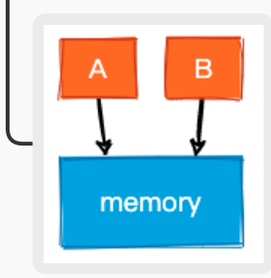
拷贝构造与拷贝赋值

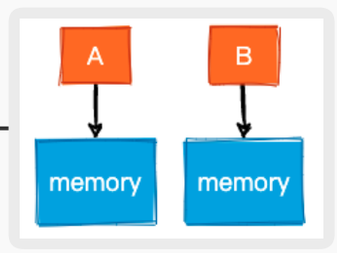
浅拷贝与深拷贝

- 提到拷贝控制，就不得不提到浅拷贝和深拷贝这一个非常重要的概念
- 我们用一根指针做一个非常简单的释义
  - 浅拷贝
    - 拷贝指针的地址，也就是拷贝之后结果是两根指针指向同一块内存区域
    - 此时我们修改任何一根指针指向的内容，另一根指针也会受到影响
    - 

    - ```
      int i = 1024;
      int *ptr = &i;

      int *ktr = nullptr;
      ktr = ptr;

      *ktr = 2048;      // 此时我们修改 ktr 指针指向的内存内容

      cout << *ptr << endl;
      ```
    - 指针默认是浅拷贝行为，也就是 ktr 和 ptr 指向的是同一块内存区域
  - 深拷贝
    - 不拷贝指针本身的地址，而是拷贝指针指向的内存内容，拷贝后两根指针互不相干，任何一方的修改都不会影响另一方
    - 

- C++ 语言中对一类的的拷贝到底是浅拷贝还是深拷贝完全交给了类的设计者，这一点和 Java 相同，和 Python、Go 由内部函数实现的方式有所不同

拷贝构造

- 拷贝构造是指使用另一个对象来直接初始化当前声明的对象，对应拷贝构造函数 ClassName(const ClassName& other)
- Example
  - ```
    class Buz {
    private:
        int m_size;
        int *m_array;
    public:
        explicit Buz(int size = 4) : m_size(size), m_array(new int[size]) {
            puts("ctor");
        }
        ~Buz() {delete[] m_array;}

        Buz(const Buz& other);

        const int &operator[] (int index) const { return m_array[index];}
        int &operator[] (int index) {return m_array[index];}
    };
    ```
  - 在 Buz 类中存在一个 int 类型的指针，我们作为 array 使用，由此可以实现动态数组
  - 浅拷贝实现
    - ```
      Buz::Buz(const Buz &other) : m_size(other.m_size), m_array(other.m_array) {
          puts("copy ctor");
      }
      ```
  - 深拷贝实现
    - ```
      Buz::Buz(const Buz &other) : m_size(other.m_size), m_array(new int[other.m_size]) {
          memcpy(m_array, other.m_array, other.m_size * sizeof(int));
          puts("copy ctor");
      }
      ```
- 要素
  - **拷贝构造函数必须接收一个常量的对象引用**
  - **添加 const 的原因在于拓宽拷贝构造函数可接收形参范围，不管是常量对象还是非常量对象，该函数均能使用**
  - **按引用传递而非按值传递形参的原因在于，当函数内部使用实参初始化形参时，实际上就是调用的拷贝构造函数。那么如果我们按值传递的话，将会发生无限递归调用，直到栈溢出。因此，当拷贝构造函数形参为值类型时，编译器会直接抛出错误**
- 发生时机
  - 和构造函数一样，我们没有办法直接调用一个类的拷贝构造函数，这个过程是由系统自动调用的
  - 显式初始化
    - ```
      Buz buz{1024};
      Buz puz = buz;    // 此时将会调用拷贝构造函数来初始化 puz 对象
      ```
  - 值类型的函数形参
    - 若某一个函数的形参为值类型的话，在使用实参初始化形参时也会调用拷贝构造函数，而非构造函数
    - ```
      void foo(Buz b) {}

      Buz buz{1024};
      foo(buz);
      ```
    - 程序运行将会输出 "copy ctor"，证明了拷贝构造函数被调用
- 合成拷贝构造函数
  - 当我们没有给一个类提供拷贝构造函数时，编译器会给我们合成一个默认的拷贝构造函数
  - 该函数会对所有的成员变量逐一地调用其拷贝构造函数

拷贝赋值

- 拷贝赋值是一个重载函数，重载函数名称为 operator =，也就是赋值运算符
- Example
  - 这里将类对象的拷贝行为设计成深拷贝，因此，下面的代码是建立在拷贝构造为深拷贝实现的基础之上的
  - ```
    Buz &Buz::operator= (const Buz &rhs) {
        Buz temp = Buz(rhs);

        using std::swap;
        swap(this->m_size, temp.m_size);
        swap(this->m_array, temp.m_array);

        return *this;
    }
    ```
  - **在该拷贝赋值函数中，我们首先使用 rhs 构造了一个临时对象，并且我们交换 this 和 temp 临时对象的相关数据**
  - **当拷贝赋值函数执行完毕时，temp 临时对象将会被析构，从而将原有 this 对象的堆内存释放**
  - 而且，这样的方式也不需要显式的去处理 "自己给自己赋值" 这样的情况，例如 buz = buz
- 发生时机
  - 除了使用其它对象直接初始化当前对象的场景以外，所有带有 "=" 的语句均会触发拷贝赋值函数
  - ```
    Buz buz(10);
    for (int i = 0; i < 10; i++) buz[i] = i;
    Buz puz;
    puz = buz;    // 拷贝赋值函数将被调用
    puz[0] = 1024;

    cout << buz[0] << endl;    // 0
    cout << puz[0] << endl;    // 1024
    ```
  - 此时 buz 和 puz 是完全不同的对象，成员变量 m_array 所指向的地址也不相同
  - 使用 CLion 可以非常轻松的确定这一点
- 合成拷贝构造函数
  - 当我们没有给一个类提供拷贝赋值函数时，编译器会给我们合成一个默认的拷贝赋值函数
  - 该函数会对所有的成员变量逐一地调用其拷贝赋值函数