<https://github.com/huihut/interview>

<https://blog.csdn.net/liupeng900605/article/details/7528368> 内存布局

<https://zhuanlan.zhihu.com/p/41309205>

g++ -fdump-class-hierarchy -c main.cpp

<https://drive.google.com/drive/u/0/folders/1HRKGtucjHR7XeS5f_j4zDvsZaHcd5xXg> MAP A CONV LAYER TO MATRIX

C++中虚函数的作用主要是为了实现多态机制。多态，简单来说，是指在继承层次中，父类的指针可以具有多种形态——当它指向某个子类对象时，通过它能够调用到子类的函数，而非父类的函数

**虚析构函数**

虚析构函数是为了解决基类的指针指向派生类对象，并用基类的指针删除派生类对象

**纯虚函数**

纯虚函数是一种特殊的虚函数，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。

### 虚函数、纯虚函数

* 类里如果声明了虚函数，这个函数是实现的，哪怕是空实现，它的作用就是为了能让这个函数在它的子类里面可以被覆盖（override），这样的话，编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。
* 虚函数在子类里面可以不重写；但纯虚函数必须在子类实现才可以实例化子类。
* 虚函数的类用于 “实作继承”，继承接口的同时也继承了父类的实现。纯虚函数关注的是接口的统一性，实现由子类完成。
* 带纯虚函数的类叫抽象类，这种类不能直接生成对象，而只有被继承，并重写其虚函数后，才能使用。抽象类被继承后，子类可以继续是抽象类，也可以是普通类。
* 虚基类是虚继承中的基类，具体见下文虚继承。

**虚函数指针、虚函数表**

* 虚函数指针：在含有虚函数类的对象中，指向虚函数表，在运行时确定。
* 虚函数表：在程序只读数据段（.rodata section，见：[目标文件存储结构](https://github.com/huihut/interview" \l "%E7%9B%AE%E6%A0%87%E6%96%87%E4%BB%B6%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84)），存放虚函数指针，如果派生类实现了基类的某个虚函数，则在虚表中覆盖原本基类的那个虚函数指针，在编译时根据类的声明创建。

### 虚继承

虚继承用于解决多继承条件下的菱形继承问题（浪费存储空间、存在二义性）。

底层实现原理与编译器相关，一般通过**虚基类指针**和**虚基类表**实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。

实际上，vbptr 指的是虚基类表指针（virtual base table pointer），该指针指向了一个虚基类表（virtual table），虚表中记录了虚基类与本类的偏移地址；通过偏移地址，这样就找到了虚基类成员，而虚继承也不用像普通多继承那样维持着公共基类（虚基类）的两份同样的拷贝，**节省了存储空间**

**虚继承、虚函数**

* 相同之处：都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）
* 不同之处：
  + 虚继承
    - 虚基类依旧存在继承类中，只占用存储空间
    - 虚基类表存储的是虚基类相对直接继承类的偏移
  + 虚函数
    - 虚函数不占用存储空间
    - 虚函数表存储的是虚函数地址

**模板类、成员模板、虚函数**

* 模板类中可以使用虚函数
* 一个类（无论是普通类还是类模板）的成员模板（本身是模板的成员函数）不能是虚函数

**抽象类、接口类、聚合类**

* 抽象类：含有纯虚函数的类
* 接口类：仅含有纯虚函数的抽象类
* 聚合类：用户可以直接访问其成员，并且具有特殊的初始化语法形式。满足如下特点：
  + 所有成员都是 public
  + 没有定义任何构造函数
  + 没有类内初始化
  + 没有基类，也没有 virtual 函数

### 内存分配和管理

#### malloc、calloc、realloc、alloca

1. malloc：申请指定字节数的内存。申请到的内存中的初始值不确定。
2. calloc：为指定长度的对象，分配能容纳其指定个数的内存。申请到的内存的每一位（bit）都初始化为 0。
3. realloc：更改以前分配的内存长度（增加或减少）。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，而新增区域内的初始值则不确定。
4. alloca：在栈上申请内存。程序在出栈的时候，会自动释放内存。但是需要注意的是，alloca 不具可移植性, 而且在没有传统堆栈的机器上很难实现。alloca 不宜使用在必须广泛移植的程序中。C99 中支持变长数组 (VLA)，可以用来替代 alloca。

#### new、delete

1. new / new[]：完成两件事，先底层调用 malloc 分配了内存，然后调用构造函数（创建对象）。
2. delete/delete[]：也完成两件事，先调用析构函数（清理资源），然后底层调用 free 释放空间。
3. new 在申请内存时会自动计算所需字节数，而 malloc 则需我们自己输入申请内存空间的字节数。

new、delete 使用

申请内存，确认是否申请成功

**定位 new**

定位 new（placement new）允许我们向 new 传递额外的地址参数，从而在预先指定的内存区域创建对象。

new (place\_address) type

new (place\_address) type (initializers)

new (place\_address) type [size]

new (place\_address) type [size] { braced initializer list }

* place\_address 是个指针
* initializers 提供一个（可能为空的）以逗号分隔的初始值列表

**delete this 合法吗？**

[Is it legal (and moral) for a member function to say delete this?](https://isocpp.org/wiki/faq/freestore-mgmt#delete-this)

合法，但：

1. 必须保证 this 对象是通过 new（不是 new[]、不是 placement new、不是栈上、不是全局、不是其他对象成员）分配的
2. 必须保证调用 delete this 的成员函数是最后一个调用 this 的成员函数
3. 必须保证成员函数的 delete this 后面没有调用 this 了
4. 必须保证 delete this 后没有人使用了

### 如何定义一个只能在堆上（栈上）生成对象的类？

#### 只能在堆上

方法：将析构函数设置为私有

原因：C++ 是静态绑定语言，编译器管理栈上对象的生命周期，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性。若析构函数不可访问，则不能在栈上创建对象。

#### 只能在栈上

方法：将 new 和 delete 重载为私有

原因：在堆上生成对象，使用 new 关键词操作，其过程分为两阶段：第一阶段，使用 new 在堆上寻找可用内存，分配给对象；第二阶段，调用构造函数生成对象。将 new 操作设置为私有，那么第一阶段就无法完成，就不能够在堆上生成对象

**智能指针**

#### C++ 11

1. shared\_ptr
2. unique\_ptr
3. weak\_ptr
4. auto\_ptr（被 C++11 弃用）

为什么要使用智能指针：我们知道c++的内存管理是让很多人头疼的事，当我们写一个new语句时，一般就会立即把delete语句直接也写了，但是我们不能避免程序还未执行到delete时就跳转了或者在函数中没有执行到最后的delete语句就返回了，如果我们不在每一个可能跳转或者返回的语句前释放资源，就会造成内存泄露。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源

<https://www.cnblogs.com/TenosDoIt/p/3456704.html>

* Class shared\_ptr 实现共享式拥有（shared ownership）概念。多个智能指针指向相同对象，该对象和其相关资源会在 “最后一个 reference 被销毁” 时被释放。为了在结构较复杂的情景中执行上述工作，标准库提供 weak\_ptr、bad\_weak\_ptr 和 enable\_shared\_from\_this 等辅助类。
* Class unique\_ptr 实现独占式拥有（exclusive ownership）或严格拥有（strict ownership）概念，保证同一时间内只有一个智能指针可以指向该对象。你可以移交拥有权。它对于避免内存泄漏（resource leak）——如 new 后忘记 delete ——特别有用。

##### shared\_ptr

多个智能指针可以共享同一个对象，对象的最末一个拥有着有责任销毁对象，并清理与该对象相关的所有资源。

* 支持定制型删除器（custom deleter），可防范 Cross-DLL 问题（对象在动态链接库（DLL）中被 new 创建，却在另一个 DLL 内被 delete 销毁）、自动解除互斥锁

##### weak\_ptr

weak\_ptr 允许你共享但不拥有某对象，一旦最末一个拥有该对象的智能指针失去了所有权，任何 weak\_ptr 都会自动成空（empty）。因此，在 default 和 copy 构造函数之外，weak\_ptr 只提供 “接受一个 shared\_ptr” 的构造函数。

* 可打破环状引用（cycles of references，两个其实已经没有被使用的对象彼此互指，使之看似还在 “被使用” 的状态）的问题

##### unique\_ptr

unique\_ptr 是 C++11 才开始提供的类型，是一种在异常时可以帮助避免资源泄漏的智能指针。采用独占式拥有，意味着可以确保一个对象和其相应的资源同一时间只被一个 pointer 拥有。一旦拥有着被销毁或编程 empty，或开始拥有另一个对象，先前拥有的那个对象就会被销毁，其任何相应资源亦会被释放。

* unique\_ptr 用于取代 auto\_ptr

##### auto\_ptr

被 c++11 弃用，原因是缺乏语言特性如 “针对构造和赋值” 的 std::move 语义，以及其他瑕疵。

##### auto\_ptr 与 unique\_ptr 比较

* auto\_ptr 可以赋值拷贝，复制拷贝后所有权转移；unqiue\_ptr 无拷贝赋值语义，但实现了move 语义；
* auto\_ptr 对象不能管理数组（析构调用 delete），unique\_ptr 可以管理数组（析构调用 delete[] ）；

### 强制类型转换运算符

[MSDN . 强制转换运算符](https://msdn.microsoft.com/zh-CN/library/5f6c9f8h.aspx)

#### static\_cast

* 用于非多态类型的转换
* 不执行运行时类型检查（转换安全性不如 dynamic\_cast）
* 通常用于转换数值数据类型（如 float -> int）
* 可以在整个类层次结构中移动指针，子类转化为父类安全（向上转换），父类转化为子类不安全（因为子类可能有不在父类的字段或方法）
* static\_cast不能转换掉expression的const、volitale、或者\_\_unaligned属性

向上转换是一种隐式转换。

#### dynamic\_cast

* 用于多态类型的转换 被转换对象obj的类型T1必须是多态类型，即T1必须公有继承自其它类，或者T1拥有虚函数（继承或自定义）。若T1为非多态类型，使用dynamic\_cast会报编译错误
* 执行行运行时类型检查
* 只适用于指针或引用
* 对不明确的指针的转换将失败（返回 nullptr），但不引发异常
* 可以在整个类层次结构中移动指针，包括向上转换、向下转换

#### const\_cast

* 用于删除 const、volatile 和 \_\_unaligned 特性（如将 const int 类型转换为 int 类型 ）

#### reinterpret\_cast

reinterpret\_cast 只能在指针之间转换

* 用于位的简单重新解释
* 滥用 reinterpret\_cast 运算符可能很容易带来风险。 除非所需转换本身是低级别的，否则应使用其他强制转换运算符之一。
* 允许将任何指针转换为任何其他指针类型（如 char\* 到 int\* 或 One\_class\* 到 Unrelated\_class\* 之类的转换，但其本身并不安全）
* 也允许将任何整数类型转换为任何指针类型以及反向转换。
* reinterpret\_cast 运算符不能丢掉 const、volatile 或 \_\_unaligned 特性。
* reinterpret\_cast 的一个实际用途是在哈希函数中，即，通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。

#### bad\_cast

* 由于强制转换为引用类型失败，dynamic\_cast 运算符引发 bad\_cast 异常。

bad\_cast 使用

try {

Circle& ref\_circle = dynamic\_cast<Circle&>(ref\_shape);

}

catch (bad\_cast b) {

cout << "Caught: " << b.what();

}

<https://zhuanlan.zhihu.com/p/33040213> 哈哈哈哈

使用reinterpret\_cast运算符把pnum从int\*转变成char\*类型并用于初始化pstr后，pstr也指向num的内存区域，但是由于pstr是char\*类型的，通过pstr读写num内存区域将不再按照整型变量的规则，而是按照char型变量规则。一个char型变量占用一个Byte，对pstr解引用得到的将是一个字符，也就是’a’。而在使用输出流输出pstr时，将输出pstr指向的内存区域的字符，那pstr指向的是一个的字符，那为什么输出三个字符呢？这是由于在输出char\*指针时，输出流会把它当做输出一个字符串来处理，直至遇到’\0’才表示字符串结束。对代码稍做改动，就会得到不一样的输出结果，例如将num的值改为0x63006261,输出的字符串就变为”ab”

**More Effective c++**

1. 仔细区别 pointers 和 **references**（当你知道你需要指向某个东西，而且绝不会改变指向其他东西，或是当你实现一个操作符而其语法需求无法由 pointers 达成，你就应该选择 references；任何其他时候，请采用 pointers）
2. 最好使用 C++ 转型操作符（static\_cast、const\_cast、dynamic\_cast、reinterpret\_cast）
3. 绝不要以多态（polymorphically）方式处理数组（多态（polymorphism）和指针算术不能混用；数组对象几乎总是会涉及指针的算术运算，所以数组和多态不要混用）
4. 非必要不提供 default constructor（避免对象中的字段被无意义地初始化）
5. 对定制的 “类型转换函数” 保持警觉（单自变量 constructors 可通过简易法（explicit 关键字）或代理类（proxy classes）来避免编译器误用；隐式类型转换操作符可改为显式的 member function 来避免非预期行为）
6. 区别 increment/decrement 操作符的前置（prefix）和后置（postfix）形式（前置式累加后取出，返回一个 reference；后置式取出后累加，返回一个 const 对象；处理用户定制类型时，应该尽可能使用前置式 increment；后置式的实现应以其前置式兄弟为基础）
7. 千万不要重载 &&，|| 和 , 操作符（&& 与 || 的重载会用 “函数调用语义” 取代 “骤死式语义”；, 的重载导致不能保证左侧表达式一定比右侧表达式更早被评估）
8. 了解各种不同意义的 new 和 delete（new operator、operator new、placement new、operator new[]；delete operator、operator delete、destructor、operator delete[]）
9. 利用 destructors 避免泄漏资源（在 destructors 释放资源可以避免异常时的资源泄漏）
10. 在 constructors 内阻止资源泄漏（由于 C++ 只会析构已构造完成的对象，因此在构造函数可以使用 try...catch 或者 auto\_ptr（以及与之相似的 classes） 处理异常时资源泄露问题）
11. 禁止异常流出 destructors 之外（原因：一、避免 terminate 函数在 exception 传播过程的栈展开（stack-unwinding）机制种被调用；二、协助确保 destructors 完成其应该完成的所有事情）
12. 了解 “抛出一个 exception” 与 “传递一个参数” 或 “调用一个虚函数” 之间的差异（第一，exception objects 总是会被复制（by pointer 除外），如果以 by value 方式捕捉甚至被复制两次，而传递给函数参数的对象则不一定得复制；第二，“被抛出成为 exceptions” 的对象，其被允许的类型转换动作比 “被传递到函数去” 的对象少；第三，catch 子句以其 “出现于源代码的顺序” 被编译器检验对比，其中第一个匹配成功者便执行，而调用一个虚函数，被选中执行的是那个 “与对象类型最佳吻合” 的函数）
13. 以 by reference 方式捕获 exceptions（可避免对象删除问题、exception objects 的切割问题，可保留捕捉标准 exceptions 的能力，可约束 exception object 需要复制的次数）
14. 明智运用 exception specifications（exception specifications 对 “函数希望抛出什么样的 exceptions” 提供了卓越的说明；也有一些缺点，包括编译器只对它们做局部性检验而很容易不经意地违反，与可能会妨碍更上层的 exception 处理函数处理未预期的 exceptions）
15. 了解异常处理的成本（粗略估计，如果使用 try 语句块，代码大约整体膨胀 5%-10%，执行速度亦大约下降这个数；因此请将你对 try 语句块和 exception specifications 的使用限制于非用不可的地点，并且在真正异常的情况下才抛出 exceptions）
16. 谨记 80-20 法则（软件的整体性能几乎总是由其构成要素（代码）的一小部分决定的，可使用程序分析器（program profiler）识别出消耗资源的代码）
17. 考虑使用 lazy evaluation（缓式评估）（可应用于：Reference Counting（引用计数）来避免非必要的对象复制、区分 operator[] 的读和写动作来做不同的事情、Lazy Fetching（缓式取出）来避免非必要的数据库读取动作、Lazy Expression Evaluation（表达式缓评估）来避免非必要的数值计算动作）
18. 分期摊还预期的计算成本（当你必须支持某些运算而其结构几乎总是被需要，或其结果常常被多次需要的时候，over-eager evaluation（超急评估）可以改善程序效率）

**封装**

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。关键字：public, protected, private。不写默认为 private。

* public 成员：可以被任意实体访问
* protected 成员：只允许被子类及本类的成员函数访问
* private 成员：只允许被本类的成员函数、友元类或友元函数访问

**继承**

* 基类（父类）——> 派生类（子类）

**多态**

* 多态，即多种状态（形态）。简单来说，我们可以将多态定义为消息以多种形式显示的能力。
* 多态是以封装和继承为基础的。
* C++ 多态分类及实现：
  1. 重载多态（Ad-hoc Polymorphism，编译期）：函数重载、运算符重载
  2. 子类型多态（Subtype Polymorphism，运行期）：虚函数
  3. 参数多态性（Parametric Polymorphism，编译期）：类模板、函数模板
  4. 强制多态（Coercion Polymorphism，编译期/运行期）：基本类型转换、自定义类型转换

**静态多态（编译期/早绑定）**

函数重载

class A{

public:

void do(int a);

void do(int a, int b);

};

**动态多态（运行期期/晚绑定）**

* 虚函数：用 virtual 修饰成员函数，使其成为虚函数
* 普通函数（非类成员函数）不能是虚函数
* 静态函数（static）不能是虚函数
* 构造函数不能是虚函数（因为在调用构造函数时，虚表指针并没有在对象的内存空间中，必须要构造函数调用完成后才会形成虚表指针）
* 内联函数不能是表现多态性时的虚函数，解释见：[虚函数（virtual）可以是内联函数（inline）吗？](https://github.com/huihut/interview#%E8%99%9A%E5%87%BD%E6%95%B0virtual%E5%8F%AF%E4%BB%A5%E6%98%AF%E5%86%85%E8%81%94%E5%87%BD%E6%95%B0inline%E5%90%97)

**虚函数（virtual）可以是内联函数（inline）吗？**

[Are "inline virtual" member functions ever actually "inlined"?](http://www.cs.technion.ac.il/users/yechiel/c++-faq/inline-virtuals.html)

* 虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。
* 内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。
* inline virtual 唯一可以内联的时候是：编译器知道所调用的对象是哪个类（如 Base::who()），这只有在编译器具有实际对象而不是对象的指针或引用时才会发生