



构造函数

构造函数与析构函数

三三三 13657136475



构造/析构函数	使用方式
默认构造函数	People a;
People(string name);	People a("hug");
People(const People &a)	拷贝构造，与 = 不等价
~People();	无

- **具有一个参数的构造函数叫转换构造。**多个参数没有名字。

拷贝构造必须传引用的原因：

变量更想要拷贝给形参，需要调用该对象的拷贝构造函数，而你现在正在实现这个拷贝构造函数，类似鸡生蛋的问题，一样会一直无线递归下去。

```
guanghu@haizei:13.封装$ g++ 5.constructor.cpp
5.constructor.cpp:88:7: error: no viable constructor copying variable of type 'A'
  A a = 45; // 2 constructor
    ^
5.constructor.cpp:22:5: note: candidate constructor not viable: no known conversion from 'A' to
'int' for 1st argument
  A(int x) : x(x) {
    ^
5.constructor.cpp:25:5: note: candidate constructor not viable: expects an l-value for 1st
argument
  A(A &a) {
    ^
1 error generated.
guanghu@haizei:13.封装$
```

此 & → 不能绑定到匿名对象

拷贝构造必须加const的3点原因：

1. 为了编译器兼容绑定匿名对象的形式。
2. 逻辑上的常量限制。拷贝函数内部是不会修改传入的对象。
3. 兼容const类型的拷贝。就是把const类型的对象绑定到非const类型的引用上，程序会报错，因此索性都加上const。

啥是引用:4.reference.cpp

```
#include<iostream>
using namespace std;
void add_one(int &n) {
    n += 1;
    return ;
}

int main(){
    int a = 1, b = 3;
    cout << "normal param : " << a << " " << b << endl;
    add_one(a);
    add_one(b);
    cout << a << " " << b << endl;

    return 0;
}
```

引用是贴在实参上的标签，实际上修改的是实参。

初始化列表:5.constructor.cpp

初始化列表为啥是完美的赋初值方式？

- 初始化列表中初始化的**顺序**只和相关的成员属性在类中**声明的顺序**有关系。先声明先初始化。
- 当定义了有参构造的时候，**默认构造会被清除掉**。
- 在大括号内，成员变量已经初始化完了。
- 规定了每一个成员属性的构造行径。
- 提高效率。减少一次初始化。

```

#include<iostream>
using namespace std;

class A {
public:
    A(int x) : x(x) {
        cout << this << " : Class A : " << x << endl;
    }
    A(const A &a) {
        cout << this << " copy from " << &a << endl;
    }
    void operator=(const A &a) {
        cout << this << " assign from " << &a << endl;
        return ;
    }
    int x;
};

class B1 {
public:
    B1() = default;
    B1(const B1 &) {
        cout << "B1 copy constructor" << endl;
    }
};

class B2 {
public:
    B2() = default;
    B2(const B2 &) {
        cout << "B2 copy constructor" << endl;
    }
};

class B3 {
public:
    B3() = default;
    B3(const B3 &) {
        cout << "B3 copy constructor" << endl;
    }
};

class B {
public:
    B1 b1;
    B2 b2;
    B3 b3;
};

```

```

class Data {
public:
    Data() : __x(0), __y(0), a(34) {
        cout << "default constructor" << endl;
    }
    int x() { return __x; }
    int y() { return __y; }
    ~Data() {
        cout << "destructor" << endl;
    }

private:
    int __x, __y;
    A a;
};

int main() {
    B b1;
    B b2 = b1;

    Data d;
    cout << d.x() << " " << d.y() << endl;
    A a = 45; // 2 constructor
    cout << "address a : " << &a << endl;
    a = 78;

    return 0;
}

```

```

// 类中的四个默认：
// 1. 默认构造
// 2. 默认析构
// 3. 默认拷贝
// 4. 默认赋值运算

```

赋值运算符=进行构造：

```

A a = 45;

```

关闭返回值优化：

g++ -fno-elide-constructors 5.constructor.cpp

- 首先编译器确定了要调用拷贝构造（等号=）
- 然后拿45这个数去想办法变成A类型的。并且绑定到引用上。
- 所以才调用了转换构造，而不是说两边类型不匹配的原因（不准确）。

```
74 ~Data() {
75     cout << "destructor" << endl;
76 }
77
78 private:
79     int x, y;
80     A a;
81 };
82
83 int main() {
84     B b1;
85     B b2 = b1;
86     Data d;
87     cout << d.x() << " " << d.y() << endl;
88     A a = 45; // 2 constructor
89     cout << "address a : " << &a << endl;
90     a = 78;
91     return 0;
92 }
```

Handwritten note: $45 \rightarrow \text{拷贝构造} \rightarrow A(\text{const } A\&a)$

- 一定要关注C++程序的执行流程。

a = 78

- 不同于上面的点：首先转换构造。然后调用赋值运算符。

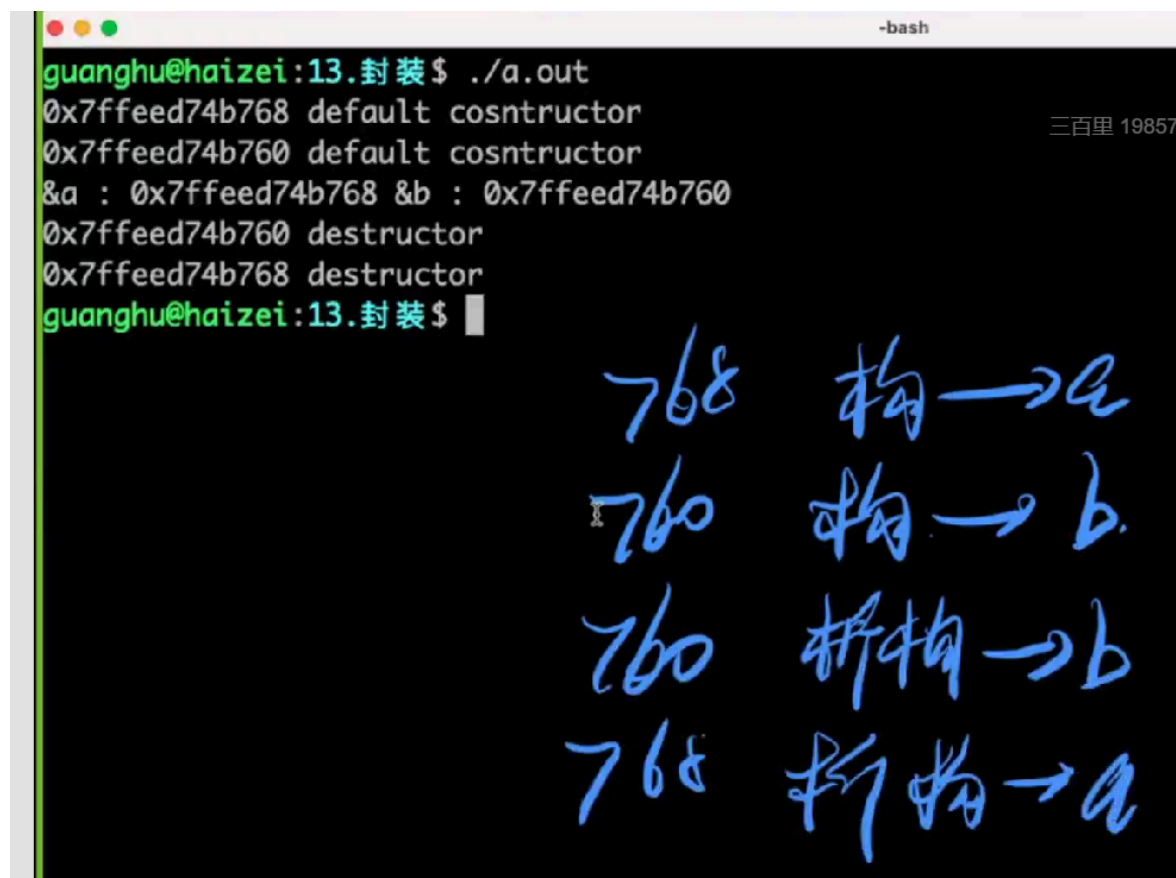
```
41 int y() { return y; }
42
43 ~Data() {
44     cout << "destructor" << endl;
45 }
46
47 private:
48     int x, y;
49     A a;
50 };
51
52 int main() {
53     Data d;
54     cout << d.x() << " " << d.y() << endl;
55     A a = 45; // 2 constructor
56     cout << "address a : " << &a << endl;
57     a = 78;
58     return 0;
59 }
```

Handwritten note: $45 \rightarrow \text{拷贝构造} \rightarrow A(\text{const } A\&a)$

- 系统自带的赋值运算符的行为方式：依次调用每个成员属性的赋值运算符。
- 系统默认会添加拷贝构造：依次调用每个成员属性的拷贝构造函数

- 拷贝和赋值是两类不同的操作，拷贝是拷贝构造。

构造跟析构的顺序:6.constructor_order.cpp



```
guanghu@haizei:13.封装$ ./a.out
0x7ffeed74b768 default constructor
0x7ffeed74b760 default constructor
&a : 0x7ffeed74b768 &b : 0x7ffeed74b760
0x7ffeed74b760 destructor
0x7ffeed74b768 destructor
guanghu@haizei:13.封装$
```

Handwritten notes on the right side of the terminal output:

- 768 构 → a
- 760 构 → b
- 760 析构 → b
- 768 析构 → a

- 构造顺序和析构顺序相反：先构造的后析构。（考虑到变量依赖问题）
- 不仅是作用在不同对象之间，还作用在成员属性和当前对象之间。

```

guanghu@haizei:13.封装$ ./a.out
A(int) constructor
A(int *size) constructor
&a : 0x7ffeeadd6730 &b : 0x7ffeeadd6710
0x7ffeeadd6710 destructor
0x7ffeeadd6730 destructor
ATTR_1 default constructor
ATTR_2 default constructor
ATTR_3 default constructor
ATTR_4 default constructor
Class A constructor
Class A destructor
ATTR_4 destructor
ATTR_3 destructor
ATTR_2 destructor
ATTR_1 destructor
guanghu@haizei:13.封装$

```

- 内部成员变量先构造，后析构：
- 构造的顺序：先完成各部分的成员属性的构造，最后才完成自身的构造，成员属性的构造顺序只和成员属性的声明顺序相关。
- 析构的顺序：当前类先完成自身的析构，再实现各成员变量的析构
- 先构造父类，再构造子类。

代码案例：

```

#include<iostream>
using namespace std;

#define BEGINS(x) namespace x {
#define ENDS(x) }

BEGINS(test1)
class A {
public:
    A() {
        cout << this << " default constructor" << endl;
    }
    A(int n, int m) :
    n(n), m(m),
    arr(nullptr), size(nullptr),
    offset(nullptr) {
        cout << "A(int) constructor" << endl;
    }
    A(int *size, int *offset) : size(size), offset(offset) {
        arr = new int[*size];
        arr += *offset;
        cout << "A(int *size) constructor" << endl;
    }
    ~A() {
        cout << this << " destructor" << endl;
        if (arr == nullptr) return ;
        arr -= *offset;
        delete[] arr;
    }
    int *arr, *size, *offset;
    int n, m;
};

void main() {
    A a(3, 4);
    A b(&a.n, &a.m);
    cout << "&a : " << &a << " &b : " << &b << endl;
    return ;
}
ENDS(test1)

BEGINS(test2)

#define ATTR_CLASS(x) class ATTR_##x { \
public: \
    ATTR_##x() { \

```



```

        cout << "ATTR_" #x " default constructor" << endl; \
    } \
    ~ATTR_##x() { \
        cout << "ATTR_" #x " destructor" << endl; \
    } \
};
ATTR_CLASS(1);
ATTR_CLASS(2);
ATTR_CLASS(3);
ATTR_CLASS(4);

class A {
public:
    A() {
        cout << "Class A constructor" << endl;
    }
    ~A() {
        cout << "destructor" << endl;
    }
    ATTR_1 a1;
    ATTR_2 a2;
    ATTR_3 a3;
    ATTR_4 a4;
};


void main() {
    A a;
    return ;
}
ENDS(test2)

int main() {
    test1::main();
    test2::main();
}

```

类属性与方法

类属性与方法



```
class People {
public :
    void say(string word);
    void run(Location &loc);
    static void is_valid_height(double height);
private :
    static int total_num;
    string __name;
    Day __birthday;
    double __height;
    double __weight;
};
```

- **类属性**：主要体现在性质和功能上：数据区里面有一个**静态数据区**，不随对象的增加而增加。**类属性**就存在**静态数据区**中。增加**static**关键字即可把成员属性变成类属性。一个对象对类属性的修改，其他对象都能够感知到。
- ****类方法**：**主要体现在逻辑上：成员方法是属于对象的行为，**类方法是非对象行为**，不能访问**this**指针。不是某个对象的方法

const 类型的方法:7.const_method.cpp

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

  #define BEGINS(x) namespace x {
  #define ENDS(x) }

  BEGINS(test1)
  class A {
  public:
      A() { x = 23800; }
      void say() const {
          x = 30000;
          cout << x << endl;
      }
      mutable int x;
  };

  void main() {
      const A a;
      a.say();
      return ;
  }
  ENDS(test1)

  int main() {
      test1::main();

      return 0;
  }
```

```
guanghu@haizei:13.封装$ ./a.out
23800
guanghu@haizei:13.封装$ vim 7.const_method.cpp
guanghu@haizei:13.封装$ g++ 7.const_method.cpp
7.const_method.cpp:36:5: error: 'this' argument to member function 'say' has type
      'const test1::A', but function is not marked const
      a.say();
      ^
7.const_method.cpp:28:10: note: 'say' declared here
      void say() {
      ^
1 error generated.
guanghu@haizei:13.封装$
```

- this指针指向了一个const对象，但是成员方法却不是const方法。换言之，**const类型的对象只能定义const类型的方法**。（原因是限制成员方法不去修改成员属性。**const**加在方法的大括号 { 前面）

- 如果仍要在const方法里面修改某个变量，可以在该变量前面加上**mutable**关键字。
- **C++**的核心精髓：用**严格准确的逻辑**，将**错误暴露在编译阶段**。