# 编程语言

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 \*ptr;

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()**