C/C++ 最常见50道面试题

C/C++经典 面试题

面试题 1：变量的声明和定义有什么区别

为变量分配地址和存储空间的称为定义，不分配地址的称为声明。一个变量可以在多个地方声明，
但是只在一个地方定义。加入 extern 修饰的是变量的声明，说明此变量将在文件以外或在文件后面部分定义。说明：很多时候一个变量，只是声明不分配内存空间，直到具体使用时才 初始化 ，分配内存空间，
如外部变量。

面试题 2：写出 bool 、int、 float、指针变量与“零值”比较的 if 语句

```
//bool 型数据:
if( flag )
{
    A;
}
else
{
    B;
}
//int 型数据:
if( 0 != flag )
{
    A;
}
else {
    B;
}
//指针型:
if( NULL == flag )
{
    A;
}
else {
    B;
}
//float 型数据:
if ( ( flag >= -NORM ) && ( flag <= NORM ) )
{
    A;
}
```

注意：应特别注意在 int、指针型变量和“零值”比较的时候，把“零值”放在左边，这样当把“==” 误写成“=”时，编译器可以报错，否则这种逻辑错误不容易发现，并且可能导致很严重的后果。

面试题 3：sizeof 和 strlen 的区别

sizeof 和 strlen 有以下区别：

1 sizeof 是一个操作符，strlen 是库函数。

2 sizeof 的参数可以是数据的类型，也可以是变量，而 strlen 只能以结尾为'\0'的字符串作参数。

3 编译器在编译时就计算出了 sizeof 的结果。而 strlen 函数必须在运行时才能计算出来。并且 sizeof 计算的是数据类型占内存的大小，而 strlen 计算的是字符串实际的长度。

4 数组做 sizeof 的参数不退化，传递给 strlen 就退化为指针了。

注意：有些是操作符看起来像是函数，而有些函数名看起来又像操作符，这类容易混淆的名称一定要加以区分，否则遇到数组名这类特殊数据类型作参数时就很容易出错。最容易混淆为函数的操作符就是 sizeof。

面试题 4：C 语言的关键字 static 和 C++ 的关键字 static 有什么区别

在 C 中 static 用来修饰局部静态变量和外部静态变量、函数。而 C++中除了上述功能外，还用来定义类的成员变量和函数。即静态成员和静态成员函数。

注意：编程时 static 的记忆性，和全局性的特点可以让在不同时期调用的函数进行通信，传递信息，而 C++的静态成员则可以在多个对象实例间进行通信，传递信息。

面试题 5：C中的 malloc 和C ++中的 new 有什么区别

malloc 和 new 有以下不同：

（1） new、delete 是操作符，可以重载，只能在 C++中使用。

（2） malloc、free 是函数，可以覆盖，C、C++中都可以使用。

（3） new 可以调用对象的构造函数，对应的 delete 调用相应的析构函数。

（4） malloc 仅仅分配内存，free 仅仅回收内存，并不执行构造和析构函数

（5） new、delete 返回的是某种数据类型指针，malloc、free 返回的是 void 指针。

注意：malloc 申请的内存空间要用 free 释放，而 new 申请的内存空间要用 delete 释放，不要混用。

因为两者实现的机理不同。

面试题 6：写一个“标准”宏 MIN

```
#define min(a,b)((a)<=(b)?(a):(b))
```

注意：在调用时一定要注意这个宏定义的副作用，如下调用：

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

p 指针就自加了两次，违背了 MIN 的本意。

7：一个指针可以是 volatile 吗

可以，因为指针和普通变量一样，有时也有变化程序的不可控性。常见例：子中断服务子程序修改一个指向一个 buffer 的指针时，必须用 volatile 来修饰这个指针。

说明：指针是一种普通的变量，从访问上没有什么不同于其他变量的特性。其保存的数值是个整型数据，和整型变量不同的是，这个整型数据指向的是一段内存地址。

面试题 8：a 和&a 有什么区别

请写出以下代码的打印结果，主要目的是考察 a 和&a 的区别。

```
#include<stdio.h>
void main( void )
{
    int a[5]={1,2,3,4,5};
    int *ptr=(int *)(&a+1);
    printf("%d,%d",*(a+1),*(ptr-1));
    return;
}
```

输出结果：2，5。

这是因为数组在内存中是连续存储的，a+1表示数组的第2个元素的地址。而ptr是一个指针，它指向的是a整个数组在内存中的后一个位置，也就是数组外的地址。所以*(a+1)等于2，表示取数组内的第2个元素。而*(ptr-1)等于5，表示取了指针指向的地址的前一个位置的元素，也就是数组内的最后一个元素。

注意：数组名 a 可以作数组的首地址，而&a 是数组的指针。思考，将原式的 int *ptr=(int *)(&a+1); 改为 int *ptr=(int *)(a+1);时输出结果将是什么呢?

面试题 9：简述 C、C++程序编译的内存分配情况

C、C++中内存分配方式可以分为三种：

（1） 从静态存储区域分配：

内存在程序编译时就已经分配好，这块内存在程序的整个运行期间都存在。速度快、不容易出错，因为有系统会善后。例如全局变量，static 变量等。

（2） 在栈上分配：

在执行函数时，函数内局部变量的存储单元都在栈上创建，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中，效率很高，但是分配的内存容量有限。

（3） 从堆上分配：

即动态内存分配。程序在运行的时候用 malloc 或 new 申请任意大小的内存，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。如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，另外频繁地分配和释放不同大小的堆空间将会产生堆内碎块。

一个 C、C++程序编译时内存分为 5 大存储区：堆区、栈区、全局区、文字常量区、程序代码区。

10：简述 strcpy、sprintf 与 memcpy 的区别

三者主要有以下不同之处：

（1） 操作对象不同，strcpy 的两个操作对象均为字符串，sprintf 的操作源对象可以是多种数据类型，目的操作对象是字符串，memcpy 的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。

（2） 执行效率不同，memcpy 最高，strcpy 次之，sprintf 的效率最低。

（3） 实现功能不同，strcpy 主要实现字符串变量间的拷贝，sprintf 主要实现其他数据类型格式到字符串的转化，memcpy 主要是内存块间的拷贝。

说明：strcpy、sprintf 与 memcpy 都可以实现拷贝的功能，但是针对的对象不同，根据实际需求，来选择合适的函数实现拷贝功能。

面试题 11：设置地址为 0x67a9 的整型变量的值为 0xaa66

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa66;
```

说明：这道题就是强制类型转换的典型例子，无论在什么平台地址长度和整型数据的长度是一样的，即一个整型数据可以强制转换成地址指针类型，只要有意义即可。

面试题 12：面向对象的三大特征

面向对象的三大特征是封装性、继承性和多态性：

 封装性：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）。

 继承性：广义的继承有三种实现形式：实现继承（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)。

 多态性：是将父类对象设置成为和一个或更多它的子对象相等的技术。用子类对象给父类对象赋值之后，父类对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。 这部分需要熟悉掌握原理虚函数，了解一些概念（静态多态、动态多态）等，面试时经常会问。

说明：面向对象的三个特征是实现面向对象技术的关键，每一个特征的相关技术都非常的复杂，程序员应该多看、多练。

面试题 13：C++的空类有哪些成员函数

 缺省构造函数。

 缺省拷贝构造函数。

 缺省析构函数。

 缺省赋值运算符。

 缺省取址运算符。

 缺省取址运算符 const。

注意：有些书上只是简单的介绍了前四个函数。没有提及后面这两个函数。但后面这两个函数也是空类的默认函数。另外需要注意的是，只有当实际使用这些函数的时候，编译器才会去定义它们。

面试题 14：谈谈你对拷贝构造函数和赋值运算符的认识

拷贝构造函数和赋值运算符重载有以下两个不同之处：

（1） 拷贝构造函数生成新的类对象，而赋值运算符不能。

（2） 由于拷贝构造函数是直接构造一个新的类对象，所以在初始化这个对象之前不用检验源对象是否和新建对象相同。而赋值运算符则需要这个操作，另外赋值运算中如果原来的对象中有内存分配要先把内存释放掉

注意：当有类中有指针类型的成员变量时，一定要重写拷贝构造函数和赋值运算符，不要使用默认的。

面试题 15：用 C++设计一个不能被继承的类

```
template <typename T> class A
{
    friend T; private:
    A() {}
    ~A() {}
};

class B : virtual public A<B>
{
public:
    B() {}
    ~B() {}
};
class C : virtual public B
{
public:
    C() {}
    ~C() {}
};
void main( void )
{
    B b;  //C c;
    return;
}
```

注意：构造函数是继承实现的关键，每次子类对象构造时，首先调用的是父类的构造函数，然后才是自己的。

面试题 16：访问基类的私有虚函数
写出以下程序的输出结果：

```
#include <iostream.h>
class A
{
    virtual void g()
    {
        cout << "A::g" << endl;
    }
private:
    virtual void f()
    {
        cout << "A::f" << endl;
    }
};
class B : public A
{
    void g()
    {
        cout << "B::g" << endl;
    }
    virtual void h()
    {
        cout << "B::h" << endl;
    }
};
typedef void( *Fun )( void ); void main()
{
    B b;
    Fun pFun;
    for(int i = 0 ; i < 3; i++)
    {
        pFun = ( Fun )*( ( int* ) * ( int* )( &b ) + i );
        pFun();
    }
}
```

输出结果：
B::g
A::f
B::h

注意：本题主要考察了面试者对虚函数的理解程度。一个对虚函数不了解的人很难正确的做出本题。
在学习面向对象的多态性时一定要深刻理解虚函数表的工作原理。

面试题 17：简述类成员函数的重写、重载和隐藏的区别

（1）重写和重载主要有以下几点不同。
 范围的区别：被重写的和重写的函数在两个类中，而重载和被重载的函数在同一个类中。
 参数的区别：被重写函数和重写函数的参数列表一定相同，而被重载函数和重载函数的参数列表一定不同。
 virtual 的区别：重写的基类中被重写的函数必须要有 virtual 修饰，而重载函数和被重载函数可以被
virtual 修饰，也可以没有。
（2）隐藏和重写、重载有以下几点不同。
 与重载的范围不同：和重写一样，隐藏函数和被隐藏函数不在同一个类中。
 参数的区别：隐藏函数和被隐藏的函数的参数列表可以相同，也可不同，但是函数名肯定要相同。当参数不相同时，无论基类中的参数是否被 virtual 修饰，基类的函数都是被隐藏，而不是被重写。
说明：虽然重载和覆盖都是实现多态的基础，但是两者实现的技术完全不相同，达到的目的也是完全不同的，覆盖是动态态绑定的多态，而重载是静态绑定的多态。

面试题 18：简述多态实现的原理

编译器发现一个类中有虚函数，便会立即为此类生成虚函数表 vtable。虚函数表的各表项为指向对应虚函数的指针。编译器还会在此类中隐含插入一个指针 vptr（对 vc 编译器来说，它插在类的第一个位置上）指向虚函数表。调用此类的构造函数时，在类的构造函数中，编译器会隐含执行 vptr 与 vtable 的关联代码，将 vptr 指向对应的 vtable，将类与此类的 vtable 联系了起来。另外在调用类的构造函数时，指向基础类的指针此时已经变成指向具体的类的 this 指针，这样依靠此 this 指针即可得到正确的 vtable，。
如此才能真正与函数体进行连接，这就是动态联编，实现多态的基本原理。
注意：一定要区分虚函数，纯虚函数、虚拟继承的关系和区别。牢记虚函数实现原理，因为多态
C++面试的重要考点之一，而虚函数是实现多态的基础。

面试题 19：链表和数组有什么区别

数组和链表有以下几点不同：
（1） 存储形式：数组是一块连续的空间，声明时就要确定长度。链表是一块可不连续的动态空间，长度可变，每个结点要保存相邻结点指针。
（2） 数据查找：数组的线性查找速度快，查找操作直接使用偏移地址。链表需要按顺序检索结点，效率低。
（3） 数据插入或删除：链表可以快速插入和删除结点，而数组则可能需要大量数据移动。
（4） 越界问题：链表不存在越界问题，数组有越界问题。
说明：在选择数组或链表数据结构时，一定要根据实际需要进行选择。数组便于查询，链表便于插入删除。数组节省空间但是长度固定，链表虽然变长但是占了更多的存储空间。

面试题 20：怎样把一个单链表反序

（1） 反转一个链表。循环算法。

```
List reverse(List   n)
{
    if(!n)                          //判断链表是否为空，为空即退出。
    {
        return n;
    }
    list cur = n.next;              //保存头结点的下个结点
    list pre = n;
    list tmp;           //保存头结点
    pre.next = null;                //头结点的指针指空，转换后变尾结点
    while ( NULL != cur.next )          //循环直到 cur.next 为空
    {
        tmp = cur;
    }
    tmp.next = pre;
    pre = tmp;
    cur = cur.next;

    return tmp;                     //f 返回头指针
}
```

（2） 反转一个链表。递归算法。

```
List *reverse( List *oldList, List *newHead = NULL )
{
    List *next = oldList-> next;        //记录上次翻转后的链表
    oldList-> next = newHead;       //将当前结点插入到翻转后链表的开头

```

```
    newHead = oldList;              //递归处理剩余的链表
    return ( next==NULL )? newHead: reverse( t, newHead );
}
```

说明：循环算法就是移动过程，比较好理解和想到。递归算法的设计虽有一点难度，但是理解了循环算法，再设计递归算法就简单多了。

面试题 21：简述队列和栈的异同

队列和栈都是线性存储结构，但是两者的插入和删除数据的操作不同，队列是“先进先出”，栈是“后进先出”。

注意：区别栈区和堆区。堆区的存取是“顺序随意”，而栈区是“后进先出”。栈由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。堆一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收。分配方式类似于链表。

它与本题中的堆和栈是两回事。堆栈只是一种数据结构，而堆区和栈区是程序的不同内存存储区域。

面试题 22：能否用两个栈实现一个队列的功能

```
//结点结构体:
typedef struct node
{
   int data;
   node *next;
}node, *LinkStack;
//创建空栈:
LinkStack CreateNULLStack(LinkStack &S)
{
   S = (LinkStack)malloc(sizeof(node)); //申请新结点
   if (NULL == S)
   {
    printf("Fail to malloc a new node.\n")
        return NULL;
   }
   S->data = 0; //初始化新结点
   S->next = NULL;
   return S;
}
//栈的插入函数:
LinkStack Push(LinkStack &S, int data)
{
   if (NULL == S) //检验栈
   {
    printf("There no node in stack!");
    return NULL;
   }
   LinkStack p = NULL;
   p = (LinkStack)malloc(sizeof(node)); //申请新结点
   if (NULL == p)
   {
    printf("Fail to malloc a new node.\n");
    return S;
   }
   if (NULL == S->next)
   {
    p->next = NULL;
   }
   else
   {
    p->next = S->next;
   }
   p->data = data; //初始化新结点
   S->next = p; //插入新结点
   return S;
}
//出栈函数:
node Pop(LinkStack &S)
{
   node temp;
   temp.data = 0;
   temp.next = NULL;

```

```
    if (NULL == S) //检验栈
    {
     printf("There no node in stack!");
     return temp;
    }
    temp = *S;
    10
     if (S->next == NULL)
     {
         printf("The stack is NULL,can't pop!\n");
         return temp;
     }
    LinkStack p = S->next; //节点出栈
    S->next = S->next->next;
    temp = *p;
    free(p);
    p = NULL;
    return temp;
}
//双栈实现队列的入队函数:
LinkStack StackToQueuePush(LinkStack &S, int data)
{
    node n;
    LinkStack S1 = NULL;
    CreateNULLStack(S1); //创建空栈
    while (NULL != S->next) //S 出栈入 S1
    {
     n = Pop(S);
     Push(S1, n.data);
    }
    Push(S1, data); //新结点入栈
    while (NULL != S1->next) //S1 出栈入 S
    {
     n = Pop(S1);
     Push(S, n.data);
    }
}
```

说明：用两个栈能够实现一个队列的功能，那用两个队列能否实现一个队列的功能呢？结果是否定的，因为栈是先进后出，将两个栈连在一起，就是先进先出。而队列是现先进先出，无论多少个连在一起都是先进先出，而无法实现先进后出。

面试题 23：计算一颗二叉树的深度

深度的计算函数：

```
int depth(BiTree T)
{
    if(!T) return 0;                    //判断当前结点是否为叶子结点
    int d1= depth(T->lchild);           //求当前结点的左孩子树的深度
    int d2= depth(T->rchild);               //求当前结点的右孩子树的深度

}   return (d1>d2?d1:d2)+1;
```

注意：根据二叉树的结构特点，很多算法都可以用递归算法来实现。

面试题 24：编码实现直接插入排序

```
#include<iostream.h>
void main( void )
{
    int ARRAY[10] = { 0, 6, 3, 2, 7, 5, 4, 9, 1, 8 };

    int i,j;
    for(   i = 0; i < 10; i++)
    {
        cout<<ARRAY[i]<<" ";
    }

    cout<<endl;

    for( i = 2; i <= 10; i++ )                  //将 ARRAY[2],…,ARRAY[n]依次按序插入
    {


        if(ARRAY[i] < ARRAY[i-1])               //如果 ARRAY[i]大于一切有序的数值,
                                                //ARRAY[i]将保持原位不动
        {
            ARRAY[0] = ARRAY[i];                //将 ARRAY[0]看做是哨兵,是 ARRAY[i]的副本
            j = i - 1;

            do{                                 //从右向左在有序区 ARRAY[1. . i-1]中
                                                //查找 ARRAY[i]的插入位置
            ARRAY[j+1] = ARRAY[j];              //将数值大于 ARRAY[i]记录后移
            j-- ;
            }while( ARRAY[0] < ARRAY[j] );
            ARRAY[j+1]=ARRAY[0];                //ARRAY[i]插入到正确的位置上
        }
    }

    for(   i = 0; i < 10; i++)
    {
        cout<<ARRAY[i]<<" ";
    }
    cout<<endl;
}
```


直接插入排序编程实现如下：

```
#include<iostream.h>
void main( void )
{
    int ARRAY[10] = { 0, 6, 3, 2, 7, 5, 4, 9, 1, 8 };
    int i,j;
    for(   i = 0; i < 10; i++)
    {
        cout<<ARRAY[i]<<" ";
    }
    cout<<endl;
    for( i = 2; i <= 10; i++ )                  //将 ARRAY[2],…,ARRAY[n]依次按序插入
    {
        if(ARRAY[i] < ARRAY[i-1])               //如果 ARRAY[i]大于一切有序的数值,
                                                //ARRAY[i]将保持原位不动
        {
   ARRAY[0] = ARRAY[i];  //将 ARRAY[0]看做是哨兵,是 ARRAY[i]的副本     j = i - 1;

            do{                                 //从右向左在有序区 ARRAY[1. . i-1]中                                    //j
   ARRAY[j+1] = ARRAY[j];  //将数值大于 ARRAY[i]记录后移     j-- ;
            }while( ARRAY[0] < ARRAY[j] );
            ARRAY[j+1]=ARRAY[0];    //ARRAY[i]插入到正确的位置上
        }
    }
    for(   i = 0; i < 10; i++)
    {
        cout<<ARRAY[i]<<" ";
    }
    cout<<endl;
}
```

注意：所有为简化边界条件而引入的附加结点（元素）均可称为哨兵。引入哨兵后使得查找循环条件的时间大约减少了一半，对于记录数较大的文件节约的时间就相当可观。类似于排序这样使用频率非常高的算法，要尽可能地减少其运行时间。所以不能把上述算法中的哨兵视为雕虫小技。

面试题 25：编码实现冒泡排序

冒泡排序编程实现如下：

```
#include <stdio.h>
#define LEN 10                                  //数组长度

 void main( void )
 {
    int ARRAY[10] = { 0, 6, 3, 2, 7, 5, 4, 9, 1, 8 };   //待排序数组
    printf( "\n" );
    for( int a = 0; a < LEN; a++ )              //打印数组内容
    {
    printf( "%d ", ARRAY[a] );
    }
    int i = 0; int j = 0;
    bool isChange;                              //设定交换标志

    for( i = 1; i < LEN; i++ )
    {                                       //最多做 LEN-1 趟排序
        isChange = 0;                       //本趟排序开始前,交换标志应为假
        for( j = LEN-1; j >= i; j-- )           //对当前无序区 ARRAY[i..LEN]自下向上扫描
    {
    if( ARRAY[j+1] < ARRAY[j] )
    {                               //交换记录
            ARRAY[0] = ARRAY[j+1];      //ARRAY[0]不是哨兵,仅做暂存单元
            ARRAY[j+1] = ARRAY[j];
            ARRAY[j] = ARRAY[0];
            isChange = 1;           //发生了交换,故将交换标志置为真
        }
}
printf( "\n" );
for( a = 0; a < LEN; a++)           //打印本次排序后数组内容
 {
    printf( "%d ", ARRAY[a] );
}
if( !isChange )
{
    break;
}   //本趟排序未发生交换,提前终止算法
printf( "\n" ); return;
}
```

面试题 26：编码实现直接选择排序

```
#include"stdio.h"
#define LEN 9
void main( void )
{
    int ARRAY[LEN]={ 5, 6, 8, 2, 4, 1, 9, 3, 7 };              //待序数组

    printf("Before sorted:\n");
    for( int m = 0; m < LEN; m++ )                          //打印排序前数组
    {
        printf( "%d   ", ARRAY[m] );
    }
    for (int i = 1; i <= LEN - 1; i++)                      //选择排序
    {
        int t = i - 1;
        int temp = 0;
        for (int j = i; j < LEN; j++)
        {
            if (ARRAY[j] < ARRAY[t])
            {
                t = j;
            }
        }
        if (t != (i - 1))
        {
            temp = ARRAY[i - 1];
            ARRAY[i - 1] = ARRAY[t];
            ARRAY[t] = temp;
        }
    }
    printf( "\n" );
    printf("After sorted:\n");
    for( i = 0; i < LEN; i++ )                          //打印排序后数组
    {
        printf( "%d   ", ARRAY[i] );
    }
    printf( "\n" );
}
```

注意：在直接选择排序中，具有相同关键码的对象可能会颠倒次序，因而直接选择排序算法是一种不稳定的排序方法。在本例中只是例举了简单的整形数组排序，肯定不会有什么问题。但是在复杂的数据元素序列组合中，只是根据单一的某一个关键值排序，直接选择排序则不保证其稳定性，这是直接选择排序的一个弱点。

面试题 27：编程实现堆排序

堆排序编程实现：

```
void createHeep(int ARRAY[], int sPoint, int Len) //生成大根堆
{
    while ((2 * sPoint + 1) < Len)
    {
        int mPoint = 2 * sPoint + 1;
        if ((2 * sPoint + 2) < Len)
        {
            if (ARRAY[2 * sPoint + 1] < ARRAY[2 * sPoint + 2])
            {
                mPoint = 2 * sPoint + 2;
            }
        }
        if (ARRAY[sPoint] < ARRAY[mPoint]) //堆被破坏，需要重新调整
        {
            int tmpData = ARRAY[sPoint]; //交换 sPoint 与 mPoint 的数据
            ARRAY[sPoint] = ARRAY[mPoint];
            ARRAY[mPoint] = tmpData;
            sPoint = mPoint;
        }
        else
        {
            break; //堆未破坏，不再需要调整
        }
    }
    return;
}
void heepSort(int ARRAY[], int Len) //堆排序
{
    int i = 0;
    for (i = (Len / 2 - 1); i >= 0; i--) //将 Hr[0，Lenght-1]建成大根堆
    {
        createHeep(ARRAY, i, Len);
    }
    for (i = Len - 1; i > 0; i--)
    {
        int tmpData = ARRAY[0]; //与最后一个记录交换
        ARRAY[0] = ARRAY[i];
        ARRAY[i] = tmpData;
        createHeep(ARRAY, 0, i); //将 H.r[0..i]重新调整为大根堆
    }
    return;
}
int main(void)
{
    int ARRAY[] = { 5, 4, 7, 3, 9, 1, 6, 8, 2 };
    printf("Before sorted:\n"); //打印排序前数组内容
    for (int i = 0; i < 9; i++)
    {
        printf("%d ", ARRAY[i]);
    }
    printf("\n");
    heepSort(ARRAY, 9); //堆排序
    printf("After sorted:\n"); //打印排序后数组内容
    for (i = 0; i < 9; i++)
    {
        printf("%d ", ARRAY[i]);
    }
    printf("\n");
}
```

说明：堆排序，虽然实现复杂，但是非常的实用。另外读者可是自己设计实现小堆排序的算法。虽然和大堆排序的实现过程相似，但是却可以加深对堆排序的记忆和理解。

面试题 28：编程实现基数排序

```
#include <stdio.h>
#include <malloc.h>
#define LEN 8
typedef struct node //队列结点
{
    int data;
    struct node * next;
}node, *QueueNode;
typedef struct Queue //队列
{
    QueueNode front;
    QueueNode rear;
}Queue, *QueueLink;
QueueLink CreateNullQueue(QueueLink &Q) //创建空队列
{
    Q = NULL;
    Q = (QueueLink)malloc(sizeof(Queue));
    if (NULL == Q)
    {
        printf("Fail to malloc null queue!\n");
        return NULL;
    }
    Q->front = (QueueNode)malloc(sizeof(node));
    Q->rear = (QueueNode)malloc(sizeof(node));
    if (NULL == Q->front || NULL == Q->rear)
    {
        printf("Fail to malloc a new queue's fornt or rear!\n");
        return NULL;
    }
    Q->rear = NULL;
    Q->front->next = Q->rear;
    return Q;
}
int lenData(node data[], int len) //计算队列中各结点的数据的最大位数
{
    int m = 0;
    int temp = 0;
    int d;
    for (int i = 0; i < len; i++)
    {
        d = data[i].data;
        while (d > 0)
        {
            d /= 10;
            temp++;
        }
        if (temp > m)
        {
            m = temp;
        }
        temp = 0;
    }
    return m;
}
QueueLink Push(QueueLink &Q, node node) //将数据压入队列
{
    QueueNode p1, p;
    p = (QueueNode)malloc(sizeof(node));
    if (NULL == p)
    {
        printf("Fail to malloc a new node!\n");
        return NULL;
    }
    p1 = Q->front;
    while (p1->next != NULL)
    {
        p1 = p1->next;
    }
    p->data = node.data;
```

```
    p1->next = p;
    p->next = Q->rear;
    return NULL;
}
node Pop(QueueLink &Q) //数据出队列
{
    node temp;
    temp.data = 0;
    temp.next = NULL;
    QueueNode p;
    p = Q->front->next;
    if (p != Q->rear)
    {
        temp = *p;
        Q->front->next = p->next;
        free(p);
        p = NULL;
    }
    return temp;
}
int IsEmpty(QueueLink Q)
{
    if (Q->front->next == Q->rear)
    {
        return 0;
    }
    return 1;
}
int main(void)
{
    int i = 0;
    int Max = 0; //记录结点中数据的最大位数
    int d = 10;
    int power = 1;
    int k = 0;
    node Array[LEN] = { { 450, NULL }, { 32, NULL }, { 781, NULL }, { 57, NULL }, 组
    { 145, NULL }, { 613, NULL }, { 401, NULL }, { 594, NULL } };
    //队列结点数
    QueueLink Queue[10];
    for (i = 0; i < 10; i++)
    {
        CreateNullQueue(Queue[i]); //初始化队列数组
    }
    for (i = 0; i < LEN; i++)
    {
        printf("%d ", Array[i].data);
    }
    printf("\n");
    Max = lenData(Array, LEN); //计算数组中关键字的最大位数
    printf("%d\n", Max);
    for (int j = 0; j < Max; j++) //按位排序
    {
        if (j == 0) power = 1;
        else power = power *d;
        for (i = 0; i < LEN; i++)
        {
            k = Array[i].data / power - (Array[i].data / (power * d)) * d;
            Push(Queue[k], Array[i]);
        }
        for (int l = 0, k = 0; l < d; l++) //排序后出队列重入数组
        {
            while (IsEmpty(Queue[l]))
            {
                Array[k++] = Pop(Queue[l]);
            }
        }
        for (int t = 0; t < LEN; t++)
        {
            printf("%d ", Array[t].data);
        }
        printf("\n");
```

```
    }
    return 0;
}
```

说明：队列为基数排序的实现提供了很大的方便，适当的数据机构可以减少算法的复杂度，让更多的算法实现更容易。

面试题 29：谈谈你对编程规范的理解或认识

编程规范可总结为：程序的可行性，可读性、可移植性以及可测试性。

说明：这是编程规范的总纲目，面试者不一定要去背诵上面给出的那几个例子，应该去理解这几个例子说明的问题，想一想，自己如何解决可行性、可读性、可移植性以及可测试性这几个问题，结合以上几个例子和自己平时的编程习惯来回答这个问题。

面试题 30：short i = 0; i = i + 1L；这两句有错吗

代码一是错的，代码二是正确的。

说明：在数据安全的情况下大类型的数据向小类型的数据转换一定要显示的强制类型转换。

面试题 31：&&和&、||和|有什么区别

（1）&和|对操作数进行求值运算，&&和||只是判断逻辑关系。（2）&&和||在在判断左侧操作数就能确定结果的情况下就不再对右侧操作数求值。

注意：在编程的时候有些时候将&&或||替换成&或|没有出错，但是其逻辑是错误的，可能会导致不可预想的后果（比如当两个操作数一个是 1 另一个是 2 时。

面试题 32：C++的引用和 C 语言的指针有什么区别

指针和引用主要有以下区别：

（1） 引用必须被初始化，但是不分配存储空间。指针不声明时初始化，在初始化的时候需要分配存储空间。

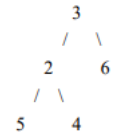
（2） 引用初始化以后不能被改变，指针可以改变所指的对象。

（3） 不存在指向空值的引用，但是存在指向空值的指针。

注意：引用作为函数参数时，会引发一定的问题，因为让引用作参数，目的就是想改变这个引用所指向地址的内容，而函数调用时传入的是实参，看不出函数的参数是正常变量，还是引用，因此可能会引发错误。所以使用时一定要小心谨慎。

面试题 33：在二元树中找出和为某一值的所有路径

输入一个整数和一棵二元树。从树的根结点开始往下访问，一直到叶结点所经过的所有结点形成一条路径。打印出和与输入整数相等的所有路径。例如，输入整数 9 和如下二元树：



则打印出两条路径：3，6 和 3，2，4。

【答案】

```
typedef struct path
{
    BiTNode* tree;                      //结点数据成员
    struct path* next;
}PATH,*pPath;               //结点指针成员
//初始化树的结点栈:
void init_path( pPath* L )
{
    *L = ( pPath )malloc( sizeof( PATH ) );
    ( *L )->next = NULL;
}       //创建空树
//树结点入栈函数:
void push_path(pPath H, pBTree T)
{
    pPath p = H->next;
    pPath q = H;
    while( NULL != p )
    {
    q = p;
    p = p->next;
}
p = ( pPath )malloc( sizeof( PATH ) );      //申请新结点
p->next = NULL;                     //初始化新结点
}   p->tree = T;
q->next = p;                        //新结点入栈
//树结点打印函数:
void print_path( pPath L )
{
```

```
    pPath p = L->next;

    while( NULL != p )
    {
        printf("%d, ", p->tree->data);   p = p->next;
    }
}       //打印当前栈中所有数据

///树结点出栈函数:

void pop_path( pPath H )
{
    pPath p = H->next;
    pPath q = H;
    if( NULL == p )                     //检验当前栈是否为空
    {
        printf("Stack is null!\n");
        return;
    }
    p = p->next;
    while( NULL != p )                  //出栈
    {
        q = q->next;
        p = p->next;
    }
    free( q->next );                    //释放出栈结点空间
    q->next = NULL;
}
//判断结点是否为叶子结点:
int IsLeaf(pBTree T)
{
    return ( T->lchild == NULL )&&( T->rchild==NULL );
}
//查找符合条件的路径:
int find_path(pBTree T, int sum, pPath L)
{
    push_path( L, T); record += T->data;
    if( ( record == sum ) && ( IsLeaf( T ) ) )      //打印符合条件的当前路径
    {
        print_path( L );
        printf( "\n" );
    }
    if( T->lchild != NULL )                     //递归查找当前节点的左孩子
    {
        find_path( T->lchild, sum, L);
    }
    if( T->rchild != NULL )                     //递归查找当前节点的右孩子
    {
        find_path( T->rchild, sum, L);
    }
    record -= T->data;  pop_path(L);  return 0;
}
```

注意：数据结构一定要活学活用，例如本题，把所有的结点都压入栈，而不符合条件的结点弹出栈，很容易实现了有效路径的查找。虽然用链表也可以实现，但是用栈更利于理解这个问题，即适当的数据结构为更好的算法设计提供了有利的条件。

面试题 34：写一个“标准”宏 MIN

写一个“标准”宏 MIN，这个宏输入两个参数并且返回较小的一个。

【答案】

```
#define min(a,b)((a)<=(b)?(a):(b))
```

注意：在调用时一定要注意这个宏定义的副作用，如下调用：

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

p 指针就自加了两次，违背了 MIN 的本意。

面试题 35：typedef 和 define 有什么区别

（1） 用法不同：typedef 用来定义一种数据类型的别名，增强程序的可读性。define 主要用来定义常量，以及书写复杂使用频繁的宏。

（2） 执行时间不同：typedef 是编译过程的一部分，有类型检查的功能。define 是宏定义，是预编译的部分，其发生在编译之前，只是简单的进行字符串的替换，不进行类型的检查。
（3） 作用域不同：typedef 有作用域限定。define 不受作用域约束，只要是在 define 声明后的引用都是正确的。
（4） 对指针的操作不同：typedef 和 define 定义的指针时有很大的区别。
注意：typedef 定义是语句，因为句尾要加上分号。而 define 不是语句，千万不能在句尾加分号。

面试题 36：关键字 const 是什么
const 用来定义一个只读的变量或对象。主要优点：便于类型检查、同宏定义一样可以方便地进行参数的修改和调整、节省空间，避免不必要的内存分配、可为函数重载提供参考。
说明：const 修饰函数参数，是一种编程规范的要求，便于阅读，一看即知这个参数不能被改变，实现时不易出错。 const修饰成员函数不可修改成员变量。

面试题 37：static 有什么作用
static 在 C 中主要用于定义全局静态变量、定义局部静态变量、定义静态函数。在 C++中新增了两种作用：定义静态数据成员、静态函数成员。
注意：因为 static 定义的变量分配在静态区，所以其定义的变量的默认值为 0，普通变量的默认值为随机数，在定义 指针变量 时要特别注意。

面试题 38：extern 有什么作用
extern 标识的变量或者函数声明其定义在别的文件中，提示编译器遇到此变量和函数时在其它模块中寻找其定义。

面试题 39：流操作符重载为什么返回引用
在程序中，流操作符>>和<<经常连续使用。因此这两个操作符的返回值应该是一个仍旧支持这两个操作符的流引用。其他的数据类型都无法做到这一点。
注意：除了在赋值操作符和流操作符之外的其他的一些操作符中，如+、-、*、/等却千万不能返回引用。因为这四个操作符的对象都是右值，因此，它们必须构造一个对象作为返回值。

面试题 40：简述指针常量与常量指针区别
指针常量是指定义了一个指针，这个指针的值只能在定义时初始化，其他地方不能改变。常量指针是指定义了一个指针，这个指针指向一个只读的对象，不能通过常量指针来改变这个对象的值。
指针常量强调的是指针的不可改变性，而常量指针强调的是指针对其所指对象的不可改变性。
注意：无论是指针常量还是常量指针，其最大的用途就是作为函数的形式参数，保证实参在被调用函数中的不可改变特性。

面试题 41：数组名和指针的区别
请写出以下代码的打印结果：

```
#include <iostream.h>
#include <string.h>
void main(void)
{
    char str[13]="Hello world!";
    char *pStr="Hello world!";
    cout<<sizeof(str)<<endl;
    cout<<sizeof(pStr)<<endl;
    cout<<strlen(str)<<endl;
    cout<<strlen(pStr)<<endl;
    return;
}
```

【答案】
打印结果：
13
4
12 12
注意：一定要记得数组名并不是真正意义上的指针，它的内涵要比指针丰富的多。但是当数组名当做参数传递给函数后，其失去原来的含义，变作普通的指针。另外要注意 sizeof 不是函数，只是操作符。

面试题 42：如何避免“野指针”
“野指针”产生原因及解决办法如下：
（1） 指针变量声明时没有被初始化。解决办法：指针声明时初始化，可以是具体的地址值，也可让它指向 NULL。
（2） 指针 p 被 free 或者 delete 之后，没有置为 NULL。解决办法：指针指向的内存空间被释放后指针应该指向 NULL。
（3） 指针操作超越了变量的作用范围。解决办法：在变量的作用域结束前释放掉变量的地址空间并且让指针指向 NULL。
注意：“野指针”的解决方法也是编程规范的基本原则，平时使用指针时一定要避免产生“野指针”，在使用指针前一定要检验指针的合法性。

面试题 43：常引用有什么作用

常引用的引入主要是为了避免使用变量的引用时，在不知情的情况下改变变量的值。常引用主要用于定义一个普通变量的只读属性的别名、作为函数的传入形参，避免实参在调用函数中被意外的改变。

说明：很多情况下，需要用常引用做形参，被引用对象等效于常对象，不能在函数中改变实参的值，这样的好处是有较高的易读性和较小的出错率。

面试题 44：编码实现字符串转化为数字

编码实现函数 atoi()，设计一个程序，把一个字符串转化为一个整型数值。例如数字：“5486321”，转化成字符：5486321。

【答案】

```
int myAtoi(const char * str)
{
    int num = 0; //保存转换后的数值
    int isNegative = 0; //记录字符串中是否有负号
    int n = 0;
    char *p = str;
    if (p == NULL) //判断指针的合法性
    {
        return -1;
    }
    while (*p++ != '\0') //计算数字符串度
    {
        n++;
    }
    p = str;
    if (p[0] == '-') //判断数组是否有负号
    {
        isNegative = 1;
    }
    char temp = '0';
    for (int i = 0; i < n; i++)
    {
        char temp = *p++;
        if (temp > '9' || temp < '0') //滤除非数字字符
        {
            continue;
        }
        if (num != 0 || temp != '0') //滤除字符串开始的 0 字符
        {
            temp -= 0x30; //将数字字符转换为数值
            num += temp *int(pow(10, n - 1 - i));
        }
    }
    if (isNegative) //如果字符串中有负号，将数值取反
    {
        return (0 - num);
    }
    else
    {
        return num; //返回转换后的数值
    }
}
```

注意：此段代码只是实现了十进制字符串到数字的转化，读者可以自己去实现 2 进制，8 进制，10 进制，16 进制的转化。

面试题 45：简述 strcpy、sprintf 与 memcpy 的区别

三者主要有以下不同之处：

（1） 操作对象不同，strcpy 的两个操作对象均为字符串，sprintf 的操作源对象可以是多种数据类型，目的操作对象是字符串，memcpy 的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。

（2） 执行效率不同，memcpy 最高，strcpy 次之，sprintf 的效率最低。

（3） 实现功能不同，strcpy 主要实现字符串变量间的拷贝，sprintf 主要实现其他数据类型格式到字符串的转化，memcpy 主要是内存块间的拷贝。

说明：strcpy、sprintf 与 memcpy 都可以实现拷贝的功能，但是针对的对象不同，根据实际需求，来选择合适的函数实现拷贝功能。

面试题 46：用 C 编写一个死循环程序

while(1)

{ }
说明：很多种途径都可实现同一种功能，但是不同的方法时间和空间占用度不同，特别是对于嵌入式软件，处理器速度比较慢，存储空间较小，所以时间和空间优势是选择各种方法的首要考虑条件。

面试题 47：编码实现某一变量某位清 0 或置 1
给定一个整型变量 a，写两段代码，第一个设置 a 的 bit 3，第二个清 a 的 bit 3，在以上两个操作中，要保持其他位不变。
【答案】

```
#define BIT3 (0x1 << 3 )
Satic int a;
//设置 a 的 bit 3:
void set_bit3( void )
{
    a |= BIT3;
}   //将 a 第 3 位置 1
//清 a 的 bit 3
void set_bit3( void )
{
    a &= ~BIT3;
}   //将 a 第 3 位清零
```

说明：在置或清变量或寄存器的某一位时，一定要注意不要影响其他位。所以用加减法是很难实现的。

面试题 48：评论下面这个中断函数
中断是嵌入式系统中重要的组成部分，这导致了很多编译开发商提供一种扩展——让标准 C 支持中断。具体代表事实是，产生了一个新的关键字__interrupt。下面的代码就使用了__interrupt 关键字去定义一个中断服务子程序(ISR)，请评论以下这段代码。

```
__interrupt double compute_area (double radius)
{
    double area = PI * radius * radius;  printf(" Area = %f", area);  return area;
}
```

【答案】
这段中断服务程序主要有以下四个问题：
（1） ISR 不能返回一个值。
（2） ISR 不能传递参数。
（3） 在 ISR 中做浮点运算是不明智的。
（4） printf()经常有重入和性能上的问题。
注意：本题的第三个和第四个问题虽不是考察的重点，但是如果能提到这两点可给面试官留下一个好印象。

面试题 49：构造函数能否为虚函数
构造函数不能是虚函数。而且不能在构造函数中调用虚函数，因为那样实际执行的是父类的对应函数，因为自己还没有构造好。析构函数可以是虚函数，而且，在一个复杂类结构中，这往往是必须的。
析构函数也可以是纯虚函数，但纯虚析构函数必须有定义体，因为析构函数的调用是在子类中隐含的。
说明：虚函数的动态绑定特性是实现重载的关键技术，动态绑定根据实际的调用情况查询相应类的虚函数表，调用相应的虚函数。

面试题 50：谈谈你对面向对象的认识
面向对象可以理解成对待每一个问题，都是首先要确定这个问题由几个部分组成，而每一个部分其实就是一个对象。然后再分别设计这些对象，最后得到整个程序。传统的程序设计多是基于功能的思想来进行考虑和设计的，而面向对象的程序设计则是基于对象的角度来考虑问题。这样做能够使得程序更加的简洁清晰。
说明：编程中接触最多的“面向对象编程技术”仅仅是面向对象技术中的一个组成部分。发挥面向对象技术的优势是一个综合的技术问题，不仅需要面向对象的分析，设计和编程技术，而且需要借助必要的建模和开发工具。

XX公司的题。

一、基本问题（80%）
1、const、static作用。
2、c++面向对象三大特征及对他们的理解，引出多态实现原理、动态绑定、菱形继承。
3、虚析构的必要性，引出内存泄漏，虚函数和普通成员函数的储存位置，虚函数表、虚函数表指针。
4、malloc、free和new、delete区别，引出malloc申请大内存、malloc申请空间失败怎么办。
5、stl熟悉吗，vector、map、list、hashMap，vector底层，map引出红黑树。优先队列用过吗，使用的场景。无锁队列听说过吗，原理是什么（比较并交换）
6、实现擅长的排序，说出原理（快排、堆排）
7、四种cast，智能指针

8、tcp和udp区别
9、进程和线程区别。
10、指针和引用作用以及区别。
11、c++11用过哪些特性，**auto作为返回值和模板一起怎么用，函数指针能和auto混用吗。**
12、boost用过哪些类，thread、asio、signal、bind、function
13、单例、工厂模式、代理、适配器、模板，使用场景。
14、QT信号槽实现机制，QT内存管理，MFC消息机制。
15、进程间通信。会选一个详细问。
16、多线程，锁和信号量，互斥和同步。
17、动态库和静态库的区别。

```cpp
//auto作为返回值和模板一起怎么用，函数指针能和auto混用吗

#include <iostream>

using namespace std;

template <typename T,typename U>
auto add(T t,U u) -> decltype(t+u)
{
    return t+u;
}

template <typename T,typename U>
auto sub(T t,U u) -> decltype(t-u)
{
        return t-u;
}

template <typename T,typename U>
auto pro(T t,U u) -> decltype(t*u)
{
        return t*u;
}

template <typename T,typename U>
auto div(T t,U u) -> decltype(t/u)
{
    try
    {
        return t/u;
    }
    catch(...)
    {
        exit(0);
    }
}


int main()
{
    int x = 520;
    double y= 13.14;
    auto z = add(x,y);
    cout<<z<<endl;

    //auto(*funp[4])(int ,double) = {add,sub,pro,div};//error
    double (*funp[4])(int ,double) = {add,sub,pro,div};
    for(unsigned char i=0;i<4;i++)
    {
       cout<<funp[i](x,y)<<endl;
    }
    return 0;
}
```

二、保留问题（20%）
1、提高c++性能，你用过哪些方式去提升（构造、析构、返回值优化、临时对象（使用operator=()消除临时对象）、内联（内联技巧、条件内联、递归内联、静态局部变量内联）、内存池、使用函数对象不使用函数指针、编码（编译器优化、预先计算）、设计

（延迟计算、高效数据结构）、系统体系结构（寄存器、缓存、上下文切换））。
2、编译原理，尝试自己写过语言或语言编译器。
3、泛型模板实用度高。
4、对多种计算机语言熟悉。
5、Git项目了解多。
6、针对网络框架（DPDK）、逆向工程（汇编）、分布式集群（docker、k8s、redis等）、CPU计算（nvidia cuda)、图像识别（opencv、opengl、tensorflow等)、AI等有研究。