

C/C++ 笔试、面试题目大汇总

这些东西有点烦，有点无聊。如果要去 C++面试就看看吧。几年前网上搜索的。刚才看到，就整理一下，里面有些被我改了，感觉之前说的不对或不完善。

1. 求下面函数的返回值（微软）

```
int func(x)
{
    int countx = 0;
    while(x)
    {
        countx++;
        x = x&(x-1);
    }
    return countx;
}
```

假定 $x = 9999$ 。 答案：8

思路：将 x 转化为 2 进制，看含有的 1 的个数。

2. 什么是“引用”？申明和使用“引用”要注意哪些问题？

答：引用就是某个目标变量的“别名”(alias)，对应用的操作与对变量直接操作效果完全相同。申明一个引用的时候，切记要对其进行初始化。引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，不能再把该引用名作为其他变量名的别名。声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。不能建立数组的引用。

3. 将“引用”作为函数参数有哪些特点？

（1）传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

（2）使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。

（3）使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用“*指针变量名”的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

4. 在什么时候需要使用“常引用”？

如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。常引用声明方式：**const** 类型标识符 **&**引用名 = 目标变量名；

例 1

```
int a;  
const int &ra = a;  
ra = 1; // 错误  
a = 1; // 正确
```

例 2

```
string foo();  
void bar(string &s)  
// 那么下面的表达式将是非法的：  
bar(foo());  
bar("hello world");
```

原因在于 `foo()` 和 "hello world" 串都会产生一个临时对象，而在 C++ 中，这些临时对象都是 **const** 类型的。因此上面的表达式就是试图将一个 **const** 类型的对象转换为非 **const** 类型，这是非法的。

引用型参数应该在能被定义为 **const** 的情况下，尽量定义为 **const**。

5. 将“引用”作为函数返回值类型的格式、好处和需要遵守的规则？

格式：

类型标识符 **&** **函数名** (形参列表及类型说明)
{
 //函数体
}

好处：在内存中不产生被返回值的副本；（注意：正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生 **runtime error**！）

注意：

(1) 不能返回局部变量的引用。这条可以参照 **Effective C++[1]** 的 Item 31。主要是因为局部变量会在函数返回后被销毁，因此被返回的引用就成为了“无所指”的引用，程序会进入未知状态。

(2) 不能返回函数内部 **new** 分配的内存的引用（**这个要注意啦，很多人没意识到，哈哈。。。**）。这条可以参照 **Effective C++[1]** 的 Item 31。虽然不存在局部变量的被动销毁问题，但对于这种情况（返回函数内部 **new** 分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由 **new** 分配）就无法释放，造成 **memory leak**。

(3) 可以返回类成员的引用，但最好是 **const**。这条原则可以参照 **Effective C++[1]** 的 Item 30。主要原因是当对象的属性是与某种业务规则（**business rule**）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规

则当中。如果其它对象可以获得该属性的非常量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

（4）流操作符重载返回值申明为“引用”的作用：

流操作符 **<< 和 >>**，这两个操作符常常希望被连续使用，例如：`cout << "hello" << endl;`；因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个 **<<** 操作符实际上是针对不同对象的！这无法让人接受。对于返回一个流指针则不能连续使用 **<<** 操作符。因此，返回一个流对象引用是唯一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这就是 C++ 语言中引入引用这个概念的原因吧。赋值操作符 **=**。这个操作符象流操作符一样，是可以连续使用的，例如：`x = j = 10;` 或者 `(x=10)=100;` **赋值操作符的返回值必须是一个左值**，以便可以被继续赋值。因此引用成了这个操作符的唯一返回值选择。

例 3

```
#include <iostream.h>
int &put(int n);
int vals[10];
int error = -1;

void main()
{
    put(0) = 10; // 以 put(0) 函数值作为左值，等价于 vals[0]=10;
    put(9) = 20; // 以 put(9) 函数值作为左值，等价于 vals[9]=20;
    cout << vals[0];
    cout << vals[9];
}

int &put(int n)
{
    if (n>=0 && n<=9 )
    {
        return vals[n];
    }
    else
    {
        cout << "subscript error";
        return error;
    }
}
```

（5）在另外的一些操作符中，却千万不能返回引用：**+-*/** 四则运算符。它们不能返回引用，**Effective C++[1]** 的 Item23 详细的讨论了这个问题。主要原因是这四个操作符没有 **side effect**，因此，它们必须构造一个对象作为返回值，可选的方案包括：返回一个对象、

返回一个局部变量的引用，返回一个 `new` 分配的对象的引用、返回一个静态对象引用。根据前面提到的引用作为返回值的三个规则，第 2、3 两个方案都被否决了。静态对象的引用又因为 $((a+b) == (c+d))$ 会永远为 `true` 而导致错误。所以可选的只剩下返回一个对象了。

6. “引用”与多态的关系？

引用是除指针外另一个可以产生多态效果的手段。这意味着，一个基类的引用可以指向它的派生类实例（见：[C++中类的多态与虚函数的使用](#)）。

例 4

```
Class A;
Class B : Class A
{
    // ...
};

B b;
A& ref = b;
```

7. “引用”与指针的区别是什么？

指针通过某个指针变量指向一个对象后，对它所指向的变量[间接操作](#)。程序中使用指针，程序的可读性差；

而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作。此外，就是上面提到的对函数传 `ref` 和 `pointer` 的区别。

8. 什么时候需要“引用”？

流操作符`<<`和`>>`、赋值操作符`=`的返回值、拷贝构造函数的参数、赋值操作符`=`的参数、其它情况都推荐使用引用。

9. 结构与联合有和区别？

1. 结构和联合都是由多个不同的数据类型成员组成，但在任何同一时刻，联合中[只存放了一个被选中的成员](#)（所有成员共用一块地址空间），而结构的[所有成员都存在](#)（不同成员的存放地址不同）。
2. 对于联合的不同成员赋值，将会对其他成员重写，原来成员的值就不存在了，而对于结构的不同成员赋值是互不影响的。

10. 下面关于“联合”的题目的输出？

a)

```
#include <stdio.h>
union
{
    int i;
    char x[2];
```

```

} a;

void main()
{
    a.x[0] = 10;
    a.x[1] = 1;
    printf("%d", a.i);
}

```

答案: 266 (低位低地址, 高位高地址, 内存占用情况是 0x010A)

b)

```

main()
{
    union{ /*定义一个联合*/
        int i;
        struct{ /*在联合中定义一个结构*/
            char first;
            char second;
        }half;
    }number;
    number.i=0x4241; /*联合成员赋值*/
    printf("%c%c\n", number.half.first, number.half.second);
    number.half.first='a'; /*联合中结构成员赋值*/
    number.half.second='b';
    printf("%x\n", number.i);
    getch();
}

```

答案: AB (0x41 对应'A', 是低位; 0x42 对应'B', 是高位)

6261 (number.i 和 number.half 共用一块地址空间)

11. 已知 **strcpy** 的函数原型: **char *strcpy(char *strDest, const char *strSrc)**
 其中 **strDest** 是目的字符串, **strSrc** 是源字符串。不调用 **C++/C** 的字符串库函数,
 请编写函数 **strcpy**。

答案:

```

/*
编写 strcpy 函数 (10 分)
已知 strcpy 函数的原型是
    char *strcpy(char *strDest, const char *strSrc);
    其中 strDest 是目的字符串, strSrc 是源字符串。
(1) 不调用 C++/C 的字符串库函数, 请编写函数 strcpy
(2) strcpy 能把 strSrc 的内容复制到 strDest, 为什么还要 char * 类型的返回值?

```

```

答: 为了 实现链式表达式。 // 2 分
例如 int length = strlen( strcpy( strDest, "hello world" ) );
*/



#include <assert.h>
#include <stdio.h>
char *strcpy(char *strDest, const char *strSrc)
{
    assert((strDest!=NULL) && (strSrc !=NULL)); // 2 分
    char* address = strDest; // 2 分
    while( (*strDest++ = *strSrc++) != '\0' ) // 2 分
        NULL;
    return address; // 2 分
}

```

另外 `strlen` 函数如下:

```

#include<stdio.h>
#include<assert.h>
int strlen( const char *str ) // 输入参数 const
{
    assert( str != NULL ); // 断言字符串地址非 0
    int len = 0;
    while( (*str++) != '\0' )
    {
        len++;
    }
    return len;
}

```

12. 已知 `String` 类定义如下:

```

class String
{
public:
    String(const char *str = NULL); // 通用构造函数
    String(const String &another); // 拷贝构造函数
    ~String(); // 析构函数
    String& operator =(const String &rhs); // 赋值函数
}

```

```
private:  
    char* m_data; // 用于保存字符串  
};
```

尝试写出类的成员函数实现。

答案：

```
String::String(const char *str)  
{  
    if ( str == NULL ) // strlen 在参数为 NULL 时会抛异常才会有这步判断  
    {  
        m_data = new char[1] ;  
        m_data[0] = '\0' ;  
    }  
    else  
    {  
        m_data = new char[strlen(str) + 1];  
        strcpy(m_data,str);  
    }  
}  
  
String::String(const String &another)  
{  
    m_data = new char[strlen(another.m_data) + 1];  
    strcpy(m_data,another.m_data);  
}  
  
String& String::operator =(const String &rhs)  
{  
    if ( this == &rhs)  
        return *this ;  
    delete []m_data; //删除原来的数据，新开一块内存  
    m_data = new char[strlen(rhs.m_data) + 1];  
    strcpy(m_data,rhs.m_data);  
    return *this ;  
}  
  
String::~String()  
{  
    delete []m_data ;  
}
```

13. .h 头文件中的 **ifndef/define/endif** 的作用?

答: 防止该头文件被重复引用。

14. #include<file.h> 与 #include "file.h" 的区别?

答: 前者是从 Standard Library 的路径寻找和引用 file.h, 而后者是从当前工作路径搜寻并引用 file.h。

15. 在 C++ 程序中调用被 C 编译器编译后的函数, 为什么要加 **extern "C"**?

首先, 作为 **extern** 是 C/C++ 语言中表明函数和全局变量 **作用范围** (可见性) 的关键字, 该关键字告诉编译器, 其声明的函数和变量可以在本模块或其它模块中使用。

通常, 在模块的头文件中对本模块提供给其它模块引用的函数和全局变量以关键字 **extern** 声明。例如, 如果模块 B 欲引用该模块 A 中定义的全局变量和函数时只需包含模块 A 的头文件即可。这样, 模块 B 中调用模块 A 中的函数时, 在编译阶段, 模块 B 虽然找不到该函数, 但是并不会报错; 它会在连接阶段中从模块 A 编译生成的目标代码中找到此函数

extern "C" 是连接申明 (linkage declaration), 被 **extern "C"** 修饰的变量和函数是按照 C 语言方式编译和连接的, 来看看 C++ 中对类似。

C 的函数是怎样编译的:

作为一种面向对象的语言, C++ 支持函数重载, 而过程式语言 C 则不支持。函数被 C++ 编译后在符号库中的名字与 C 语言的不同。例如, 假设某个函数的原型为:

```
void foo( int x, int y );
```

该函数被 C 编译器编译后在符号库中的名字为 `_foo`, 而 C++ 编译器则会产生像 `_foo_int_int` 之类的名字 (不同的编译器可能生成的名字不同, 但是都采用了相同的机制, 生成的新名字称为 "**mangled name**")。

`_foo_int_int` 这样的名字包含了函数名、函数参数数量及类型信息, C++ 就是靠这种机制来实现函数重载的。例如, 在 C++ 中, 函数 `void foo(int x, int y)` 与 `void foo(int x, float y)` 编译生成的符号是不相同的, 后者为 `_foo_int_float`。

同样地, C++ 中的变量除支持局部变量外, 还支持类成员变量和全局变量。用户所编写程序的类成员变量可能与全局变量同名, 我们以 `."` 来区分。而本质上, 编译器在进行编译时, 与函数的处理相似, 也为类中的变量取了一个独一无二的名字, 这个名字与用户程序中同名的全局变量名字不同。

未加 **extern "C"** 声明时的连接方式

假设在 C++ 中, 模块 A 的头文件如下:

```
// 模块 A 头文件 moduleA.h
#ifndef MODULE_A_H
#define MODULE_A_H
int foo( int x, int y );
#endif
```

在模块 B 中引用该函数:

```
// 模块 B 实现文件 moduleB.cpp
#include "moduleA.h"
foo(2,3);
```

实际上, 在连接阶段, 连接器会从模块 A 生成的目标文件 moduleA.obj 中寻找 _foo_int_int 这样的符号!

加 **extern "C"** 声明后的编译和连接方式

加 **extern "C"** 声明后, 模块 A 的头文件变为:

```
// 模块 A 头文件 moduleA.h
#ifndef MODULE_A_H
#define MODULE_A_H
extern "C" int foo( int x, int y );
#endif
```

在模块 B 的实现文件中仍然调用 **foo(2,3)**, 其结果是:

(1) 模块 A 编译生成 **foo** 的目标代码时, 没有对其名字进行特殊处理, 采用了 C 语言的方式;

(2) 连接器在为模块 B 的目标代码寻找 **foo(2,3)** 调用时, 寻找的是未经修改的符号名 **_foo**。

如果在模块 A 中函数声明了 **foo** 为 **extern "C"** 类型, 而模块 B 中包含的是 **extern int foo(int x, int y)**, 则模块 B 找不到模块 A 中的函数; 反之亦然。

所以, 可以用一句话概括 **extern "C"** 这个声明的 **真实目的** (任何语言中的任何语法规则的诞生都不是随意而为的, 来源于真实世界的需求驱动。我们在思考问题时, 不能只停留在这个语言是怎么做的, 还要问一问它为什么要这么做, 动机是什么, 这样我们可以更深入地理解许多问题): **实现 C++ 与 C 及其它语言的混合编程**。

明白了 C++ 中 **extern "C"** 的设立动机, 我们下面来具体分析 **extern "C"** 通常的使用技巧:

extern "C" 的惯用法

(1) 在 C++ 中引用 C 语言中的函数和变量, 在包含 C 语言头文件(假设为 **cExample.h**)时, 需进行下列处理:

```
extern "C"
{
    #include "cExample.h"
}
```

而在 C 语言的头文件中, 对其外部函数只能指定为 **extern** 类型, C 语言中不支持 **extern "C"** 声明, 在.c 文件中包含了 **extern "C"** 时会出现编译语法错误。

C++ 引用 C 函数例子工程中包含的三个文件的源代码如下:

```
/* c 语言头文件: cExample.h */
#ifndef C_EXAMPLE_H
#define C_EXAMPLE_H
```

```
extern int add(int x, int y);  
#endif
```

```
/* c 语言实现文件: cExample.c */  
#include "cExample.h"  
int add( int x, int y )  
{  
    return x + y;  
}
```

```
// c++实现文件, 调用 add: cppFile.cpp  
extern"C"  
{  
    #include"cExample.h"  
}  
int main(int argc, char* argv[]){  
    add(2,3);  
    return 0;  
}
```

如果 C++ 调用一个 C 语言编写的.DLL 时, 当包括.DLL 的头文件或声明接口函数时, 应加 **extern "C" { }**。

(2) 在 C 中引用 C++ 语言中的函数和变量时, C++ 的头文件需添加 **extern "C"**, 但是在 C 语言中不能直接引用声明了 **extern "C"** 的该头文件, 应该仅将 C 文件中将 C++ 中定义的 **extern "C"** 函数声明为 **extern** 类型。

C 引用 C++ 函数例子工程中包含的三个文件的源代码如下:

```
//C++头文件 cppExample.h  
#ifndef CPP_EXAMPLE_H  
#define CPP_EXAMPLE_H  
extern "C" int add( int x, int y );  
#endif
```

```
//C++实现文件 cppExample.cpp  
#include"cppExample.h"  
int add( int x, int y )  
{
```

```

    return x + y;
}

/* C 实现文件 cFile.c
/* 这样会编译出错: #i nclude "cExample.h" */
extern int add( int x, int y );
int main( int argc, char* argv[] )
{
    add( 2, 3 );
    return 0;
}

```

16. 关联、聚合(Aggregation)以及组合(Composition)的区别?

涉及到 UML 中的一些概念:

关联是表示两个类的一般性联系, 比如“学生”和“老师”就是一种关联关系;

聚合表示 **has-a** 的关系, 是一种相对松散的关系, 聚合类不需要对被聚合类负责, 如下图所示, 用空的菱形表示聚合关系:

从实现的角度讲, 聚合可以表示为:

```
class A {...} class B { A* a; .....}
```

组合表示 **contains-a** 的关系, 关联性强于聚合: 组合类与被组合类有相同的生命周期, 组合类要对被组合类负责, 采用实心的菱形表示组合关系:

实现的形式是:

```
class A{...} class B{ A a; ...}
```

17. 面向对象的三个基本特征, 并简单叙述之?

1. 封装: 将客观事物抽象成类, 每个类对自身的数据和方法实行 **protection(private, protected,public)**
2. 继承: 广义的继承有三种实现形式: 实现继承 (指使用基类的属性和方法而无需额外编码的能力)、可视继承 (子窗体使用父窗体的外观和实现代码)、接口继承 (仅使用属性和方法, 实现滞后到子类实现)。前两种 (类继承) 和后一种 (对象组合=>接口继承以及纯虚函数) 构成了功能复用的两种方式。
3. 多态: 系统能够在运行时, 能够根据其类型确定调用哪个重载的成员函数的能力, 称为多态性。(见: [C++中类的多态与虚函数的使用](#))

18. 重载 (overload)和重写(overried, 有的书也叫做“覆盖”) 的区别?

常考的题目。

从定义上来说:

重载: 是指允许存在多个同名函数, 而这些函数的参数表不同 (或许参数个数不同, 或许参数类型不同, 或许两者都不同)。

重写：是指子类重新定义父类虚函数的方法。

从实现原理上来说：

重载：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：`function func(p:integer):integer;`和`function func(p:string):integer;`。那么编译器做过修饰后的函数名称可能是这样的：`int_func`、`str_func`。对于这两个函数的调用，在编译器间就已经确定了，是静态的。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关！

重写：和多态真正相关。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态的调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚绑定）。

19. 多态的作用？

主要是两个：

1. 隐藏实现细节，使得代码能够模块化；扩展代码模块，实现代码重用；
2. 接口重用：为了类在继承和派生的时候，保证使用家族中任一类的实例的某一属性时的正确调用。

20. Ado 与 Ado.net 的相同与不同？

除了“能够让应用程序处理存储于 DBMS 中的数据”这一基本相似点外，两者没有太多共同之处。但是 Ado 使用 OLE DB 接口并基于微软的 COM 技术，而 ADO.NET 拥有自己的 ADO.NET 接口并且基于微软的.NET 体系架构。众所周知.NET 体系不同于 COM 体系，ADO.NET 接口也就完全不同于 ADO 和 OLE DB 接口，这也就是说 ADO.NET 和 ADO 是两种数据访问方式。ADO.net 提供对 XML 的支持。

21. New delete 与 mallocfree 的联系与区别？

答案：都是在堆(heap)上进行动态的内存操作。用 `malloc` 函数需要指定内存分配的字节数并且不能初始化对象，`new` 会自动调用对象的构造函数。`delete` 会调用对象的 `destructor`，而 `free` 不会调用对象的 `destructor`。

（可以看看：[显式调用构造函数和析构函数](#)）

22. #define DOUBLE(x) x+x , i = 5*DOUBLE(5); i 是多少？

答案：i 为 30。（注意直接展开就是了） $5 * 5 + 5$

23. 有哪几种情况只能用 initializationlist 而不能用 assignment？

答案：当类中含有 `const`、`reference` 成员变量；基类的构造函数都需要初始化表。

24. C++是不是类型安全的？

答案：[不是](#)。两个不同类型的指针之间可以强制转换（用 `reinterpret cast`）。C#是类型安

全的。

25. main 函数执行以前，还会执行什么代码？

答案：全局对象的构造函数会在 **main** 函数之前执行，为 **malloc** 分配必要的资源，等等。

26. 描述内存分配方式以及它们的区别？

- 1) 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，**static** 变量。
- 2) 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集。
- 3) 从堆上分配，亦称动态内存分配。程序在运行的时候用 **malloc** 或 **new** 申请任意多少的内存，程序员自己负责在何时用 **free** 或 **delete** 释放内存。动态内存的生存期由程序员决定，使用非常灵活，但问题也最多。
- 4) 代码区。

27. struct 和 class 的区别

答案：**struct** 的成员默认是公有的，而类的成员默认是私有的。**struct** 和 **class** 在其他方面是功能相当的。

从感情上讲，大多数的开发者感到类和结构有很大的差别。感觉上结构仅仅象一堆缺乏封装和功能的开放的内存位，而类就象活的并且可靠的社会成员，它有智能服务，有牢固的封装屏障和一个良好定义的接口。既然大多数人都这么认为，那么只有在你的类很少的方法并且有公有数据（这种事情在良好设计的系统中是存在的！）时，你也许应该使用 **struct** 关键字，否则，你应该使用 **class** 关键字。

28. 当一个类 **A 中没有生命任何成员变量与成员函数，这时 **sizeof(A)** 的值是多少，如果不是零，请解释一下编译器为什么没有让它为零。（Autodesk）**

答案：肯定不是零。举个反例，如果是零的话，声明一个 **class A[10]** 对象数组，而每一个对象占用的空间是零，这时就没办法区分 **A[0],A[1]...** 了。

29. 在 8086 汇编下，逻辑地址和物理地址是怎样转换的？（Intel）

答案：通用寄存器给出的地址，是段内偏移地址，相应段寄存器地址*10H+通用寄存器内地址，就得到了真正要访问的地址。

30. 比较 C++ 中的 4 种类型转换方式？

重点是 **static_cast**, **dynamic_cast** 和 **reinterpret_cast** 的区别和应用。（以后再补上吧）

31. 分别写出 **BOOL,int,float, 指针类型的变量 **a** 与“零”的比较语句。**

答案：

```
BOOL :      if ( !a ) or if(a)
```

```
int :      if ( a == 0)
float :  const EXPRESSION EXP = 0.000001
          if ( a < EXP&& a >-EXP)
pointer : if ( a != NULL) or if(a == NULL)
```

32. 请说出 **const** 与 **#define** 相比, 有何优点?

1) **const** 常量有数据类型, 而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换, 没有类型安全检查, 并且在字符替换可能会产生意想不到的错误。

2) 有些集成化的调试工具可以对 **const** 常量进行调试, 但是不能对宏常量进行调试。

33. 简述数组与指针的区别?

数组要么在静态存储区被创建 (如全局数组), 要么在栈上被创建。指针可以随时指向任意类型的内存块。

(1) 修改内容上的差别

```
char a[] = "hello";
a[0] = 'X';
char *p = "world"; // 注意 p 指向常量字符串
p[0] = 'X'; // 编译器不能发现该错误, 运行时错误
```

(2) 用运算符 **sizeof** 可以计算出数组的容量 (字节数)。**sizeof(p)**, **p** 为指针得到的是一个指针变量的字节数, 而不是 **p** 所指的内存容量。**C++/C** 语言没有办法知道指针所指的内存容量, 除非在申请内存时记住它。注意当数组作为函数的参数进行传递时, 该数组自动退化为同类型的指针。

```
char a[] = "hello world";
char *p = a;
cout<< sizeof(a) << endl; // 12 字节
cout<< sizeof(p) << endl; // 4 字节
```

计算数组和指针的内存容量

```
void Func(char a[100])
{
    cout<< sizeof(a) << endl; // 4 字节而不是 100 字节
}
```

34. 类成员函数的重载、覆盖和隐藏区别?

答案:

a. 成员函数被重载的特征:

- (1) 相同的范围 (在同一个类中);
- (2) 函数名字相同;
- (3) 参数不同;
- (4) **virtual** 关键字可有可无。

b. 覆盖是指派生类函数覆盖基类函数, 特征是:

- (1) 不同的范围 (分别位于派生类与基类);
- (2) 函数名字相同;
- (3) 参数相同;
- (4) 基类函数必须有 **virtual** 关键字。

c.“隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

- (1) 如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无 **virtual** 关键字，基类的函数将被隐藏 (注意别与重载混淆)。
- (2) 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 **virtual** 关键字。此时，基类的函数被隐藏 (注意别与覆盖混淆)

35. There are two int variables: a and b, don't use "if", "? :", "switch" or other judgement statements, find out the biggest one of the two numbers.

答案: $((a + b) + \text{abs}(a - b)) / 2$

36. 如何打印出当前源文件的文件名以及源文件的当前行号？

答案:

```
cout << __FILE__ ;  
cout << __LINE__ ;
```

__FILE__ 和 **__LINE__** 是系统预定义宏，这种宏并不是在某个文件中定义的，而是由编译器定义的

。

37. main 主函数执行完毕后，是否可能会再执行一段代码，给出说明？

答案：可以，可以用 **_onexit** 注册一个函数，它会在 **main** 之后执行 **int fn1(void), fn2(void), fn3(void), fn4 (void);**

```
void main( void )  
{  
    String str("zhanglin");  
    _onexit( fn1 );  
    _onexit( fn2 );  
    _onexit( fn3 );  
    _onexit( fn4 );  
    printf( "This is executed first.\n" );  
}  
int fn1()  
{  
    printf( "next.\n" );  
    return 0;  
}  
int fn2()  
{  
    printf( "executed " );  
    return 0;
```

```

}

int fn3()
{
    printf( "is " );
    return 0;
}

int fn4()
{
    printf( "This " );
    return 0;
}

```

The `_onexit` function is passed the address of a function (`func`) to be called when the program terminates normally. Successive calls to `_onexit` create a register of functions that are executed in LIFO (last-in-first-out) order. The functions passed to `_onexit` cannot take parameters.

38. 如何判断一段程序是由 **C** 编译程序还是由 **C++** 编译程序编译的?

答案:

```

#ifndef __cplusplus
    cout<<"c++";
#else
    cout<<"c";
#endif

```

注意, 后面很多代码啊。代码不看也罢。

39. 文件中有一组整数, 要求排序后输出到另一个文件中 (面试官, 超级喜欢考排序的。你要去面试, 数据结构的那几个排序一定要非常熟悉, 用笔也可以写出代码来, 用笔写代码, 就是这样变态啊, 其实感觉没有必要这样笔试)

答案:

```

#include<iostream>
#include<fstream>
using namespace std;

void Order(vector<int>& data) //bubble sort
{
    int count = data.size() ;
    int tag = false ; // 设置是否需要继续冒泡的标志位
    for ( int i = 0 ; i < count ; i++)
    {
        for ( int j = 0 ; j < count - i - 1 ; j++)
        {

```

```

if ( data[j] > data[j+1])
{
    tag = true ;
    int temp = data[j] ;
    data[j] = data[j+1] ;
    data[j+1] = temp ;
}
}

if ( !tag )
break ;
}

void main( void )
{
vector<int>data;
ifstream in("c:\\data.txt");
if ( !in)
{
cout<<"file error!";
exit(1);
}
int temp;
while ( !in.eof())
{
in>>temp;
data.push_back(temp);
}
in.close(); //关闭输入文件流
Order(data);
ofstream out("c:\\result.txt");
if ( !out)
{
cout<<"file error!";
exit(1);
}
for ( i = 0 ; i < data.size() ; i++)
out<<data[i]<<" ";
out.close(); //关闭输出文件流
}

```

40. 链表题：一个链表的结点结构

```

struct Node
{

```

```

int data ;
Node *next ;
};

typedef struct Node Node ;

```

(1)已知链表的头结点 **head,写一个函数把这个链表逆序 (Intel)**

```

Node * ReverseList(Node *head) //链表逆序
{
    if ( head == NULL || head->next == NULL )
        return head;

    Node *p1 = head ;
    Node *p2 = p1->next ;
    Node *p3 = p2->next ;
    p1->next = NULL ;
    while ( p3 != NULL )
    {
        p2->next = p1 ;
        p1 = p2 ;
        p2 = p3 ;
        p3 = p3->next ;
    }
    p2->next = p1 ;
    head = p2 ;
    return head ;
}

```

(2)已知两个链表 **head1 和 **head2** 各自有序, 请把它们合并成一个链表依然有序。(保留所有结点, 即便大小相同)**

```

Node * Merge(Node *head1 , Node *head2)
{
    if ( head1 == NULL)
        return head2 ;
    if ( head2 == NULL)
        return head1 ;
    Node *head = NULL ;
    Node *p1 = NULL;
    Node *p2 = NULL;
    if ( head1->data < head2->data )
    {
        head = head1 ;
        p1 = head1->next;
        p2 = head2 ;
    }

```

```

else
{
head = head2 ;
p2 = head2->next ;
p1 = head1 ;
}

Node *pcurrent = head ;
while ( p1 != NULL && p2 != NULL)
{
if ( p1->data <= p2->data )
{
pcurrent->next = p1 ;
pcurrent = p1 ;
p1 = p1->next ;
}
else
{
pcurrent->next = p2 ;
pcurrent = p2 ;
p2 = p2->next ;
}
}

if ( p1 != NULL )
pcurrent->next = p1 ;
if ( p2 != NULL )
pcurrent->next = p2 ;
return head ;
}

```

(3)已知两个链表 **head1** 和 **head2** 各自有序，请把它们合并成一个链表依然有序，这次要求用递归方法进行。**(Autodesk)**

答案：

```

Node * MergeRecursive(Node *head1 , Node *head2)
{
if ( head1 == NULL )
return head2 ;
if ( head2 == NULL)
return head1 ;
Node *head = NULL ;
if ( head1->data < head2->data )
{
head = head1 ;
head->next = MergeRecursive(head1->next,head2) ;
}

```

```

else
{
head = head2 ;
head->next = MergeRecursive(head1,head2->next);
}
return head ;
}

```

41. 分析一下这段程序的输出(Autodesk)

```

class B
{
public:
B()
{
cout<<"default constructor"<<endl;
}
~B()
{
cout<<"destructed"<<endl;
}
B(int i):data(i) //B(int) works as a converter ( int ->instance of B)
{
cout<<"constructed by parameter " << data <<endl;
}
private:
int data;
};

B Play( B b)
{
return b ;
}
(1)                               results:
int main(int argc, char* argv[])      constructed by parameter 5
{
                               destructed B(5) 形参析构
B t1 = Play(5); B t2 = Play(t1);      destructed t1 形参析构
return 0;                           destructed t2 注意顺序!
}
                               destructed t1
(2)                               results:
int main(int argc, char* argv[])      constructed by parameter 5
{
                               destructed B(5) 形参析构
B t1 = Play(5); B t2 = Play(10);      constructed by parameter 10
return 0;                           destructed B(10) 形参析构
}
                               destructed t2 注意顺序!

```

```
destructed t1
```

42. 写一个函数找出一个整数数组中，第二大的数 (microsoft)

答案：

```
const int MINNUMBER = -32767 ;
int find_sec_max( int data[] , int count)
{
    int maxnumber = data[0] ;
    int sec_max = MINNUMBER ;
    for ( int i = 1 ; i < count ; i++)
    {
        if ( data[i] > maxnumber )
        {
            sec_max = maxnumber ;
            maxnumber = data[i] ;
        }
        else
        {
            if ( data[i] > sec_max )
                sec_max = data[i] ;
        }
    }
    return sec_max ;
}
```

43. 写一个在一个字符串(n)中寻找一个子串(m)第一个位置的函数。

KMP 算法效率最好，时间复杂度是 $O(n+m)$ 。

44. 多重继承的内存分配问题：

比如有 **class A : public class B, public classC {}**

那么 **A** 的内存结构大致是怎么样的？

这个是 **compiler-dependent** 的，不同的实现其细节可能不同。

如果不考虑有虚函数、虚继承的话就相当简单；否则的话，相当复杂。

可以参考《深入探索 C++ 对象模型》

45. 如何判断一个单链表是有环的？（注意不能用标志位，最多只能用两个额外指针）

```
struct node { char val; node* next; }

bool check(const node* head) {} //return false : 无环; true: 有环一种  $O(n)$  的办法就是（搞两个指针，一个每次递增一步，一个每次递增两步，如果有环的话两者必然重合，反之亦然）:
```

```

bool check(const node* head)
{
    if(head==NULL)  return false;
    node *low=head, *fast=head->next;
    while(fast!=NULL & fast->next!=NULL)
    {
        low=low->next;
        fast=fast->next->next;
        if(low==fast)  return true;
    }
    return false;
}

```

继续 ~~~~~~

一. 找错题

试题 1:

```

void test1()
{
    char string[10];
    char* str1 = "0123456789";
    strcpy( string, str1 );
}

```

试题 2:

```

void test2()
{
    char string[10],str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1 = 'a';
    }
    strcpy( string, str1 );
}

```

试题 3:

```

void test3(char* str1)
{
    char string[10];
    if( strlen( str1 ) <= 10 )
    {

```

```
    strcpy( string, str1 );
}
}
```

解答：

试题 1 字符串 str1 需要 11 个字节才能存放下（包括末尾的'\0'），而 string 只有 10 个字节的空间，strcpy 会导致数组越界；

对试题 2，如果面试者指出字符数组 str1 不能在数组内结束可以给 3 分；如果面试者指出 strcpy(string,str1) 调用使得从 str1 内存起复制到 string 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 strcpy 工作方式的给 10 分；

对试题 3，if(strlen(str1)<= 10) 应改为 if(strlen(str1) < 10)，因为 strlen 的结果未统计'\0'所占用的 1 个字节。

剖析：

考查对基本功的掌握：

- (1)字符串以'\0'结尾；
- (2)对数组越界把握的敏感度；
- (3)库函数 strcpy 的工作方式，如果编写一个标准 strcpy 函数的总分值为 10，下面给出几个不同得分的答案：

试题 4：

```
void GetMemory( char *p )
{
    p = (char *) malloc( 100 );
}

void Test( void )
{
    char *str = NULL;
    GetMemory( str );
    strcpy( str, "hello world" );
    printf( str );
}
```

试题 5：

```
char *GetMemory( void )
{
    char p[] = "hello world";
    return p;
}

void Test( void )
{
    char *str = NULL;
    str = GetMemory();
    printf( str );
}
```

```
}
```

试题 6:

```
void GetMemory( char **p, int num )
{
    *p = (char *) malloc( num );
}

void Test( void )
{
    char *str = NULL;
    GetMemory( &str, 100 );
    strcpy( str, "hello" );
    printf( str );
}
```

试题 7:

```
void Test( void )
{
    char *str = (char *) malloc( 100 );
    strcpy( str, "hello" );
    free( str );
    ... //省略的其它语句
}
```

解答：

试题 4 传入中 `GetMemory(char *p)` 函数的形参为字符串指针，在函数内部修改形参并不能真正的改变传入形参的值，执行完

```
char *str = NULL;
GetMemory( str );
后的 str 仍然为 NULL;
```

试题 5 中

```
char p[] = "hello world";
return p;
```

的 `p[]` 数组为函数内的局部自动变量，在函数返回后，内存已经被释放。这是许多程序员常犯的错误，其根源在于不理解变量的生存期。

试题 6 的 `GetMemory` 避免了试题 4 的问题，传入 `GetMemory` 的参数为字符串指针的指针，但是在 `GetMemory` 中执行申请内存及赋值语句

```
*p = (char *) malloc( num );
后未判断内存是否申请成功，应加上：
if ( *p == NULL )
{
    ...//进行申请内存失败处理
```

```
}
```

试题 7 存在与试题 6 同样的问题，在执行

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

后未进行内存是否申请成功的判断；另外，在 `free(str)` 后未置 `str` 为空，导致可能变成一个“野”指针，应加上：

```
str = NULL;
```

试题 6 的 `Test` 函数中也未对 `malloc` 的内存进行释放。

剖析：

试题 4~7 考查面试者对内存操作的理解程度，基本功扎实的面试者一般都能正确的回答其中 50~60 的错误。但是要完全解答正确，却也绝非易事。

对内存操作的考查主要集中在：

- 1) 指针的理解；
- 2) 变量的生存期及作用范围；
- 3) 良好的动态内存申请和释放习惯。

再看看下面的一段程序有什么错误：

```
swap( int* p1,int* p2 )  
{  
    int *p;  
    *p = *p1;  
    *p1 = *p2;  
    *p2 = *p;  
}
```

在 `swap` 函数中，`p` 是一个“野”指针，有可能指向系统区，导致程序运行的崩溃。在 VC++ 中 DEBUG 运行时提示错误“AccessViolation”。该程序应该改为：

```
swap( int* p1,int* p2 )  
{  
    int p;  
    p = *p1;  
    *p1 = *p2;  
    *p2 = p;  
}
```

二. 内功题

试题 1：分别给出 **BOOL**, **int**, **float**, 指针变量 与“零值”比较的 **if** 语句（假设变量名为 **var**）

解答：

BOOL 型变量： `if(!var)`

int 型变量： `if(var==0)`

float 型变量:

```
const float EPSINON = 0.00001;  
if ((x >= - EPSINON) && (x <= EPSINON)  
指针变量: if(var==NULL)
```

剖析:

考查对 0 值判断的“内功”，BOOL 型变量的 0 判断完全可以写成 if(var==0)，而 int 型变量也可以写成 if(!var)，指针变量的判断也可以写成 if(!var)，上述写法虽然程序都能正确运行，但是未能清晰地表达程序的意思。

一般的，如果想让 if 判断一个变量的“真”、“假”，应直接使用 if(var)、if(!var)，表明其为“逻辑”判断；如果用 if 判断一个数值型变量(short、int、long 等)，应该用 if(var==0)，表明是与 0 进行“数值”上的比较；而判断指针则适宜用 if(var==NULL)，这是一种很好的编程习惯。

浮点型变量并不精确，所以不可将 float 变量用“==”或“!=”与数字比较，应该设法转化成“>=”或“<=”形式。如果写成 if(x == 0.0)，则判为错，得 0 分。

试题 2: 以下为 WindowsNT 下的 32 位 C++ 程序，请计算 sizeof 的值

```
void Func ( char str[100] )  
{  
    sizeof( str ) = ?  
}  
void *p = malloc( 100 );  
sizeof ( p ) = ?
```

解答:

sizeof(str) = 4

sizeof (p) = 4

剖析:

Func (char str[100]) 函数中数组名作为函数形参时，在函数体内，数组名失去了本身的内涵，仅仅只是一个指针；在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。

数组名的本质如下：

(1) 数组名指代一种数据结构，这种数据结构就是数组；

例如：

```
char str[10];  
cout << sizeof(str) << endl;
```

输出结果为 10，str 指代数据结构 char[10]。

(2) 数组名可以转换为指向其指代实体的指针，而且是一个指针常量，不能作自增、自减等操作，不能被修改；

```
char str[10];  
str++; //编译出错，提示 str 不是左值
```

(3) 数组名作为函数形参时，沦为普通指针。

Windows NT 32 位平台下，指针的长度(占用内存的大小)为 4 字节，故 `sizeof(str)` 、 `sizeof(p)` 都为 4。

试题 3：写一个“标准”宏 **MIN**，这个宏输入两个参数并返回较小的一个。另外，当你写下面的代码时会发生什么事？

```
least = MIN(*p++, b);
```

解答：

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

`MIN(*p++, b)` 会产生宏的副作用

剖析：

这个面试题主要考查面试者对宏定义的使用，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。

程序员对宏定义的使用要非常小心，特别要注意两个问题：

(1) 谨慎地将宏定义中的“参数”和整个宏用括弧括起来。所以，严格地讲，下述解答：

```
#define MIN(A,B) (A) <= (B) ? (A) : (B)
```

`#define MIN(A,B) (A <= B ? A : B)` 都应判 0 分；

(2) 防止宏的副作用。

宏定义 `#define MIN(A,B) ((A) <= (B) ? (A) : (B))` 对 `MIN(*p++, b)` 的作用结果是：

```
((*p++) <= (b) ? (*p++) : (*p++))
```

这个表达式会产生副作用，指针 `p` 会作三次 `++` 自增操作。

除此之外，另一个应该判 0 分的解答是：

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B));
```

这个解答在宏定义的后面加“；”，显示编写者对宏的概念模糊不清，只能被无情地判 0 分并被面试官淘汰。

函数头是这样的：

```
// pStr 是指向以'\0'结尾的字符串的指针
// steps 是要求移动的 n
void LoopMove ( char * pStr, int steps )
{
    // 请填充...
}
```

解答：

正确解答 1：

```
void LoopMove ( char *pStr, int steps )
{
```

```

int n = strlen( pStr ) - steps;
char tmp[MAX_LEN];
strcpy ( tmp, pStr + n );
strcpy ( tmp + steps, pStr );
*( tmp + strlen ( pStr ) ) = '\0';
strcpy( pStr, tmp );
}

```

正确解答 2:

```

void LoopMove ( char *pStr, int steps )
{
    int n = strlen( pStr ) - steps;
    char tmp[MAX_LEN];
    memcpy( tmp, pStr + n, steps );
    memcpy(pStr + steps, pStr, n );
    memcpy(pStr, tmp, steps );
}

```

剖析:

这个试题主要考查面试者对标准库函数的熟练程度,在需要的时候引用库函数可以很大程度上简化程序编写的工作量。

最频繁被使用的库函数包括:

- (1) `strcpy`
- (2) `memcpy`
- (3) `memset`

试题 6: 已知 **WAV** 文件格式如下表, 打开一个 **WAV** 文件, 以适当的数据结构组织 **WAV** 文件头并解析 **WAV** 格式的各项信息。

WAVE 文件格式说明表

	偏移地址	字节数	节数	数据类型	类别	内容
文件头	00H	4	Char	"RIFF"标志		
	04H	4	int32	文件长度		
	08H	4	Char	"WAVE"标志		
	0CH	4	Char	"fmt"标志		
	10H	4		过渡字节 (不定)		
	14H	2	int16	格式类别		
	16H	2	int16	通道数		

18H	2	int16	采样率(每秒样本数), 表示每个通道的播放速度
1CH	4	int32	波形音频数据传送速率
20H	2	int16	数据块的调整数(按字节算的)
22H	2		每样本的数据位数
24H	4	Char	数据标记符 " data "
28H	4	int32	语音数据的长度

解答:

将 WAV 文件格式定义为结构体 WAVEFORMAT:

```
typedef struct tagWaveFormat
{
    char cRiffFlag[4];
    UIN32 nFileLen;
    char cWaveFlag[4];
    char cFmtFlag[4];
    char cTransition[4];
    UIN16 nFormatTag ;
    UIN16 nChannels;
    UIN16 nSamplesPerSec;
    UIN32 nAvgBytesperSec;
    UIN16 nBlockAlign;
    UIN16 nBitNumPerSample;
    char cDataFlag[4];
    UIN16 nAudioLength;

} WAVEFORMAT;
```

假设 WAV 文件内容读出后存放在指针 `buffer` 开始的内存单元内, 则分析文件格式的代码很简单, 为:

```
WAVEFORMAT waveFormat;
memcpy( &waveFormat, buffer, sizeof( WAVEFORMAT ) );
```

直接通过访问 `waveFormat` 的成员, 就可以获得特定 WAV 文件的各项格式信息。

剖析:

试题 6 考查面试者组织数据结构的能力, 有经验的程序设计者将属于一个整体的数据成员组织为一个结构体, 利用指针类型转换, 可以将 `memcpy`、`memset` 等函数直接用于结构体地址, 进行结构体的整体操作。透过这个题可以看出面试者的程序设计经验是否丰富。

试题 7: 编写类 **String** 的构造函数、析构函数和赋值函数, 已知类 **String** 的原型为:

```
class String
{
```

```

public:
    String(const char *str = NULL); // 普通构造函数
    String(const String &other); // 拷贝构造函数
    ~String(void); // 析构函数
    String & operate =(const String &other); // 赋值函数
private:
    char *m_data; // 用于保存字符串
};

解答:
//普通构造函数
String::String(const char *str)
{
    if(str==NULL)
    {
        m_data = new char[1]; // 得分点: 对空字符串自动申请存放结束标志'\0'的空
        //加分点: 对 m_data 加 NULL 判断
        *m_data = '\0';
    }
    else
    {
        int length = strlen(str);
        m_data = new char[length+1]; // 若能加 NULL 判断则更好
        strcpy(m_data, str);
    }
}

// String 的析构函数
String::~String(void)
{
    delete [] m_data; // 或 deletem_data;
}

//拷贝构造函数
String::String(const String &other) // 得分点: 输入参数为 const 型
{
    int length = strlen(other.m_data);
    m_data = new char[length+1]; //加分点: 对 m_data 加 NULL 判断
    strcpy(m_data, other.m_data);
}

//赋值函数
String & String::operator =(const String &other) // 得分点: 输入参数为 const 型
{
    if(this == &other) //得分点: 检查自赋值
        return *this;
    delete [] m_data; //得分点: 释放原有的内存资源
    int length = strlen( other.m_data );
}

```

```

    m_data = new char[length+1]; //加分点: 对 m_data 加 NULL 判断
    strcpy( m_data, other.m_data );
    return *this; //得分点: 返回本对象的引用
}

```

剖析:

能够准确无误地编写出 **String** 类的构造函数、拷贝构造函数、赋值函数和析构函数的面试者至少已经具备了 C++ 基本功的 60% 以上!

在这个类中包括了指针类成员变量 **m_data**, **当类中包括指针类成员变量时, 一定要重载其拷贝构造函数、赋值函数和析构函数, 这既是对 C++ 程序员的基本要求, 也是《Effective C++》中特别强调的条款。**

仔细学习这个类, 特别注意加注释的得分点和加分点的意义, 这样就具备了 60% 以上的 C++ 基本功!

试题 8: 请说出 **static** 和 **const** 关键字尽可能多的作用

解答:

static 关键字至少有下列 n 个作用:

- (1) 函数体内 **static** 变量的作用范围为该函数体, 不同于 **auto** 变量, 该变量的内存只被分配一次, 因此其值在下次调用时仍维持上次的值;
- (2) 在模块内的 **static** 全局变量可以被模块内所用函数访问, 但不能被模块外其它函数访问;
- (3) 在模块内的 **static** 函数只可被这一模块内的其它函数调用, 这个函数的使用范围被限制在声明它的模块内;
- (4) 在类中的 **static** 成员变量属于整个类所拥有, 对类的所有对象只有一份拷贝;
- (5) 在类中的 **static** 成员函数属于整个类所拥有, 这个函数不接收 **this** 指针, 因而只能访问类的 **static** 成员变量。

const 关键字至少有下列 n 个作用:

- (1) 欲阻止一个变量被改变, 可以使用 **const** 关键字。在定义该 **const** 变量时, 通常需要对它进行初始化, 因为以后就没有机会再去改变它了;
- (2) 对指针来说, 可以指定指针本身为 **const**, 也可以指定指针所指的数据为 **const**, 或二者同时指定为 **const**;
- (3) 在一个函数声明中, **const** 可以修饰形参, 表明它是一个输入参数, 在函数内部不能改变其值;
- (4) 对于类的成员函数, 若指定其为 **const** 类型, 则表明其是一个常函数, 不能修改类的成员变量;
- (5) 对于类的成员函数, 有时候必须指定其返回值为 **const** 类型, 以使得其返回值不为“左值”。例如:

```
const classA operator*(const classA& a1,const classA& a2);
```

operator* 的返回结果必须是一个 **const** 对象。如果不是, 这样的变态代码也不会编译出错:

```
classA a, b, c;
```

```
(a * b) = c; // 对 a*b 的结果赋值
```

操作 **(a * b) = c** 显然不符合编程者的初衷, 也没有任何意义。

剖析:

惊讶吗？小小的 **static** 和 **const** 居然有这么多功能，我们能回答几个？如果只能回答 1~2 个，那还真得闭关再好好修炼修炼。

这个题可以考查面试者对程序设计知识的掌握程度是初级、中级还是比较深入，没有一定的知识广度和深度，不可能对这个问题给出全面的解答。大多数人只能回答出 **static** 和 **const** 关键字的部分功能。

三. 技巧题

试题 1：写一个函数返回 **1+2+3+...+n** 的值（假定结果不会超过长整型变量的范围）

解答：

```
int Sum( int n )
{
    return ( (long)1 + n ) * n / 2;    //或 return (1 + n)* n / 2;
}
```

剖析：

对于这个题，只能说，也许最简单的答案就是最好的答案。下面的解答，或者基于下面的解答思路去优化，不管怎么“折腾”，其效率也不可能与直接 `return(1 + n) * n / 2` 相比！

```
int Sum( int n )
{
    long sum = 0;
    for( int i=1; i<=n; i++ )
    {
        sum += i;
    }
    return sum;
}
```

所以程序员们需要敏感地将数学等知识用在程序设计中。