封装, 继承, 多态

封装

封装是将对象的属性和行为结合在一起, 对外部隐藏对象的内部细节, 只提供接口与外部通信

继承

继承是指一个类可以派生出一个或多个子类, 子类可以继承父类的属性和行为, 并且可以重写父类的属性和行为

- 公有继承: public
- 保护继承: protected: 可以被派生类访问, 不能被外部访问
- 私有继承: private: 派生类不能访问基类的成员

多态

多态是指同一个函数调用, 由于对象不同, 可以有不同的行为

- 静态多态: 函数重载: 同一个类中, 函数名相同, 参数列表不同
- 动态多态: 虚函数: 父类指针指向子类对象, 调用虚函数时, 会调用子类的虚函数

重载

- 重载的条件:
  - 参数个数不同
  - 参数类型不同
  - 参数顺序不同
- 返回值不同, 不构成重载

虚函数

```
class A{
public:
    void func1(){}
    void func2(){}
public:
    virtual void vfunc1(){}
    virtual void vfunc2(){}
    virtual ~A(){} // 虚析构函数
private:
    int m_a;
    int m_b;
};
```

虚函数

虚函数是在基类中使用关键字`virtual`声明的成员函数

- 虚函数的作用是实现多态
- 虚函数的调用是在运行时确定的

虚函数表

当一个类中包含虚函数时，编译器会为该类生成一个虚函数表

虚函数表指针

虚函数表指针: Vptr 虚函数表: Vtbl

- 每个类的构造函数都会有一条语句, 将该类的虚函数表指针指向该类的虚函数表
- 如果程序员已经定义了构造函数, 则编译器一日会在构造函数中插入该语句

类对象在内存中的布局

内存中:

类A对象:

- 虚函数表指针Vptr -> 虚函数表Vtbl
- int m_a
- int m_b

类A对象的虚函数表Vtbl:

- 指针1: A::vfunc1
- 指针2: A::vfunc2
- 指针3: A::~A

虚函数的工作原理及多态性的体现

多态性

父类:

- 虚函数A

子类:

- 重写虚函数A

当通过父类指针 new 一个子类对象时, 调用虚函数A时, 会调用子类的虚函数A

例如:

```
A* p = new B(); // B继承A
p->A();
```

> 在动态多态的范畴中: 多态必须存在虚函数, 没有虚函数就没有多态

```
class Base{
public:
    virtual void func(){}
};
Base* p = new Base();
p->func(); // 是多态: 因为是根据虚函数的地址来调用的
Base base;
base.func(); // 不是多态: 因为是根据对象的地址来调用的
Base* ybase = &base;
ybase->func(); // 是多态: 因为是根据虚函数的地址来调用的
```

表现形式:

1. 程序中既存在父类也存在子类, 父类中必须有虚函数, 子类中必须重写父类的虚函数

2. 父类指针指向子类对象, 或者父类引用绑定子类对象
    - 父类指针指向子类对象: `Father* p = new Son();`
    - 父类引用绑定子类对象: `Father& p = son;`
3. 通过父类指针或者父类引用, 调用虚函数子类中重写的虚函数
    - `p->func();`

智能指针

指针管理的问题

1. 资源释放了, 但是指针没有置空: 野指针/指针悬挂/踩内存
    - 野指针: 指针指向的内存已经释放, 但是指针没有置空
    - 指针悬挂: 指针指向的内存已经释放, 其他指针还在使用它
    - 踩内存: 资源delete后, 其他地方已经使用了这块内存, 但是指针仍在使用此内存
2. 指针置空了, 但是资源没有释放: 内存泄漏
    - 忘记delete
    - if判断中直接return, 没有delete
    - 异常处理中没有delete
3. 重复释放资源, 引发core dump(段错误)

解决办法

RAII: 利用对象的生命周期来管理资源

- 构造函数中申请资源(如lock)
- 析构函数中释放资源(如unlock)

智能指针的种类

shared_ptr:

- 实现原理: 多个指针指向同一块内存, 内部有引用计数, 当引用计数为0时, 释放内存
- 使用场景: 多个指针指向同一块内存, 且不知道哪个指针会最后释放内存

unique_ptr:

- 实现原理: 独享所有权, 不能被复制, 只能被移动
- 使用场景: 只有一个指针指向内存, 且需要释放内存

weak_ptr:

- 怎么解决shared_ptr的循环引用问题: 循环引用问题是因为shared_ptr的引用计数, unique_ptr没有引用计数, 所以不会有循环引用问题
- 循环引用问题: A指向B, B指向A, 两个指针的引用计数都不为0, 导致内存泄漏

栈内存与堆内存的区别

1. 栈内存

自动分配, 用于存放局部变量, 函数参数以及返回地址等(一般在函数体内直接声明的变量)

- 由编译器自动分配和释放
- 存放函数的参数和局部变量
- 存放函数的返回地址

2. 堆内存

手动分配, 通常用 `new` 或 `malloc` 分配内存, 用 `delete` 或 `free` 释放内存

- 动态分配内存: 通过 `new` 或 `malloc` 分配内存
- 手动管理: 需要程序员手动释放内存

new 和 malloc 的区别

表格:

特性	**new**	**malloc**
内存分配	自动分配适当的大小	需要手动计算分配的大小
返回值	返回指定类型的指针	返回 `void*` 类型的指针
构造函数	会调用构造函数	不会调用构造函数
内存释放	delete 释放内存(同时调用析构函数)	free 释放内存(不调用析构函数)
错误处理	抛出std::bad_alloc异常	返回NULL

三次握手与四次挥手

三次握手:

- 第一次：客户端向服务端发送tcp报文，请求和服务端建立连接。（有客户端初始序列号为j）
- 第二次：服务器向客户端返回tcp报文，确认收到客户端的请求并给出回应。（有ack字段为j+1表示收到了序列号为j的报文，以及服务器的初始序列号k）
- 第三次：客户端响应给服务器一个tcp报文，来确立连接。（有ack为k+1）

四次挥手:

- 第一次：客户端向服务端发送tcp报文，请求断开连接。（有客户端初始序列号为j）
- 第二次：服务器向客户端返回tcp报文，确认收到客户端的请求并给出回应。（有ack字段为j+1表示收到了序列号为j的报文）
- 第三次：服务器向客户端发送tcp报文，请求断开连接。（有服务器初始序列号为k）
- 第四次：客户端向服务器返回tcp报文，确认收到服务器的请求并给出回应。（有ack字段为k+1表示收到了序列号为k的报文）

struct与class的区别

1. 默认访问权限

    - struct: 默认访问权限为public
    - class: 默认访问权限为private

2. 继承方式

    - struct: 默认继承方式为public
    - class: 默认继承方式为private

3. 成员变量的默认访问权限

    - struct: 默认成员变量的访问权限为public
    - class: 默认成员变量的访问权限为private

4. 类型别名

    - struct: 可以使用typedef定义类型别名
    - class: 不能使用typedef定义类型别名

5. 类型转换

    - struct: 可以使用C风格的类型转换
    - class: 不能使用C风格的类型转换

STL标准库

unordered_map与map的区别

unordered_map: 无序map

- 底层实现: 哈希表
- 时间复杂度: O(1)
- 适用场景: 查找速度快, 不需要有序

map: 有序map

- 底层实现: 红黑树
- 时间复杂度: O(logn)
- 适用场景: 需要有序
- 优点: 查找速度快, 插入删除慢

vector与数组的区别

vector: 动态数组

- 动态数组: 可以动态扩容
- 优点: 可以动态扩容, 可以使用STL的算法
- 缺点: 动态扩容会导致内存重新分配, 会导致性能下降

数组: 静态数组

- 静态数组: 长度固定

- 优点: 不会导致内存重新分配, 性能更好

vector扩容:

- 当vector的size达到capacity时, 会重新分配内存, 将原来的元素拷贝到新的内存中
- 扩容策略: 一般是扩容为原来的两倍
- 扩容次数: 一般是log2(n)
- 扩容代价: O(n)
- 扩容时机: 当size达到capacity时, 会触发扩容
- 预留空间: 可以使用reserve预留空间, 避免频繁扩容

vector与list的区别

vector: 动态数组

- 底层实现: 数组
- 优点: 随机访问速度快
- 缺点: 插入删除元素慢

list: 双向链表

- 底层实现: 双向链表
- 优点: 插入删除元素快
- 缺点: 随机访问慢

vector里面reserve和resize的区别

reserve: 预留空间

- 功能: 预留空间, 避免频繁扩容
- 用法: `vector.reserve(n)`
- 作用: 预留n个元素的空间, 但是size不变

resize: 改变大小

- 功能: 改变vector的大小
- 用法: `vector.resize(n)`
- 作用: 改变vector的大小为n, 多出的元素会被初始化

多线程

线程资源冲突的解决办法

1. 互斥锁: 保护临界区资源, 保证同一时间只有一个线程访问
2. 读写锁: 读多写少的场景, 读锁共享, 写锁独占
3. 自旋锁: 适用于临界区执行时间短的场景
4. 条件变量: 线程间通信(当一个线程等待到某个条件时, 另一个线程通知它)
5. 原子操作: 保证操作的原子性, 适用于简单的操作

线程与进程的区别

线程	进程
线程是进程的一部分	进程是程序的一次执行
线程共享进程的资源	进程拥有独立的资源
线程间共享内存	进程间不共享内存

死锁

产生的条件:

1. 互斥条件: 一个资源每次只能被一个进程使用
2. 请求和保持条件: 一个进程因请求资源而阻塞时, 不释放已经获得的资源
3. 不剥夺条件: 进程已获得的资源, 在未使用完之前, 不能被其他进程强行剥夺
4. 循环等待条件: 若干进程之间形成一种头尾相接的循环等待资源关系

左值与右值

左值: 表示某个特定内存位置的表达式

- 可以被赋值(可以出现在赋值号的左边和右边)
- 有明确的内存地址
- 例如: `int a = 10;` 中的a是左值

右值: 不能被赋值的表达式

- 不能出现在赋值号的左边
- 临时变量
- 例如: `int a = 10;` 中的10是右值

++i: i是左值, 返回的是自增之后的变量本身, 可以被赋值 i++: i是右值, 返回的是自增之前的变量, 不能被赋值