三五法则

三/五法则

- 在 C++98 中，由于没有移动语义的存在，所以我们只会定义 3 个成员函数来控制对象在销毁、拷贝和赋值时做什么操作，这些函数分别为析构函数、拷贝构造函数和拷贝赋值函数。此时的拷贝控制称之为 C++ 三法则
- 而在 C++11 中，额外的添加了移动语义，使得程序可以通过转移对象控制权的方式实现对象的移动，也就是移动构造函数和移动赋值函数。加上这两个函数以后，能够影响对象拷贝控制的函数增加至 5 个所以也称之为 C++ 五法则
- 后面为了统一，干脆称为"三/五法则"，指的其实就是析构函数，拷贝构造函数，拷贝赋值函数，移动构造函数以及移动赋值函数

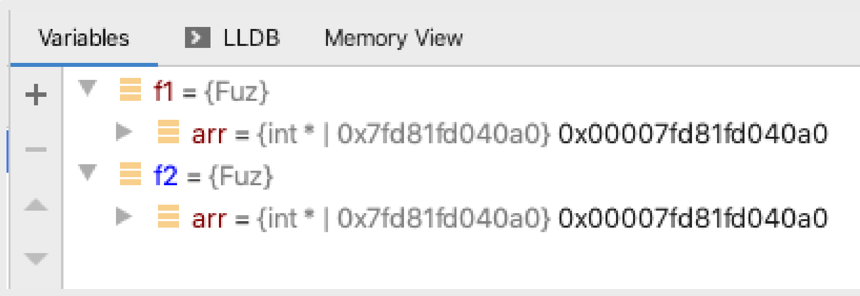
拷贝控制

需要析构函数的类必须定义拷贝和赋值函数

- 当一个类需要析构函数时，一定是因为其在实例化对象时申请了系统资源，并且这些资源必须在对象被销毁时归还给系统。最典型的资源莫过于动态分配的内存，打开的文件描述符，一个 TCP 连接
- 这些资源如果依赖于系统所默认的拷贝控制操作，那么在程序运行过程中就会埋下巨大的隐患
- **比如如果一个类中持有 malloc 所分配的堆内存，并使用成员指针进行管理。由于系统默认的拷贝操作只会拷贝指针的地址，那么执行拷贝操作后将会有两个对象指向同一块堆内存。当两个对象先后析构时，就会对同一块堆内存释放两次，这是系统所不允许的**
-

```
class Fuz {
private:
    int *arr;
public:
    Fuz() : arr(new int){}
    ~Fuz() {delete arr;}
};


int main() {
    Fuz f1;
    Fuz f2 = f1;
}
```

  - Fuz 类虽然定义了析构函数，但是拷贝构造/赋值却使用了系统提供的默认实现
  - 

  - 在 DUBUG 模式下我们可以非常清晰的看到 f1.arr 和 f2.arr 指向的是同一个内存地址
  - 当我们对同一块堆内存释放两次时，底层将直接调用 abort() 终止进程
- **因此，当一个类需要析构函数时，并且该类确实是应该可以被拷贝的，那么应正确的实现拷贝构造和拷贝赋值函数。否则，应禁用这两个函数，显式地标注为 = delete**

如果可能的话，尽量定义移动构造函数

- 在移动构造中，我们提到了如果某个类的移动构造被声明为 noexcept 的话，那么容器在扩容进行元素迁移时，将调用该类的移动构造函数，从而提高系统运行效率
- 在一般的拷贝构造函数中，我们通常会需要重新申请动态内存，每一次的 malloc/free 都是有代价的。尽管 STL 使用内存池进行了优化，但是相比于移动的方式，拷贝的代价仍然要更大一些
-

```
class Fuz {
public:
    Fuz() {puts("ctor");}

    Fuz(const Fuz& other) {puts("copy ctor");}

    Fuz(Fuz&& other) noexcept {puts("move ctor");}

};
```

  -

```
int main() {
    vector<Fuz> fuzes;
    fuzes.reserve(1);

    fuzes.emplace_back();
    fuzes.emplace_back();
}
```

  - 扩容时将调用移动构造函数

需要拷贝的类也需要赋值，移动语义同理

- 通常来说，当一个类定义了拷贝构造函数，那么就应该要有拷贝赋值函数，否则这个类的功能是不完整的。移动构造和移动赋值也是一样的，很少有一个类只需要拷贝而不需要赋值
- 也就是说，拷贝和赋值这两个兄弟应该是成对出现的，不应该落下其中任何一个