**C++面试**

**const**

**作用**

1. 修饰变量，说明该变量不可以被改变；
2. 修饰指针，分为指向常量的指针（pointer to const）和自身是常量的指针（常量指针，const pointer）；
3. 修饰引用，指向常量的引用（reference to const），用于形参类型，即避免了拷贝，又避免了函数对值的修改；
4. 修饰成员函数，说明该成员函数内不能修改成员变量。

**const 的指针与引用**

* 指针
  + 指向常量的指针（pointer to const）
  + 自身是常量的指针（常量指针，const pointer）
* 引用
  + 指向常量的引用（reference to const）
  + 没有 const reference，因为引用本身就是 const pointer

（为了方便记忆可以想成）被 const 修饰（在 const 后面）的值不可改变，如下文使用例子中的 p2、p3

**使用**

const 使用

// 类

class A

{

private:

const int a; // 常对象成员，只能在初始化列表赋值

public:

// 构造函数

A() : a(0) { };

A(int x) : a(x) { }; // 初始化列表

// const可用于对重载函数的区分

int getValue(); // 普通成员函数

int getValue() const; // 常成员函数，不得修改类中的任何数据成员的值

};

void function()

{

// 对象

A b; // 普通对象，可以调用全部成员函数、更新常成员变量

const A a; // 常对象，只能调用常成员函数

const A \*p = &a; // 指针变量，指向常对象

const A &q = a; // 指向常对象的引用

// 指针

char greeting[] = "Hello";

char\* p1 = greeting; // 指针变量，指向字符数组变量

const char\* p2 = greeting; // 指针变量，指向字符数组常量（const 后面是 char，说明指向的字符（char）不可改变）

char\* const p3 = greeting; // 自身是常量的指针，指向字符数组变量（const 后面是 p3，说明 p3 指针自身不可改变）

const char\* const p4 = greeting; // 自身是常量的指针，指向字符数组常量

}

// 函数

void function1(const int Var); // 传递过来的参数在函数内不可变

void function2(const char\* Var); // 参数指针所指内容为常量

void function3(char\* const Var); // 参数指针为常量

void function4(const int& Var); // 引用参数在函数内为常量

// 函数返回值

const int function5(); // 返回一个常数

const int\* function6(); // 返回一个指向常量的指针变量，使用：const int \*p = function6();

int\* const function7(); // 返回一个指向变量的常指针，使用：int\* const p = function7();

**static**

**作用**

1. 修饰普通变量，修改变量的存储区域和生命周期，使变量存储在静态区，在 main 函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。
2. 修饰普通函数，表明函数的作用范围，仅在定义该函数的文件内才能使用。在多人开发项目时，为了防止与他人命名空间里的函数重名，可以将函数定位为 static。
3. 修饰成员变量，修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。
4. 修饰成员函数，修饰成员函数使得不需要生成对象就可以访问该函数，但是在 static 函数内不能访问非静态成员。

**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。

**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。

**使用**

inline 使用

// 声明1（加 inline，建议使用）

inline int functionName(int first, int second,...);

// 声明2（不加 inline）

int functionName(int first, int second,...);

// 定义

inline int functionName(int first, int second,...) {/\*\*\*\*/};

// 类内定义，隐式内联

class A {

int doA() { return 0; } // 隐式内联

}

// 类外定义，需要显式内联

class A {

int doA();

}

inline int A::doA() { return 0; } // 需要显式内联

**编译器对 inline 函数的处理步骤**

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**

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。

**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。

虚函数内联使用

#include <iostream>

using namespace std;

class Base

{

public:

inline virtual void who()

{

cout << "I am Base\n";

}

virtual ~Base() {}

};

class Derived : public Base

{

public:

inline void who() // 不写inline时隐式内联

{

cout << "I am Derived\n";

}

};

int main()

{

// 此处的虚函数 who()，是通过类（Base）的具体对象（b）来调用的，编译期间就能确定了，所以它可以是内联的，但最终是否内联取决于编译器。

Base b;

b.who();

// 此处的虚函数是通过指针调用的，呈现多态性，需要在运行时期间才能确定，所以不能为内联。

Base \*ptr = new Derived();

ptr->who();

// 因为Base有虚析构函数（virtual ~Base() {}），所以 delete 时，会先调用派生类（Derived）析构函数，再调用基类（Base）析构函数，防止内存泄漏。

delete ptr;

ptr = nullptr;

system("pause");

return 0;

}

**volatile**

volatile int i = 10;

* 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

**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

**sizeof()**

* sizeof 对数组，得到整个数组所占空间大小。
* sizeof 对指针，得到指针本身所占空间大小。

**#pragma pack(n)**

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

**位域**

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

**extern "C"**

* 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

**struct 和 typedef struct**

**C 中**

// c

typedef struct Student {

int age;

} S;

等价于

// c

struct Student {

int age;

};

typedef struct Student S;

此时 S 等价于 struct Student，但两个标识符名称空间不相同。

另外还可以定义与 struct Student 不冲突的 void Student() {}。

**C++ 中**

由于编译器定位符号的规则（搜索规则）改变，导致不同于C语言。

一、如果在类标识符空间定义了 struct Student {...};，使用 Student me; 时，编译器将搜索全局标识符表，Student 未找到，则在类标识符内搜索。

即表现为可以使用 Student 也可以使用 struct Student，如下：

// cpp

struct Student {

int age;

};

void f( Student me ); // 正确，"struct" 关键字可省略

二、若定义了与 Student 同名函数之后，则 Student 只代表函数，不代表结构体，如下：

typedef struct Student {

int age;

} S;

void Student() {} // 正确，定义后 "Student" 只代表此函数

//void S() {} // 错误，符号 "S" 已经被定义为一个 "struct Student" 的别名

int main() {

Student();

struct Student me; // 或者 "S me";

return 0;

}

**C++ 中 struct 和 class**

总的来说，struct 更适合看成是一个数据结构的实现体，class 更适合看成是一个对象的实现体。

**区别**

* 最本质的一个区别就是默认的访问控制
  1. 默认的继承访问权限。struct 是 public 的，class 是 private 的。
  2. struct 作为数据结构的实现体，它默认的数据访问控制是 public 的，而 class 作为对象的实现体，它默认的成员变量访问控制是 private 的。

**union 联合**

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;

}

**C 实现 C++ 类**

C 实现 C++ 的面向对象特性（封装、继承、多态）

* 封装：使用函数指针把属性与方法封装到结构体中
* 继承：结构体嵌套
* 多态：父类与子类方法的函数指针不同

[Can you write object-oriented code in C? [closed]](https://stackoverflow.com/a/351745)

**explicit（显式）关键字**

* 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

explicit 使用

struct A

{

A(int) { }

operator bool() const { return true; }

};

struct B

{

explicit B(int) {}

explicit operator bool() const { return true; }

};

void doA(A a) {}

void doB(B b) {}

int main()

{

A a1(1); // OK：直接初始化

A a2 = 1; // OK：复制初始化

A a3{ 1 }; // OK：直接列表初始化

A a4 = { 1 }; // OK：复制列表初始化

A a5 = (A)1; // OK：允许 static\_cast 的显式转换

doA(1); // OK：允许从 int 到 A 的隐式转换

if (a1); // OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换

bool a6(a1); // OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换

bool a7 = a1; // OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换

bool a8 = static\_cast<bool>(a1); // OK ：static\_cast 进行直接初始化

B b1(1); // OK：直接初始化

B b2 = 1; // 错误：被 explicit 修饰构造函数的对象不可以复制初始化

B b3{ 1 }; // OK：直接列表初始化

B b4 = { 1 }; // 错误：被 explicit 修饰构造函数的对象不可以复制列表初始化

B b5 = (B)1; // OK：允许 static\_cast 的显式转换

doB(1); // 错误：被 explicit 修饰构造函数的对象不可以从 int 到 B 的隐式转换

if (b1); // OK：被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 bool 的按语境转换

bool b6(b1); // OK：被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 bool 的按语境转换

bool b7 = b1; // 错误：被 explicit 修饰转换函数 B::operator bool() 的对象不可以隐式转换

bool b8 = static\_cast<bool>(b1); // OK：static\_cast 进行直接初始化

return 0;

}

**friend 友元类和友元函数**

* 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

**using**

**using 声明**

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**

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**

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**

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;

**:: 范围解析运算符**

**分类**

1. 全局作用域符（::name）：用于类型名称（类、类成员、成员函数、变量等）前，表示作用域为全局命名空间
2. 类作用域符（class::name）：用于表示指定类型的作用域范围是具体某个类的
3. 命名空间作用域符（namespace::name）:用于表示指定类型的作用域范围是具体某个命名空间的

:: 使用

int count = 11; // 全局（::）的 count

class A {

public:

static int count; // 类 A 的 count（A::count）

};

int A::count = 21;

void fun()

{

int count = 31; // 初始化局部的 count 为 31

count = 32; // 设置局部的 count 的值为 32

}

int main() {

::count = 12; // 测试 1：设置全局的 count 的值为 12

A::count = 22; // 测试 2：设置类 A 的 count 为 22

fun(); // 测试 3

return 0;

}

**enum 枚举类型**

**限定作用域的枚举类型**

enum class open\_modes { input, output, append };

**不限定作用域的枚举类型**

enum color { red, yellow, green };

enum { floatPrec = 6, doublePrec = 10 };

**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

}

**引用**

**左值引用**

常规引用，一般表示对象的身份。

**右值引用**

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**

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

**宏**

* 宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对 “参数” 进行的是一对一的替换。

**成员初始化列表**

好处

* 更高效：少了一次调用默认构造函数的过程。
* 有些场合必须要用初始化列表：
  1. 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
  2. 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
  3. 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化

**initializer\_list 列表初始化**

用花括号初始化器列表初始化一个对象，其中对应构造函数接受一个 std::initializer\_list 参数.

initializer\_list 使用

#include <iostream>

#include <vector>

#include <initializer\_list>

template <class T>

struct S {

std::vector<T> v;

S(std::initializer\_list<T> l) : v(l) {

std::cout << "constructed with a " << l.size() << "-element list\n";

}

void append(std::initializer\_list<T> l) {

v.insert(v.end(), l.begin(), l.end());

}

std::pair<const T\*, std::size\_t> c\_arr() const {

return {&v[0], v.size()}; // 在 return 语句中复制列表初始化

// 这不使用 std::initializer\_list

}

};

template <typename T>

void templated\_fn(T) {}

int main()

{

S<int> s = {1, 2, 3, 4, 5}; // 复制初始化

s.append({6, 7, 8}); // 函数调用中的列表初始化

std::cout << "The vector size is now " << s.c\_arr().second << " ints:\n";

for (auto n : s.v)

std::cout << n << ' ';

std::cout << '\n';

std::cout << "Range-for over brace-init-list: \n";

for (int x : {-1, -2, -3}) // auto 的规则令此带范围 for 工作

std::cout << x << ' ';

std::cout << '\n';

auto al = {10, 11, 12}; // auto 的特殊规则

std::cout << "The list bound to auto has size() = " << al.size() << '\n';

// templated\_fn({1, 2, 3}); // 编译错误！“ {1, 2, 3} ”不是表达式，

// 它无类型，故 T 无法推导

templated\_fn<std::initializer\_list<int>>({1, 2, 3}); // OK

templated\_fn<std::vector<int>>({1, 2, 3}); // 也 OK

}

**面向对象**

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

**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

[The Four Polymorphisms in C++](https://catonmat.net/cpp-polymorphism)

**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
* 动态绑定：当使用基类的引用或指针调用一个虚函数时将发生动态绑定

**注意：**

* 可以将派生类的对象赋值给基类的指针或引用，反之不可
* 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)

动态多态使用

class Shape // 形状类

{

public:

virtual double calcArea()

{

...

}

virtual ~Shape();

};

class Circle : public Shape // 圆形类

{

public:

virtual double calcArea();

...

};

class Rect : public Shape // 矩形类

{

public:

virtual double calcArea();

...

};

int main()

{

Shape \* shape1 = new Circle(4.0);

Shape \* shape2 = new Rect(5.0, 6.0);

shape1->calcArea(); // 调用圆形类里面的方法

shape2->calcArea(); // 调用矩形类里面的方法

delete shape1;

shape1 = nullptr;

delete shape2;

shape2 = nullptr;

return 0;

}

**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；

}

**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。

virtual int A() = 0;

**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。

[CSDN . C++ 中的虚函数、纯虚函数区别和联系](https://blog.csdn.net/u012260238/article/details/53610462)

**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#%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/)

**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。

**malloc、free**

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;

**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

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;

}

**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？**

[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**

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++ 标准库（STL）中**

头文件：#include <memory>

**C++ 98**

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

**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**

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）

向上转换是一种隐式转换。

**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**

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

**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**

* 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();

}

**运行时类型信息 (RTTI)**

**dynamic\_cast**

* 用于多态类型的转换

**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**

* 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;

}