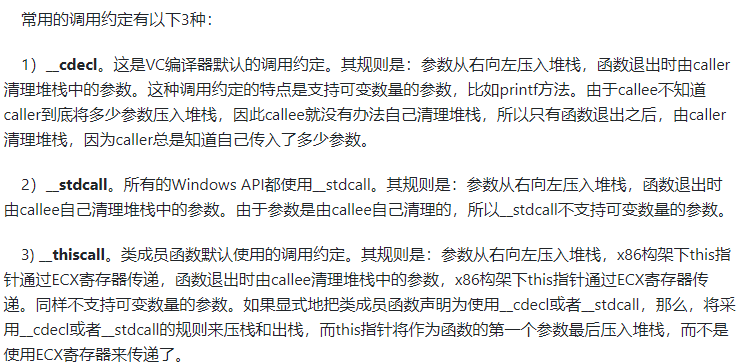
## 基础篇

### 1.常用的3种调用约定



### 2.this指针

1. this 指针是一个隐含于每一个非静态成员函数中的特殊指针。它指向调用该成员函数的那个对象。
2. 当对一个对象调用成员函数时，编译程序先将对象的地址赋给 this 指针，然后调用成员函数，每次成员函数存取数据成员时，都隐式使用 this 指针。
3. 当一个成员函数被调用时，自动向它传递一个隐含的参数，该参数是一个指向这个成员函数所在的对象的指针。
4. this 指针被隐含地声明为: ClassName \*const this，这意味着不能给 this 指针赋值；在 ClassName 类的 const 成员函数中，this 指针的类型为：const ClassName\* const，这说明不能对 this 指针所指向的这种对象是不可修改的（即不能对这种对象的数据成员进行赋值操作）；
5. this 并不是一个常规变量，而是个右值，所以不能取得 this 的地址（不能 &this）。
6. 在以下场景中，经常需要显式引用 this 指针：
   1. 为实现对象的链式引用；
   2. 为避免对同一对象进行赋值操作；
   3. 在实现一些数据结构时，如 list。

* **delete this合法吗**

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

### 3.inline内联函数

* 特征
* 相当于把内联函数里面的内容写在调用内联函数处；
* 相当于不用执行进入函数的步骤，直接执行函数体；
* 相当于宏，却比宏多了类型检查，真正具有函数特性；
* 编译器一般不内联包含循环、递归、switch 等复杂操作的内联函数；
* 在类声明中定义的函数，除了虚函数的其他函数都会自动隐式地当成内联函数。
* [编译器对 inline 函数的处理步骤](https://interview.huihut.com/#/?id=%e7%bc%96%e8%af%91%e5%99%a8%e5%af%b9-inline-%e5%87%bd%e6%95%b0%e7%9a%84%e5%a4%84%e7%90%86%e6%ad%a5%e9%aa%a4)

1. 将 inline 函数体复制到 inline 函数调用点处；
2. 为所用 inline 函数中的局部变量分配内存空间；
3. 将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；
4. 如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）。

* [优缺点](https://interview.huihut.com/#/?id=%e4%bc%98%e7%bc%ba%e7%82%b9)

优点

1. 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
2. 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
3. 在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量，宏定义则不能。
4. 内联函数在运行时可调试，而宏定义不可以。

缺点

1. 代码膨胀。内联是以代码膨胀（复制）为代价，消除函数调用带来的开销。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。
2. inline 函数无法随着函数库升级而升级。inline函数的改变需要重新编译，不像 non-inline 可以直接链接。
3. 是否内联，程序员不可控。内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

### 4.volatile

* volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素（操作系统、硬件、其它线程等）更改。所以使用 volatile 告诉编译器不应对这样的对象进行优化。
* volatile 关键字声明的变量，每次访问时都必须从内存中取出值（没有被 volatile 修饰的变量，可能由于编译器的优化，从 CPU 寄存器中取值）
* const 可以是 volatile （如只读的状态寄存器）
* 指针可以是 volatile

1) 一个参数既可以是const还可以是volatile吗？解释为什么。  
2) 一个指针可以是volatile 吗？解释为什么。  
3) 下面的函数有什么错误：  
  
int square(volatile int \*ptr)  
{  
        return \*ptr \* \*ptr;  
}  
  
下面是答案：  
1) 是的。一个例子是只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。  
2) 是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个buffer的指针时。  
3) 这段代码有点变态。这段代码的目的是用来返指针\*ptr指向值的平方，但是，由于\*ptr指向一个volatile型参数，编译器将产生类似下面的代码：  
  
int square(volatile int \*ptr)  
{  
    int a,b;  
    a = \*ptr;  
    b = \*ptr;  
    return a \* b;  
}  
  
由于\*ptr的值可能被意想不到地该变，因此a和b可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：  
  
long square(volatile int \*ptr)  
{  
    int a;  
    a = \*ptr;  
    return a \* a;  
}

### 5.assert()

断言，是宏，而非函数。assert 宏的原型定义在 <assert.h>（C）、<cassert>（C++）中，其作用是如果它的条件返回错误，则终止程序执行。可以通过定义 NDEBUG 来关闭 assert，但是需要在源代码的开头，include <assert.h> 之前。

assert() 使用

#define NDEBUG // 加上这行，则 assert 不可用

#include <assert.h>

assert( p != NULL ); // assert 不可用

### 6.[#pragma pack(n)](https://interview.huihut.com/#/?id=pragma-packn)

设定结构体、联合以及类成员变量以 n 字节方式对齐

#pragma pack(n) 使用

#pragma pack(push) // 保存对齐状态

#pragma pack(4) // 设定为 4 字节对齐

struct test

{

char m1;

double m4;

int m3;

};

#pragma pack(pop) // 恢复对齐状态

### 7.[位域](https://interview.huihut.com/#/?id=%e4%bd%8d%e5%9f%9f)

Bit mode: 2; // mode 占 2 位

类可以将其（非静态）数据成员定义为位域（bit-field），在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域。

* 位域在内存中的布局是与机器有关的
* 位域的类型必须是整型或枚举类型，带符号类型中的位域的行为将因具体实现而定
* 取地址运算符（&）不能作用于位域，任何指针都无法指向类的位域

### 8.[extern "C"](https://interview.huihut.com/#/?id=extern-quotcquot)

* 被 extern 限定的函数或变量是 extern 类型的
* 被 extern "C" 修饰的变量和函数是按照 C 语言方式编译和链接的

extern "C" 的作用是让 C++ 编译器将 extern "C" 声明的代码当作 C 语言代码处理，可以避免 C++ 因符号修饰导致代码不能和C语言库中的符号进行链接的问题。

extern "C" 使用

#ifdef \_\_cplusplus

extern "C" {

#endif

void \*memset(void \*, int, size\_t);

#ifdef \_\_cplusplus

}

#endif

### 9.[union 联合](https://interview.huihut.com/#/?id=union-%e8%81%94%e5%90%88)

联合（union）是一种节省空间的特殊的类，一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当某个成员被赋值后其他成员变为未定义状态。联合有如下特点：

* 默认访问控制符为 public
* 可以含有构造函数、析构函数
* 不能含有引用类型的成员
* 不能继承自其他类，不能作为基类
* 不能含有虚函数
* 匿名 union 在定义所在作用域可直接访问 union 成员
* 匿名 union 不能包含 protected 成员或 private 成员
* 全局匿名联合必须是静态（static）的

union 使用

#include<iostream>

union UnionTest {

UnionTest() : i(10) {};

int i;

double d;

};

static union {

int i;

double d;

};

int main() {

UnionTest u;

union {

int i;

double d;

};

std::cout << u.i << std::endl; // 输出 UnionTest 联合的 10

::i = 20;

std::cout << ::i << std::endl; // 输出全局静态匿名联合的 20

i = 30;

std::cout << i << std::endl; // 输出局部匿名联合的 30

return 0;

}

### 10.[explicit（显式）关键字](https://interview.huihut.com/#/?id=explicit%ef%bc%88%e6%98%be%e5%bc%8f%ef%bc%89%e5%85%b3%e9%94%ae%e5%ad%97)

* explicit 修饰构造函数时，可以防止隐式转换和复制初始化
* explicit 修饰转换函数时，可以防止隐式转换，但 [**按语境转换**](https://zh.cppreference.com/w/cpp/language/implicit_conversion) 除外

### 11. [friend 友元类和友元函数](https://interview.huihut.com/#/?id=friend-%e5%8f%8b%e5%85%83%e7%b1%bb%e5%92%8c%e5%8f%8b%e5%85%83%e5%87%bd%e6%95%b0)

* 能访问私有成员
* 破坏封装性
* 友元关系不可传递
* 友元关系的单向性
* 友元声明的形式及数量不受限制

### 12.using

#### [using 声明](https://interview.huihut.com/#/?id=using-%e5%a3%b0%e6%98%8e)

一条 using 声明 语句一次只引入命名空间的一个成员。它使得我们可以清楚知道程序中所引用的到底是哪个名字。如：

using namespace\_name::name;

#### [构造函数的 using 声明](https://interview.huihut.com/#/?id=%e6%9e%84%e9%80%a0%e5%87%bd%e6%95%b0%e7%9a%84-using-%e5%a3%b0%e6%98%8e)

在 C++11 中，派生类能够重用其直接基类定义的构造函数。

class Derived : Base {

public:

using Base::Base;

/\* ... \*/

};

如上 using 声明，对于基类的每个构造函数，编译器都生成一个与之对应（形参列表完全相同）的派生类构造函数。生成如下类型构造函数：

Derived(parms) : Base(args) { }

#### [using 指示](https://interview.huihut.com/#/?id=using-%e6%8c%87%e7%a4%ba)

using 指示 使得某个特定命名空间中所有名字都可见，这样我们就无需再为它们添加任何前缀限定符了。如：

using namespace\_name name;

#### [尽量少使用 using 指示 污染命名空间](https://interview.huihut.com/#/?id=%e5%b0%bd%e9%87%8f%e5%b0%91%e4%bd%bf%e7%94%a8-using-%e6%8c%87%e7%a4%ba-%e6%b1%a1%e6%9f%93%e5%91%bd%e5%90%8d%e7%a9%ba%e9%97%b4)

**一般说来，使用 using 命令比使用 using 编译命令更安全，这是由于它只导入了指定的名称。如果该名称与局部名称发生冲突，编译器将发出指示。using编译命令导入所有的名称，包括可能并不需要的名称。如果与局部名称发生冲突，则局部名称将覆盖名称空间版本，而编译器并不会发出警告。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。**

* using 使用

尽量少使用 using 指示

using namespace std;

应该多使用 using 声明

int x;

std::cin >> x ;

std::cout << x << std::endl;

或者

using std::cin;

using std::cout;

using std::endl;

int x;

cin >> x;

cout << x << endl;

### 13.decltype

decltype 关键字用于检查实体的声明类型或表达式的类型及值分类。语法：

decltype ( expression )

decltype 使用

// 尾置返回允许我们在参数列表之后声明返回类型

template <typename It>

auto fcn(It beg, It end) -> decltype(\*beg)

{

// 处理序列

return \*beg; // 返回序列中一个元素的引用

}

// 为了使用模板参数成员，必须用 typename

template <typename It>

auto fcn2(It beg, It end) -> typename remove\_reference<decltype(\*beg)>::type

{

// 处理序列

return \*beg; // 返回序列中一个元素的拷贝

}

### 14.[引用](https://interview.huihut.com/#/?id=%e5%bc%95%e7%94%a8)

#### [左值引用](https://interview.huihut.com/#/?id=%e5%b7%a6%e5%80%bc%e5%bc%95%e7%94%a8)

常规引用，一般表示对象的身份。

#### [右值引用](https://interview.huihut.com/#/?id=%e5%8f%b3%e5%80%bc%e5%bc%95%e7%94%a8)

右值引用就是必须绑定到右值（一个临时对象、将要销毁的对象）的引用，一般表示对象的值。

右值引用可实现转移语义（Move Sementics）和精确传递（Perfect Forwarding），它的主要目的有两个方面：

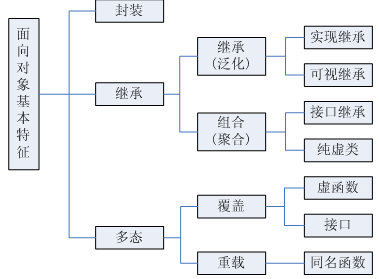
* 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
* 能够更简洁明确地定义泛型函数。

#### [引用折叠](https://interview.huihut.com/#/?id=%e5%bc%95%e7%94%a8%e6%8a%98%e5%8f%a0)

* X& &、X& &&、X&& & 可折叠成 X&
* X&& && 可折叠成 X&&

### 15.面向对象三大特征

面向对象程序设计（Object-oriented programming，OOP）是种具有对象概念的程序编程典范，同时也是一种程序开发的抽象方针。



面向对象三大特征 —— 封装、继承、多态

### 16.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。

虚析构函数使用

class Shape

{

public:

Shape(); // 构造函数不能是虚函数

virtual double calcArea();

virtual ~Shape(); // 虚析构函数

};

class Circle : public Shape // 圆形类

{

public:

virtual double calcArea();

...

};

int main()

{

Shape \* shape1 = new Circle(4.0);

shape1->calcArea();

delete shape1; // 因为Shape有虚析构函数，所以delete释放内存时，先调用子类析构函数，再调用基类析构函数，防止内存泄漏。

shape1 = NULL;

return 0；

}

### 17.虚函数和纯虚函数

* 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。

### 18.[虚函数指针、虚函数表](https://interview.huihut.com/#/?id=%e8%99%9a%e5%87%bd%e6%95%b0%e6%8c%87%e9%92%88%e3%80%81%e8%99%9a%e5%87%bd%e6%95%b0%e8%a1%a8)

* 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://interview.huihut.com/#/?id=%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。

[**C++中的虚函数(表)实现机制以及用C语言对其进行的模拟实现**](https://blog.twofei.com/496/)

### 19.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。

### 20.[虚继承、虚函数](https://interview.huihut.com/#/?id=%e8%99%9a%e7%bb%a7%e6%89%bf%e3%80%81%e8%99%9a%e5%87%bd%e6%95%b0)

* 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

### 21.[模板类、成员模板、虚函数](https://interview.huihut.com/#/?id=%e6%a8%a1%e6%9d%bf%e7%b1%bb%e3%80%81%e6%88%90%e5%91%98%e6%a8%a1%e6%9d%bf%e3%80%81%e8%99%9a%e5%87%bd%e6%95%b0)

* 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

### 22.[抽象类、接口类、聚合类](https://interview.huihut.com/#/?id=%e6%8a%bd%e8%b1%a1%e7%b1%bb%e3%80%81%e6%8e%a5%e5%8f%a3%e7%b1%bb%e3%80%81%e8%81%9a%e5%90%88%e7%b1%bb)

* 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

### 23.[智能指针](https://interview.huihut.com/#/?id=%e6%99%ba%e8%83%bd%e6%8c%87%e9%92%88)

#### [C++ 标准库（STL）中](https://interview.huihut.com/#/?id=c-%e6%a0%87%e5%87%86%e5%ba%93%ef%bc%88stl%ef%bc%89%e4%b8%ad)

头文件：#include <memory>

#### [C++ 98](https://interview.huihut.com/#/?id=c-98)

std::auto\_ptr<std::string> ps (new std::string(str))；

#### [C++ 11](https://interview.huihut.com/#/?id=c-11)

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

* 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](https://interview.huihut.com/#/?id=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](https://interview.huihut.com/#/?id=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](https://interview.huihut.com/#/?id=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](https://interview.huihut.com/#/?id=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](https://interview.huihut.com/#/?id=auto_ptr-%e4%b8%8e-unique_ptr-%e6%af%94%e8%be%83)

* 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[] ）；

### 24.[强制类型转换运算符](https://interview.huihut.com/#/?id=%e5%bc%ba%e5%88%b6%e7%b1%bb%e5%9e%8b%e8%bd%ac%e6%8d%a2%e8%bf%90%e7%ae%97%e7%ac%a6)

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

#### [static\_cast](https://interview.huihut.com/#/?id=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）

**向上转换是一种隐式转换。**

#### [dynamic\_cast](https://interview.huihut.com/#/?id=dynamic_cast)

* 用于多态类型的转换
* 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](https://interview.huihut.com/#/?id=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](https://interview.huihut.com/#/?id=reinterpret_cast)

* 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](https://interview.huihut.com/#/?id=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();

}

### 25.[运行时类型信息 (RTTI)](https://interview.huihut.com/#/?id=%e8%bf%90%e8%a1%8c%e6%97%b6%e7%b1%bb%e5%9e%8b%e4%bf%a1%e6%81%af-rtti)

#### [typeid](https://interview.huihut.com/#/?id=typeid)

* typeid 运算符允许在运行时确定对象的类型
* type\_id 返回一个 type\_info 对象的引用
* 如果想通过基类的指针获得派生类的数据类型，基类必须带有虚函数
* 只能获取对象的实际类型

#### [type\_info](https://interview.huihut.com/#/?id=type_info)

* type\_info 类描述编译器在程序中生成的类型信息。 此类的对象可以有效存储指向类型的名称的指针。 type\_info 类还可存储适合比较两个类型是否相等或比较其排列顺序的编码值。 类型的编码规则和排列顺序是未指定的，并且可能因程序而异。
* 头文件：typeinfo

typeid、type\_info 使用

#include <iostream>

using namespace std;

class Flyable // 能飞的

{

public:

virtual void takeoff() = 0; // 起飞

virtual void land() = 0; // 降落

};

class Bird : public Flyable // 鸟

{

public:

void foraging() {...} // 觅食

virtual void takeoff() {...}

virtual void land() {...}

virtual ~Bird(){}

};

class Plane : public Flyable // 飞机

{

public:

void carry() {...} // 运输

virtual void takeoff() {...}

virtual void land() {...}

};

class type\_info

{

public:

const char\* name() const;

bool operator == (const type\_info & rhs) const;

bool operator != (const type\_info & rhs) const;

int before(const type\_info & rhs) const;

virtual ~type\_info();

private:

...

};

void doSomething(Flyable \*obj) // 做些事情

{

obj->takeoff();

cout << typeid(\*obj).name() << endl; // 输出传入对象类型（"class Bird" or "class Plane"）

if(typeid(\*obj) == typeid(Bird)) // 判断对象类型

{

Bird \*bird = dynamic\_cast<Bird \*>(obj); // 对象转化

bird->foraging();

}

obj->land();

}

int main(){

Bird \*b = new Bird();

doSomething(b);

delete b;

b = nullptr;

return 0;

}

## 内存分配与管理

### 1.[malloc、calloc、realloc、alloca](https://interview.huihut.com/#/?id=malloc%e3%80%81calloc%e3%80%81realloc%e3%80%81alloca)

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。

### 2.[malloc、free](https://interview.huihut.com/#/?id=malloc%e3%80%81free)

用于分配、释放内存

malloc、free 使用

申请内存，确认是否申请成功

char \*str = *(char\*) malloc(100);*

assert(str != nullptr);

释放内存后指针置空

free(p);

p = nullptr;

### 3.[new、delete](https://interview.huihut.com/#/?id=new%e3%80%81delete)

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

int main()

{

T\* t = new T(); // 先内存分配 ，再构造函数

delete t; // 先析构函数，再内存释放

return 0;

}

### 4.[定位 new](https://interview.huihut.com/#/?id=%e5%ae%9a%e4%bd%8d-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

### 5.[如何定义一个只能在堆上（栈上）生成对象的类？](https://interview.huihut.com/#/?id=%e5%a6%82%e4%bd%95%e5%ae%9a%e4%b9%89%e4%b8%80%e4%b8%aa%e5%8f%aa%e8%83%bd%e5%9c%a8%e5%a0%86%e4%b8%8a%ef%bc%88%e6%a0%88%e4%b8%8a%ef%bc%89%e7%94%9f%e6%88%90%e5%af%b9%e8%b1%a1%e7%9a%84%e7%b1%bb%ef%bc%9f)

[**如何定义一个只能在堆上（栈上）生成对象的类?**](https://www.nowcoder.com/questionTerminal/0a584aa13f804f3ea72b442a065a7618)

#### [只能在堆上](https://interview.huihut.com/#/?id=%e5%8f%aa%e8%83%bd%e5%9c%a8%e5%a0%86%e4%b8%8a)

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](https://interview.huihut.com/#/?id=%e5%8f%aa%e8%83%bd%e5%9c%a8%e6%a0%88%e4%b8%8a)

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。

## 进程线程篇

### 1. 进程之间通信的途径有哪些？

答：进程间通信主要通过管道、消息、信号、信号量、共享内存、socket等途径进行。

1、无名管道( pipe )：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

2、高级管道(popen)：将另一个程序当做一个新的进程在当前程序进程中启动，则它算是当前程序的子进程，这种方式我们成为高级管道方式。

3、有名管道 (named pipe) ：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

4、消息队列( message queue ) ：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

5、信号量( semophore ) ：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

6、信号 ( sinal ) ：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

7、共享内存( shared memory ) ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

8、套接字( socket ) ：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

### 2. 死锁的4个必要条件

1、互斥：某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，其他进程就不能再访问，直到该进程访问结束。

2、占有且等待：一个进程本身占有资源（一种或多种），同时还有资源未得到满足，正在等待其他进程释放该资源。

3、不可抢占：别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。

4、循环等待：存在一个进程链，使得每个进程都占有下一个进程所需的至少一种资源。

当以上四个条件均满足，必然会造成死锁，发生死锁的进程无法进行下去，它们所持有的资源也无法释放。这样会导致CPU的吞吐量下降。所以死锁情况是会浪费系统资源和影响计算机的使用性能的。那么，解决死锁问题就是相当有必要的了。

### 3.死锁的处理方式有哪些？

死锁的处理方式主要从预防死锁、避免死锁、检测与解除死锁这四个方面来进行处理。

预防死锁：

1、资源一次性分配：（破坏请求和保持条件）

2、可剥夺资源：即当某进程新的资源未满足时，释放已占有的资源（破坏不可剥夺条件）

3、资源有序分配法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）

避免死锁:

预防死锁的几种策略，会严重地损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得 较满意的系统性能。由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，进程等待。其中最具有代表性的避免死锁算法是银行家算法。

检测死锁：

首先为每个进程和每个资源指定一个唯一的号码；

然后建立资源分配表和进程等待表

解除死锁：

当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的方法有：

1、剥夺资源：从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态；

* 1. 撤消进程：可以直接撤消死锁进程或撤消代价最小的进程，直至有足

### 4.进程和线程有什么区别？

**进程**是并发执行的程序在执行过程中**分配和管理资源的基本单位**。线程是进程的一个执行单元，是比进程还要小的独立运行的基本单位。一个程序至少有一个进程，一个进程至少有一个线程。两者的区别主要有以下几个方面：

1. 进程是资源分配的最小单位。

2. 线程是程序执行的最小单位，也是处理器调度的基本单位，但进程不是，两者均可并发执行。

3. 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据，使用相同的地址空间，因此，CPU切换一个线程的花费远比进程小很多，同时创建一个线程的开销也比进程小很多。

4. 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也跟着死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

5. 进程切换时，消耗的资源大，效率低。所以涉及到频繁的切换时，使用线程要好于进程。同样如果要求同时进行并且又要共享某些变量的并发操作，只能用线程不能用进程。

6. 执行过程：每个独立的进程有一个程序运行的入口、顺序执行序列和程序入口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

**优缺点：**

线程执行开销小，但是不利于资源的管理和保护。线程适合在SMP机器（双CPU系统）上运行。

进程执行开销大，但是能够很好的进行资源管理和保护，可以跨机器迁移。

**何时使用多进程，何时使用多线程？**

对资源的管理和保护要求高，不限制开销和效率时，使用多进程。

要求效率高，频繁切换时，资源的保护管理要求不是很高时，使用多线程。

### 5.线程是否具有相同的堆栈?

答：真正的程序执行都是线程来完成的，程序启动的时候操作系统就帮你创建了一个主线程。

每个线程有自己的堆栈。

### 6.TCP与UDP有啥区别？

答：TCP和UDP是OSI模型中的运输层中的协议。TCP提供可靠的通信传输，而UDP则常被用于广播和细节控制交给应用的通信传输，两者主要的不同体现在一下几个方面：

1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接

2、TCP提供可靠的服务。它通过校验和，丢包时的重传控制，序号标识，滑动窗口、确认应答，次序乱掉的分包进行顺序控制实现可靠传输。即通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达; UDP尽最大努力交付，即不保证可靠交付。

3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高要求的通信或广播通信场景。

4、每一条TCP连接只能是点到点的; UDP支持一对一，一对多，多对一和多对多的交互通信方式。

5、TCP对系统资源要求较多，UDP对系统资源要求较少。

UDP有时比TCP更有优势:

UDP以其简单、传输快的优势，在越来越多场景下取代了TCP, 如实时游戏。

（1）网速的提升给UDP的稳定性提供可靠网络保障，丢包率很低，如果使用应用层重传，能够确保传输的可靠性。

（2）TCP为了实现网络通信的可靠性，使用了复杂的拥塞控制算法，建立了繁琐的握手过程，由于TCP在内置的系统协议栈中，极难对其进行改进。

采用TCP，一旦发生丢包，TCP会将后续的包缓存起来，等前面的包重传并接收到后再继续发送，延时会越来越大。

基于UDP对实时性要求较为严格的情况下，采用自定义重传机制，能够把丢包产生的延迟降到最低，尽量减少网络问题造成的影响。