模板是编译器产生具体代码的规则。

先补充两个知识点：**左值跟右值**

左值右值

> C++11标准推出让C++重回神坛，其中一个点就是区分了左值跟右值。

值	左值	右值
引用	左值引用（int &）	右值引用（int &&）

左值引用优先匹配左值，右值引用则优先匹配右值。

是不是左值就看，能不能放到赋值运算符的左边。

左值表常态，右值表临时

==窍门：==在代码下一行的时候能不能直接通过一个单一的变量访问到表达式的值。可以就是左值，不行就是右值。

比如n++表达式返回的是n加一之前的值，在下一行是无法通过单一变量访问到的，所以这个是右值。

但是左值跟右值是有作用域的，会出现在外界作用域是右值，但是当前作用域是左值的情况。

```
void judge2(int &x) {
    cout << "(left value - 2) ";
    return ;
}
void judge2(int &&x) {
    cout << "(right value - 2) ";
    return ;
}
void judge(int &x) {
    judge2(x);
    cout << "left value" << endl;
    return ;
}
void judge(int &&x) {
    judge2(x);// 下一行能获得x的值，显然是左值
    cout << "right value" << endl;
    return ;
}
#define TEST(a) { \
    cout << "judge " << #a << " : "; \
    judge(a); \
}
int main() {
    int n = 12;
    (n += 2) = 1000;
    cout << n << endl;
    TEST(n);
    TEST(n + 1);
    TEST(n++);
    TEST(++n);
    TEST(1 + 2);
    TEST(n += 2);
    return 0;
}
```

```
1000
judge n : (left value - 2) left value
judge n + 1 : (left value - 2) right value
judge n++ : (left value - 2) right value
judge ++n : (left value - 2) left value
judge 1 + 2 : (left value - 2) right value
judge n += 2 : (left value - 2) left value
```

为什么要分成左值和右值？——为了处理不同的情况。

- **move()运算**：将任何表达式直接变右值。

```
// 如何向下准确地进行参数传递?
void judge(int &&x) {
    judge2(move(x));
    cout << "right value" << endl;
    return ;
}
```

- **forward<>()：**将相应表达式变成固定类型。

```
void judge(int &&x) {
    judge2(forward<int &&>(x));
    cout << "right value" << endl;
    return ;
}
```

在某些运算符重载的时候。设计就是参照：左值返回引用，右值返回值。

```
class Point;
void judge(Point &x) {
    cout << "left value" << endl;
    return ;
}
void judge(Point &&x) {
    cout << "right value" << endl;
    return ;
}
class Point {
public:
    Point(int x = 0, int y = 0) : __x(x), __y(y) {}
    int x() { return __x; }
    int y() { return __y; }
    Point &operator+=(int d) {
        __x += d;
        __y += d;
        return *this;
    }
    Point operator+(int d) {
        Point p(__x + d, __y + d);
        return p;
    }
private:
    int __x, __y;
};
#define TEST(a) { \
    cout << "judge " << #a << " : "; \
    judge(a); \
}
int main() {
    Point p(3, 4);
    TEST(p);
    TEST(p += 1); // += 运算符，判断是左值合理一点，运算符重载就返回引用
    TEST(p + 1);  // + 运算符，显然是右值，运算符重载就返回值
    return 0;
}
```

移动构造

右值的一个非常重要的功能。在左值右值场景下，做区分的一个产物。

移动构造是面对**右值拷贝**的情形。

常见的左值拷贝，因为被拷贝的变量是一个左值，是我们不能动的，所以我们需要把全部东西**深拷贝**过来。

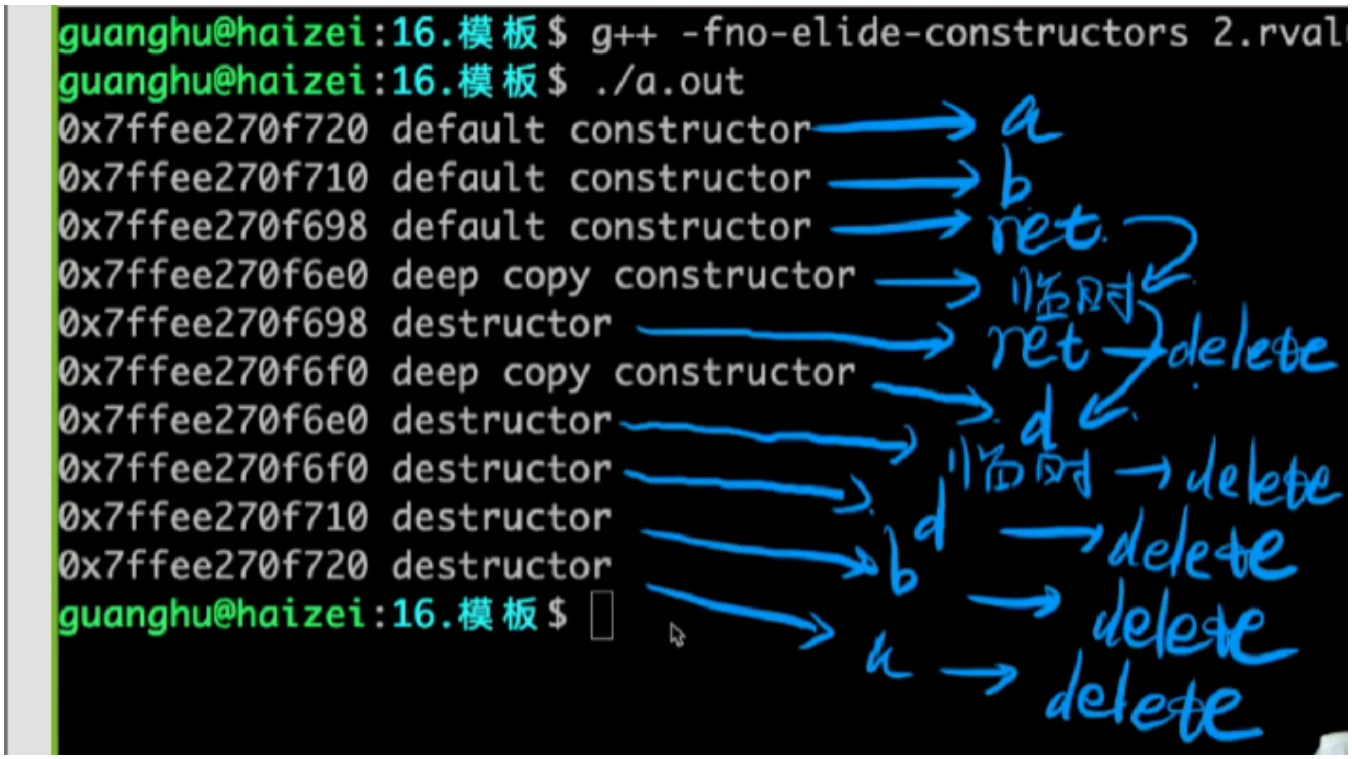
时间复杂度是**O(n)**的

```
Array(const Array &a) : __size(a.__size), data(new int[__size]) {
    for (int i = 0; i < __size; i++) {
        data[i] = a[i];
    }
    cout << this << " deep copy constructor " << endl;
}
```

但是右值就不一样，右值是一个临时值，牺牲掉不可惜。我可以直接把他的东西拿过来，而不需要再拷贝一份。

时间复杂度是**O(1)**的

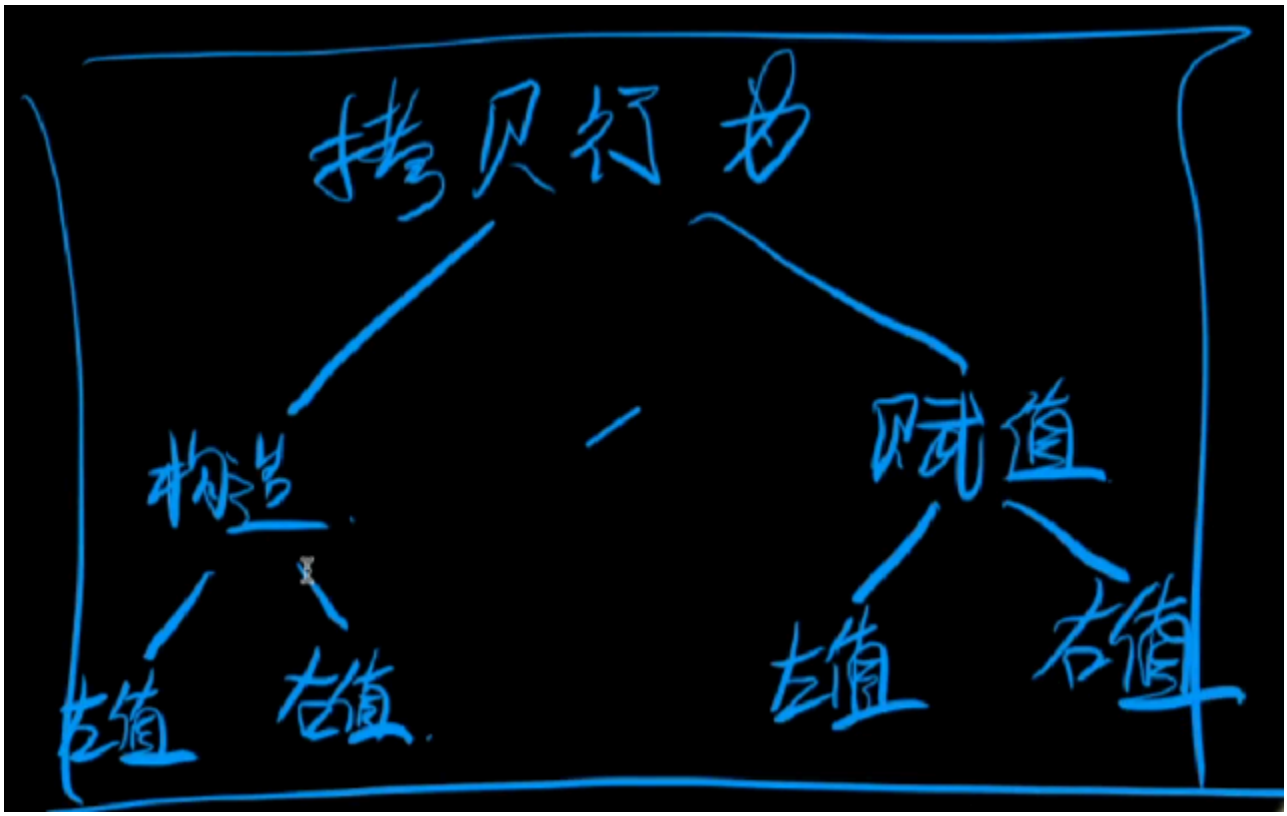
```
Array(Array &&a) : __size(a.__size), data(a.data) {
    // move constructor
    a.data = nullptr;
    a.__size = 0;
    cout << this << " move constructor " << endl;
}
```



```
g++ -fno-elide-constructors 2.rvalue_move.cpp
```

```
// 通过move函数变成右值，调用移动构造
Array d(move(a));
```

拷贝行为：**构造+赋值运算符。**



模板的特化

针对某些场景去做特定的处理。其实就是固定参数。

- 所谓**全特化就是所有参数都被固定**下来的情况，**偏特化就是部分参数被固定**下来的情况。

1. 函数模板的全特化：

```
template<typename T>
T add(T a, T b) {
    return a + b;
}
// 在输出整型的时候额外输出一段信息（用到全特化）
template<>
int add<int>(int a, int b) {
    cout << "int template " << endl;
    return a + b;
}
int main() {
    cout << add(3, 4) << endl;
    cout << add(3.1, 4.2) << endl;
    return 0;
}
```

2. 类模板的偏特化与全特化

```
template<typename T, typename u> // 正常的
class Test {
public:
    Test() {
        cout << "normal template<T, U>" << endl;
    }
};
template<> // 全特化
class Test<int, double> {
public:
    Test() {
        cout << "specialization template<int, double>" << endl;
    }
};
template<typename T> // 偏特化
class Test<int, T> {
public:
    Test() {
        cout << "partial specialization template<int, T>" << endl;
    }
};
int main() {
    Test<string, int> t1; // 正常的
    Test<int, double> t2; // 全特化,比偏特化优先级高
    Test<int, string> t3; // 偏特化
    return 0;
}
```

优先级：**全特化 > 偏特化 > 正常的。**

要求实现：判断有没有构造函数

1. 判断内置类型没有构造函数
2. 判断指针没有构造函数
3. 其他有构造函数

```
class A {};
class B {};

template<typename T>
class type_trais {
public:
    static const char *has_constructor;
};
template<typename T>
const char *type_trais<T>::has_constructor = "yes";
// 对所有的指针,没有构造函数,输出no
template<typename T>
class type_trais<T *> {
public:
    static const char *has_constructor;
};
template<typename T>
const char *type_trais<T *>::has_constructor = "no";

template<>
class type_trais<int> {
public:
    static const char *has_constructor;
};
const char *type_trais<int>::has_constructor = "no";

template<>
class type_trais<double> {
public:
    static const char *has_constructor;
};
const char *type_trais<double>::has_constructor = "no";

#define TEST(type) \
    cout << #type << " : " << type_trais<type>::has_constructor << endl;

int main() {
    TEST(int);
    TEST(A);
    TEST(double);
    TEST(B);
    TEST(string);
    TEST(string *);
    return 0;
}
```

方案1：

发现上述代码中，每次写特化版本的时候，都要重复写一遍成员属性，有什么办法能够解决呢？

——**继承**

```
class A {};
class B {};

class yes_constructor {
public:
    static const char *has_constructor;
};
const char *yes_constructor::has_constructor = "yes";

class no_constructor{
public:
    static const char *has_constructor;
};
const char *no_constructor::has_constructor = "no";


template<typename T>
class type_trais : public yes_constructor {};

// 对所有的指针,没有构造函数,输出no
template<typename T>
class type_trais<T *> : public no_constructor {};

template<>
class type_trais<int> : public no_constructor {};

template<>
class type_trais<double> : public no_constructor {};

#define TEST(type) \
    cout << #type << " : " << type_trais<type>::has_constructor << endl;

int main() {
    TEST(int);
    TEST(A);
    TEST(double);
    TEST(B);
    TEST(string);
    TEST(string *);
    return 0;
}
#undef TEST
```

方案2：

还有一种设计方式——设置空类，**并作为成员属性**，**然后重载输出方式**：

```cpp
class A {};
class B {};
//设置为不同的类
class yes_constructor {};
class no_constructor{};

template<typename T>
class type_trais {
public:
    typedef yes_constructor has_constructor;
};

// 对所有的指针,没有构造函数,输出no
template<typename T>
class type_trais<T *> {
public:
    typedef no_constructor has_constructor;
};

template<>
class type_trais<int> {
public:
    typedef no_constructor has_constructor;
};

template<>
class type_trais<double> {
public:
    typedef no_constructor has_constructor;
};
// 只需要重载输出这两个类的对象即可
ostream &operator<<(ostream &out, const yes_constructor &) {
    out << "yes";
    return out;
}
ostream &operator<<(ostream &out, const no_constructor &) {
    out << "no";
    return out;
}
#define TEST(type) \
    cout << #type << " : " << type_trais<type>::has_constructor() << endl;
int main() {
    TEST(int);
    TEST(A);
    TEST(double);
    TEST(B);
```

```
    TEST(string);
    TEST(string *);
    return 0;
}
#undef TEST
```