# 编程语言

参考链接：Github：https://github.com/huihut/interview

## C/C++

### 1.1 const

#### 1.1.1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和指针常量；
3. 常量引用，经常用于形参类型，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。

#### 1.1.2 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; // 常指针

const A &q = a; // 常引用

// 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

char\* const p3 = greeting; // 常指针，指向字符数组变量

const char\* const p4 = greeting; // 常指针，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); // 参数指针为常指针

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();

### **1.2 static**

#### 1.2.1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。Static函数只能访问static成员。

### 1.3 this指针

#### 1.3.1 作用

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，都隐含使用 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：
7. 为实现对象的链式引用；
8. 为避免对同一对象进行赋值操作；
9. 在实现一些数据结构时，如 list。

### 1.4 inline 内联函数

#### 1.4.1 特征

1.相当于把内联函数里面的内容写在调用内联函数处；

2.相当于不用执行进入函数的步骤，直接执行函数体；

3.相当于宏，却比宏多了类型检查，真正具有函数特性；

4.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；

5.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。

#### 1.4.2 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

#### 1.4.3 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）。

#### 1.4.4 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。

#### 1.4.5 虚函数（virtual）可以是内联函数（inline）吗？

1.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。

2.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。

3.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。

#### 1.4.6 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;

}

### **1.5 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

### 1.6 **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。

#define NDEBUG // 加上这行，则 assert 不可用

#include <assert.h>

assert( p != NULL ); // assert 不可用

### 1.7 **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。

### 1.8 **#pragma pack(n)**

#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

### 1.9 **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

### 1.10 **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。

#ifdef \_\_cplusplusextern "C" {

#endif

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

#ifdef \_\_cplusplus

}

#endif

### **1.11 struct 和 typedef struct**

#### 1.11.1 在C 中

// ctypedef struct Student {

int age;

} S;

等价于

// cstruct 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() {}。

#### 1.11.2 在**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;

}

### 1.12 **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。

### **1.13 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

#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;

}

### 1.14 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

### 1.15 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

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 b ool 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;

}

### 1.16 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

### 1.17 **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;

### 1.18 **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。

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。

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;

### 1.19 :: 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

int count = 0; // 全局（::）的 count

class A {public:

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

};

int main() {

::count = 1; // 设置全局的 count 的值为 1

A::count = 2; // 设置类 A 的 count 为 2

int count = 0; // 局部的 count

count = 3; // 设置局部的 count 的值为 3

return 0;

}

### 1.20 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 };

### 1.21 **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：

// 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 typenametemplate <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

}

### 1.22 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&&