合并优化，有些更新更"激进"的编译器还会进行跨行跨表达式的合并优化。
- linux下可以将下面代码拷贝到test.cpp文件，编译时用 `g++ test.cpp -fno-elide-constructors` 的方式关闭构造相关的优化。

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

class A
{
public:
	A(int a = 0)
		:_a1(a)
	{
		cout << "A(int a)" << endl;
	}

	A(const A& aa)
		:_a1(aa._a1)
	{
		cout << "A(const A& aa)" << endl;
	}

	A& operator=(const A& aa)
	{
		cout << "A& operator=(const A& aa)" << endl;

		if (this != &aa)
		{
			_a1 = aa._a1;

		}

		return *this;
	}

	~A()
	{
		cout << "~A()" << endl;
	}
private:
```

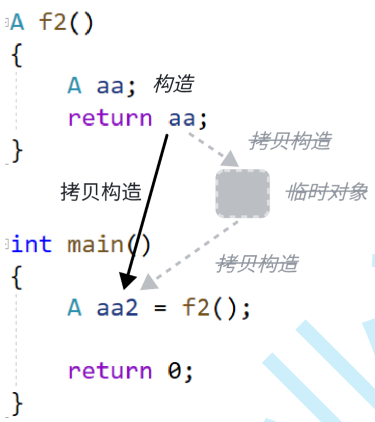
```
        int _a1 = 1;
};

void f1(A aa)
{}

A f2()
{
        A aa;
        return aa;
}

int main()
{
        // 传值传参
        // 构造+拷贝构造
        A aa1;
        f1(aa1);
        cout << endl;

        // 隐式类型，连续构造+拷贝构造->优化为直接构造
        f1(1);

        // 一个表达式中，连续构造+拷贝构造->优化为一个构造
        f1(A(2));
        cout << endl;

        cout << "********************************************" << endl;

        // 传值返回
        // 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式的返回值

        // 无优化 （vs2019 debug）
        // 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（vs2022 debug）
        f2();
        cout << endl;

        // 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）
        // 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和拷贝的临时对象和接收返回值对象aa2优化为一个直接构造。（vs2022 debug）
        A aa2 = f2();
        cout << endl;

        // 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和拷贝临时对象合并为一个直接构造（vs2022 debug）
81          aa1 = f2();
82          cout << endl;
83
84          return 0;
85  }
```

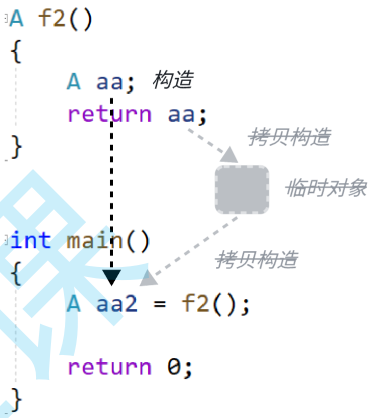
g++ test.cpp -fno-elide-constructors

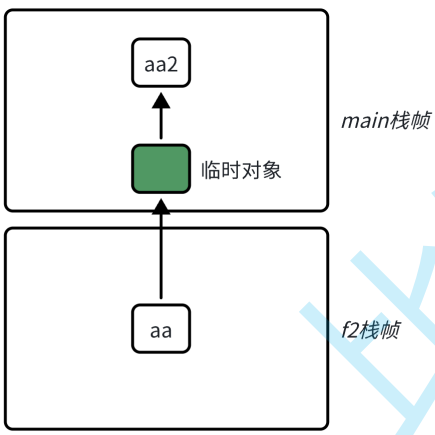


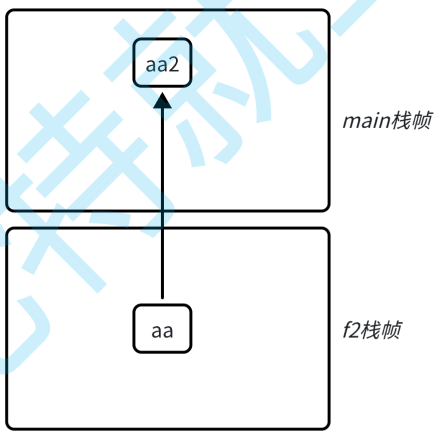
vs2019 debug

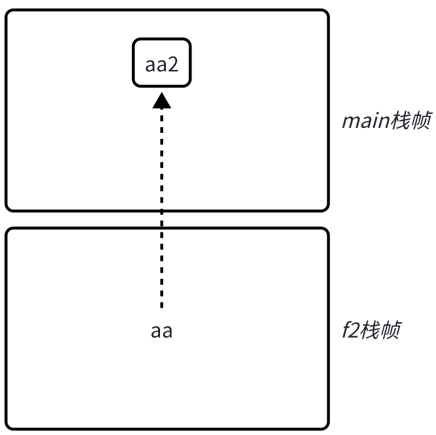


vs2022 debug









底层实现的角度，没有构造aa
构造的是aa2，aa是aa2的引用