C++2

* 面向对象
  + 面向过程和面向对象
    - 面向过程
      * 怎么做：分析解决问题所需的步骤，用函数一步步实现这些步骤，使用的时候一步步调用步骤
      * 性能高：因为类的调用需要实例化，开销较大，消耗资源
      * 代码复用率低，扩展能力差
    - 面向对象
      * 谁来做事情：相比于函数，面向对象是更大的封装，根据问题在一个对象里封装多个方法，注重对象和职责
      * 封装，继承，多态。低耦合，容易维护，复用，扩展。
    - 理解C++是面向对象的编程
  + 用C 实现 C++ 的面向对象特性
    - 封装
      * 结构体
    - 继承
      * 结构体的嵌套
    - 多态
      * 函数指针
  + 双冒号、using和namespace
    - ::作用域符：用在类名称，用以区分不同类的同名成员；全局作用域符号，当全局变量与在局部函数中的某个变量重名时，用::区分
    - 命名空间namespace：C++标准程序库的所有标识符都定义在namespace中，为解决标识符命名冲突问题，通过namespace MY{int x;...};可以全局嵌套定义
    - using使命名空间可用，接下来就不用写::作用域标识符，也不能定义同名变量
  + 对象的引用方式
    - 成员引用
      * MyClass m; m.show();
    - 对象指针引用
      * MyClass \* mp; mp->show();//或者， (\*mp).show();
  + 类默认的六个成员函数
    - 构造函数
    - 拷贝构造函数
    - 析构函数
    - 赋值操作符=重载
    - 取地址操作符&重载
      * 返回this指针
    - const修饰的取地址操作符&重载
      * 返回this指针
  + const成员
    - 常成员变量
      * 只能通过构造函数的初始化列表进行初始化
      * C++11开始可以在类中初始化
        + const int a = 666;//C++11开始允许static const int b = 999;//静态常成员变量，可以在类中直接初始化
    - 常成员函数
      * 在声明和定义的时候在函数头部的结尾加上 const 关键字
      * 不能修改类的非静态成员变量
        + const修饰的this指针
      * 不能调用非const修饰的成员函数
  + 构造函数
    - 与类同名，无返回值，可以被内联（inline），可以在类外定义
    - 建立一个对象，构造函数就会被自动调用；用于初始化对象，例如对数据成员进行赋值设置类的属性
    - 成员初始化列表
      * 必须要用初始化列表的情况
        + 非静态const数据成员

和引用一样，初始化之后不能改变，不能对其赋值（声明和赋值是一起的）

* + - * + 引用&数据成员
        + 成员类型是没有默认构造函数的类
      * 为什么更快
        + 针对初始化类类型成员更快

调用构造函数时，对象在程序进入构造函数函数体之前被创建。被分配了内存空间，但是还没有初始化。也就是说，调用构造函数的时候，先创建对象，再进入函数体。若不使用初始化成员列表会浪费一次赋值机会，进入函数体后另外初始化（赋值）。

* + - 如何禁止构造函数的使用
      * 私有化
    - 如何减少构造函数开销
  + 拷贝构造函数
    - classname (const classname &obj){}
      * 是否可以是值传递
        + 不行，值传递又要调用拷贝构造函数，造成死循环。
    - 用途
      * 初始化，用一个已知的对象来初始化生成一个一模一样的对象
      * 值传递，函数参数为对象，复制对象把它作为参数传递
      * 返回值为对象，复制对象并从函数返回这个对象
      * 编译器生成临时对象
    - 拷贝构造> 有参构造 > 默认构造
      * 写了拷贝构造，编译器就不提供其他构造函数
      * 写了有参构造，编译器不提供默认构造，但是提供默认拷贝
    - 拷贝构造函数参数必须为const引用
    - 不要用拷贝构造函数初始化匿名对象
    - 如何避免使用拷贝函数
      * 使用引用传递参数，而不用值传递
      * 使拷贝构造函数私有化
  + 深拷贝与浅拷贝
    - 浅拷贝
      * 默认拷贝构造函数，只是对指针的拷贝，拷贝后两个指针指向同一个内存空间
    - 深拷贝
      * 自定义拷贝构造函数，重新开辟内存空间，使指针成员有自己的内存空间
    - 深拷贝不仅是对指针进行拷贝，还对指针所指向的内容进行拷贝，经深拷贝后的指针是指向不同地址的，
    - 深拷贝避免了同一块内存释放多次，造成内存泄漏和崩溃
  + 析构函数
    - ~MyClass();//同类名，无返回值，多了~
    - 析构函数不能被重载，只能有一个，不能带参数
    - 释放一个对象，在对象删除前用它做一些清理工作
    - 在多态中，析构函数要声明为虚函数，避免内存泄漏
    - 析构函数中不能抛出异常
      * Effective C++条款8
  + 封装
    - 抽象成一个类，将数据和操作封装在一起，使用public，protected和private描述成员的访问属性
  + 类成员的访问属性
    - 访问方式
      * public
        + proteected

private

* + - 类内/友元
      * 可以
        + 可以

可以

* + - 派生类
      * 可以
        + 可以

不可以

* + - 对象、类外
      * 可以
        + 不可以

不可以

* + - private和protected对外表现特性一样，只是对派生类不同
  + this指针
    - 指向被调用对象本身
      * 每个非静态成员函数（包括构造函数/析构函数）都有一个this指针
        + 所有成员函数的**隐含参数**
      * this是调用对象的地址，\*this才是被调用对象
      * 每个对象都有自己的非静态数据成员，但是成员函数共享一个通过this指针指向被调用的成员函数所属的对象，调用成员函数找到自己的数据成员
  + 静态成员
    - 静态成员static允许使用类名直接访问，且也可通过对象访问
    - 静态成员变量
      * 普通的成员函数只有对象创建了才会被分配内存，而静态数据成员属于整个类即使没有对象创建，类的静态成员变量也存在
      * 类内定义，类外初始化
    - 静态成员函数
      * 只能访问静态成员变量和静态成员函数
      * 静态成员函数不与任何对象绑定，不能使用this指针，不能使用const、virtual
  + 友元
    - friend让指定的类或函数可以直接访问类的私有数据，赋予函数与类成员函数相同的访问权限
    - 友元关系不能继承
    - 使用
      * 在类中声明，friend+函数原型
        + friend 返回值类型 函数名(参数表);
        + friend 返回值类型 其他类的类名::成员函数名(参数表);

不能把其他类的私有成员函数声明为友元

* + - * + friend class 类名;

一个类 A 可以将另一个类 B 声明为自己的友元类 B 的所有成员函数就都可以访问类 A 对象的私有成员

* + - * 在类外定义函数，不需要使用friend
      * 使用场景
        + 要访问非static成员时，需要对象做参数；要访问static成员或全局变量时，则不需要对象做参数；如果做参数的对象是全局对象，则不需要对象做参数
  + 重载(overload)、覆盖(overwrite)和重定义
    - 重载
      * 函数名相同，参数列表必须不同，返回值可以不同，同一个作用域
      * 将功能相似的函数进行重载，减少对于函数名的记忆
      * 运算符重载
        + 输入输出运算符重载

声明为友元

返回值是输入输出对象的引用

* + - * + 对++和--重载

运算符分为前置和后置，在后面跟上（int）表示为后置

* + - * + 不能重载的运算符

.

?:

sizeof

::

\*

* + - 覆盖（重写）
      * 派生类重新定义基类的虚函数
      * 不同作用域，基类和派生类
      * 函数名，参数列表，返回值都必须相同，发生在多态期间
      * 重写函数的访问修饰符可以不同。尽管 virtual 是 private 的，派生类中重写改写为 public,protected 也是可以的
    - 重定义（隐藏）
      * 处于不同的作用域中，分别是基类和派生类
      * 函数名相同，返回值可以不同
      * 参数不同
        + 不论有无 virtual 关键字，基类的函数将被隐藏
      * 参数相同
        + 基类函数没有 virtual关键字，基类的函数被隐藏
      * 在基类和派生类中只要不构成重写就是重定义
  + 继承
    - 概念
      * 是一个类可以从现有类中派生，而不必重新定义一个类
    - 继承关系
      * 派生类型
        + public

protected

private

* + - * 基类公有
        + public

protected

private

* + - * 基类保护
        + protected

protected

private

* + - * 基类私有
        + private

private

private

* + - 访问顺序
      * 构造函数
        + 基类->派生类
      * 析构函数
        + 派生类->基类
    - 多重继承会出现什么状况，如何解决
  + 多态
    - 概念
      * 不同功能的函数可以用同一函数名，一个接口，多种方法
    - 类型
      * 编译时多态
        + 编译期间确定函数的调用地址

通过函数重载实现，调用速度快，效率高，但不灵活

泛型编程

* + - * 运行时多态
        + 程序运行时确认函数的调用地址

通过虚函数实现

* + 虚函数
    - 实现多态的关键，派生类重写基类的虚函数，改变函数的功能
    - 使用成本
      * 使用虚函数会增加一个指向虚函数的指针的内存开销
      * 对于每个类，编译器都会创建一个虚函数地址表（数组）
      * 对于每个函数调用，都要执行额外操作：到虚函数表中查找地址
    - 只有类的非静态成员函数才能为虚函数
    - 析构函数通常是虚函数
      * 防止内存泄漏
    - 构造函数不能是虚函数
      * 虚函数的调用过程
        + 找到对象内存里的虚函数指针
        + 通过虚函数指针找到虚函数表
        + 在虚函数表里找到该虚函数的地址，并调用
      * 构造函数的作用
        + 创建对象并初始化（实例化），初始化之后才有虚函数指针
      * 对象的生成
        + 开辟内存空间
        + 编译器调用构造函数进行初始化（实例化）

在调用构造函数的时候已经有了内存，只是没有实例化此时没有虚函数指针

* + - 虚函数表
      * 每一个有虚函数的类（以及其派生类）都有一个虚函数表（数组），表里存放着虚函数的地址
      * 该类的任何对象都存放着指向该虚函数表的指针
      * 虚函数表是在编译的过程中创建
      * 虚函数指针
        + 在运行时创建，实例化对象时
        + 存放在对象的开头，内存的前四个字节
      * 虚函数表的存放位置
        + 存放在目标文件和可执行文件的常量段（VS）
        + 存放在可执行文件的只读数据段（.rodata/Linux）
      * 单继承和多继承的表结构
        + 多继承就会有多个虚函数表。因为每个父类的虚函数是不同的，指针也是不同的
        + 每一个基类都会有自己的虚函数表，派生类的虚函数表的数量根据继承的基类的数量来定
        + 派生类的虚函数表的顺序，和继承时的顺序相同
        + 派生类自己的虚函数放在第一个虚函数表的后面，顺序也是和定义时顺序相同
        + 对于派生类如果要覆盖父类中的虚函数，那么会在虚函数表中代替其位置
    - 父类构造函数中能调用虚函数，但是会屏蔽多态机制
      * 不要在构造函数和析构函数中使用虚函数
        + 《Effective C++ 》条款9

虚函数调用从不会被传递到派生类中

* + - * 会把基类中的虚函数作为普通函数调用，而不会调用派生类中被重写的函数
      * 原因
        + 父类的构造函数在子类构造函数之前执行，此时子类对象还未实例化，没有虚函数指针等，为了避免调用到未初始化的内存，C++规定：这种情况下使用的是静态绑定，调用父类函数
    - 安全性问题
      * 通过父类型指针访问子类自己的虚函数（错误）
      * 子类访问父类非public的虚函数（通过虚函数表的方式）
    - 虚函数中的默认参数，比如父类中是0，子类是1，那么调用虚函数的时候，默认参数是0！，跟随父类
  + 抽象类（纯虚函数）
    - 抽象类至少有一个纯虚函数
    - 不能被实例化，只能作为基类来派生子类
      * 子类若不是抽象类，必须实现父类中所有的纯虚函数
    - 纯虚函数
      * 不具体实现的虚成员函数：virtual 类型 函数名（参数列表）=0；
    - 可以定义指向抽象类的指针变量，用来实现多态
  + 模板
    - 函数模板
      * 代表了一个函数家族，该函数与类型无关，根据传递的实参类型，来决定其功能
      * 如template<class T ,int A> 返回类型 函数名 (T a, T b, T num[A]) // 其中模板类型参数class也能用typedef
      * 模板函数可以定义为inline函数，inline关键字必须放在模板形参之后，返回值之前，不能放在template之前
    - 模板是泛型编程的基础，泛型编程就是与类型无关的编程
    - 类模板
      * 使得类的一些数据成员和成员函数的参数或返回值可以取任意数据类型
      * template<class T> class 类名
    - 类型萃取
      * 通过类型萃取的方式来区分自定义类型和内置类型
      * 类型萃取是通过在模板的基础上区分内置类型和其他类型
      * 原理是将内置类型进行特化再进行具体区分
    - 全特化与偏特化
    - 可变参数模板
  + 类所占内存
    - 空的类会占1字节的内存
      * 空类也会被实例化，所以编译器会给空类隐含的添加一个字节C++要求每个实例在内存中都有独一无二的地址
    - 空类自动生成的6个函数
      * 两个构造，一个析构，三个重载
    - 占内存
      * 非静态成员变量
        + 注意内存对齐
      * 虚函数指针
    - 不占内存
      * 成员函数
        + 函数存放在代码区内联函数也是，它只是在调用的时候避免了函数的跳转和保护现场的开销
      * 友元函数
        + 友元函数只是在类中声明，属于类外
      * 静态成员变量
        + 存放在全局变量区（静态区）
  + 设计类
    - 不能被继承
      * 将构造函数析构函数申明为私有
      * 使用静态函数来创建和释放类的实例
    - 类的创建
      * 静态创建，在栈上
        + 类名 对象名；A a;

只能在栈上创建

operator new和operator delete访问权限修改为私有

* + - * 动态创建，在堆上
        + 使用new

只能在堆上创建

设置构造函数和析构函数的访问权限为protected调用静态成员函数创建和销毁对象

将析构函数声明为私有

对象建立在栈上时，编译器分配空间，调用构造函数来实例化对象，当对象使用完之后，编译器还会访问析构函数来释放对象的空间，编译器管理了对象的整个生命周期，编译器为对象分配空间的时候，只要是非静态成员函数都会检查，这时检查析构函数时，析构函数不能访问，编译器就无法在栈上分配对象。

* + - * 只能创建一个对象
        + 在类中创建一个静态变量，用来限制可创建实例的数量
    - 禁止类被实例化
      * 构造函数私有化
      * 纯虚函数
  + 静态绑定和动态绑定
    - 静态绑定（默认）
      * 在编译时刻，根据指针或引用变量的静态类型来决定成员函数属于哪一个类
    - 动态绑定（多态）
      * 在运行时刻，根据指针或引用变量实际指向或引用的对象类型（动态类型）来确定成员函数属于哪一个类
        + 类的定义中成员函数声明为虚函数
        + 通过引用或指针来访问对象的虚函数
  + 实例化对象过程
    - 哪几个阶段