目录

[C++内存分配 2](#_Toc99278050)

[堆和栈的区别 2](#_Toc99278051)

[内存泄漏 3](#_Toc99278052)

[C++文件从代码到可执行程序的过程 4](#_Toc99278053)

[include <> 和“” 5](#_Toc99278054)

[定义与申明 5](#_Toc99278055)

[引用和指针 6](#_Toc99278056)

[数组和指针 6](#_Toc99278057)

[野指针和悬空指针 7](#_Toc99278058)

[递增递减运算符 7](#_Toc99278059)

[显式类型转换 7](#_Toc99278060)

[数组和引用结合 7](#_Toc99278061)

[数组、函数和指针结合 7](#_Toc99278062)

[函数传递参数的方式 8](#_Toc99278063)

[指针传递和引用传递 8](#_Toc99278064)

[C++函数压栈 8](#_Toc99278065)

[构造函数、析构函数、虚函数可不可以声明为内联函数 8](#_Toc99278066)

[为什么不能把函数都写成内联函数 9](#_Toc99278067)

[访问说明符 9](#_Toc99278068)

[lambda表达式 9](#_Toc99278069)

[智能指针 10](#_Toc99278070)

[循环引用问题 11](#_Toc99278071)

[拷贝构造函数 12](#_Toc99278072)

[深拷贝和浅拷贝 12](#_Toc99278073)

[移动和拷贝 12](#_Toc99278074)

[左值和右值 13](#_Toc99278075)

[面向对象三大特性 13](#_Toc99278076)

[类的大小 14](#_Toc99278077)

[静态绑定和动态绑定 14](#_Toc99278078)

[构造函数为什么不能是虚函数 14](#_Toc99278079)

[析构函数为什么一般写成虚函数 15](#_Toc99278080)

[哪些函数不能是虚函数 15](#_Toc99278081)

[构造函数或者析构函数可以调用虚函数吗 15](#_Toc99278082)

[构造函数和析构函数顺序 16](#_Toc99278083)

[虚函数原理（多态） 16](#_Toc99278084)

[虚函数为什么不能是静态的 17](#_Toc99278085)

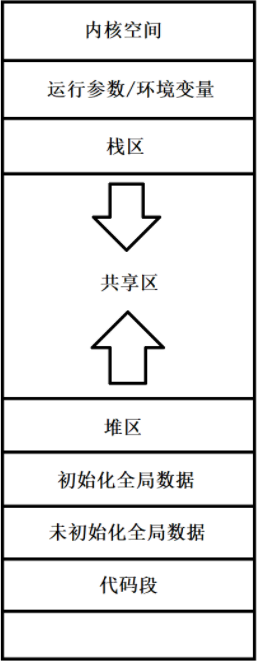
[纯虚函数与虚函数 17](#_Toc99278086)

[抽象基类 17](#_Toc99278087)

[继承中的成员访问权限 18](#_Toc99278088)

[重载、重写（覆盖）和隐藏 18](#_Toc99278089)

## C++内存分配

内存分配：

1、**从静态存储区域分配**：内存在程序编译的时候就已经分配好了，这块内存在程序的整个运行期间都存在。例如全局变量，static静态成员变量

2、**从栈上创建**：执行函数时，函数内部变量的存储单位可以在栈上创建，函数执行结束时，这些存储单元自动释放。

3、**从堆上分配**：也称为动态内存分配。程序在运行的时候用malloc或new申请任意多少内存，程序员自己负责在何时用free或delete来释放这块内存。动态内存的生命周期由程序员决定，使用非常灵活，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现泄漏，频繁分配不同大小空间会产生内存碎片



**栈区**：由编译器自动分配释放，存放为函数运行的局部变量，函数参数，返回数据，返回地址等。

**堆区**：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。

**全局数据区**：也叫做静态区，存放全局变量，静态数据。程序结束后由系统释放

**文字常量区**：可以理解为常量区，常量字符串存放这里。程序结束后由系统释放

**程序代码区**：存放函数体的二进制代码。但是代码段中也分为代码段和数据段。

## 堆和栈的区别

* 管理方式：栈是在函数运行时，由系统自动分配；而堆是通过程序员自己调用malloc函数或者new运算符去申请一个需要的大小空间。
* 空间大小：栈的空间大小并不大，一般最多为2M，超过之后会报Overflow错误。堆的空间非常大，最大可到达4G，可操作的空间非常大。
* 能否产生碎片：栈的操作与数据结构中的栈用法是类似的。‘后进先出’的原则，以至于不可能有一个空的内存块从栈被弹出。因为在它弹出之前，在它上面的后进栈的数据已经被弹出。它是严格按照栈的规则来执行。但是堆是通过new/malloc随机申请的空间，频繁的调用它们，则会产生大量的内存碎片。这是不可避免地。
* 生长方向：栈的生长方向是由高地址向低地址增长，是自上而下的。堆的生长方向是由低地址向高地址增长，是自下而上的。
* 分配方式：堆都是动态分配的，没有静态分配。但是栈有两种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由malloc函数实现，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行和释放，无需程序员进行操作。
* 分配效率 栈是机器系统提供的数据结构，计算机底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行。这就决定了栈有着很高的效率。堆需要通过C/C++的库函数进行一个复杂的算法，在对内存中搜寻一个足够大小的空间，如果没有足够的空间(内存碎片空间太多)，就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低的多。

## 内存泄漏

* 定义：一般我们常说的内存泄漏是指堆内存的泄漏。堆内存是指程序从堆中分配的，大小任意的(内存块的大小可以在程序运行期决定)内存块，使用完后必须显式释放的内存。应用程序般使用malloc,、realloc、 new等函数从堆中分配到块内存，使用完后，程序必须负责相应的调用free或delete释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了
* 避免方式：

1、计数法：使用new或者malloc时，让该数+1，delete或free时，该数-1，程序执行完打印这个计数，如果不为0则表示存在内存泄露

2、一定要将基类的析构函数声明为虚函数

3、有new就有delete，有malloc就有free，保证它们一定成对出现，对象数组的释放一定要用delete []

* 检测

1、Linux中可以使用swap指令观察还有多少交换空间，如果在减少可能存在泄漏

2、用于内存调试、内存泄漏检测以及性能分析的软件开发工具valgrind（Linux）或者CRT库（Windows）

## C++文件从代码到可执行程序的过程

* 预编译：主要处理源代码文件中的以“#”开头的预编译指令。规则如下：

1、删除所有的#define，展开所有的宏定义。

2、处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。

3、处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件。

4、删除所有的注释，“//”和“/\*\*/”。

5、保留所有的#pragma编译器指令，编译器需要用到他们，如：#pragma once是为了防止有文件被重复引用。

6、添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是能够显示行号。

* 编译：把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。

1、词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分割成一系列的记号。

2、语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的语法树是一种以表达式为节点的树。

3、语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定的语义。

4、优化：源代码级别的一个优化过程。

5、目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示。

6、目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等。

* 汇编：将汇编代码转变成机器可以执行的指令(机器码文件)。

汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过来，汇编过程有汇编器as完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Windows下)、xxx.obj(Linux下)

* 链接：将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：

1、静态链接：函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

2、动态链接：动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

## include <> 和“”

* <>用于引用编译器提供的头文件，在编译器指定include目录找不到则报错；
* “”主要用于项目中自己实现的头文件引用，先在项目目录寻找，找不到会去编译器目录中找，找不到报错

## 定义与申明

* 变量：从编译原理上来说，声明是仅仅告诉编译器，有个某类型的变量会被使用，但是编译器并不会为它分配任何内存。而定义就是分配了内存。
* 函数：

1、声明：一般在头文件里，对编译器说：这里我有一个函数叫function()让编译器知道这个函数的存在。

2、定义：一般在源文件里，具体就是函数的实现过程写明函数体。

## 引用和指针

引用：定义引用时，程序把引用和它的初始值绑定在一起，而不是将初始值拷贝给引用

指针：本身是一个对象，指向另外一种类型的复合类型

区别：

1、指针本身是一个对象，允许对指针赋值和拷贝，而且在生命周期内可以指向几个不同的对象；而引用只能绑定到一个对象

2、指针不需要在定义时赋初始值（最好初始化）；而引用必须初始化

3、指针有自己的内存空间；而引用只是一个别名

4、不存在指向空值的引用；但存在指向空值的指针

5、作为参数时，传指针的实质是传值，传递的值是指针的地址；传引用的实质是传地址，传递的是变量的地址

## 数组和指针

数组：存储多个相同类型数据的集合

指针：相当于一个变量，但是存放的是变量在内存中的地址

区别：

1、赋值

* 同类型指针之间可以相互赋值
* 数组只能一个一个赋值或拷贝

2、存储方式

* 数组在内存中是连续存放的，开辟一块连续的内存空间。数组是根据数组的下进行访问的，多维数组在内存中是按照一维数组存储的，只是在逻辑上是多维的。
* 指针很灵活，它可以指向任意类型的数据。指针的类型说明了它所指向地址空间的内存。

3、求sizeof

* 数组所占存储空间的内存：sizeof（数组名）

数组的大小：sizeof（数组名）/sizeof（数据类型）

* 在32位平台下，无论指针的类型是什么，sizeof（指针名）都是4，在64位平台下，无论指针的类型是什么，sizeof（指针名）都是8

4、访问数据

* 数组是直接访问
* 指针是间接访问，首先获得指针的内容，然后根据指针去提取数据

## 野指针和悬空指针

野指针：没有被初始化过的指针；指针尽量初始化，要么置空

悬空指针：指针指向的内存已被释放的一种指针；释放操作后立即置空

## 递增递减运算符

* ++i ：先自增后返回
* i++：先返回再自增

## 显式类型转换

形式：case-name<type>(expression);

* static\_cast：具有明确定义的显式类型转换，都可用除了底层const
* const\_cast：只能改变运算对象的底层const，为对象添加或去除const特性
* reinterpret\_cast：常用于改变指针或引用的类型，整型->指针，指针->数组等
* dynamic\_cast：其他三种都是编译时完成，这个是运行时处理的，要进行类型检查，不能用于内置的基本数据类型的强制转换，专门用于派生类之间的转换，返回的是指向类的指针或引用，基类中一定要有虚函数

## 数组和引用结合

* &arr[10]：引用的数组，数组中10个引用类型
* (&arr)[10]：10个整数的数组的引用

## 数组、函数和指针结合

* int (\*p)[4] // 数组指针，p是指针，指向一维数组，4个int元素
* int \*p[4] // 指针数组，p是数组，数组元素是指针，4个int指针
* bool (\*pf) (const string&); // 函数指针，指向函数的指针变量，指向一个具体的函数，有了它可以使用指针变量调用函数

## 函数传递参数的方式

* 值传递：形参是实参的拷贝，函数内部对形参的操作不影响外部的实参
* 指针传递：传地址值，形参是指向实参地址的指针，当对形参的指向操作时，相当于对实参本身操作
* 引用传递：实质上就是把引用对象的地址放到了开辟的栈空间中，可以改变外部实参

## 指针传递和引用传递

* 指针传递
* 虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量（地址），那就得使用指向指针的指针或者指针引用。

## C++函数压栈

main函数调用func函数：

当函数从入口函数main函数开始执行时，编译器会将我们操作系统的运行状态，main函数的返回地址、 main的参数、main函数中的变量、进行依次压栈；

当main函数开始调用func()函数时，编译器此时会将main函数的运行状态进行压栈，再将func()函数的返回地址、func()函数的参数从右到左、func()定义变量依次压栈；

## 构造函数、析构函数、虚函数可不可以声明为内联函数

* 构造函数、析构函数

class中的函数默认是inline型的，编译器也只是有选择性的inline，将构造函数和析构函数声明为内联函数是没有什么意义的。并且，编译器会在构造和析构函数中添加额外的操作（申请/释放内存，构造/析构对象等），致使构造函数/析构函数并不像看上去的那么精简，因此译器并不真正对声明为inline的构造和析构函数进行内联操作

* 虚函数

inline是编译器决定的，虚函数是运行时决定的，如果虚函数在编译器就能够决定将要调用哪个函数时，就能够内联，当用对象调用虚函数（此时不具有多态性）时，就内联展开

综上，当是指向派生类的指针（多态性）调用声明为inline的虚函数时，不会内联展开；当是对象本身调用虚函数时，会内联展开。当然前提依然是函数并不复杂的情况下。

## 为什么不能把函数都写成内联函数

内联函数以代码复杂为代价，它以省去函数调用的开销来提高执行效率。所以一方面如果内联函数体内代码执行时间相比函数调用开销较大，则没有太大的意义；另一方面每一处内联函数的调用都要复制代码，消耗更多的内存空间，因此以下情况不宜使用内联函数：

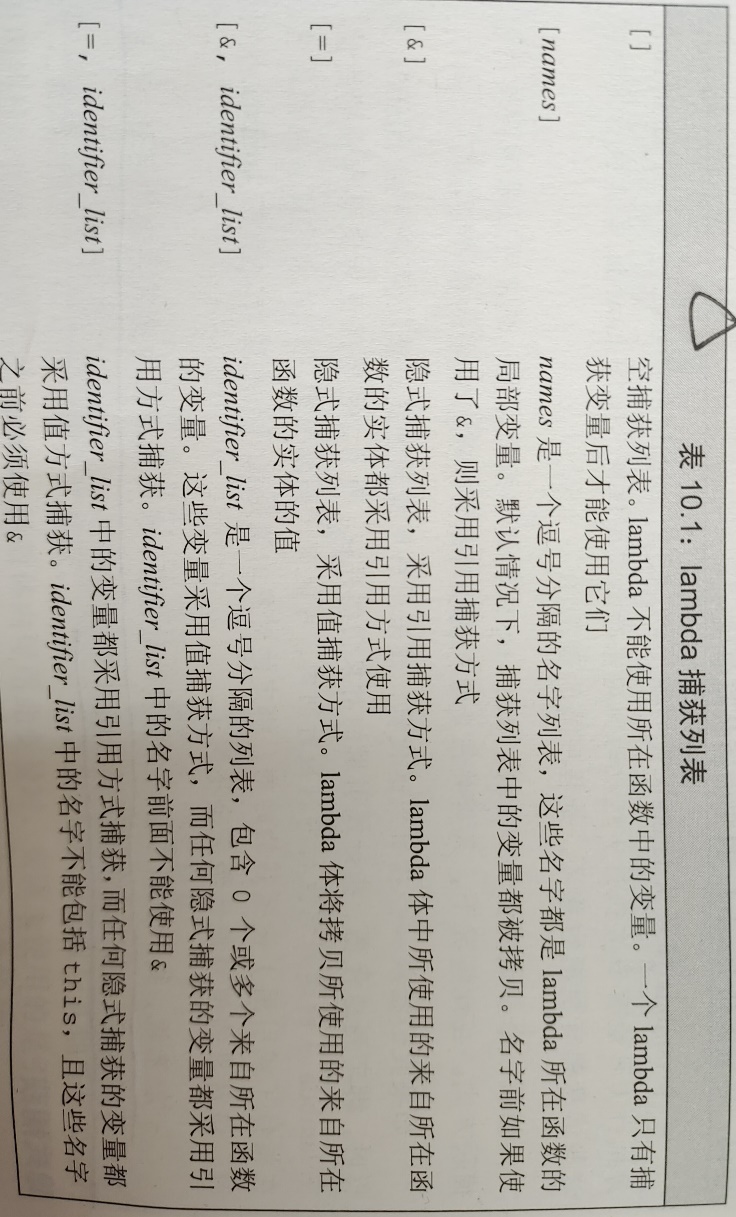
* 函数体内的代码比较长，将导致内存消耗代价
* 函数体内有循环，函数执行时间要比函数调用开销大

## 访问说明符

* public的变量和函数在类的内部外部都可以访问
* protected的变量和函数只能在类的内部和其派生类中访问
* private修饰的元素只能在类内访问
* friend：可以访问类的所有成员

## lambda表达式

[capture list] (parameter list) -> return type {function body}



## 智能指针

原理：智能指针是一个类，用来存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏。动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源

分类：

* shared\_ptr

实现原理：采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

用法：

1、智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针

2、每次创建类的新对象时，初始化指针并将引用计数置为1

3、当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数

4、对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数

5、调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）

* unique\_ptr

unique\_ptr采用的是独享所有权语义，一个非空的unique\_ptr总是拥有它所指向的资源。转移一个unique\_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique\_ptr不支持普通的拷贝和赋值操作，不能用在STL标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将要被销毁）；如果你拷贝一个unique\_ptr，那么拷贝结束后，这两个unique\_ptr都会指向相同的资源，造成在结束时对同一内存指针多次释放而导致程序崩溃。

* weak\_ptr

weak\_ptr：弱引用。引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用weak\_ptr打破环形引用。weak\_ptr是一个弱引用，它是为了配合shared\_ptr而引入的一种智能指针，它指向一个由shared\_ptr管理的对象而不影响所指对象的生命周期，也就是说，它只引用，不计数。如果一块内存被shared\_ptr和weak\_ptr同时引用，当所有shared\_ptr析构了之后，不管还有没有weak\_ptr引用该内存，内存也会被释放。所以weak\_ptr不保证它指向的内存一定是有效的，在使用之前可使用函数lock()检查weak\_ptr是否为空指针。

* auto\_ptr

主要是为了解决“有异常抛出时发生内存泄漏”的问题 。因为发生异常而无法正常释放内存。

auto\_ptr有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而unique\_ptr则无拷贝语义，但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用std::move()进行转移。

auto\_ptr不支持拷贝和赋值操作，不能用在STL标准容器中。STL容器中的元素经常要支持拷贝、赋值操作，在这过程中auto\_ptr会传递所有权，所以不能在STL中使用。

## 循环引用问题

循环引用是指使用多个智能指针share\_ptr时，出现了指针之间相互指向，从而形成环的情况，有点类似于死锁的情况，这种情况下，智能指针往往不能正常调用对象的析构函数，从而造成内存泄漏。如果不可避免，可以使用弱指针——weak\_ptr，它不增加引用计数，只要出了作用域就会自动析构。

## 拷贝构造函数

* 为什么拷贝构造函数必须传引用不能传值？

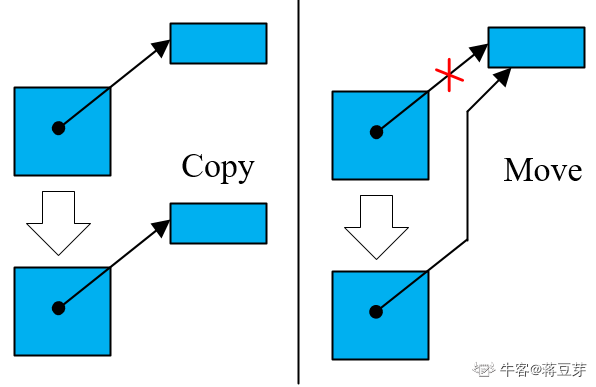
如果一个对象以值进行传递，编译器会生成代码调用它的拷贝构造函数生成一个副本，如果类A的拷贝构造函数不是引用传递，那么就又需要创建一个传递给拷贝构造函数的副本，这是一个无限递归。

## 深拷贝和浅拷贝

* 浅拷贝：浅拷贝只是拷贝一个指针，并没有新开辟一个地址，拷贝的指针和原来的指针指向同一块地址，如果原来的指针所指向的资源释放了，那么再释放浅拷贝的指针的资源就会出现错误。
* 深拷贝：深拷贝不仅拷贝值，还开辟出一块新的空间用来存放新的值，即使原先的对象被析构掉，释放内存了也不会影响到深拷贝得到的值。在自己实现拷贝赋值的时候，如果有指针变量的话是需要自己实现深拷贝的

## 移动和拷贝

移动构造函数与拷贝构造不同，它并不是重新分配一块新的空间同时将要拷贝的对象复制过来，而是"拿"了过来，将自己的指针指向别人的资源，然后将别人的指针修改为nullptr，这一步很重要，如果不将别人的指针修改为空，那么临时对象析构的时候就会释放掉这个资源，那么就没有“拿”过来。



## 左值和右值

* 左值：表示存储在计算机内存的对象，而不是常量或计算的结果。或者说左值是代表一个内存地址值，并且通过这个内存地址，就可以对内存进行读并且写（主要是能写）操作；左值代表地址值。
* 右值：当一个符号或者常量放在操作符右边的时候，计算机就读取他们的“右值”，也就是其代表的真实值。右值代表数据值。

1、左值持久；右值短暂

如何区分：左值有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象，右值引用指向将要被销毁的对象

2、变量是左值：我们不能将一个右值引用类型绑定到另一个变量或者右值引用类型上

3、标准库move函数：使用std::move函数可显式地将一个左值转换为对应的右值引用类型

## 面向对象三大特性

* 封装

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变。

* 继承

是指可以让某个类型的对象获得另一个类型的对象的属性的方法。它可以使用现有类的所有功能，并在无需重新编写原来类的情况下对这些功能进行扩展。

* 多态

就是向不同的对象发送同一个消息，不同的对象在接收时会产生不同的行为。简单一句话：允许将子类类型的指针赋值给父类类型的指针（重载实现编译时多态，虚函数实现运行时多态）

## 类的大小

* C++空类的大小不为0，不同编译器设置不一样，vs设置为1
* 带有虚函数的C++类大小不为1，因为每一个对象会有一个vptr指向虚函数表，具体大小根据指针大小确定
* C++中要求对于类的每个实例都必须有独一无二的地址,那么编译器自动为空类分配一个字节大小，这样便保证了每个实例均有独一无二的内存地址
* 当该空白类作为基类时，该类的大小就优化为0了，子类的大小就是子类本身的大小。这就是所谓的空白基类最优化

## 静态绑定和动态绑定

对象的静态类型：对象在声明时采用的类型。是在编译期确定的。

对象的动态类型：所指对象的实际类型。是在运行期决定的。

对象的动态类型可以更改，但是静态类型无法更改

* 静态绑定：绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期；
* 动态绑定：绑定的是动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期

## 构造函数为什么不能是虚函数

* 创建一个对象时需要确定对象的类型，而虚函数是在运行时动态确定其类型的。在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型
* 虚函数的调用需要虚函数表指针vptr，而该指针存放在对象的内存空间中，若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表vtable地址用来调用虚构造函数了
* 虚函数的作用在于通过父类的指针或者引用调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类或者引用去调用，因此就规定构造函数不能是虚函数

## 析构函数为什么一般写成虚函数

由于类的多态性，基类指针可以指向派生类的对象，如果删除该基类的指针，就会调用该指针指向的派生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全，造成内存泄漏。

## 哪些函数不能是虚函数

* 构造函数：构造函数初始化对象，派生类必须知道基类函数干了什么，才能进行构造；当有虚函数时，每一个类有一个虚表，每一个对象有一个虚表指针，虚表指针在构造函数中初始化；
* 内联函数：内联函数表示在编译阶段进行函数体的替换操作，而虚函数意味着在运行期间进行类型确定，所以内联函数不能是虚函数
* 静态函数：静态函数不属于对象属于类，静态成员函数没有this指针，因此静态函数设置为虚函数没有任何意义
* 友元函数：友元函数不属于类的成员函数，不能被继承。对于没有继承特性的函数没有虚函数的说法
* 普通函数：普通函数不属于类的成员函数，不具有继承特性，因此普通函数没有虚函数

## 构造函数或者析构函数可以调用虚函数吗

* 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本；
* 因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数时不安全的，故而C++不会进行动态联编
* 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函数没有任何意义

## 构造函数和析构函数顺序

* 构造函数顺序

1、基类构造函数：如果有多个基类，则构造函数的调用顺序是某类是类派生表中出现的顺序，而不是它们在成员初始化列表中的顺序

2、成员类对象构造函数：如果有多个成员类对象，则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们在成员初始化列表中的顺序

3、派生类构造函数

* 析构函数

1、调用派生类的析构函数

2、调用成员类对象的析构函数

3、调用基类的析构函数

## 虚函数原理（多态）

C++中虚函数表位于只读数据段（.rodata），也就是C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是C++内存模型中的代码区。

* 编译器在发现基类中有虚函数时，会自动为每个含有虚函数的类生成一份虚表，该表是一个一维数组，虚表里保存了虚函数的入口地址
* 编译器会在每个对象的前四个字节中保存一个虚表指针，即vptr，指向对象所属类的虚表。在构造时，根据对象的类型去初始化虚指针vptr，从而让vptr指向正确的虚表，从而在调用虚函数时，能找到正确的函数
* 所谓的合适时机，在派生类定义对象时，程序运行会自动调用构造函数，在构造函数中创建虚表并对虚表初始化。在构造子类对象时，会先调用父类的构造函数，此时，编译器只“看到了”父类，并为父类对象初始化虚表指针，令它指向父类的虚表；当调用子类的构造函数时，为子类对象初始化虚表指针，令它指向子类的虚表
* 当派生类对基类的虚函数没有重写时，派生类的虚表指针指向的是基类的虚表；当派生类对基类的虚函数重写时，派生类的虚表指针指向的是自身的虚表；当派生类中有自己的虚函数时，在自己的虚表中将此虚函数地址添加在后面



## 虚函数为什么不能是静态的

* static成员不属于任何类对象或类实例，所以即使给此函数加上virutal也是没有任何意义的
* 静态与非静态成员函数之间有一个主要的区别，那就是静态成员函数没有this指针。虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来访问它，因为它是类的一个成员，并且vptr指向保存虚函数地址的vtable.对于静态成员函数，它没有this指针，所以无法访问vptr

## 纯虚函数与虚函数

纯虚函数：首先它是虚函数，通过=0声明，不能在类的内部为纯虚函数提供函数体。它的函数指针会被存在虚函数表中，由于纯虚函数并没有具体的函数体，因此它在虚函数表中的值就为0，而具有函数体的虚函数则是函数的具体地址

## 抽象基类

定义：含有纯虚函数的类是抽象基类

作用：抽象基类负责定义接口，不能给出虚函数具体意义的实现，派生类负责重写该接口

* 抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。
* 不能创建抽象基类的对象。一个纯虚函数不需要（但是可以，须定义在类的外部）被定义

## 继承中的成员访问权限

* public：基类成员在派生类中的访问权限保持不变，也就是说，基类中的成员访问权限，在派生类中仍然保持原来的访问权限
* private：基类所有成员在派生类中的访问权限都会变为私有(private)权限
* protected：类的公有成员和保护成员在派生类中的访问权限都会变为保护(protected)权限，私有成员在派生类中的访问权限仍然是私有(private)权限

## 重载、重写（覆盖）和隐藏

* 重载：是指在同一范围定义中的同名成员函数才存在重载关系。主要特点是函数名相同，参数类型和数目有所不同，不能出现参数个数和类型均相同，仅仅依靠返回值不同来区分的函数。重载和函数成员是否是虚函数无关。
* 重写：指的是在派生类中覆盖基类中的同名函数，重写就是重写函数体，要求基类函数必须是虚函数且：

1、与基类的虚函数有相同的参数个数

2、与基类的虚函数有相同的参数类型

3、与基类的虚函数有相同的返回值类型

* 隐藏指的是某些情况下，派生类中的函数屏蔽了基类中的同名函数，包括以下情况：

1、两个函数参数相同，但是基类函数不是虚函数。和重写的区别在于基类函数是否是虚函数。

2、两个函数参数不同，无论基类函数是不是虚函数，都会被隐藏。和重载的区别在于两个函数不在同一个类中。